© 2022 Jaemin Choi

SCALABLE HETEROGENEOUS COMPUTING WITH ASYNCHRONOUS MESSAGE-DRIVEN EXECUTION

BY

JAEMIN CHOI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Emeritus Laxmikant V. Kale, Chair
Professor Lawrence Rauchwerger
Research Associate Professor Volodymyr Kindratenko
Assistant Professor Abhinav Bhatele, University of Maryland, College Park
Dr. Michael Garland, NVIDIA Research

ABSTRACT

Computer systems today are becoming increasingly heterogeneous, in response to increasingly demanding performance requirements of both traditional and emerging workloads including computational science, data science, and machine learning, pushing the limits of power and energy imposed by the silicon. Although the problem of data movement costs has been exacerbating as a consequence of increasingly complex memory hierarchies and heterogeneous computing resources, the popular approaches to parallel programming have largely remained to be a mixture of the Message Passing Interface (MPI) and a GPU programming model such as CUDA. Asynchronous message-driven execution, realized in the Charm++ parallel programming system, is an emerging model that has been proven to be effective in traditional CPU-based systems and large-scale parallel execution due to its adaptive features such as computation-communication overlap and dynamic load balancing. However, when applied to modern heterogeneous and GPU-accelerated systems, asynchronous message-driven execution presents many challenges when it comes to realizing overdecomposition and asynchronous progress which are necessary to achieve low overhead and minimal synchronization between the host and device as well as between the parallel work units for performance.

In this dissertation, we analyze the issues in realizing efficient asynchronous message-driven execution on modern heterogeneous systems, and introduce new capabilities and approaches to address them in the form of runtime support in the Charm++ parallel programming system. To mitigate communication costs and minimize unnecessary synchronization overheads, we exploit automatic computation-communication overlap driven by overdecomposition and enable GPU-aware communication in the asynchronous message-driven execution model. We also combine these two approaches together to further improve performance and scalability on heterogeneous systems, and explore the effectiveness of techniques such as kernel fusion and CUDA Graphs to reduce the impact of kernel launch overheads especially with strong scaling. Finally, we investigate the possibilities of an entirely GPU-driven runtime system, CharminG, which seeks to realize asynchronous message-driven execution on the GPU with more user-level control of the GPU computing resources, enabled by GPU-resident scheduling, memory management, and messaging mechanisms. We discuss the challenges, limitations and potential improvements of such a GPU-centric approach of parallel programming towards the goal of developing an overarching runtime system that can efficiently utilize all of the available heterogeneous computing resources.

To my family, for their everlasting love and support.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Dissertation Overview 3

CHAPTER 2 REALIZING OVERDECOMPOSITION AND EFFICIENT ASYNCHRONOUS MESSAGE-DRIVEN EXECUTION WITH GPUS 5
2.1 Charm++ Parallel Programming System 6
2.2 Using GPUs in Charm++ 9
2.3 Hybrid API (HAPI) 14
2.4 Concluding Remarks 20

CHAPTER 3 EXPLOITING AUTOMATIC COMPUTATION-COMMUNICATION OVERLAP ON GPU SYSTEMS 22
3.1 Exploiting Overlap on Traditional CPU-Based Systems 23
3.2 Maximizing Overlap on GPU-Accelerated Systems 25
3.3 Experimental Setup 28
3.4 Performance Evaluation 31
3.5 Related Work 35
3.6 Concluding Remarks 36

CHAPTER 4 GPU-AWARE COMMUNICATION FOR MESSAGE-DRIVEN EXECUTION 37
4.1 Background 38
4.2 Design and Implementation 40
4.3 Performance Evaluation 50
4.4 Related Work 58
4.5 Concluding Remarks 58

CHAPTER 5 IMPROVING SCALABILITY WITH GPU-AWARE ASYNCHRONOUS TASKS 60
5.1 Background 61
5.2 Design and Implementation 66
5.3 Performance Evaluation 71
5.4 Related Work 77
5.5 Concluding Remarks 77

CHAPTER 6 TOWARDS A HETEROGENEOUS MESSAGE-DRIVEN PARALLEL PROGRAMMING SYSTEM 79
6.1 Design Overview 80

6.2 Implementation Details . . . 85
6.3 Performance Evaluation . . . 103
6.4 Challenges and Limitations . . . 107
6.5 Concluding Remarks . . . 110

CHAPTER 7 CONCLUSION . . . 112
7.1 Future Directions . . . 113

REFERENCES . . . 115

CHAPTER 1: INTRODUCTION

High-performance computing (HPC) has been at the forefront of scientific and technological advances, leading developments in both hardware and software to solve the world's largest and most critical problems. The architecture of HPC systems has evolved drastically in the last decades, from multicore CPUs to GPU acceleration, now with multi-GPU nodes driving the bulk of the computational performance. So called the "fat nodes" phenomenon, each compute node of a supercomputing system is becoming more powerful while the total number of nodes decreases. As the throughput of floating point calculations improves rapidly with such changes in the hardware, the gap between computational performance and network bandwidth continues to grow. The memory hierarchy and interconnect topology have also grown in complexity with heterogeneous computing, placing more emphasis on the ability to mitigate the increasing costs of data movement and efficiently utilize the available computing resources, using techniques such as asynchronous execution and computation-communication overlap.

Despite the rapid changes in hardware and system architecture over the years, the Message Passing Interface (MPI) remains the dominant form of programming for distributed memory environments, with the addition of a GPU programming model such as CUDA to build applications for heterogeneous systems. MPI maintains its popularity due to the simplicity of the Single Program Multiple Data (SPMD) model, performance portability, and general availability. The blocking send-receive semantics of MPI makes it easy to build distributed-memory applications using a sequential flow of execution, and generally provides good raw communication performance. A notable limitation of MPI, however, is the difficulty of hiding communication latency, which is critical for performance and scalability on modern cluster systems with high data movement costs. The programmer must manually and carefully structure non-blocking MPI routines and their synchronization to exploit computation-communication overlap, which degrades software modularity and sustainability. This also limits the amount of attainable overlap to the regions identified by the programmer.

As portrayed by the development of programming models and runtime systems in the Exascale Computing Project (ECP) [1], next-generation parallel programming frameworks will play a vital role in realizing massive parallelism with high levels of utilization on leadership-class systems. The message-driven execution model [2] employed by the Charm++ parallel programming system [3] is one such approach, which provides application programmers a natural, object-oriented method of expressing interactions between work units. The Charm++ runtime system is empowered by overdecomposition, which allows multiple work units to

be created and assigned to each computational resource, e.g., a CPU core. Asynchronous parallel execution driven by schedulers and active messages enables the runtime system to efficiently switch between different work units mapped to the same resource. This facilitates automatic overlap of computation and communication of the work units, providing substantial improvements in performance and scalability even on GPU-accelerated systems by mitigating data movement costs.

Load imbalance is another significant hurdle for many classes of applications, especially those that exhibit dynamism in the computational load and communication pattern of the work units during execution. MPI by itself does not directly solve this problem; static load imbalance may be mitigated with manual work distribution at the beginning of the application (which is possible only after the assessment of the degree of load imbalance), but addressing dynamic load imbalance is infeasible with MPI constructs. Overdecomposition in Charm++ serves as a useful feature in such scenarios as work units can be migrated to underutilized computing resources to reduce load imbalance. This is done automatically and dynamically by the runtime system as it tracks the load of each work unit during execution.

To combat the increasing data movement costs of modern supercomputing systems and cater to the dynamic behavior of applications running on such machines, an asynchronous message-driven execution model built on overdecomposition can be a powerful weapon. However, there has not been much exploration into how message-driven execution can be realized efficiently on GPU systems, especially with regard to ensuring asynchronous execution and supporting fast communication mechanisms that are vital to performance on modern heterogeneous architectures. To maximize resource utilization, the runtime system must be able to efficiently manage the parallel execution of the work units and their interactions with the GPU devices as well as the network, in order to mitigate the high data movement costs common to state-of-the-art supercomputers.

In this thesis, we validate the hypothesis that asynchronous message-driven execution supported by runtime techniques improves performance and scalability on modern heterogeneous platforms, and serves as a promising model for designing GPU-centric parallel programming systems. In other words, the benefits of an overdecomposition-based migratable objects model can be combined with the computational advantage provided by data-parallel GPU accelerators. Building on the Charm++ runtime system, we propose techniques to facilitate asynchronous execution and reduce communication costs on GPU-accelerated systems. We first enable automatic computation-communication overlap using overdecomposition and asynchronous message-driven execution, minimizing synchronizations between the host and GPU devices as well as between work units to further exploit overlap. We also develop mechanisms to reduce data movement costs in message-driven execution by supporting GPU-aware

communication, and combine them with automatic computation-communication overlap to achieve high levels of resource utilization and performance even at scale. In preparation for increasingly GPU-focused computing resources, we explore the possibilities of a GPU-resident runtime system that employs message-driven execution with its core scheduling and communication mechanisms driven by the GPU instead of the host CPU. We showcase what can be achieved with currently available GPU programming and communication frameworks, and explore new hardware and software features that can improve support for heterogeneous execution on future exascale systems and beyond.

1.1 DISSERTATION OVERVIEW

The content in this dissertation can be largely seen as the combination of two parts, first where we leverage the CPU-driven Charm++ parallel programming system to improve the performance and scalability on modern GPU-accelerated systems (Chapters 2-5), and second where possibilities of GPU-driven parallel execution are explored with the CharminG GPU-resident runtime system (Chapter 6). Chapter 2 describes how Charm++ provides support for GPU-accelerated applications including PE-GPU mapping and asynchronous completion notification with the Hybrid API (HAPI) module, building the foundation for the following chapters. Chapter 3 discusses how automatic overlap of computation and communication can be achieved using overdecomposition and asynchronous message-driven execution in Charm++, which was published in ACM/IEEE International Workshop on Extreme Scale Programming Models and Middleware (ESPM2) [4]. The asynchronous completion notification mechanism provided by HAPI and prioritizing communication-related operations are the core features necessary to maximize automatic computation-communication overlap on GPU systems. Chapter 4 provides the design and implementation details on integrating GPU-aware communication in message-driven execution to improve programmer productivity and reduce communication costs, which is achieved by extending the UCX machine layer that lies at the bottom of the Charm++ software stack. This allows multiple front-end programming models of the Charm++ ecosystem including Adaptive MPI (AMPI) and Charm4py to seamlessly benefit from GPU-aware communication. This chapter contains materials in a paper published in the International Workshop on Accelerators and Hybrid Emerging Systems (AsHES) [5]. Chapter 5 combines automatic computation-communication overlap enabled by overdecomposition and asynchronous message-driven execution with GPU-aware communication to further push scalability on GPU-accelerated systems. It also explores the performance impact of techniques such as kernel fusion and CUDA Graphs on strong scaling. The work in this chapter was published in the International Workshop on High-Level

Parallel Programming Models and Supportive Environments (HIPS) [6]. Chapter 6 explores GPU-driven parallel execution using an asynchronous message-driven execution model with the CharminG GPU-resident runtime system. It contains details on how task scheduling and communication can be performed from inside the GPU device, building on the concept of persistent thread blocks and GPU-initiated communication using NVSHMEM. The chapter ends with a discussion on the current limitations and future prospects of GPU-driven parallel execution from the experiences of building the CharminG runtime system. Chapter 7 concludes the dissertation with its main contributions and directions for future research.

CHAPTER 2: REALIZING OVERDECOMPOSITION AND EFFICIENT ASYNCHRONOUS MESSAGE-DRIVEN EXECUTION WITH GPUS

Charm++ [3] is one of the first parallel programming systems built on the foundation of the message-driven execution model, which is still widely used by computational science and engineering applications today, including NAMD [7], an award-winning molecular dynamics simulation framework, and ChaNGa [8], a tree-based cosmological simulation code. Perhaps the most appealing aspect of Charm++ to scientists and application developers is its adaptive runtime system, which provides functionalities to automatically and dynamically adapt to the application behavior such as changes in the distribution of work and data during execution. Runtime features such as automatic load balancing and fault tolerance are valuable for increasingly dynamic and large-scale applications of today. Overdecomposition is a technique that underpins the capabilities of the Charm++ runtime system, which allows the programmer to construct their application with as many work units as they would like, as migratable parallel objects, without being concerned about the amount of computing resources (such as the number of CPU cores) in the machine that it will run on. This empowers the runtime system as multiple work units are available per processor, allowing the scheduler resident on each processor to switch between work units for automatic overlap of computation and communication and migrate them between different processors with the goal of improving load balance or supporting fault tolerance.

The benefits from these adaptive features of the Charm++ parallel programming system, enabled by overdecomposition and asynchronous message-driven execution, have been well studied and demonstrated on traditional CPU-based systems. Realizing the same benefits from overdecomposition and asynchronous message-driven execution on modern heterogeneous systems, however, is not straightforward. As we will observe in the following sections and chapters, there are many challenges such as issues with synchronization, communication costs, and fine grained execution and kernel launches. This chapter lays down the foundation in the Charm++ runtime system upon which solutions to these problems are built, designed towards reaping the benefits of overdecomposition and asynchronous message-driven execution on GPU-accelerated systems. First, the Charm++ parallel programming system and its core design principles, overdecomposition and asynchronous message-driven execution, are introduced along with the software stack of the Charm++ ecosystem. Then we explore how Charm++ applications can be adapted to run on today's GPU-accelerated systems, and discuss on how GPU support in Charm++ has evolved to overcome various roadblocks to usability and performance. Finally, the Hybrid API (HAPI) module in the Charm++ runtime is presented, which facilitates the management of GPU tasks and provides support for

asynchronous completion detection that serves as the foundation for improvements discussed in the subsequent chapters.

2.1 CHARM++ PARALLEL PROGRAMMING SYSTEM

Charm++ [3] is a parallel programming system based on the C++ language that employs an object-oriented approach to parallel and distributed-memory execution. The problem domain is decomposed into first-class objects called *chares* using a technique called *overdecomposition*, which allows the application to perform a logical decomposition without being constrained by the number of physically available processors. This is one of the key features of the Charm++ programming model, as it allows applications to naturally express their algorithms in terms of work units executing and communicating with each other in parallel, and applications can be run with any number of such units regardless of the available computing resources. Overdecomposition also empowers the Charm++ runtime system with the ability to dynamically orchestrate the execution of the resulting chares, and when combined with asynchronous message-driven execution, enables adaptive runtime features such as automatic computation-communication overlap, dynamic load balancing, and fault tolerance.

Chare objects interact with one another by invoking *entry methods*, which are C++ methods that can be remotely triggered via messages. The invocation of an entry method transfers the control flow from the source chare to the target chare, although its asynchronous nature allows the source chare to continue its execution to the end of entry method body. This mechanism realizes asynchronous message-driven execution and facilitates automatic computation-communication overlap, with a runtime-level software implementation of the active messages model [9]. As chare objects can also be migrated between processors, usually to improve load balance, the runtime system keeps track of where chares currently reside and directs messages accordingly.

2.1.1 Overdecomposition

With overdecomposition, a Charm++ application can be run with any number of work units (chares) regardless of the underlying hardware resources, and is encouraged to be run with a larger number of chares than the number of processing elements (PEs, generally CPU cores). This provides more control over the parallel execution of chares to the Charm++ runtime system, enabling its adaptive features. Using a two-dimensional particle simulation as an example, Figure 2.1 compares a typical decomposition strategy in MPI, where each

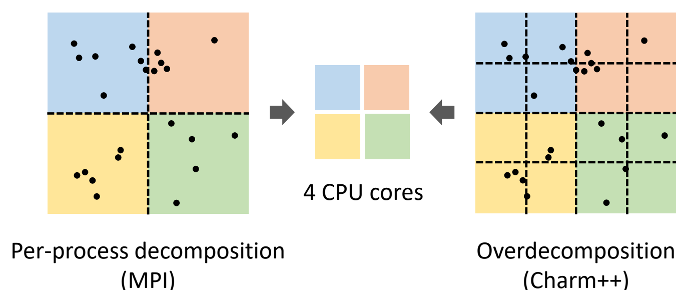


Figure 2.1: Comparison of a typical per-process decomposition in MPI and overdecomposition in Charm++. Particles are scattered across the two-dimensional global grid. MPI decomposition assigns one sub-domain to each process (which runs on a CPU core in pure MPI), whereas overdecomposition assigns four smaller sub-domains to each processing element (generally a single core).

sub-domain is mapped to an MPI process, against overdecomposition in Charm++ where multiple (four) chares are mapped to each PE. Overdecomposition decouples computational work and data of the application from the underlying hardware resources, providing control of task granularity to the programmer. Without overdecomposition, there is only a single chare object executing on each core, providing no computation-communication overlap as each work unit sequentially performs communication after computation in each iteration. On the other hand, with increasing degrees of overdecomposition, each work unit is split into smaller work units (four objects per core with an overdecomposition factor of four), which enables the Charm++ scheduler to switch between work units on the same processor to hide communication of one work unit with computation of another. Asynchronous message-driven execution, as described in Section 2.1.2, allows the parallel objects to progress with minimal synchronization and is thus crucial in achieving such automatic overlap of computation and communication.

Finding the right level of granularity is not trivial, however, as finer granularity typically provides more parallelism, better load balance and more opportunities for computation-communication overlap, but suffers from increased overheads related to scheduling and communication. If the overdecomposition factor becomes too high, effects of finer work granularity such as scheduler overheads and increases in communication costs can start to outweigh the benefits from overlap. The Performance-analysis-based Introspective Control System (PICS) [10] is a feature that automates the process of finding the optimal granularity, but is not yet available for production use; currently Charm++ applications are expected to make granularity decisions based on empirical observations. Such granularity issues become

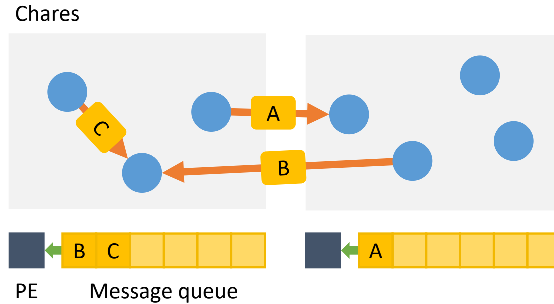


Figure 2.2: Asynchronous message-driven execution in Charm++. Chares exchange messages, which are stored in the message queue and picked up by the scheduler on each PE for execution.

more challenging on GPU-accelerated systems, as GPU utilization by the work units involve additional overheads such as kernel launches and other calls into the GPU runtime. This is discussed in more detail in the context of strong scaling in Section 5.2.4. Nevertheless, overdecomposition provides the underlying Charm++ runtime system with the flexibility needed to implement its adaptive features, especially when combined with asynchronous message-driven execution described below.

2.1.2 Asynchronous Message-Driven Execution

Charm++ employs message-driven execution to drive the parallel program flow, taking inspiration from the active messages model [9]. As illustrated in Figure 2.2, chare objects send messages to one another without expecting a reply, and the arrival of a message triggers the execution of an entry method that performs useful work for the target chare. These messages are sent asynchronously by the runtime system, and are stored in message queues maintained by the PEs. The Charm++ scheduler on each PE continuously checks for incoming messages in the queue and processes them consecutively, by default in FIFO order. Each message contains information and data needed for the execution of the target entry method, which occurs when the scheduler picks up a message from the queue. This mechanism, *asynchronous message-driven execution*, continuously creates workloads for execution on the PEs while allowing the runtime system to progress unhindered by redundant synchronization, which facilitates adaptive features such as automatic computation-communication overlap. The research discussed in this thesis aims to preserve and improve the benefits from the Charm++ design principles, such as overdecomposition and asynchronous message-driven execution, on today's heterogeneous systems.

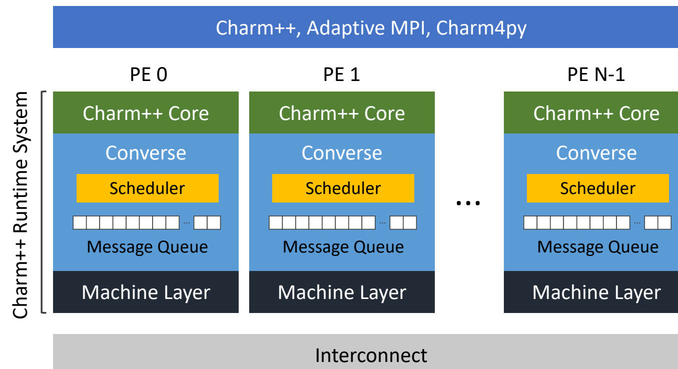


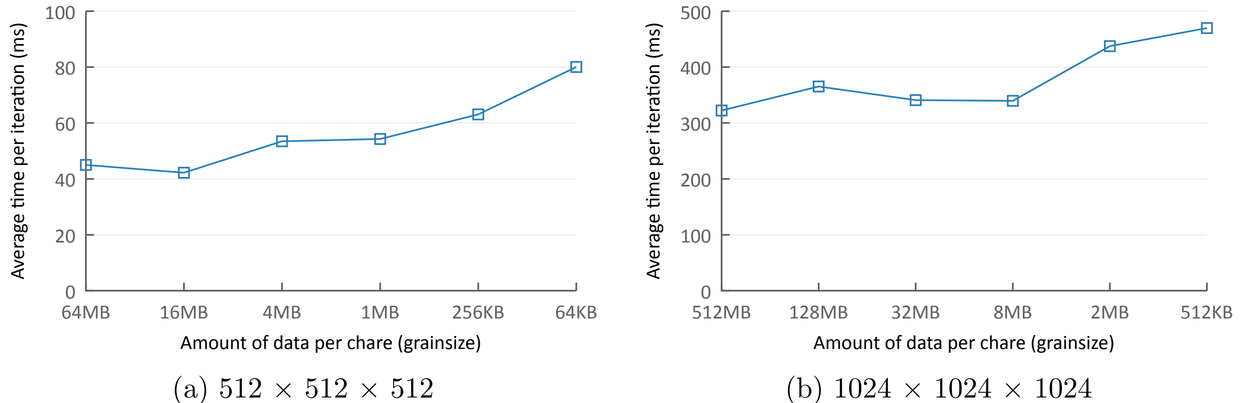
Figure 2.3: Software stack of the Charm++ family of parallel programming models.

2.1.3 The Charm++ Ecosystem

To ease application development for programmers familiar with other programming models such as MPI or Python and make Charm++'s adaptive features widely accessible, the Charm++ ecosystem provides Adaptive MPI (AMPI) [11] and Charm4py [12] (Python-based) as alternative models. All such models including Charm++ share a common runtime system (RTS) that manages message-driven execution and scheduling of chare objects. Figure 2.3 describes the software stack of the Charm++ RTS. The Charm++ Core layer provides the high-level API for creating and utilizing chare objects, which is used to implement ranks (virtualized MPI processes) in AMPI or Pythonized chare objects in Charm4py. It also contains implementations for the core functionalities of the Charm++ programming model used by the application. The Converse layer maintains much of the runtime system capabilities such as the scheduling and messaging mechanisms, and interfaces with the underlying machine layers to perform communication. Machine layers lie at the bottom of the Charm++ RTS stack and enable communication over the wire, providing support for various low-level transports including TCP/IP, Mellanox Infiniband, Cray uGNI, IBM PAMI, and UCX.

2.2 USING GPUS IN CHARM++

We want to ensure that the beneficial characteristics of the parallel execution with the Charm++ programming system are retained when applications are adapted to use GPU devices available on modern HPC systems. We describe two of such properties here in detail: adequate overdecomposition and asynchronous progress.



(a) $512 \times 512 \times 512$ (b) $1024 \times 1024 \times 1024$

Figure 2.4: Performance of Jacobi3D with varying granularity, with a three-dimensional simulation grid of double-precision values.

2.2.1 Adequate Overdecomposition

Many chares, created from overdecomposition, run on each individual PE for concurrent execution. Messages from itself or other chare objects trigger work to be performed on the PE, which can be offloaded to the device on GPU-accelerated systems. As Charm++ relies on overdecomposition to offer adaptive runtime features such as dynamic load balancing and automatic computation-communication overlap, it is important for a Charm++ application to be able to run efficiently with small work granularity. The fine-grained work units resulting from overdecomposition empower the Charm++ runtime system to manage their execution on the available computing resources in a way that mitigates data movement costs and increases hardware utilization.

On CPU-only architectures, Charm++ is able to support very small grain sizes (corresponding to a high degree of overdecomposition), primarily because of its low-overhead task scheduling capability. Figure 2.4 demonstrates the effect of overdecomposition on the performance of a proxy application, Jacobi3D, which performs the Jacobi iterative method in a three-dimensional space, with two different problem sizes. The simulation domain is decomposed into chare objects, each of which performs halo exchanges with up to six neighbors before carrying out the stencil computation. The granularity of each chare is continuously reduced by a factor of four with overdecomposition, up to 1024x overdecomposition (1024 chares per processor). With a smaller domain of $512 \times 512 \times 512$ doubles (Figure 2.4a), performance begins to degrade with overdecomposition factors that result in 1MB or less data per chare. With a small overdecomposition factor such as the one that results in a grain size of 16MB per chare, we can observe that Charm++ provides performance improvement by automatically overlapping computation of one chare with communication of

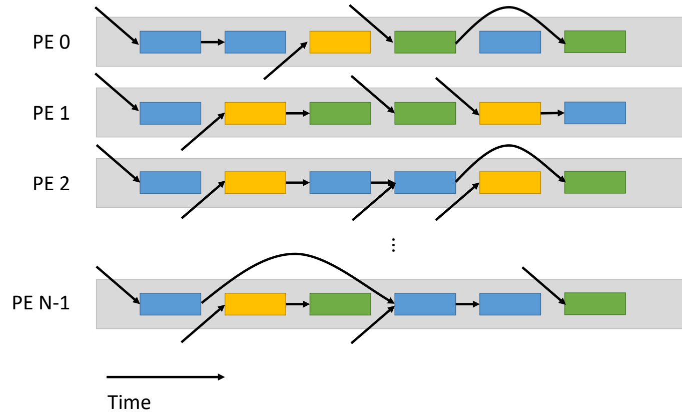


Figure 2.5: Execution timeline of chares created from overdecomposition on multiple PEs. Each chare on a PE has a separate color. Lines depict dependencies derived from message exchanges between chares, including those from external messages (lines incoming from other PEs) and internal (lines from chares on the same PE).

another. With a larger domain of 1024 × 1024 × 1024 doubles (Figure 2.4b), there is relatively less degradation in performance due to the larger granularity. With grain sizes of 2MB or less per chare, we start to observe more reduction in performance. As another example of fine-grained execution, the biomolecular simulation program NAMD, written in Charm++, was able to execute each timestep in less than a millisecond on a BlueGene/Q machine [13], which was further improved to 220 microseconds per step on IBM PERCS machine. This was in spite of the fact that each processor received and scheduled tens of messages in each step.

As such, users of Charm++ are encouraged to use as a small work granularity as possible to empower the runtime system with many work units per processor and maximize the potential for overlap and load balancing. This requires the overhead for runtime components such as task scheduling and communication to be low, which become challenges on GPU systems as will be discussed in later chapters.

2.2.2 Asynchronous Progress

For chares, the work units in Charm++, to efficiently utilize the GPU computing sources, it is crucial to retain the degree of asynchronous progress observed in CPU-driven execution. As illustrated in Figures 2.5 and 2.6, the scheduler switches between different chare objects (each shown in a distinct color) or tasks of the same chare in an asynchronous manner, allowing communication to occur in the background as chares perform their computation.

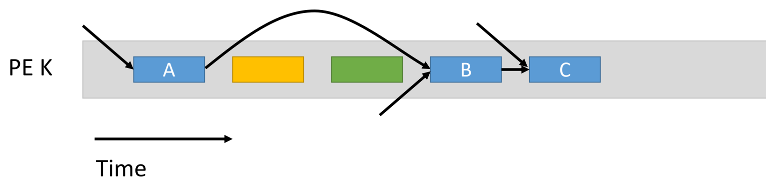


Figure 2.6: Asynchronous progress of chares on a single PE. Each chare has a separate color. Here the blue chare has three different tasks, A, B, and C, which depend on the arrival of messages from itself as a result of a task completion or from other chares.

The computational tasks of chares are only executed once their dependencies are fulfilled, which are in this case message arrivals from remote or local chare objects. For example, for task C of the blue chare to execute, task B must have completed its computation and sent a message to itself and a message should have arrived from a remote chare. The dependency graphs (DAGs) of multiple chare objects are allowed to interleave freely, subject only to internal dependencies and availability of remote data via messages. As the execution chain of chare objects involve computation (offloaded to GPU devices in heterogeneous systems) and messages, the initiation and completion notification/continuation of computational tasks as well as communication must be progressed asynchronously to improve the efficiency of the parallel execution and maximize benefits such as computation-communication overlap.

2.2.3 Specific Goals

The main challenge that we aim to address in this and the following chapters is to allow multiple objects per computing resource (e.g., CPU core, GPU device, compute node) to progress their execution asynchronously based on the availability of remote data as well as their internal task dependencies, while offloading computational work to the GPU devices as needed, and to do this with low overhead so that it is possible to support a reasonably large degree of overdecomposition adequate to facilitate effective computation-communication overlap and load balancing.

2.2.4 Development of GPU Support in Charm++

Charm++ programmers should use a GPU parallel programming model, such as CUDA for NVIDIA GPUs, to build applications for GPU-accelerated systems. Currently Charm++ only supports NVIDIA GPUs and CUDA, but it can be easily extended to other vendors such as AMD and Intel. Although the discussions in this thesis assume the use of NVIDIA GPUs,

Approach	Issues
Synchronous	Blocks host CPU, can cause GPU underutilization, limits computation-communication overlap
User Polling	CPU overhead, difficult for user to determine when and how often to poll
Work Request	Complex API, does not compose with separate CUDA streams per chare, does not allow concurrent kernel execution
HAPI	Static polling frequency, potential thread collision with CUDA Callback-based mechanism

Table 2.1: Approaches to using GPUs in Charm++, in the order of improvement.

most topics can also be applied to other types of GPUs with similar features. As chare objects take the role of work units in Charm++, each chare should be able to individually offload computational work to the GPU, which can be achieved with the use of CUDA Streams [14]. GPU operations such as data transfers between the host and device and kernel launches can be issued to a CUDA stream, and the sequence of operations is guaranteed to execute in the issue order on the target GPU device. Moreover, independent operations in separate streams can be executed concurrently on the same device under the management of the CUDA runtime, as long as there are enough hardware resources available.

Table 2.1 summarizes the approaches to using GPUs in a Charm++ application, from the simplest but least performant method (synchronous) to the most recent and improved mechanism (HAPI). To ensure that the Charm++ scheduler running on the host is not hindered by usage of the GPU and maximize the amount of attainable computation-communication overlap, it is important that the chare objects interact with the GPU as fast and asynchronously as possible. The fact that the scheduler constantly switches between different chares that may offload work to the GPU and multiple chares can concurrently utilize the same GPU should also be taken into consideration. The synchronous method, however, which is the simplest and most naive way to program for GPUs, has significant deficiencies that make it unsuitable for use in applications that aim for performance. If a chare object performs a synchronization call, such as `cudaStreamSynchronize`, it blocks the host CPU until all of the GPU work submitted by the chare are complete. This prevents the Charm++ scheduler from progressing and thus can cause underutilization of the GPU and limit overlap of computation and communication. With overdecomposition, synchronization performed by a chare can also delay the initiation of GPU operations of other chares mapped to the same PE until the synchronization is complete. To avoid such synchronous calls for completion notification and continuation of GPU operations, the user can poll for their status

on their own, using features such as CUDA Events. In addition to the overhead incurred on the CPU, however, it is also extremely difficult for the user to determine when and how often polling should be performed.

To tackle such difficulties with regard to efficiently tracking work offloaded to the GPU, the Work Request API [15, 16] was introduced. A work request is tied to a GPU kernel and contains information about buffers on host and device memory that need to be transferred before and after the execution of the kernel. A Charm++ callback object is also associated with each work request, allowing the program flow to continue once the work request is complete. The Charm++ runtime system maintains three separate streams for each GPU device, one each for host-to-device transfers, kernel launches, and device-to-host transfers. All work requests submitted by the application are tracked by the runtime, and data transfers and kernels are initiated as necessary within the scheduler loop. This design enables overlap between the different types of GPU operations issued by the chare objects. However, aside from the complexity of interface that forces the user to provide verbose details about the usage of the GPU, the Work Request API also disallows concurrent kernel execution as all computational kernels are enqueued into a single stream. It also prevents the CUDA runtime from finding independent operations in different streams to fill in the execution pipeline. To resolve these issues and improve support for GPU-accelerated Charm++ applications that rely on overdecomposition and asynchronous message-driven execution, we develop a more CUDA-stream friendly interface called Hybrid API (HAPI), which is described in detail in the following section.

2.3 HYBRID API (HAPI)

The limitations of previous approaches to GPU usage, especially those pertinent to limiting concurrency and asynchrony that are critical for the efficient execution of Charm++ applications, render them unsuitable for driving runtime-level support on modern GPUs and heterogeneous systems. Support for GPUs from the runtime system must evolve to utilize the latest hardware and software features, towards the goal of providing easy-to-use functionalities for improving performance to the end user. To seamlessly support the use of CUDA Streams, which has become the foundation upon which many modern CUDA features have been built to facilitate asynchronous and concurrent execution, we develop **Hybrid API (HAPI)** [17], a software module for GPU support in the Charm++ runtime system. HAPI encapsulates the core logic and data structures for implementing GPU support capabilities in the context of overdecomposition and asynchronous message-driven execution, including establishing mappings between Charm++ PEs and GPUs, and mechanisms for asynchronous

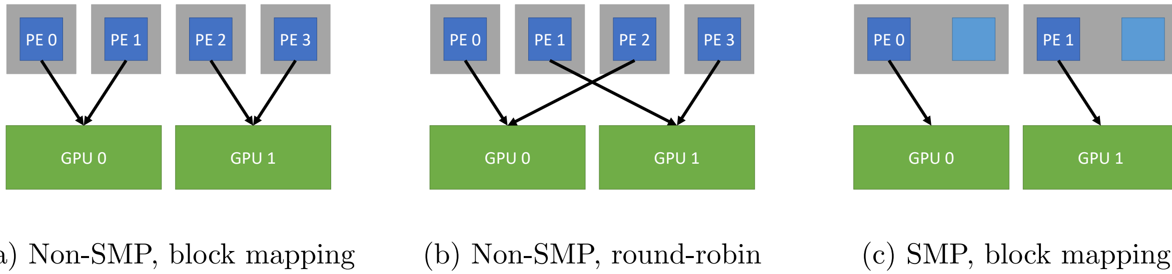
completion detection of GPU operations.

2.3.1 Mapping of Charm++ PEs to GPU Devices

There are often many more CPU cores available than GPU devices on a compute node of a HPC system, which leads to multiple Charm++ PEs having to utilize the same GPU device. To evenly distribute the GPU workloads, a similar number of PEs must be mapped to each GPU. Instead of leaving the responsibility of assigning PEs to GPUs to the user, HAPI provides simple command line options to achieve efficient PE-GPU mappings.

Before discussing how HAPI performs such mappings, it is worth introducing here the SMP (Symmetric Multiprocessing) and non-SMP modes of Charm++ in a CPU-only context. Non-SMP is the simpler mode, where a single Charm++ PE is contained in a OS process and memory address space. Because single-threaded execution is enforced within a process and each PE sequentially executes one chare after another, the user does not have to worry about multi-threading issues. As such, it is much easier to debug Charm++ programs in non-SMP mode, making it a good starting point for application development. In SMP mode, on the other hand, multiple PEs may reside in each process, which are implemented as worker threads, as well as a separate communication thread that handles incoming messages that target PEs (worker threads) of the same process. The user can choose the number of PEs per SMP process, which is another tunable parameter that affects performance, especially on machines with multiple CPU sockets and NUMA domains. Because PEs in a SMP process share the same address space, having multiple PEs per process often helps reduce communication overheads. In GPU-accelerated environments, PEs must be properly mapped to the GPU devices, to ensure that the scheduled chares utilize the correct devices. These mappings from PEs to GPUs are currently assumed to be static as is the norm for GPU-accelerated applications, but they can be made dynamic if PEs need to utilize multiple devices concurrently.

HAPI currently supports block and round-robin mapping of PEs to GPU devices, with block mapping being the default. For instance, a block mapping with four PEs and two GPU devices maps PEs 0 and 1 to GPU 0, and PEs 2 and 3 to GPU 1, whereas a round-robin mapping will map PEs 0 and 2 to GPU 0, and PEs 1 and 3 to GPU 1. Example mappings are shown in Figure 2.7. Mappings by HAPI can also be disabled if the user wants to utilize a more sophisticated mapping strategy. HAPI supports platforms where all GPUs on a physical node are visible to all processes, as well as environments where the job launcher limits the visible GPU devices to each process (e.g., `jsrun` on OLCF Summit). A potential improvement is performing the mapping in a topology-aware manner, using software such



(a) Non-SMP, block mapping (b) Non-SMP, round-robin (c) SMP, block mapping

Figure 2.7: Various PE-GPU mapping strategies, with four PEs and two GPU devices on a single node. Grey boxes represent processes, blue boxes are PEs (worker threads), sky-blue boxes are communication threads, and green boxes are GPU devices.

as NVTAGS [18] so that the runtime system can automatically map PEs to the GPUs that are physically closer and/or have higher bandwidth connections.

To manage the GPU devices utilized by Charm++, HAPI maintains a GPU Manager data structure per process, and a Device Manager per GPU device utilized by each process. In non-SMP mode of Charm++, each GPU Manager will only have one Device Manager since there is a single PE in the process associated with the GPU Manager which will be tied to one GPU device. In SMP mode, if there are more PEs in a process than the number of GPU devices, HAPI will designate one of the Charm++ worker threads as the GPU handler thread for each device to be responsible for its management.

2.3.2 Asynchronous Completion Detection

As introduced in Section 2.2, streams underpin the asynchronous execution capabilities of the CUDA framework, and can be utilized by chare objects to manage GPU workloads in Charm++ applications. However, there is a discrepancy between what CUDA provides and the functionalities required by asynchronous message-driven and overdecomposition-based parallel programming systems such as Charm++. Synchronous mechanisms in CUDA for detecting completion of GPU operations enqueued into a stream, e.g., `cudaStreamSynchronize`, are semantically simple and easily ensure program correctness but hinder asynchrony and can degrade performance as other useful work are prevented from being performed on the host CPU or offloaded to the GPU.

To ensure that the host, and consequently, the overall parallel execution are not impeded, the progress of GPU workloads must be tracked in an asynchronous and timely manner. With message-driven execution, the importance of such asynchronous completion detection mechanisms is even greater, as the scheduler running on the CPU needs to continue pro-

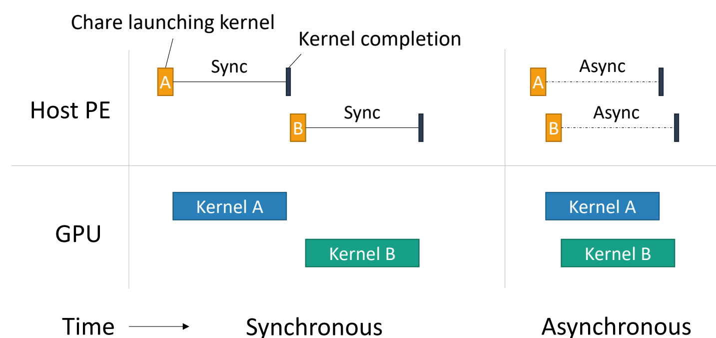


Figure 2.8: Execution timelines demonstrating the benefits of asynchronous completion detection. Two chares on a PE are offloading kernels to the same GPU, with the assumption that the two kernels are small enough to execute concurrently on the same GPU.

```
__host__ cudaError_t cudaStreamAddCallback(cudaStream_t stream, cudaStreamCallback_t
    callback, void* userData, unsigned int flags)
```

Figure 2.9: CUDA Callback API.

cessing messages and executing CPU workloads while GPUs are being utilized, and not be held back by synchronizations between the host and device. Furthermore, with overdecomposition, where multiple work units (chares in Charm++) can concurrently utilize the same device, host-device synchronizations can effectively serialize the execution of GPU operations and significantly degrade performance. The effect of synchronous and asynchronous completion detection mechanisms on performance are illustrated in Figure 2.8. With two chares concurrently utilizing the same GPU device, each launching one kernel, synchronous completion detection forces one chare to wait until the other chare's kernel is complete, whereas asynchronous completion detection allows the Charm++ scheduler to switch to the other chare as soon as a kernel is offloaded, improving GPU utilization. Synchronizations also limit opportunities for computation-communication overlap as the schedulers are blocked from forward progress, as discussed in more detail in Section 3.2.

CUDA provides a couple of mechanisms to support asynchronous completion detection, on top of the built-in asynchronous initiation feature[1], in the form of CUDA Callback and CUDA Events. With CUDA Callback [19], the application can enqueue a callback function of type `cudaStreamCallback_t` in a CUDA stream, which will be executed on the host after all previous operations in the stream have completed. Figure 2.9 shows the CUDA Callback API, which also has an updated version, `cudaLaunchHostFunc`, with the same

[1]Calls to most CUDA Stream APIs return immediately before the corresponding operations are complete.

```
__host__ __device__ cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream)
__host__ cudaError_t cudaEventQuery(cudaEvent_t event)
```

Figure 2.10: CUDA Events API.

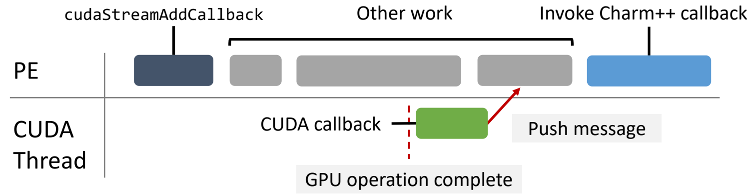
```
// CkCallback is a Charm ++ callback object
void hapiAddCallback(cudaStream_t stream, CkCallback callback)
```

Figure 2.11: HAPI asynchronous completion detection.

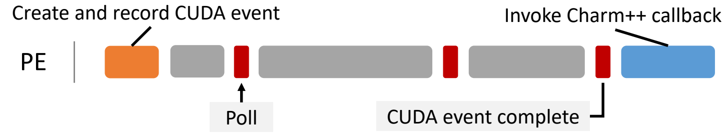
functionality but with the added support of the CUDA Graphs feature. A pointer to some user data can also be provided so that it is available when the callback function is executed. A major limitation of CUDA Callback when it is to be used with a runtime system such as Charm++, however, is that the host-side CUDA callback function does not have access to any of the runtime constructs as it is executed on a separate thread generated by the CUDA runtime. To continue the execution flow of the Charm++ application, the callback function must have access to the chare object that initiated the original GPU operation and functionalities of the Charm++ runtime system. Moreover, the CUDA runtime does not expose any mechanisms to control the placement of the CUDA-generated thread that tracks the status of the specified CUDA stream, such as binding it to a CPU core, which is problematic for runtime systems such as Charm++ that maintain their own set of threads for parallel execution. If the CUDA runtime thread is not given a separate core to run on, it may cause performance degradation as it collides with another thread of the Charm++ runtime.

An alternative mechanism provided by CUDA is CUDA Events [20], which allows the user to enqueue a CUDA event into a stream and query its status. The recording of an event into a stream and querying its status can be done as illustrated in Figure 2.10. It is now up to the user to determine when and how often to check the status of the events, which can be challenging for many applications. One potential method is to spawn a user-level thread in the application to continuously poll for the statuses of the events, which would require the thread to fully utilize a separate CPU core or share a core with the main worker threads and lose performance.

To overcome the limitations of the native CUDA mechanisms, we introduce an API that enables efficient asynchronous completion detection in HAPI. This API utilizes a Charm++ callback object for continuation, which encapsulates information about the entry method and chare object to be executed once completion is detected. Figure 2.11 demonstrates the function signature of the asynchronous completion detection API. The user calls `hapiAddCallback` after the necessary GPU operations are enqueued into a CUDA stream,



(a) Callback-based mechanism.



(b) Polling-based mechanism.

Figure 2.12: Mechanisms for asynchronous completion detection in HAPI.

to schedule a Charm++ callback (most often a chare entry method) on their completion, which serves as a continuation of the parallel execution. Internally in the runtime system, triggering a Charm++ callback tied to a HAPI call leads to enqueueing an entry to the scheduler of the PE that houses the chare that made the call. As for the underlying implementation, there are two compile-time configurable mechanisms built on top of CUDA Callback and CUDA Events, respectively.

The first mechanism utilizes the CUDA callback feature to execute a codelet registered by the Charm++ runtime system once all GPU operations previously enqueued to the specified stream are complete. As the registered codelet of type `cudaStreamCallback_t` has no association with the Charm++ runtime, it creates and sends a Converse-level message (see Figure 2.3) containing the Charm++ callback object to the PE which originally executed `hapiAddCallback`. Once this message is picked up by the resident scheduler, the entry method designated in the Charm++ callback is run, continuing the execution. Figure 2.12a illustrates this process. In essence, this mechanism returns control from the CUDA-generated thread back to the Charm++ scheduler to enable continuation. The overall process is entirely asynchronous as the Charm++ runtime progresses freely until it is notified of completions via messages sent by the host-side codelet.

The second mechanism, which is based on polling, makes use of CUDA events to track the progress of GPU operations. We take advantage of the scheduler-driven nature of Charm++ introduced in Section 2.1, where message arrivals are routinely checked, to probe the status of CUDA events in the scheduler loop and determine the completion of GPU operations. When the user calls `hapiAddCallback`, a CUDA event is created and then recorded. A HAPI event data structure, which encapsulates the CUDA event and information about the

user-specified Charm++ callback, is subsequently created. The HAPI event is then pushed to a FIFO queue maintained by each PE, which is checked every iteration of the scheduler loop. The frequency of the polling can be configured to be a number of scheduler loops or a given absolute time value. Once the scheduler detects HAPI events that are complete, the associated Charm++ callbacks are invoked before processing the next message in the message queue. Figure 2.12b outlines this mechanism.

As mentioned earlier, performing asynchronous detection with the callback-based mechanism of HAPI can degrade performance due to the CUDA-generated thread. It performs almost identically as the polling-based mechanism for applications where the CPU cores are not too busy (e.g., with large GPU task granularity), but suffers from the overheads observed on the CPU core that houses both the CUDA runtime thread and a Charm++ scheduler thread with more fine-grained applications. With the Jacobi3D proxy application, we observe about a slowdown of about 500 us per iteration with the callback-based HAPI when compared to the polling-based mechanism. We have experimented with pinning the CUDA runtime thread that tracks the GPU operations to a separate core using `pthread_setaffinity`, but this did not succeed according to the thread affinity outputs from the NVIDIA Visual Profiler tool. As such, the polling-based asynchronous completion detection mechanism is configured to be the default mode of HAPI when building Charm++ with GPU support.

It is worth noting here that asynchronous completion detection provided by HAPI is one of the key features that enable automatic and effective overlap of computation and communication on GPU-accelerated systems, as discussed in Section 3.2.

2.4 CONCLUDING REMARKS

The impedance mismatch between the highly asynchronous and overdecomposition-based execution of Charm++ and the existing capabilities of the CUDA framework presents a challenge for productive and efficient exploitation of GPU computing resources. We have developed new mechanisms to bridge this gap, allowing the Charm++ programmer to utilize familiar mechanisms such as callbacks and continuations to drive the parallel execution without losing performance. Hybrid API (HAPI), as a software module in the Charm++ runtime system, provides the necessary functionalities for enabling efficient asynchronous message-driven execution on modern GPU-accelerated systems. Its features include strategies for mapping Charm++ PEs to GPU devices and mechanisms for the asynchronous detection of GPU operations with a stream-friendly interface.

It should be noted that hardware support from the GPU could further improve the per-

formance of asynchronous completion detection. For instance, a hardware-controlled FIFO queue of events that can be directly polled by the Charm++ runtime system would enable efficient completion notification without relying on a CUDA-runtime-generated thread (with CUDA Callbacks) or CPU polling (with CUDA Events), preventing unnecessary CPU usage and collision with scheduler threads of parallel programming systems such as Charm++. With the addition of the ability to interact with pthreads and user-level threads (ULTs) specified by the user, it would not be too difficult to implement a highly efficient mechanism for asynchronous completion detection in an asynchronous task-based runtime system.

CHAPTER 3: EXPLOITING AUTOMATIC COMPUTATION-COMMUNICATION OVERLAP ON GPU SYSTEMS

A major hurdle to performance on modern HPC systems is the increasing cost of data movement, as the memory hierarchy becomes more complex with the addition of GPU accelerators, non-uniform memory access (NUMA), and non-volatile memory, and as network performance improves at a relatively slower pace compared to computational power. Hiding communication latency by overlapping computation and communication is a popular and effective technique, but many existing methods require the programmer to manually identify the regions of potential overlap. This not only hampers productivity and code maintainability, but also limits the performance gains from overlap to the programmer's implementation. As introduced in Section 2.1, one of the main advantages of the Charm++ parallel programming model is *automatic* computation-communication overlap, enabled by overdecomposition and asynchronous message-driven execution. With the application written in terms of chare objects (units of work and/or data) and message exchanges between them, the runtime system is able to overlap asynchronous communication of one chare with computational work of another (or itself). Although automatic computation-communication overlap has been extensively researched and evaluated on traditional CPU-based systems, there has been less work on how it can be achieved at the level of distributed-memory execution on GPU-accelerated systems, and very few focus on the runtime features and application design considerations needed to exploit maximal overlap. As such, in this chapter, we discuss techniques that facilitate automatic overlap of computation and communication on modern heterogeneous systems, at the level of both the application and underlying parallel programming model. Note that communication of GPU data between work units is yet staged through host memory, with GPU-aware communication introduced in Chapter 4 and its integration with computation-communication overlap discussed in Chapter 5.

A useful feature that empowers the runtime system with more work units and promotes runtime-driven overlap is overdecomposition, which allows the application programmer to partition the problem domain into work and data units without being constrained by the number of available processors. It decouples the application's computational work from the hardware resources and provides more control of the work units to the underlying runtime system. This also benefits programmer productivity as the units of decomposition become first-class citizens of the program and can be addressed with logical names. With regards to performance, an important benefit of overdecomposition is automatic overlap of computation and communication. With multiple work units assigned to each processor, a work unit can perform computation while another mapped to the same processor is waiting for com-

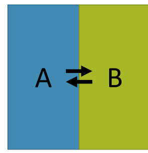
munication. Computation and communication of the same work unit can also be overlapped, provided that they are independent of one another. In fact, overdecomposition is one of the core design principles of GPUs applied at the hardware level; memory access latencies of a thread group running on a symmetric multiprocessor (SM) are hidden by switching to another group that can perform independent computation. It should be noted that automatic overlap requires a high degree of asynchrony for distributed-memory heterogeneous execution, where both the initiation and completion/continuation do not block the host CPU, to maximize concurrency and opportunities for overlap between independent operations.

With distinct phases of computation and communication phases, as in bulk-synchronous models such as MPI, time spent in communication directly translates into idle time that affects the overall execution. This issue can be mitigated by employing non-blocking communication primitives and manually performing independent computation during the communication phase, but resources can still remain idle if blocking synchronization calls such as `MPI_Waitall` are invoked before the arrival of messages [21]. Overdecomposition with asynchronous message-driven execution, supported by the Charm++ parallel programming system, provides a natural remedy to this problem. While a chare object is waiting for communication on a PE, another chare on the same PE can be scheduled to perform computation, effectively hiding communication latency. Furthermore, the injection of messages into the network is spread throughout the execution, in contrast to traditional MPI applications where communication is often clustered at certain points of the execution timeline.

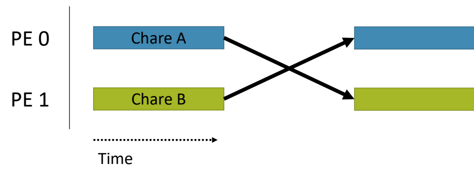
3.1 EXPLOITING OVERLAP ON TRADITIONAL CPU-BASED SYSTEMS

To describe how automatic computation-communication can be achieved on traditional CPU-based systems, let us take a simple two-dimensional computational fluid dynamics (CFD) simulation using a structured mesh as an example. The discretization method is not important for the purposes of this illustration, but we can assume it to be the finite volume method. The simulation domain is split into equal-sized, overlapping sub-domains for parallel execution, with mesh point values exchanged between the sub-domains along the boundary. Physical boundary conditions are used along the domain boundary, which constrains communication to only occur inside the domain. The simulation is run iteratively, with computation using the governing equations followed by communication between the neighboring sub-domains in each iteration.

To demonstrate how overdecomposition can provide automatic computation-communication overlap, let us first look at the case without overdecomposition, where the simulation is run on two PEs (CPU cores). The domain is split in half into two chares, A and B, as can



(a) Decomposition

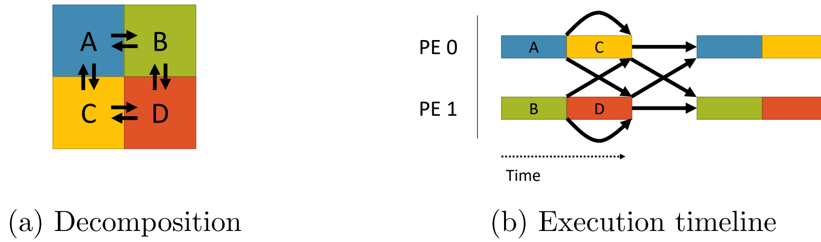


(b) Execution timeline

Figure 3.1: Domain decomposition and execution timeline without overdecomposition. Two chares are running on two PEs, each mapped to a different PE. Rectangles represent computation and black arrows depict messages (communication).

be seen in Figure 3.1a. Since there is only one chare object per PE, the scheduler would not have any other chare to run after performing computation and sending a message with sub-domain boundary data, causing it to idle. This is depicted in Figure 3.1b, where there is no overlap and the overall execution time is simply the sum of computation time and communication time. On the other hand, with overdecomposition, the domain decomposition could look something like Figure 3.2a, where four chares (instead of two) are created to divide the domain into four equal-sized sub-domains and also run on two PEs. Although each chare now needs to perform halo exchanges with two chares, the size of each exchange is halved. More importantly, there is potential for automatic overlap, as the scheduler can perform the computation of another chare while one is done with its computation and waiting for communication to finish, reducing idle time and increasing the utilization of compute resources. For example, chare A sends messages containing its halo data to chares B and C, and the scheduler immediately switches to chare C, which is the other chare mapped to the same PE. Because communication is progressed asynchronously, the computation of chare C can overlap with these messages. Before moving on to the next iteration on PE 0, chare A needs to receive the halo data from chare C, and chare C from chare D, which causes idle time, but it is half of what it was without overdecomposition as one half of the communication overlapped with computation. This is illustrated in Figure 3.2b, where the idle time between iterations caused by waiting for communication is reduced in half due to automatic overlap.

From the discussion so far, it may seem that higher degrees of overdecomposition always result in better performance from more potential of computation-communication overlap. It is unfortunately not the case, as there are costs associated with the finer granularity that comes with overdecomposition, including increases in the number of communication calls and scheduling overheads. As such, there is often a best performing degree of overdecomposition for each application that provides a good balance between these tradeoffs, depending



(a) Decomposition (b) Execution timeline

Figure 3.2: Domain decomposition and execution timeline with 2x overdecomposition. Four chares are running on two PEs, with two chares mapped to each PE. Rectangles represent computation and black arrows depict messages (communication).

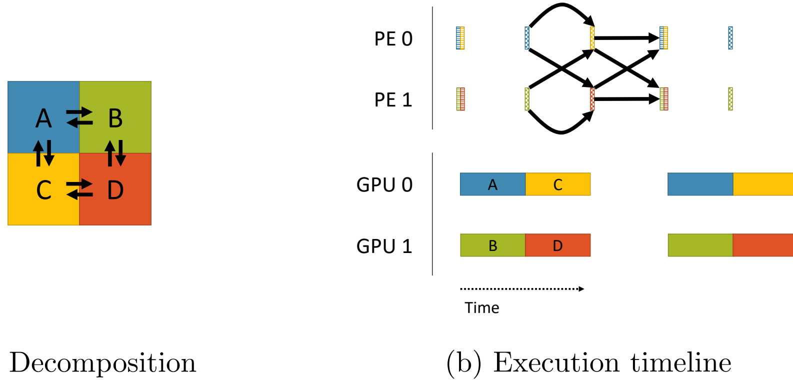
on the work granularity and other characteristics of the application. Nevertheless, once an application is built using overdecomposition, the Charm++ runtime system can automatically exploit overlap of computation and communication by actively seeking independent work made available with asynchronous message-driven execution.

3.2 MAXIMIZING OVERLAP ON GPU-ACCELERATED SYSTEMS

On GPU-accelerated systems, chares can offload work to the devices using CUDA Streams as described in Section 2.2. Computational kernels of these chares should be able to execute concurrently with data transfers between the host and device as well as message exchanges between them to exploit computation-communication overlap. To maximize concurrency and opportunities for overlap, both the initiation and completion of stream-based GPU operations (including computational kernels and data transfers) must be asynchronous, which is enabled in Charm++ by the asynchronous completion detection mechanism of HAPI, discussed in Section 2.3.2. Another critical factor in exploiting overlap on GPUs is executing communication-related operations as early as possible, by placing them on higher priority. As illustrated in Figure 3.3 and discussed in the following sections, these approaches allow us to extend automatic computation-communication overlap from traditional CPU-based systems to modern GPU-accelerated platforms, demonstrating substantial performance improvements enabled by overdecomposition and asynchronous message-driven execution of Charm++.

3.2.1 Prioritizing Communication in the Application

A straightforward method of integrating GPUs into an application is associating a CUDA stream with each work unit. GPU operations including kernel launches and data transfers

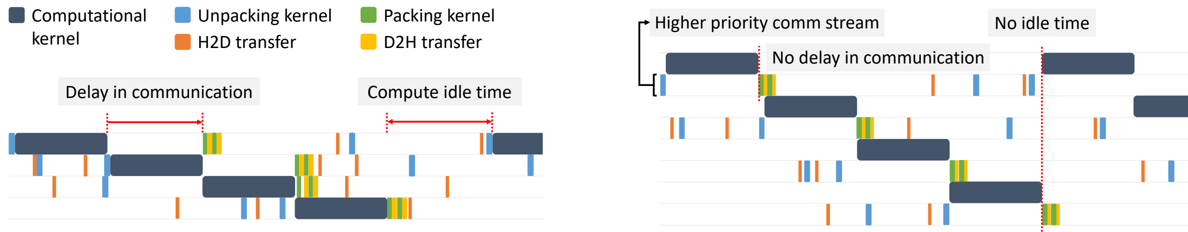


(a) Decomposition (b) Execution timeline

Figure 3.3: Decomposition and execution timeline with 2x overdecomposition, with chares offloading computational kernels (filled rectangles) to the GPU. The domain is split into four chares (sub-domains), with two chares mapped to each PE. PE 0 utilizes GPU 0 and PE 1 utilizes GPU 1. Rectangles with a horizontal line pattern represent kernel launches, whereas rectangles with a checkered pattern represent (asynchronous) kernel completion and subsequent initiation of host-side communication depicted by black arrows. Data transfers between the host and device are omitted for brevity.

enqueued in the same stream are guaranteed to execute in order. For MPI applications where a single process is used to manage each GPU, it is often sufficient to use the default stream per process. In a Charm++ application, each chare can maintain a separate stream to enforce dependencies between GPU operations performed by the same chare. While assigning a single CUDA stream to each chare ensures that dependencies are observed between operations performed within the same work unit, it can diminish opportunities for computation-communication overlap, especially with overdecomposition where multiple chares can concurrently utilize the same GPU device. One substantial factor is delay in communication-related operations such as host-device data transfers and associated packing/unpacking kernels, which is caused by computational kernels offloaded by other chares taking up the GPU resources. This ultimately delays communication on the wire, reducing the amount of attainable overlap and degrading the overall performance.

Figure 3.4 depicts execution timelines of two different implementations of Jacobi2D, a simple Charm++ program that performs the Jacobi iteration in a two-dimensional grid. The global grid is overdecomposed into four chares running on a single PE and one GPU device. Each chare is responsible for a quadrant of the grid and performs halo exchanges with its two neighboring chares after the Jacobi update, which is repeated for a given number of iterations. The communication-related GPU operations for each neighbor comprise of a packing kernel and a device-to-host (D2H) transfer enqueued after the main Jacobi update kernel, as well as a host-to-device (H2D) transfer and a unpacking kernel that are enqueued



(a) Single CUDA stream per chare. Communication is delayed by a computational kernel enqueued from another chare, causing idle time between iterations.

(b) Separate compute/communication CUDA streams per chare, with the communication stream given higher priority. Iterations continue without idle times in between.

Figure 3.4: Execution timelines of Jacobi2D with four chares mapped to a single GPU.

once a halo message is received. Figure 3.4a shows the execution of the implementation that uses a single CUDA stream per chare to enqueue all computation kernels (i.e., Jacobi update) and communication-related operations. Although there is some overlap of computation and communication from four work units (chares) asynchronously utilizing the GPU, still a large amount of idle time is observed between iterations due to the delay in the execution of the communication-related operations (packing kernels and D2H transfers). This is caused by computational kernels offloaded by other chare objects being executed before the communication-related operations, as can be seen in Figure 3.4a.

This issue can be resolved by utilizing separate streams for compute kernels and communication operations in each chare, as in Figure 3.4b, and assigning a higher priority to the communication stream using `cudaStreamCreateWithPriority`. Because there are now more than a single stream within the same work unit, CUDA events and `cudaStreamWaitEvent` must be used to enforce dependencies between streams. An example is enforcing the Jacobi update kernel enqueued in the compute stream to be executed only after the completion of the unpacking kernels enqueued in the communication stream. It should be noted that creating only two separate streams may not be enough to obtain good computation-communication overlap, an example being the MiniMD benchmark described in Section 3.3.2.

3.2.2 Support for Asynchronous Progress in the Runtime

To maximize the opportunities for overlap on heterogeneous systems, it is important to ensure that the host application and runtime system interact with the GPU devices asynchronously, in addition to prioritizing communication-related operations. This allows the host to initiate and process other necessary work while the GPU is busy, avoiding redundant barriers to execution and increasing the level of parallelism. As discussed in Section 2.3.2,

<table>
<tr><th>Platform</th><th>Nodes</th><th>CPU</th><th>GPU</th><th>GPUs/node</th></tr>
<tr><td>Summit</td><td>4,608</td><td rowspan="2">IBM Power9</td><td rowspan="2">NVIDIA Tesla V100</td><td>6</td></tr>
<tr><td>Lassen</td><td>792</td><td>4</td></tr>
</table>

Table 3.1: Summary of the experimental platforms

CUDA streams inherently support asynchronous initiation or offload of GPU operations by the host, but the native CUDA mechanisms for asynchronous completion detection are not sufficient by themselves for realizing asynchronous message-driven execution. HAPI builds on the native CUDA mechanisms to provide an efficient method of asynchronously detecting the completion of the user's GPU operations, which frees the Charm++ scheduler to perform other tasks and increase opportunities for exploiting computation-communication overlap. Experiments in the following sections utilize the polling-based asynchronous completion detection mechanism in HAPI, to achieve overlap between computation and communication of overdecomposed chare objects.

3.3 EXPERIMENTAL SETUP

Next, we describe the set of platforms and proxy applications used to evaluate the performance impact of our approach in achieving automatic computation-communication overlap.

3.3.1 Platforms Used for Experiments

Two leadership-class GPU-accelerated supercomputers are used for performance evaluations: Summit at Oak Ridge National Laboratory and Lassen at Lawrence Livermore National Laboratory (a non-classified Sierra-like system). A brief summary of the hardware and software of these systems are provided in Table 3.1. IBM Spectrum MPI and CUDA 10.1 are used on both platforms. Note that Lassen has a limit of 256 nodes (1,024 GPUs) for regular jobs. The main architectural differences are the number of GPUs per node, intra-node GPU interconnect, and inter-node network topology. A more detailed comparison of these systems can be found in [22]. Because the two systems employ the same type of GPU, we expect the difference in performance to be derived largely from communication. The same number of PEs as the total number of GPUs are used in the execution of Charm++ programs, with each PE assigned to one CPU core. On a single node of Summit, for example, six PEs (CPU cores) are mapped to six GPUs, with one PE per GPU.

3.3.2 Benchmarks

Jacobi3D. Jacobi3D is a simple Charm++ proxy application that performs the Jacobi iterative method on the GPU in a three-dimensional domain. The global grid is decomposed into cuboids, each contained within a chare. For the purposes of this work, Jacobi3D is configured to run a fixed number of iterations without convergence checks. Each Jacobi iteration consists of the following steps:

1. Perform Jacobi update on GPU (Equation 3.1)
2. For each halo to be sent to a neighbor,
   (a) Invoke packing kernel to move halo data to contiguous buffer if necessary
   (b) Device-to-host (D2H) transfer of halo buffer
3. Non-blocking exchange of halo data with neighbors
4. On receiving a message from a neighbor,
   (a) Host-to-device (H2D) transfer of halo buffer
   (b) Invoke unpacking kernel to move halo data into non-contiguous memory if necessary

The Jacobi update is a 3D stencil computation of the following:

$$A_{i,j,k} = \frac{1}{7} \times (A_{i,j,k} + A_{i-1,j,k} + A_{i+1,j,k} + A_{i,j-1,k} + A_{i,j+1,k} + A_{i,j,k-1} + A_{i,j,k+1}) \quad (3.1)$$

where $A_{i,j,k}$ is the block at position (i, j, k) of the global grid.

Each chare maintains separate compute and higher-priority communication CUDA streams as discussed in Section 3.2.1. Packing/unpacking kernels and transfers between host and device are enqueued in the communication stream, whereas the Jacobi update kernel is offloaded in the compute stream.

MiniMD. MiniMD [23] is a proxy application for molecular dynamics simulation of a Lennard-Jones or embedded atom model (EAM) system, designed to be representative of the performance of the widely used LAMMPS [24] package. In this work, we employ a Lennard-Jones system without re-neighboring and Newton's third law for ghost atoms. MPI and Kokkos [25] performance portability framework are used for execution on distributed

GPU systems, where CUDA-aware MPI handles inter-GPU communication and Kokkos manages GPU execution through its CUDA backend.

To enable overdecomposition, we convert an MPI process to a chare array element and port the MPI communication routines to Charm++. Kokkos is retained as the performance portability layer for GPU execution, but several significant modifications are made to enable asynchronous progress. These include modifying Kokkos deep copies and parallel loops to use CUDA execution instances [26] associated with CUDA streams, and forced asynchronous kernel launches [27] to disable unwanted synchronization behaviors. It is important to note that the near-neighbor communication is modified from a set of `MPI_Sendrecv` calls to non-blocking communication routines in Charm++, in order to maximize overlap of computation and communication between different chares mapped to the same GPU. This trades off memory usage (as a separate set of send and receive buffers is needed for each neighbor exchange) for improvement in communication performance and more potential for computation-communication overlap.

Because of the current lack of support for direct GPU-GPU transfers in Charm++, CUDA-aware MPI calls are converted to explicit host-device transfers and host-to-host messages. However, the ability to hide the communication latency with overdecomposition allows the Charm++ version to outperform even the CUDA-aware MPI version, as discussed in Section 3.4.

The following main iteration loop is executed by each chare in the Charm++-Kokkos version of MiniMD:

1. Initial integration
2. Exchange of atom information
   (a) Packing kernels
   (b) Device-to-host (D2H) transfers
   (c) Neighbor exchanges via host-to-host messages
   (d) Host-to-device (H2D) transfers
   (e) Unpacking kernels
3. Lennard-Jones force calculation
4. Final integration

Our first attempt at integrating CUDA streams in MiniMD involved using two streams per chare as in Jacobi3D, but it did not yield satisfactory performance due to the lack of

computation-communication overlap. The issue was that the communication-related GPU operations (Steps 2a, 2b, 2d and 2e) were not prioritized as expected. Many of these operations were held back by force calculation kernels (Step 3) from other chares. Nevertheless, this is not an erroneous behavior, as CUDA stream priorities are merely hints to the CUDA scheduler and does not guarantee preemption of lower priority work in favor of higher priority work. In such situations, we need a more sophisticated design of CUDA streams interlaced with CUDA events to enforce inter-stream dependencies.

Our design utilizes a total of five streams per *GPU* (instead of two streams per *chare*): one stream each for computational kernels (Steps 1, 3 and 4), packing kernels, D2H transfers, H2D transfers, and unpacking kernels. All streams aside from the compute stream are given higher priority. This allows communication-related operations to be properly prioritized and also overlap packing/unpacking kernels with D2H/H2D transfers. A potential drawback to this design is that compute kernels enqueued from different chares cannot execute concurrently, since there is only one compute stream per GPU. This can be fixed with a more complicated design with multiple compute streams, but it is left as future work. CUDA events are used to asynchronously enforce the following dependencies between streams: compute → packing, packing → D2H transfer, H2D transfer → unpacking, and unpacking → compute. With this design, we are able to effectively overlap computational kernels with host-device data transfers and host-to-host communication. In both Jacobi3D and MiniMD, asynchronous progress of GPU operations is supported by the Charm++ runtime system through HAPI.

3.4 PERFORMANCE EVALUATION

We evaluate the performance of our approach using two proxy applications, Jacobi3D and MiniMD, on two different GPU-accelerated platforms, Summit and Lassen. Performance is averaged across nine different measurements: three jobs each performing three runs of the same configuration.

3.4.1 Jacobi3D

Single-node. We first evaluate the performance of Jacobi3D on a single node of Summit, with a global grid of 1,536 × 1,536 × 1,536. Figure 3.5 compares different mechanisms used to ensure halo data have been moved to host memory before performing neighbor exchanges: calling `cudaStreamSynchronize` on the communication stream which is the simplest approach, and using the Charm++ runtime support (callback-based and polling-based

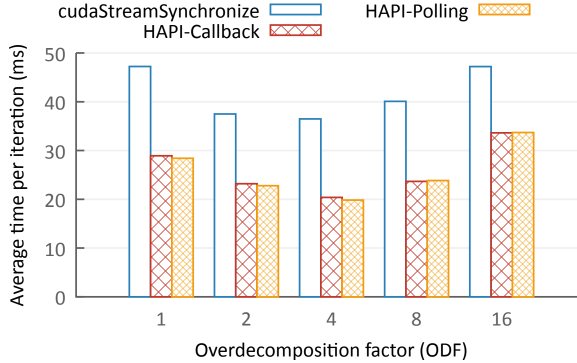


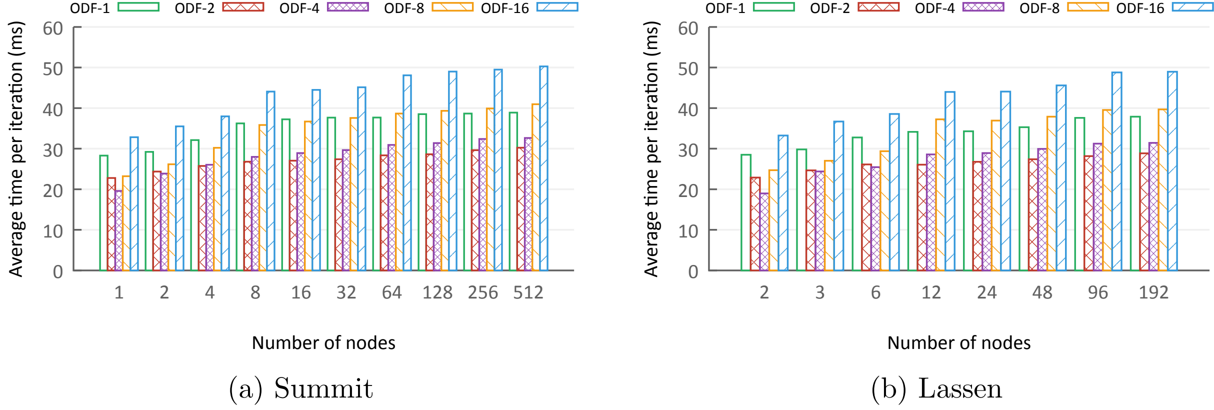
Figure 3.5: Performance of Jacobi3D with varying overdecomposition factors on a single node of OLCF Summit.

HAPI) to asynchronously invoke a Charm++ callback function once the operations in the communication stream are complete. The overdecomposition factor (ODF) is varied from one chare per GPU (MPI-like decomposition) to 16 chares per GPU; we expect overdecomposition to provide performance improvements due to computation-communication overlap up to a certain point, after which overheads from the finer granularity start to dominate performance.

`cudaStreamSynchronize` yields significant slowdowns over the versions with runtime support as PEs are fully blocked until all operations in the communication stream complete. Hence the Charm++ scheduler can neither initiate GPU operations of other chares nor progress host-side communication including the handling of incoming messages. The two HAPI mechanisms demonstrate up to 83% increased performance, with ODF-4 providing the largest performance improvement over ODF-1 of 43%. As expected, as the overdecomposition factor grows further, the increase in overall communication volume and overheads caused by smaller work units start to degrade performance.

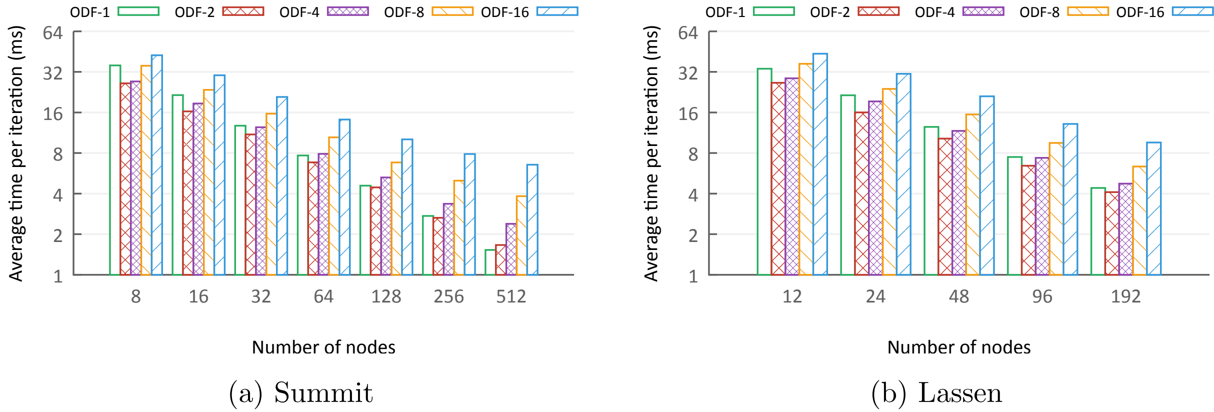
As can be seen in Figure 3.5, the polling-based mechanism of HAPI performs slightly better (about 500 us per iteration) than the callback-based mechanism. This is due to reasons discussed in Section 2.3.2, where the CUDA-generated thread sharing a physical core with a Charm++ runtime thread degrades performance. We therefore adopt the polling-based mechanism for the following scaling studies.

Weak scaling. We perform weak scaling of Jacobi3D with a base problem size of 1,536 × 1,536 × 1,536. Each dimension of the grid increases twofold as the number of GPUs double, in x, y, z order. As shown in Figures 3.6a and 3.6b, ODF-4 performs the best until 12 GPUs on Summit and 24 GPUs on Lassen, obtaining up to 44% and 50% performance improvement over ODF-1, respectively. On larger node counts, however, ODF-2 begins to



(a) Summit (b) Lassen

Figure 3.6: Weak scaling performance of Jacobi3D.

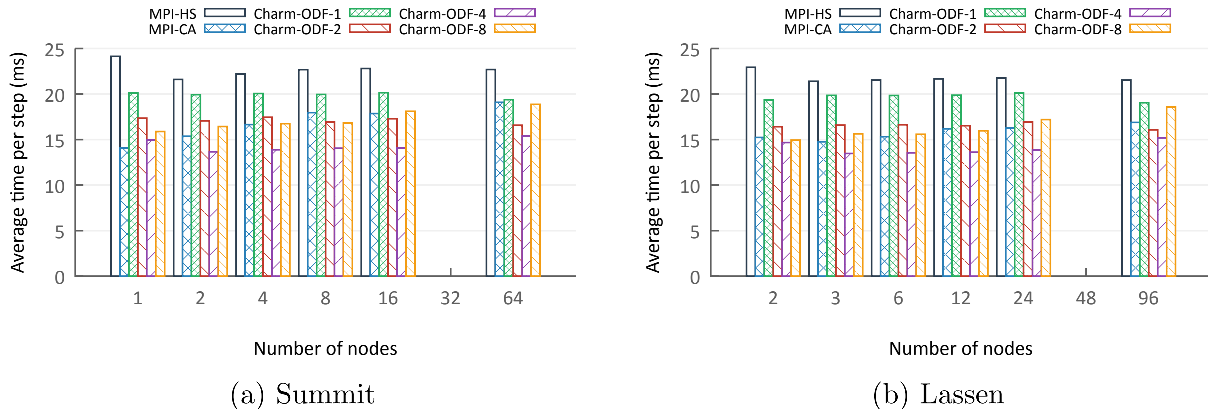


(a) Summit (b) Lassen

Figure 3.7: Strong scaling performance of Jacobi3D.

outperform ODF-4 with performance improvements compared to ODF-1 ranging between 24%-37% on Summit and 28%-33% on Lassen. We were unable to determine the exact cause of this crossover behavior and only observed longer idle times between iterations with ODF-4 after the crossover point. Nevertheless, an adequate degree of overdecomposition significantly improves performance by achieving computation-communication overlap.

Strong scaling. Jacobi3D is strong scaled with a problem size of 3,072 × 3,072 × 3,072, from 48 GPUs to 3,072 and 768 GPUs on Summit and Lassen, respectively. As in Figures 3.7a and 3.7b, ODF-2 provides the best performance until 1,536 GPUs on Summit and 768 GPUs on Lassen, but its performance improvement over ODF-1 decreases from 35% to 3% on Summit and 27% to 7% on Lassen. With 3,072 GPUs on Summit, overdecomposition degrades performance as observed by the 8% slowdown with ODF-2. This is within our expectations, however, as the performance improvement achievable with overdecomposition diminishes as the size of each work unit decreases with strong scaling. At large node counts,



(a) Summit

(b) Lassen

Figure 3.8: Weak scaling performance of MiniMD.

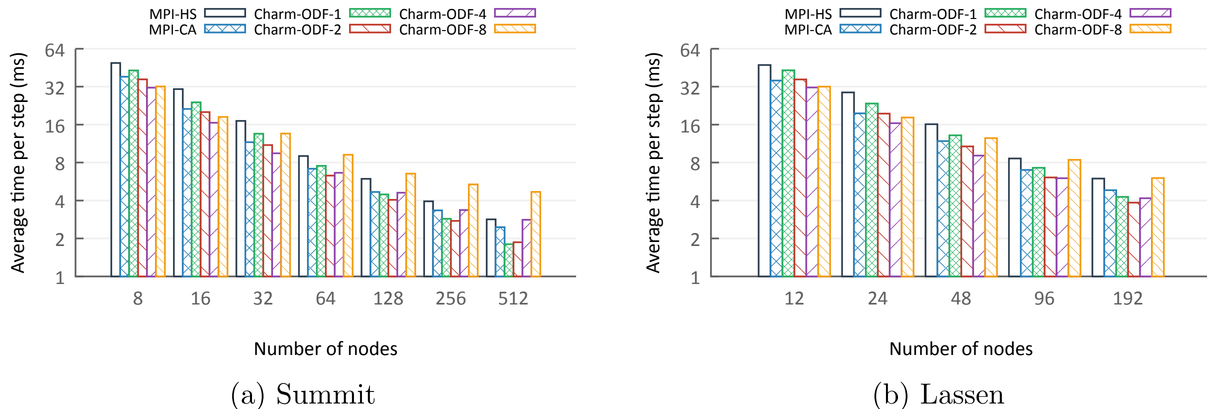
overdecomposition results in a small work unit being split up into even smaller pieces, aggravating fine-grained overheads.

3.4.2 MiniMD

The original CUDA-aware MPI version (marked MPI-CA) and modified host-staged version (marked MPI-HS, uses explicit copies between host and device) of MiniMD are benchmarked alongside the Charm++ versions employing overdecomposition. It should be noted that their performance is provided only for reference, since the Charm++ versions exercise a different communication pattern to facilitate computation-communication overlap as described in Section 3.3.2.

Weak scaling. We perform weak scaling of MiniMD with a base domain size of 192 $\times$ 192 $\times$ 192, which results in 28 million atoms that are split across 6 GPUs. As the number of GPUs double, each dimension of the grid is doubled, in x, y, z order. Figures 3.8a and 3.8b show the weak scaling performance up to 384 GPUs with domain size of 768 $\times$ 768 $\times$ 768 and atom count of 1.8 billion. We do not obtain results from 768 GPUs and onwards as an integer overflow occurs in the number of atoms. Results with 192 GPUs are not plotted as a NaN error causes computational kernels to run abnormally fast. These errors have been reported to the MiniMD developers.

It can be observed that the Charm++ version of MiniMD with an overdecomposition factor of four (Charm-ODF-4) performs the best except on a single node of Summit, where the CUDA-aware MPI version (MPI-CA) performs better. ODF-4 achieves speedups over ODF-1 ranging 26%-45% on Summit and 25%-47% on Lassen. Despite the lack of direct GPU-GPU transfers in the Charm++ versions, overlap of computation and communica-



(a) Summit (b) Lassen

Figure 3.9: Strong scaling performance of MiniMD.

tion achieved from overdecomposition allows Charm-ODF-4 to outperform MPI-CA in most configurations.[1]

Strong scaling. MiniMD is strong scaled with a domain size of 512 × 512 × 512 that contains 536 million atoms, from 48 GPUs to 3,072 GPUs on Summit and 768 GPUs on Lassen. As shown in Figures 3.9a and 3.9b, ODF-4 performs the best with performance improvements over ODF-1 between 36%-42% until 192 GPUs on Summit and 21%-44% until 384 GPUs on Lassen. Afterwards, ODF-2 provides the best performance with improvements decreasing from 19% to 3% on Summit and 11% on Lassen, except 3,072 GPUs on Summit where ODF-1 outperforms ODF-2 by 3%. Again, the results align with our expectation that the performance improvement obtainable with overdecomposition diminishes at the tail end of strong scaling, due to the smaller size of work units.

3.5 RELATED WORK

There has been extensive research on optimizing performance with overlap of computation and communication. Task-based runtime systems including HPX [28], OmpSs [29], Legion [30], and StarPU [31] exploit overlap of computation and communication through different mechanisms, most of which support execution on GPU-accelerated systems. In particular, techniques to optimize computation-communication overlap by addressing the inefficient interactions between OmpSs and MPI are discussed in the work by Castillo et al. [21] With a focus on overdecomposition and GPU execution, our work presents application design considerations and implementation details of a runtime feature for asynchronous

[1]The difference in communication pattern should also be taken into account.

progress, which can also be utilized by other task-based runtime systems and applications seeking to maximize computation-communication overlap on modern GPU systems.

3.6 CONCLUDING REMARKS

We discussed important considerations for achieving computation-communication overlap with overdecomposition on GPU systems, including the need to prioritize communication in the application and avoid synchronization with support from the runtime system. Techniques to address these issues have been presented and implemented in proxy applications and the runtime of the Charm++ parallel programming system. We demonstrated significant improvements in weak scaling performance of proxy applications on today's leadership-class GPU systems, albeit diminishing but expected returns with strong scaling. We also established interoperability of Charm++ with the Kokkos performance portability model, which is a valuable milestone towards the preparation for GPUs from different vendors in upcoming leadership-class systems.

CHAPTER 4: GPU-AWARE COMMUNICATION FOR MESSAGE-DRIVEN EXECUTION

Communication is an indispensable component of modern computing, as increasingly more applications run on distributed-memory clusters and cloud systems. With the distinction of work and data, data needs to be transferred to where work is, or vice versa; it is generally data that moves in preparation for work (tasks) to be carried out, and the time taken for such data movement is not avoidable but should be minimized. The problem of data movement is becoming more complicated today, with increasingly heterogeneous system architectures and deep levels of memory hierarchy. Extensive work has been done to support the transfer of data in GPU memory, i.e., GPU-aware communication, in the context of MPI, but there has been little work on such support for message-driven execution, which is amenable to the active messages model [9]. In this chapter, we explore how GPU-aware communication can be integrated into message-driven execution to improve both programmer productivity and performance on heterogeneous systems.

Although vendors provide GPU programming models such as CUDA for executing kernels and transferring data, their limited functionality makes it challenging to implement a general communication backend for parallel programming models on distributed-memory machines. Direct GPU-GPU communication crossing the process boundary can be implemented using CUDA Inter-process Communication (IPC), but requires extensive optimizations such as IPC handle cache and pre-allocated device buffers [32]. Direct inter-node transfers of GPU data cannot be implemented solely with CUDA and requires additional support from the networking stack [33]. Adding support for GPUs from other vendors such as AMD or Intel requires more development and optimization efforts that could be spent elsewhere.

A number of software frameworks, such as GASNet [34], libfabric [35], and UCX [36], aim to provide a unified communication layer over diverse networking hardware. While a few have been successfully adopted in parallel programming models including MPI and PGAS, UCX is one of the first communication frameworks to support production-grade inter-GPU communication on a wide range of modern GPUs and interconnects. We take advantage of UCX's capability to perform direct GPU-GPU transfers to support GPU-aware communication in multiple parallel programming models from the Charm++ ecosystem: Charm++, Adaptive MPI (AMPI), and Charm4py. We extend the UCX machine layer in the Charm++ runtime system to enable the transfer of GPU buffers and expose this functionality to the parallel programming models, with model-specific implementations to support their user applications. Our tests on a leadership-class system show that this approach substantially improves the performance of GPU-aware communication for all models.

4.1 BACKGROUND

4.1.1 GPU-aware Communication

GPU-aware communication has developed out of the need to rectify productivity and performance issues with data transfers involving GPU buffers. Without GPU-awareness, additional code is required to explicitly move data between host and device memory, which also substantially increases latency and reduces attainable bandwidth.

The GPUDirect [37] family of technologies have been leading the effort to resolve such issues on NVIDIA GPUs. Version 1.0 allows Network Interface Controllers (NICs) to have shared access to pinned system memory with the GPU and avoid unnecessary memory copies, and version 2.0 (GPUDirect P2P) enables direct memory access and data transfers between GPU devices on the same PCIe bus. GPUDirect RDMA [38] utilizes Remote Direct Memory Access (RDMA) technology to allow the NIC to directly access memory on the GPU. Based on GPUDirect RDMA, the GDRCopy library [39] provides an efficient low-latency transport for small messages. The Inter-Process Communication (IPC) feature introduced in CUDA 4.1 enables direct transfers between GPU data mapped to different processes, improving the performance of communication crossing the process boundary [32].

MPI is one of the first parallel programming models and communication standards to adopt these technologies and support GPUs in the form of CUDA-aware MPI, which is available in most MPI implementations. Other parallel programming models have added support for GPU-aware communication by either implementing their own mechanisms with GPUDirect and CUDA IPC or adopting a communication library such as UCX.

4.1.2 UCX

Unified Communication X (UCX) [36] is an open-source, high-performance communication framework that provides abstractions over various networking hardware and drivers, including TCP, OpenFabrics Alliance (OFA) verbs, Intel Omni-Path, and Cray uGNI. It is currently being developed at a fast pace with contributions from multiple hardware vendors as well as the open-source community.

UCX provides support for tag-matching send/receive, stream-oriented send/receive, active messages, remote memory access, and atomic operations. UCP is the high-level protocol layer in UCX, whose API can be used by parallel programming models to implement a performance-portable communication backend. Projects using UCX include Dask, Open-MPI, MPICH, and Charm++. GPU-aware communication is supported on NVIDIA and

AMD GPUs through its tagged and stream APIs. When provided with pointers to GPU memory, these APIs utilize the respective CUDA or ROCm libraries to perform efficient GPU-GPU transfers.

4.1.3 Adaptive MPI

Adaptive MPI (AMPI) [11] is an MPI library implementation developed on top of the Charm++ runtime system. AMPI virtualizes the concept of an MPI rank: whereas a traditional MPI library equates ranks with operating system processes, AMPI supports execution with multiple ranks per process by associating each rank with a chare object. This empowers AMPI to co-schedule ranks that are located on the same PE based on the delivery of messages. Users can tune the number of ranks they run with based on performance. AMPI ranks are also migratable at runtime for the purposes of dynamic load balancing or checkpoint/restart-based fault tolerance.

Communication in AMPI is handled through Charm++ and its optimized networking layers. AMPI optimizes communication based on locality of the recipient rank as well as the size and datatype of the message buffer. Small buffers are packed inside a regular Charm++ message in an eager fashion, and the Zero Copy API [40] is used to implement a rendezvous protocol for larger buffers. The underlying runtime optimizes message transmission based on locality over user-space shared memory, Cross Memory Attach (CMA) for within-node, or RDMA across nodes. This work extends such optimizations to the context of multi-GPU nodes connected by a high performance network programmable with UCX.

4.1.4 Charm4Py

Charm4Py [41] is a parallel programming framework based on the Python language, developed on top of the Charm++ runtime system. It seeks to provide an easily-accessible parallel programming environment with improved programmer productivity through Python, while maintaining high scalability and performance of the adaptive C++-based runtime. Being based on Python, Charm4py can readily take advantage of many widely-used software libraries such as NumPy, SciPy, and pandas.

Chare objects in Charm4py communicate with each other by asynchronously invoking entry methods like Charm++. The parameters are serialized and packed into a message that is handled by the underlying Charm++ runtime system. This allows our extension of the UCX machine layer to also support Charm4py. Charm4py also provides a functionality to establish streamed connections between a pair of chares, called *Channels* [42]. Channels

```
// Sender object's method
void Sender::foo() {
  // Send a message to the receiver object
  // to execute the 'bar' entry method
  receiver.bar(my_val1, my_val2);
}

// Receiver object's entry method,
// executed once the sender's message
// is picked up by the scheduler
void Receiver::bar(int val1, double val2) {
  // val1 and val2 are available
  ...
}
```

Figure 4.1: Message-driven execution in Charm++.

provide explicit send/receive semantics to exchange messages similar to MPI, but retains asynchrony by suspending the caller object until communication is complete. We extend the Channels feature to support GPU-aware communication in Charm4py.

4.2 DESIGN AND IMPLEMENTATION

To accelerate communication of GPU data, we utilize the capability of UCX to directly send and receive GPU data. UCX is supported as a machine layer in Charm++, positioned at the lowest level of the software stack directly interfacing the interconnect. As AMPI and Charm4py are also built on top of the Charm++ RTS, all host-side communication travels through the various layers of the RTS where layer-specific headers are added or extracted, with actual communication primitives executed by the machine layer.

The message-driven execution model in Charm++, as shown in Figure 4.1, necessitates additional metadata to be attached to each message, so that the receiver can figure out how to handle the message. This involves determining which chare object and entry method a message is targeting. Charm++ in traditional CPU-based systems achieves this by allocating a message that is big enough for both the metadata and user's payload on host memory, copying the payload into the message, and passing the prepared message to the machine layer to be sent. On GPU systems, however, this becomes a challenging problem since the user's payload data can now be in GPU memory while the metadata still needs to be in host memory (because the RTS is running on the CPU).

Our first approach of supporting GPU-aware communication in the Charm++ RTS, the *GPU Messaging API*, maintains the message-driven execution model. This is achieved by

retaining the host-side message that contains the necessary metadata and user's data in host memory, while separately sending the user's data in GPU memory through the UCX machine layer. Once the host-side message arrives on the receiver (which is possible with pre-posted receives for the message queue), a receive for the incoming GPU data can be posted using information extracted from the metadata in the host-side message. The destination GPU buffer is provided by the user using a mechanism that builds on the *Zero Copy API* [40], which is explained in more detail a later section. A noticeable limitation of the GPU Messaging API is the delay in posting the receive for the GPU data, as it can only be performed after the arrival of the host-side message.

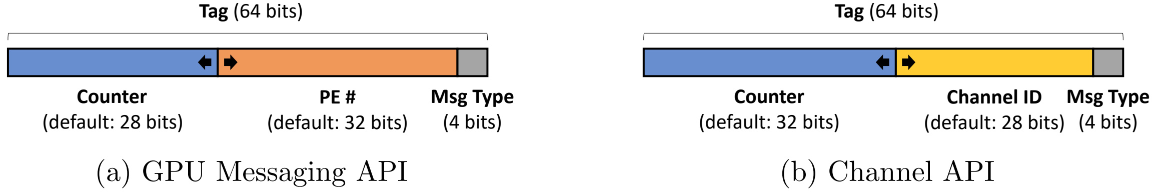
To further improve performance from the GPU Messaging API, we have developed the *Channel API* in Charm++. It allows a 'channel' to be created between a pair of chare objects, which exposes two-sided send and receive semantics that directly translate into communication primitives in UCX. This eliminates the overhead from the host-side message needed by the GPU Messaging API, but it can be seen as a deviation from the message-driven execution model; only data moves between chares, not the flow of execution. This may make it difficult for some applications (especially irregular applications that benefit from a message-driven model) to adopt the Channel API. While Charm++ supports both the GPU Messaging API and the Channel API, GPU-aware communication in AMPI and Charm4py currently only utilize the GPU Messaging API. Integration of the Channel API into AMPI and Charm4py is being worked on as it requires extensive changes in the respective codebases to move away from the message-driven communication model.

In the following sections, we discuss the updates to the UCX machine layer and the various parallel programming models in the Charm++ ecosystem to enable GPU-aware communication.

4.2.1 UCX Machine Layer

Originally contributed by Mellanox, the UCX machine layer in Charm++ is designed to handle low-level communication on networks supported by UCX. It utilizes the Tagged API exposed by the UCP layer in UCX, enabling messages to be exchanged with two-sided communication routines. Although the UCP Active Message API is potentially a better fit for Charm++, it was presumably not production-ready at the time of the UCX machine layer implementation and it also currently lacks support for GPUs.

During the initialization phase of the Charm++ runtime, each process creates a UCP worker and establishes endpoints between all workers using PMIx [43]. In the non-SMP mode of Charm++ where each PE is contained in a single process, there is a UCP worker



(a) GPU Messaging API

(b) Channel API

Figure 4.2: Tag generation schemes to support GPU-aware communication in the UCX machine layer.

for each PE. In the SMP mode, PEs are implemented using threads and multiple PEs can be contained in each process; the UCP worker in each process is managed by a separate communication thread. In this work, the non-SMP mode of Charm++ is used.

Once the endpoints are established, UCP tagged receives for eager messages (`ucp_tag_recv_nb`) are posted in advance to handle incoming messages. All eager host-side communication uses the same tag, `UCX_MSG_TAG_EAGER`, allowing the UCX machine layer to receive messages from any other PE. This conforms to the message-driven execution model in Charm++, since the runtime system must handle incoming messages without relying on receives issued by the user application. In addition to the pre-posted receives for eager messages, the UCP worker also probes for rendezvous messages (`ucp_tag_probe_nb`) with another tag, `UCX_MSG_TAG_PROBE`. When a probe is successful, the message is received with `ucp_tag_msg_recv_nb`. The invocation of an entry method from a chare object creates a message, which travels down to the machine layer. It is then sent to the target endpoint using UCP tagged sends (`ucp_tag_send_nb`), with `UCX_MSG_TAG_EAGER` for eager and `UCX_MSG_TAG_PROBE` for rendezvous messages. The non-blocking send and receive UCP calls are advanced by the progress function, `ucp_worker_progress`, which is executed by the Charm++ scheduler. When the UCX machine layer receives a message, it is stored in the corresponding PE's message queue to be picked up by the scheduler and execute the target entry method.

To support GPU-aware communication in the Charm++ family of parallel programming models, we extend the UCX machine layer to provide an additional interface for sending and receiving data in GPU memory with the UCP Tagged API. We have added support for both the GPU Messaging API and Channel API, with the former used by all three programming models (Charm++, AMPI, and Charm4py) whereas the latter is currently only used in Charm++. Since both APIs build on the two-sided send and receive semantics, we implement different schemes to generate tags for passing to the underlying UCX communication primitives as shown in Figure 4.2.

For the GPU Messaging API, the first four bits of the 64-bit tag are used to store the mes-

```
// GPU Messaging API
void LrtsDeviceSend(int dest_pe, const void*& ptr, size_t size, uint64_t& tag);
void LrtsDeviceRecv(DeviceRdmaOp* op, DeviceRecvType type);

// Channel API
void LrtsChannelSend(int dest_pe, const void*& ptr, size_t size, void* cb, uint64_t tag);
void LrtsChannelRecv(const void*& ptr, size_t size, void* cb, uint64_t tag);
```

Figure 4.3: GPU-aware communication APIs exposed by the UCX machine layer.

sage type, which is set to a new value called `UCX_MSG_TAG_DEVICE`. This allows the Charm++ RTS to recognize the arrival of a host-side metadata message used by the GPU Messaging API. The remainder of the tag is split into storing the source PE index (32 bits by default) and the value of the counter maintained by the source PE (28 bits by default) which is incremented on each transmission of GPU data. This division can be modified at compile time to accommodate different scenarios.

With the Channel API, tag counters are not maintained by the source PE but by each endpoint chare of the channel. Each chare increments its counter when a send or receive call is made to the channel. A new message type, `UCX_MSG_TAG_CHANNEL`, is assigned to the first four bits of the 64-bit tag to distinguish messages used by the Channel API. By default, 28 bits are used to store the channel ID, which must be unique in the program. The remaining 32 bits are retrieved from the per-channel counters maintained by the sender or receiver chare participating in the channel.

Figure 4.3 describes the functions that expose the core functionalities of GPU-aware communication in the UCX machine layer to the upper layers, and their usage is described in the following sections.

4.2.2 Charm++

Two different mechanisms have been implemented to support GPU-aware communication in the Charm++ runtime system: (1) GPU Messaging API and (2) Channel API. We discuss why and how these mechanisms are designed, along with their implications on performance.

GPU Messaging API. Taking inspiration from the Zero Copy API, the GPU Messaging API retains the entry method syntax and message-driven execution model, where the flow of execution is transferred from the sender chare to the receiver. As shown in Figure 4.4, we provide an additional attribute, `device`, to allow users to annotate GPU buffers in the Charm++ Interface (CI) file. The CI file contains declarations of chare objects and their

```
// Charm++ Interface (CI) file
// Declares chare objects and their entry methods
chare MyChare {
  entry MyChare();
  entry void recv(device char data[size], size_t size);
};
```

```
// C++ source file
// (1) Sender chare
void MyChare::send() {
  peer.recv(CkDeviceBuffer(send_gpu_data), size);
}

// (2) Receiver's post entry method
void MyChare::recv(char*& data, size_t& size) {
  // Set the destination GPU buffer
  // Receive size is optional
  data = recv_gpu_data;
}

// (3) Receiver's regular entry method
void MyChare::recv(char* data, size_t size) {
  // Receive complete, GPU data is available
  ...
}
```

```
// Converse layer metadata
struct CmiDeviceBuffer {
  const void* ptr; // Source GPU buffer address
  size_t size;
  uint64_t tag; // Set in the UCX machine layer
  ...
};

// Charm++ core layer metadata
struct CkDeviceBuffer : CmiDeviceBuffer {
  CkCallback cb; // Support Charm++ callbacks
  ...
};
```

Figure 4.4: Example usage of the GPU Messaging API in Charm++.

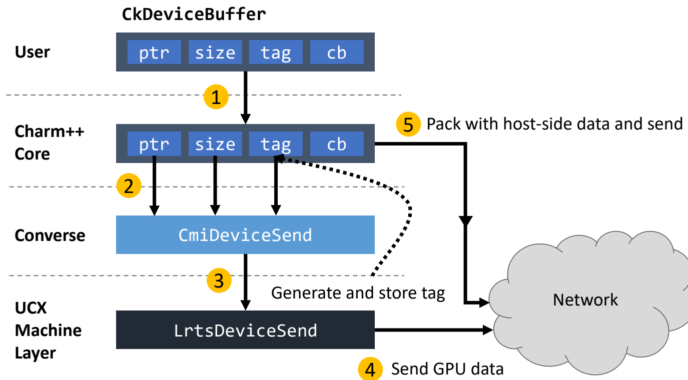


Figure 4.5: Sender-side logic of the GPU Messaging API in Charm++.

entry methods, and potentially the overall structure of parallel execution which are used for code generation. An entry method invocation such as `peer.recv` executes a generated code block, which is modified to send both the source GPU buffer and a separate host-side message that contains information about the GPU data transfer. This information is encapsulated in a structure called `CkDeviceBuffer`, which is passed to the Charm++ core layer and then down to the Converse layer in the form of `CmiDeviceBuffer`, then finally to the UCX machine layer to perform the send of GPU data with `LrtsDeviceSend`. Charm++ callbacks, stored as `cb` in `CKDeviceBuffer`, are used to notify the sender or receiver that the GPU data transfer is complete. This process is illustrated in Figure 4.5.

The 64-bit tag stored in `CmiDeviceBuffer` (and by inheritance in `CkDeviceBuffer`) is set to the correct value by the UCX machine layer in the `LrtsDeviceSend` function. It is used in sending the GPU data, and is also transferred to the receiver chare inside the host-side message, to be used as the tag for receiving the incoming GPU data. Once the host-side message arrives on the destination PE, the corresponding receive for the incoming GPU data is posted in `LrtsDeviceRecv`. The `DeviceRdmaOp` struct stores and maintains information necessary for the receive operations in the various layers of the Charm++ RTS, including the address of the destination GPU buffer, size of the data, and the tag set by the sender. `DeviceRecvType` denotes which of the parallel programming models (Charm++, AMPI, or Charm4py) has posted the receive, allowing the appropriate handler function to be invoked once the GPU data has been received.

To receive the incoming GPU data directly into the user's destination buffer and avoid an extra copy, we provide a *post entry method* that allows the user to specify the address of the destination GPU buffer in advance. An example usage is shown in Figure 4.4. The post entry method is executed by the runtime system when the host-side message associated with

```
// C++ source file
// Chare init
void MyChare::init() {
  // Channel ID is 0
  channel = CkChannel(0, thisProxy[peer]);
}

// Data exchange
void MyChare::exchange() {
  channel.send(send_buf, size, CkCallbackResumeThread());
  channel.recv(recv_buf, size, CkCallbackResumeThread());
}
```

Figure 4.6: Example usage of the Channel API in Charm++.

the GPU data transfer arrives. Once the RTS is informed of the destination GPU buffer address, it posts a receive for the incoming GPU data with `LrtsDeviceRecv`, also using information contained in the host-side message such as the tag used in the UCP send. Once the GPU data arrives, the regular entry method of the receiver chare is executed, at which point the received GPU buffer is available to the user. One downside of the GPU Messaging API is that performance may degrade from the delay in posting the receive for the incoming GPU data, which arises from the receiver not knowing which UCX tag was used until the host-side message arrives.

Channel API. The Channel API aims to improve communication performance by avoiding the need of a host-side message in the GPU Messaging API. A channel is first created between a pair of chare objects, with an ID provided by the user that has to be unique in the program. The channel can then be used to send and receive data by providing the address of source or destination buffer, size of the data, and a `CkCallback` object or a Charm++ future [44] that will be invoked on completion of the channel primitive. An example is shown in Figure 4.6, where a special type of callback, `CkCallbackResumeThread` is used to suspend the calling thread to perform an asynchronous send or receive. The thread will be awakened when the channel primitive completes, allowing the chare to continue executing. When a Charm++ future is provided, it can be later waited on for completion similar to non-blocking MPI communication that uses an MPI_Request.

The channel object maintains all information needed for GPU data exchanges between the participating pair of chares, including the 64-bit tag. Because both the sender and receiver chares keep track of which UCX tag is being used for communication, and the destination GPU buffer address is provided by the user in the channel receive call, there is no more need for a host-side message. The receiver chare has all the information needed to post a receive

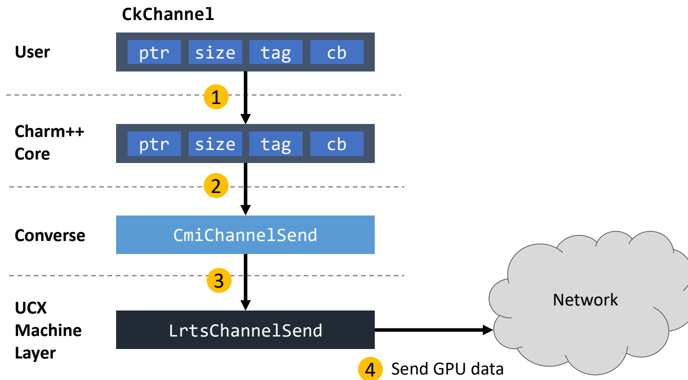


Figure 4.7: Sender-side logic of the Channel API in Charm++.

for the incoming GPU data, and the UCX machine layer is accessed almost directly by the Channel API through `LrtsChannelSend` and `LrtsChannelRecv` calls. These properties allow the Channel API to demonstrate better performance than the GPU Messaging API, as discussed in Section 4.3. The sender side logic of the Channel API is illustrated in Figure 4.7.

4.2.3 Adaptive MPI

Each AMPI rank is implemented as a chare object on top of the Charm++ runtime system, to enable virtualization and adaptive runtime features such as load balancing. Communication between AMPI ranks occurs through an exchange of AMPI messages between the respective chare objects. An AMPI message adds AMPI-specific data such as the MPI communicator and user-provided tag to a Charm++ message. We modify how an AMPI message is created to integrate GPU-aware communication with the GPU Messaging API and `CkDeviceBuffer` metadata object. This change is transparent to the user, and GPU buffers can be directly provided to AMPI communication primitives such as `MPI_Send` and `MPI_Recv` like any CUDA-aware MPI implementation. We are also currently exploring the integration of the Channel API into AMPI to potentially further improve performance, which can be done by creating a channel between the pair of chare objects each mapped to an AMPI rank.

An AMPI application can send GPU data by invoking a MPI send call with parameters including the address of the source buffer, number of elements and their datatype, destination rank, tag, and MPI communicator. The chare object that manages the destination rank is first determined, and the source buffer's address is checked to see if it is located on GPU memory. A software cache containing addresses known to be on the GPU is maintained

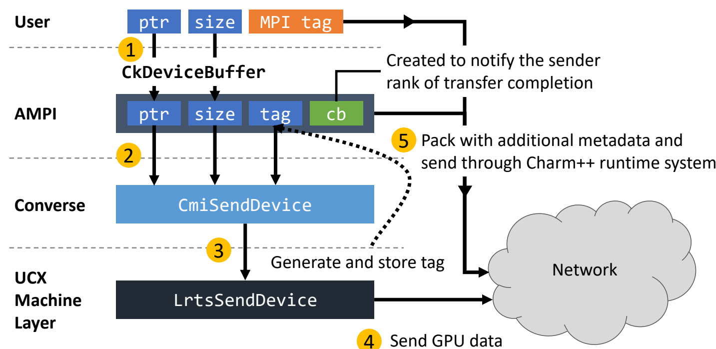


Figure 4.8: Sender-side logic of GPU-aware communication in AMPI.

on each PE to optimize this process. Figure 4.8 illustrates the mechanism that is executed when the source buffer is found to be on the GPU, where a `CkDeviceBuffer` object is first created in the AMPI runtime to store the information provided by the user. A Charm++ callback object is also created and stored as metadata, which is used by AMPI to notify the sender rank when the communication is complete. The source GPU buffer is sent in an identical manner as Charm++ through the UCX machine layer with `LrtsDeviceSend`. The tag that is needed by the receiver rank to post a receive for the incoming GPU data is also generated and stored inside the `CkDeviceBuffer` object. Note that this tag is separate from the MPI tag provided by the user, which is used to match the host-side send and receive.

Because there are explicit receive calls in the MPI model in contrast to Charm++, there are two possible scenarios regarding the host-side message that contains metadata: the message arrives before the receive is posted, and vice versa. If the message arrives first, it is stored in an unexpected message queue, which is searched for a match when the receive is posted later. If the receive is posted first, it is stored in a request queue to be matched when the message arrives. The receive for the incoming GPU data is posted after this match of the host-side message, with `LrtsDeviceRecv` in the UCX machine layer. Another Charm++ callback is created for the purpose of notifying the destination rank, which is invoked by the machine layer when the GPU data arrives.

4.2.4 Charm4py

GPU-aware communication is supported in Charm4py through its Channels feature, which allows streamed communication between a pair of chares. This is what the Channel API in Charm++ takes inspiration from, but GPU-aware communication in Charm4py is currently

```python
if not gpu_direct:
  # Host-staging mechanism (not GPU-aware)
  # Transfer GPU buffer to host memory and send
  charm.lib.CudaDtoH(h_send_buf, d_send_buf, size, stream)
  charm.lib.CudaStreamSynchronize(stream)
  channel.send(h_send_buf)

  # Receive on host and transfer to GPU
  h_recv_buf = channel.recv()
  charm.lib.CudaHtoD(d_recv_buf, h_recv_buf, size, stream)
  charm.lib.CudaStreamSynchronize(stream)
else:
  # GPU-aware communication
  # Send and receive GPU buffers directly
  channel.send(d_send_buf, size)
  channel.recv(d_recv_buf, size)
```

Figure 4.9: Channel-based communication in Charm4py. CUDA functions are included in the Charm++ library as C++ functions and exposed through Charm4py's Cython layer.

built on top of the GPU Messaging API of Charm++, not its Channel API. Like AMPI, the messaging mechanism in Charm4py is in the process of being updated to be able to adopt Charm++'s Channel API. While the Channels interface in Charm4py is in Python, its core functionalities are implemented with Cython [45] and the underlying Charm++ runtime system is comprised of C++. Cython generates C extension modules to support C constructs and types to be used with Python for interoperability and performance, and is used extensively in the Charm4py runtime. The Cython layer is also used to interface with the Charm++ runtime, which performs the bulk of the work for GPU-aware communication with the UCX machine layer. Note that the Python interface for UCX, UCX-Py [46], is not used in this work as Charm4py can directly utilize the UCX functionalities in C/C++ through the Charm++ runtime system.

Figure 4.9 compares our GPU-aware communication support against the host-staging mechanism in a ping-pong exchange of GPU data. Each of the two chares opens a channel to the other, which is used to exchange data either on the host or GPU memory determined by the `gpu_direct` flag. The host-staging version needs to explicitly move data between host and device memory using the CUDA API, adding complexity to the programmer and degrading performance. Note that the Charm4py channel send and receive calls are asynchronous; a send call returns control as soon as it is initiated, and the coroutine posting a receive is suspended and returns control back to the scheduler until the message arrives. Such asynchronous mechanisms are implemented with futures [47], a key component of Charm4py.

As can be seen from Figure 4.9, addresses of the source and destination GPU buffers

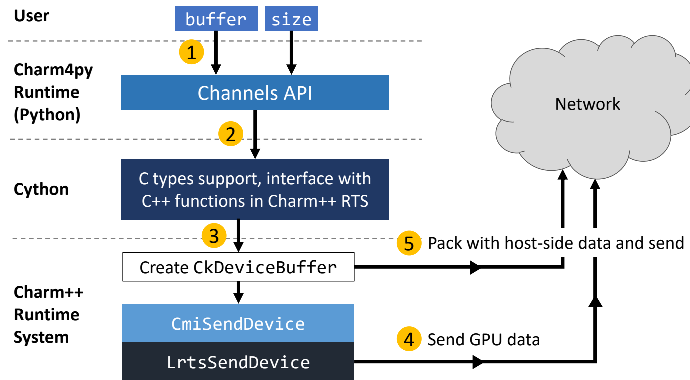


Figure 4.10: Sender-side logic of GPU-aware communication in Charm4py.

can be directly provided to Charm4py's Channel API. The address and size of the buffer are propagated to the Charm++ runtime system through the Cython layer, which are used to construct the `CkDeviceBuffer` metadata object. The steps after this point follow the GPU Messaging API, where the UCX machine layer sends the source GPU buffer using the provided metadata. The metadata is packed together with the host-side user data (if any) and Charm4py-specific information and sent in a separate code path to the receiver object. This process is illustrated in Figure 4.10.

When the host-side message containing metadata about the GPU data transfer arrives, `LrtsDeviceRecv` in the UCX machine layer posts a receive for the incoming GPU data. A Charm++ callback is created and tied to the `LrtsDeviceRecv` function, for the purpose of handling completion of the GPU communication. The invocation of this callback fulfills the Charm4py future object that was created on the channel receive call, which allows the user application (coroutine) to continue executing.

4.3 PERFORMANCE EVALUATION

In this section, we describe the hardware platform and software configurations, as well as the set of micro-benchmarks and proxy application used to evaluate the performance of our GPU-aware communication designs.

4.3.1 Experimental Setup

The Summit supercomputer at Oak Ridge National Laboratory is used to evaluate the performance of GPU-aware communication mechanisms implemented in Charm++, AMPI

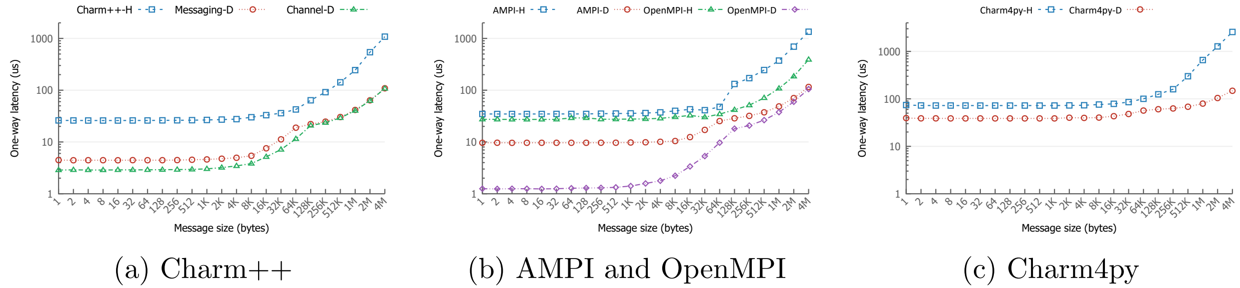
and Charm4py. The experiments are scaled up to 256 nodes of Summit, where each IBM AC922 node contains two IBM Power9 CPUs and six NVIDIA Tesla V100 GPUs. Each CPU is connected to three GPUs, which are interconnected via NVLink with a theoretical peak bandwidth of 50 GB/s. For a GPU to communicate with another GPU connected to the other CPU, data needs to travel through the X-Bus that connects the CPUs with a bandwidth of 64 GB/s. The network interconnect is based on Mellanox Enhanced Data Rate (EDR) Infiniband, providing up to 12.5 GB/s of bandwidth.

Charm++, AMPI and Charm4py are configured to use the non-SMP build, using one CPU core as the single PE for each process and one process per GPU device. On a single node of Summit, for example, up to six PEs and GPUs can be used. To accurately evaluate the impact of GPU-awareness on communication performance by separating communication from computation, we do not employ overdecomposition and instead decompose the problem domain into the same number of chare objects or AMPI ranks as the number of PEs and GPUs.

For reference, benchmark results with OpenMPI (which also maps one process to each GPU) are provided along with the performance of AMPI. Since both AMPI and OpenMPI utilize UCX to transfer GPU data, this comparison isolates the performance differential incurred by the layers above UCX. We expect the performance of AMPI to be slower because of the overheads associated with message-driven execution such as copies between the user application and the runtime system. We expect the performance of AMPI to be slower than OpenMPI because of the overheads associated with message-driven execution in the Charm++ RTS such as memory copies between the user application and the underlying runtime. This is in contrast to OpenMPI which can directly utilize UCX for communication.

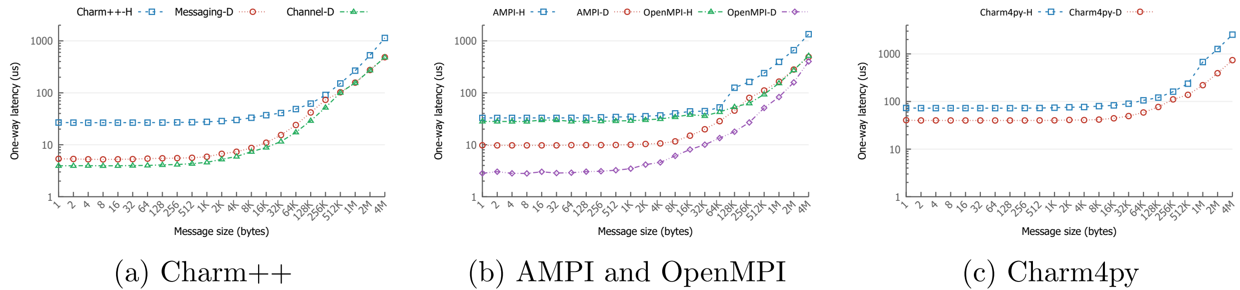
4.3.2 Micro-benchmarks

To evaluate the performance of point-to-point communication primitives involving GPU memory, we adapt the widely used OSU micro-benchmark suite [48] to Charm++ and Charm4py. For Charm++, we compare the performance of both the GPU Messaging API and the Channel API to the host-staging method. We expect the Channel API to perform better due to the lack of a metadata message delaying the receive for the GPU buffer. We also compare our GPU-aware communication mechanisms against host-staging in AMPI and OpenMPI. Performance results are presented with both axes in log-scale, comparing the GPU-aware version(s) of the benchmark (suffixed with D) against the host-staging version (suffixed with H). Results with the two different mechanisms in Charm++, the GPU Messaging API and Channel API, are provided as Messaging-D and Channel-D, respectively.



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

Figure 4.11: Comparison of intra-node latency between host-staging and direct GPU-GPU mechanisms.



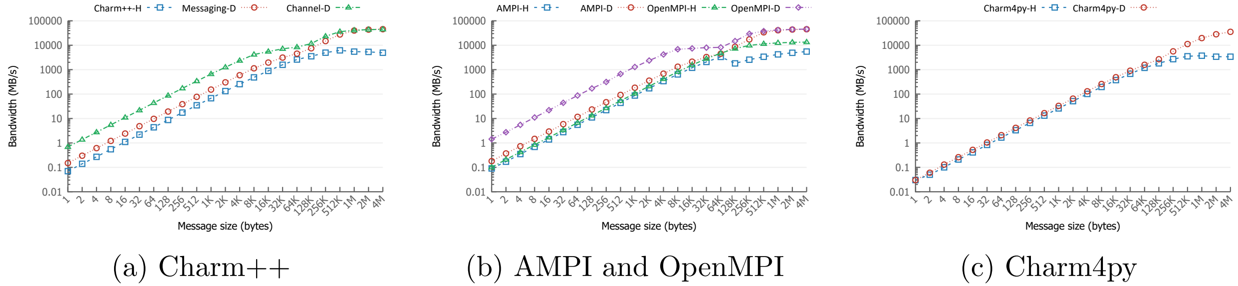
(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

Figure 4.12: Comparison of inter-node latency between host-staging and direct GPU-GPU mechanisms.

Latency. The OSU latency benchmark repeats ping-pong iterations for different message sizes, where the sender sends a message to the receiver and waits for a reply. Once the message arrives, the receiver sends a message with the same size back to the sender, completing the round trip. GPU-aware communication allows the message buffers to be supplied directly to the communication primitives, whereas the host-staging version requires additional data transfers between the host and device.

Figures 4.11 and 4.12 illustrate the improvements in intra-node and inter-node latency with GPU-awareness in Charm++, AMPI and Charm4py. The performance improvements in the latency benchmark are summarized in Tables 4.1 and 4.2, where the achieved speedups with small messages using the eager protocol are denoted in a separate row. As the Channel API performs better than the GPU Messaging API, its results are used for comparison against the host-staging mechanism. The observed improvement in latency increases with message size with large messages in all three programming models, as the host-staging mechanism suffers from performance degradation caused by host memory copies performed by the Charm++ runtime system.

Although the performance of AMPI improves substantially with GPU-aware communication, it does not quite match the latency of CUDA-aware OpenMPI. To further investigate



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

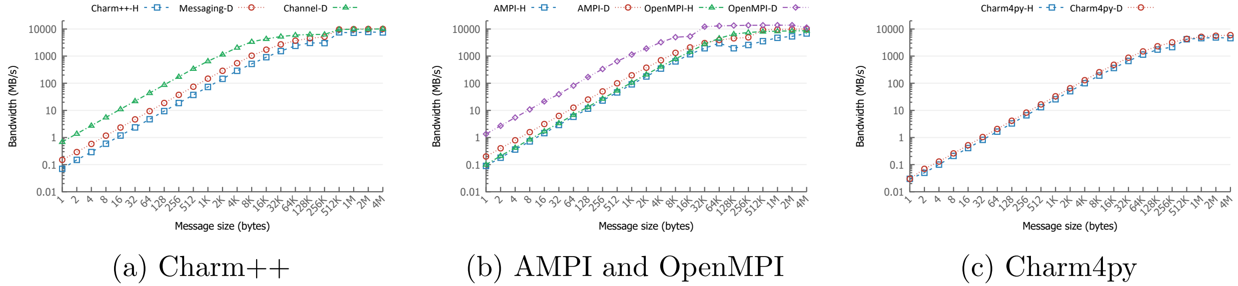
Figure 4.13: Comparison of intra-node bandwidth between host-staging and direct GPU-GPU mechanisms.

Improvement	Type	Charm++	AMPI	Charm4py
Latency	Range	3.1x – 10.1x	1.9x – 11.7x	1.8x – 17.4x
	Eager	9.1x	3.6x	1.9x
Bandwidth	Range	1.7x – 10.1x	1.3x – 10.0x	1.3x – 10.5x

Table 4.1: Performance improvement with intra-node GPU-aware communication.

this issue, we isolate the time taken in UCX by taking advantage of the modular structure of the UCX machine layer. We can easily disable the `CmiSend/RecvDevice` calls in the Converse layer and directly invoke the receive handlers, mimicking instant completion of the respective communication routines. This allows us to determine the time that is taken outside of UCX, which turns out to be about 8 μs. This tells us that the GPU data transfer itself with UCX has a latency of less than 2 μs, similar to OpenMPI. Thus most of the overhead is AMPI-specific, which includes multiple factors: message packing and unpacking, additional host-side message which contains metadata, Charm++ callback invocations, and the fact that the receiver rank cannot post a receive until the metadata message is received. There are also a couple of heap memory allocations that are needed to store metadata in the UCX machine layer to enable asynchronous communication. We plan to further analyze and optimize the code to get AMPI's performance as close to OpenMPI as possible. Redesigning the AMPI code path to utilize the newly developed Channel API of Charm++ instead of the GPU Messaging API could also bring AMPI's performance closer to OpenMPI.

It should be noted that the detection of the GDRCopy library by UCX is essential in order to achieve low latencies with small messages, which is not included in the default library search path on Summit. With the rendezvous protocol, UCX switches to the CUDA IPC transport for intra-node transfers, and to the pipelined host-staging mechanism that stages GPU data on host memory in chunks for inter-node communication.



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

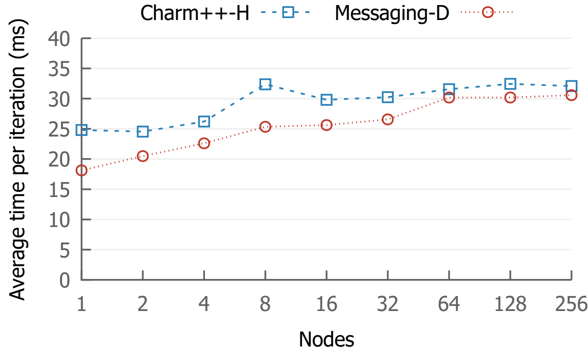
Figure 4.14: Comparison of inter-node bandwidth between host-staging and direct GPU-GPU mechanisms.

Improvement	Type	Charm++	AMPI	Charm4py
Latency	Range	1.5x – 6.8x	1.8x – 3.5x	1.5x – 3.4x
	Eager	6.8x	3.4x	1.8x
Bandwidth	Range	1.3x – 9.4x	1.3x – 2.6x	1.0x – 1.5x

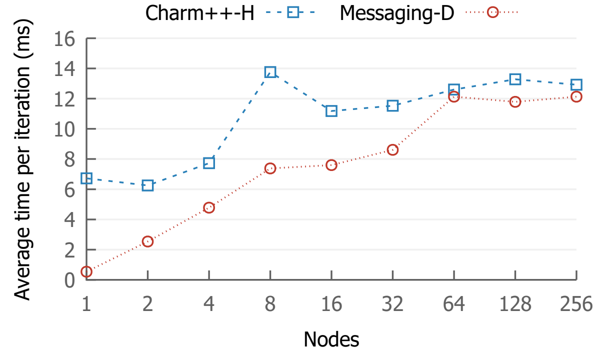
Table 4.2: Performance improvement with inter-node GPU-aware communication.

Bandwidth. In the OSU bandwidth benchmark, the sender performs a number of back-to-back non-blocking sends specified by the window size for each message size, then waits for a reply from the receiver. The receiver performs the reverse, posting multiple non-blocking receives followed by a send. The increases in bandwidth achieved by our GPU-aware communication mechanisms are illustrated in Figures 4.13 and 4.14, with the range of improvements summarized in Tables 4.1 and 4.2.

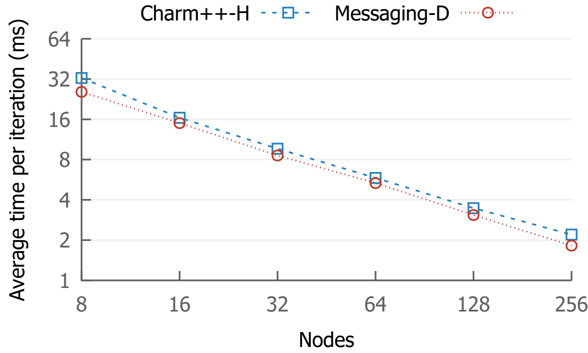
Charm++ and AMPI achieve close to the maximum attainable bandwidth (50 GB/s for intra-node, 12.5 GB/s for inter-node), with Charm++ demonstrating up to 44.7 GB/s and 10 GB/s, and AMPI up to 45.4 GB/s and 10 GB/s for intra-node and inter-node, respectively. As expected in Charm++, the Channel API (Channel-D) achieves higher bandwidth compared to the GPU Messaging API (Messaging-D) especially with smaller messages. This is because the Channel API is able to post the receive for the incoming GPU buffer without being delayed by the metadata message. It is worth noting that the host-staging version of AMPI (AMPI-H) suffers a degradation in bandwidth at 128 KB due to a sudden drop in performance, which is an issue that is being investigated. Charm4py's bandwidth only reaches 35.5 GB/s for intra-node and 6.0 GB/s for inter-node in the given range of message sizes, but we observe that it keeps increasing as messages become larger than 4 MB.



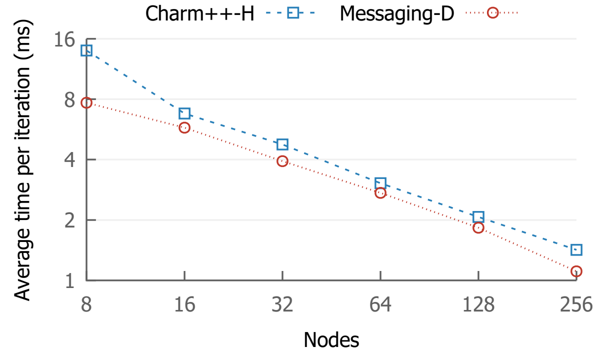
(a) Weak scaling, overall time



(b) Weak scaling, communication time



(c) Strong scaling, overall time



(d) Strong scaling, communication time

Figure 4.15: Comparison of Charm++ Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.

4.3.3 Proxy Application: Jacobi3D

To assess the impact of GPU-aware communication on application performance, we implement a proxy application, Jacobi3D, on all three parallel programming models: Charm++, AMPI, and Charm4py. Jacobi3D performs the Jacobi iterative method in a three-dimensional space, using CUDA kernels to perform stencil computations on the GPU. The problem domain is decomposed into equal-size cuboid blocks, using a decomposition strategy that minimizes surface area to reduce communication volume. Each block exchanges its halo data on the GPU with up to six neighbors, which are either provided directly to the communication primitives (if GPU-aware) or staged through host memory. Note that Jacobi3D is configured to run for a set number of iterations without convergence checks, to be able to evaluate the performance of point-to-point communication.

We evaluate both weak and strong scaling performance of Jacobi3D using up to 256 nodes (1,536 GPUs) of Summit, comparing the per-iteration execution times and communication times of the host-staging and GPU-aware communication mechanisms. Jacobi3D is weak-

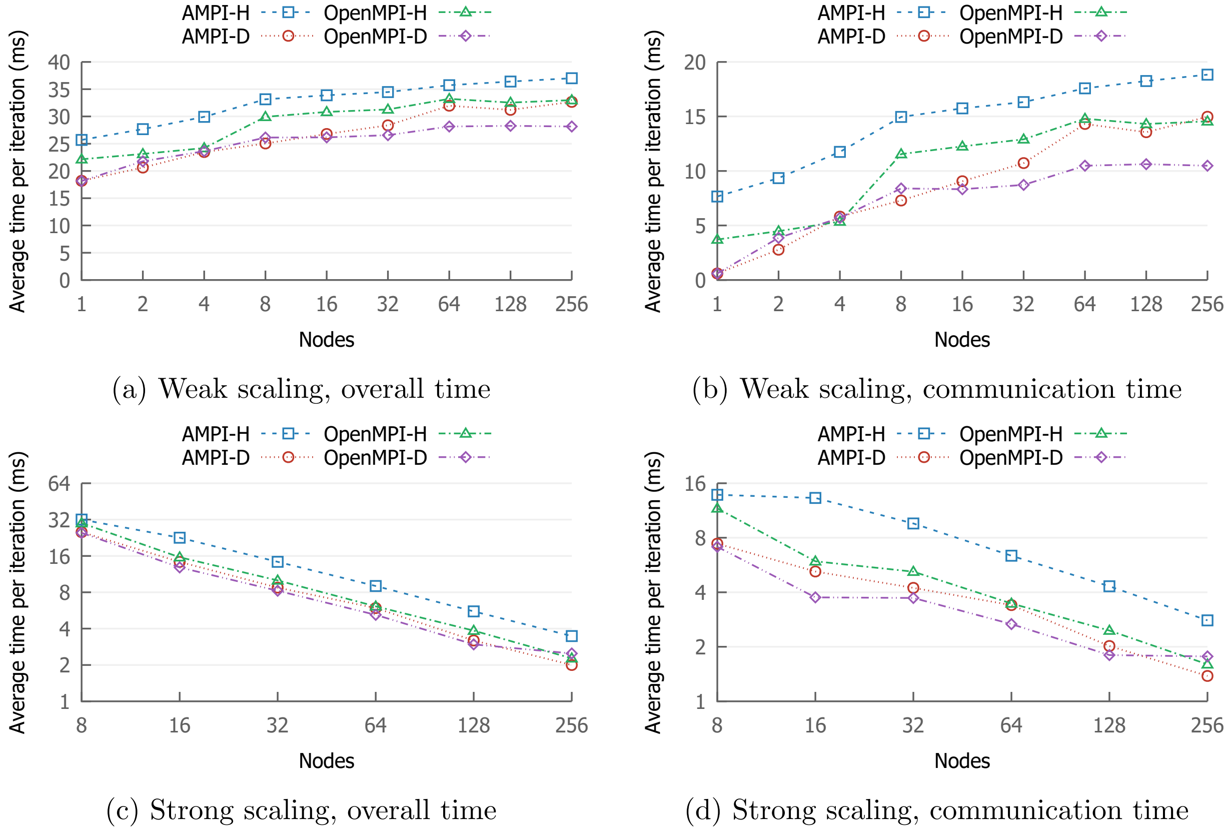
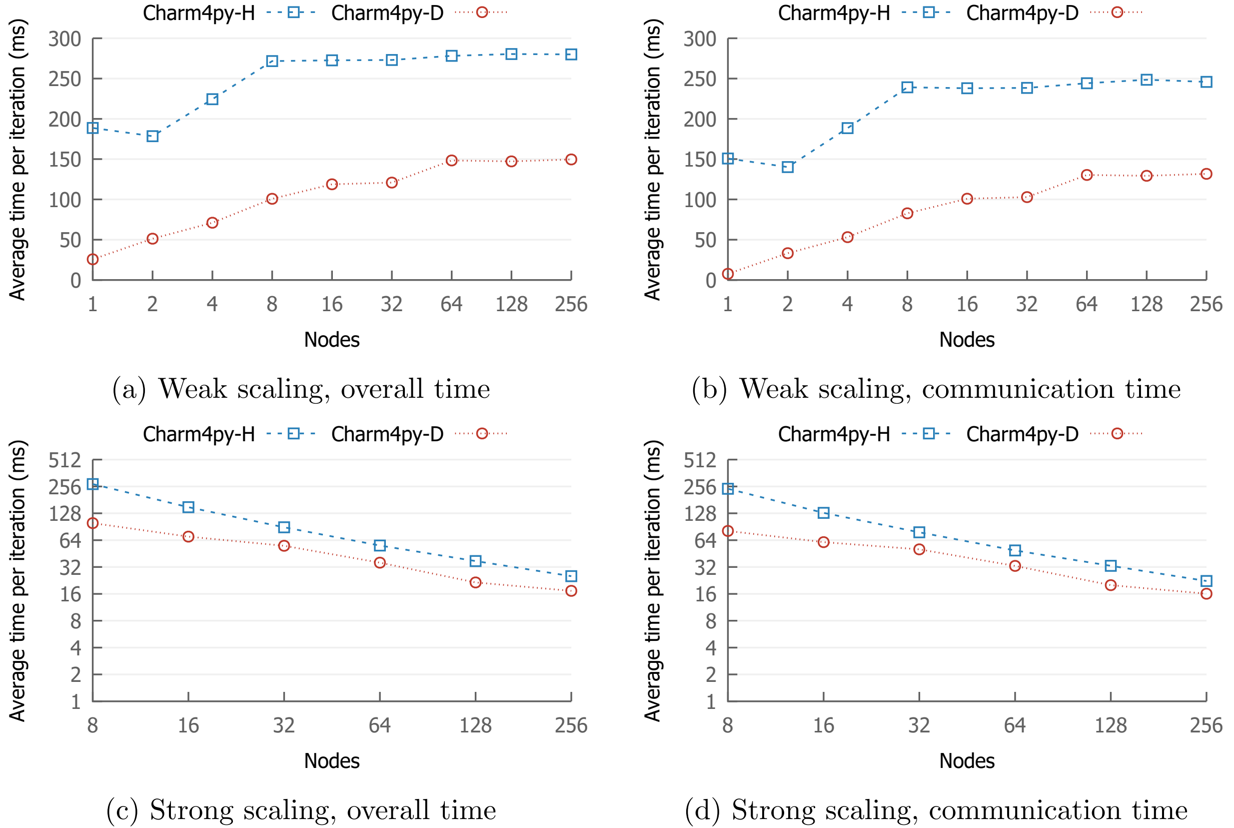


Figure 4.16: Comparison of AMPI Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.

scaled with a base domain size of $1,536^3$ double values and each dimension doubled in x, y, z order. Strong scaling experiments are performed from eight to 256 nodes while maintaining a constant domain size of $3,072^3$ doubles.

Charm++. Figure 4.15 shows the weak and strong scaling performance of the Charm++ versions of Jacobi3D. With weak scaling, the implementation with the GPU Messaging API (Messaging-D) demonstrates a speedup between 1.1x and 12.4x in communication performance, with the largest speedup obtained on a single node. This is an expected result as the improvements in latency and bandwidth are more pronounced for intra-node communication. The improved communication performance entails reductions in the overall execution time, ranging between 5% and 37%. The relative speedup obtained with GPU-aware communication decreases as the number of nodes increases, as slower inter-node communication starts to dominate intra-node communication. With strong scaling, the improvement in communication performance ranges between 12% and 82% and overall iteration time between 9% and 27%, with the largest speedup obtained on a single node.

AMPI. Figure 4.16 illustrates the weak and strong scaling performance of the AMPI



(a) Weak scaling, overall time

(b) Weak scaling, communication time

(c) Strong scaling, overall time

(d) Strong scaling, communication time

Figure 4.17: Comparison of Charm4py Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.

versions of Jacobi3D, with the performance of OpenMPI provided as reference. With weak scaling, GPU-awareness improves the communication performance by factors between 1.3x and 12.8x, accelerating the overall performance up to 41%. The GPU-aware communication performance in AMPI is similar to that of OpenMPI up to 16 nodes, but starts to fall behind at larger scales. We suspect that this is due to the additional metadata exchange performed in AMPI, whose performance impact becomes more pronounced at large node counts. With strong scaling, AMPI achieves a speedup between 1.9x and 2.6x in communication performance and an improvement in overall iteration time between 27% and 74%.

Charm4py. The weak and strong scaling performance of Charm4py are depicted in Figure 4.17. As the support for GPU-aware communication in Charm4py significantly improves performance especially for large messages as seen in Figures 4.11c and 4.12c, communication performance is improved by factors between between 1.9x and 19.7x with weak scaling. Because communication performance has a greater impact on the overall performance in Charm4py compared to other parallel programming models, we observe speedups in overall execution time between 1.9x and 7.3x. With strong scaling, the improvement in communi-

cation performance ranges between 1.4x and 3.0x, resulting in speedups between 1.5x and 2.7x in the overall iteration times.

4.4 RELATED WORK

There have been many publications on supporting GPU-aware communication in the context of parallel programming models. Works from the MVAPICH group [32, 33, 49] utilize CUDA and GPUDirect technologies to optimize inter-GPU communication in MPI. Hanford et al. [50] highlights shortcomings of current GPU communication benchmarks and shares experiences with tuning different MPI implementations. Khorassani et al. [51] evaluates the performance of various MPI implementations on GPU-accelerated OpenPOWER systems. Chen et al. [52] proposes compiler extensions to support GPU communication in the UPC programming model. This work distinguishes itself from other related studies in the discussion of designs for GPU-aware communication and their performance in a message-driven runtime system and multiple parallel programming models built on top of it, utilizing a state-of-the-art communication library, UCX.

4.5 CONCLUDING REMARKS

In this chapter, we have discussed the importance of GPU-aware communication in today's GPU-accelerated supercomputers, and the associated technologies that are involved in supporting direct GPU data transfers for several parallel programming models: Charm++, AMPI, and Charm4py. We leverage the capabilities of the UCX library to implement an extension to the UCX machine layer in the Charm++ runtime system, providing a performance-portable communication layer for the Charm++ family of parallel programming models. With the GPU Messaging API in Charm++, we are able to retain the semantics of message-driven execution while demonstrating substantial performance improvements. We also discuss the design of the Channel API in Charm++, which deviates from message-driven execution to provide data-only communication that can be useful for certain types of applications. The Channel API demonstrates superior performance to the GPU Messaging API due to its simpler design and a more direct interface to the underlying UCX library. Our GPU-aware communication mechanisms demonstrate superior performance over the host-staging methods in micro-benchmarks adapted from the OSU benchmark suite, as well as a proxy application that represents a widely used stencil algorithm.

While UCX proves to be an effective framework for universally accelerating GPU com-

munication, there is still room for performance improvement as indicated by the differences between AMPI and OpenMPI. One of the potential areas of improvement is GPU support in the Active Messages API of UCX, which could better fit the message-driven execution model of Charm++. Another is replacing the use of the GPU Messaging API for AMPI and Charm4py with the new Channel API, which would eliminate the need to delay the posting of the receive for GPU data until the arrival of the metadata message.

CHAPTER 5: IMPROVING SCALABILITY WITH GPU-AWARE ASYNCHRONOUS TASKS

The sheer degree of computational power and data parallelism provided by GPUs are enabling applications to achieve groundbreaking performance. However, due to the relatively slower improvement of network bandwidth compared to the computational capabilities of GPUs over time, communication overheads often hold applications back from achieving high compute utilization and scalability. Overlapping computation and communication is a widely used technique to mitigate this issue, but it is generally up to the application programmer to identify potential regions of overlap and implement the necessary mechanisms. This becomes increasingly difficult in applications with convoluted code structures and interleavings of computation and communication. *Automatic* computation-communication overlap can be achieved with overdecomposition and asynchronous task execution, features supported by the Charm++ runtime system and its family of parallel programming models [3], substantially improving performance and scalability on both CPU and GPU based systems [4].

However, performance gains from overdecomposition-driven overlap can degrade with finer task granularity. In weak scaling scenarios with a small base problem size or at the limits of strong scaling, fine-grained overheads associated with communication, scheduling, and management of GPU operations can outweigh the benefits from computation-communication overlap. In this work, we propose the integration of GPU-aware communication into asynchronous execution of overdecomposed tasks, to reduce communication overheads and enable higher degrees of overdecomposition at scale. In addition to improving performance and scalability, overdecomposition enables adaptive runtime features such as load balancing and fault tolerance. Asynchronous execution of overdecomposed tasks also provide the benefit of spreading out communication over time, allowing more efficient use of the network when bandwidth is limited [53].

We also demonstrate the importance of minimizing synchronizations between the host and device and increasing the concurrency of independent GPU operations, by comparing the performance of a proxy application against the implementation described in our previous work [4]. In addition to these optimizations, we explore techniques such as kernel fusion [54] and CUDA Graphs [55] to mitigate overheads related to fine-grained GPU execution, which can be exposed at the limits of strong scaling. We show how these mechanisms improve performance especially for relatively high degrees of overdecomposition, which can be useful for taking advantage of runtime adaptivity.

The major contributions of this work can be summarized as the following:

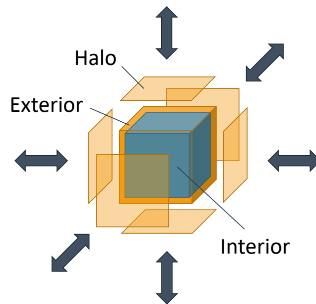
- We present the integration of overdecomposed asynchronous tasks and GPU-aware communication to exploit computation-communication overlap and reduce exposed communication overheads.
- We demonstrate the impact of our approach by evaluating the weak and strong scaling performance of a scientific proxy application on a large-scale GPU-accelerated system.
- We illustrate the importance of minimizing synchronizations between the host and device as well as ensuring concurrency of independent GPU operations.
- We explore kernel fusion and CUDA Graphs as techniques to reduce fine-grained overheads at scale and evaluate their impact on performance.

5.1 BACKGROUND

5.1.1 Automatic Computation-Communication Overlap

Overlapping computation and communication is a widely used and researched technique, which has been proven to be effective in both CPU-based and GPU-accelerated systems for hiding communication latency [4]. Non-blocking communication is one of the primary mechanisms used to expose opportunities for overlap, allowing processors to perform useful work while communication is being progressed [56]. With the Message Passing Interface (MPI), a distributed memory communication standard broadly used in HPC, it is the application programmer's responsibility to identify regions of potential overlap [57]. Not only is this often challenging due to complex code structure and flow of execution, but it also limits the amount of attainable overlap to the identified regions.

For example, let us have a look at how a three-dimensional Jacobi iterative method, hereafter called Jacobi3D, can be implemented using MPI. Each MPI process is responsible for a block of the global 3D grid, as described in Figure 5.1a. Halo data are first exchanged among the neighbors using non-blocking MPI communication routines. After all halo data are received and unpacked, each MPI process can perform the Jacobi update on its block. However, since updating only the interior of the block does not depend on the neighbors' halo data, it can overlap with the halo exchanges. Implementations with and without this manual overlap are described in Figure 5.1b. Finding such regions of potential overlap, however, can be much more challenging in larger applications. Furthermore, the execution could be blocked at synchronization points (e.g., `MPI_Waitall`) if such calls are made too early,



(a) Diagram

```
for (int iter = 0; iter < n_iters; iter++) {
  // Pack halo data for sending
  packHalos(send_halo, ...);

  // Post non-blocking receives and sends
  for (int dir = 0; dir < 6; dir++) {
      MPI_Irecv(recv_halo[dir], ..., requests[2*dir]);
      MPI_Isend(send_halo[dir], ..., requests[2*dir+1]);
  }

  if (overlap) {
    // Perform Jacobi update on the interior
    interiorUpdate();
  }

  // Wait for halo exchanges to complete
  MPI_Waitall(12, requests, statuses);

  // Unpack received halos to the 3D block
  unpackHalos(recv_halo, ...);

  if (overlap) {
    // Perform Jacobi update on the exterior
    exteriorUpdate();
  } else {
    // Perform Jacobi update on the whole block
    update();
  }
}
```

(b) Code

Figure 5.1: MPI 3D Jacobi example (Jacobi3D) with a manual overlap option. The non-blocking MPI communication can overlap with the interior Jacobi update which is independent of the halo data coming from the neighbors.

limiting the amount of attainable overlap. Periodically polling for the completion of the communication routines is an alternative, but it is not compatible with the sequential execution flow of typical MPI applications and can also unnecessary consume CPU cycles [21].

Automatic computation-communication overlap relieves the programmer from the responsibility of manually orchestrating tasks for overlap. It is achieved in the Charm++ parallel programming system [3] on the foundation of two core features: **overdecomposition** and **asynchronous task execution**. In a Charm++ program, the problem domain can be decomposed into more units of work and/or data, called *chares*, than the number of available processing elements (PEs). This is in contrast to conventional MPI applications where a single MPI process is assigned to each PE. In addition to being able to automatically overlap computation of one chare object with communication of another[1], overdecomposition empowers the runtime system to support adaptive features such as dynamic load balancing and fault tolerance. Another benefit of overdecomposition is that the injection of messages into the network can be spread out over time, alleviating pressure on the network [53].

Charm++ employs an asynchronous message-driven execution model where the arrival of a message triggers a certain task of the target chare to be executed. This message encapsulates information about which C++ method of the target chare, i.e., *entry method*, should be executed, along with the necessary data. Incoming messages are accumulated in a message queue that is continuously checked by a scheduler that runs on each PE. The execution of a Charm++ application begins with the Main Chare, which is defined by the user to play the role similar to that of the main function in regular C++. The Main Chare can can create other chare objects and initiate the flow of execution by invoking their entry methods. The invocation of a chare's entry method translates into a message transmission by the runtime system, which is by default asynchronous. This increases opportunities for computation-communication overlap by allowing the execution to continue after minimal processing when a Charm++ communication primitive is called. Once a chare entry method finishes executing, the scheduler will pick up another message from the queue to execute the next entry method.

Reducing unnecessary synchronization, between work units (chares in Charm++) as well as between the host and GPU devices, is another critical factor in exploiting computation-communication overlap. Asynchronous execution can minimize idle time and expose more opportunities for overlap by allowing each work unit to progress as freely as possible, enforcing only the necessary dependencies between tasks. Taking Jacobi3D as an example, it is not necessary to perform a global synchronization across all work units after every itera-

[1]Computation and communication of the same chare can also be overlapped, as long as they are asynchronous.

tion; in fact, each unit only needs to ensure that it is exchanging halo data from the same iteration with its neighbors. On NVIDIA GPUs, kernel launches and data transfers can be made asynchronous with the use of CUDA Streams, allowing work to be offloaded to the GPU without blocking the progress of the host CPU. However, asynchronously detecting the completion of GPU tasks requires a different mechanism especially for scheduler-driven runtime systems such as Charm++, which is discussed in Section 5.2.1.

Figure 5.2 describes the code for a Charm++ version of Jacobi3D. The Charm Interface (CI) file in Figure 5.2a is written by the user to declare components of parallel execution such as chares, entry methods, and proxies. Other codes including function bodies can be written in regular C++. The execution begins with `Main::Main` on PE 0, where an indexed collection of chares, called a *chare array*, is created. By default, Chares are distributed to all the available PEs using a block mapping; if a chare array of size eight is created on two PEs, each PE will be responsible for four consecutive chare elements. The creation of chares returns a handle to their proxy, which is used for invoking entry methods. For example, calling `block_proxy(0,0,0).run` will invoke the `run` entry method on that element of the 3D chare array. An entry method invocation on the entire proxy (e.g., `block_proxy.run`) will perform a broadcast to invoke the same entry method on all chare elements managed by that proxy.

In Charm++ Jacobi3D, the overall flow of parallel execution is encapsulated in the `Block::run` entry method. Its body is composed using Structured Dagger (SDAG) [58], which prevents the program sequence from becoming obscured by the message-driven nature of Charm++. The `serial` construct wraps regular C++ code including function calls, and the `when` construct allows the calling chare to asynchronously wait for message arrivals. Reference numbers are used in Jacobi3D to match the iteration number of an incoming message (`r` in `recvHalo`) with the block's (`iter`), to ensure that blocks progress in step with its neighbors. Control is returned back to the scheduler at the execution of the `when` construct, allowing other messages to be processed. Once an awaited message arrives, the runtime system schedules the designated entry method (e.g., `recvHalo`) to be executed.

5.1.2 GPU-Aware Communication

Without support for GPU memory from the underlying communication library, applications need explicit host-device data transfers to stage GPU buffers on host memory for communication. Not only do such host-staging methods require more code, but they also suffer from longer latency and reduction in attainable bandwidth. GPU-aware communication aims to mitigate these issues, addressing both programmer productivity and communication

```
readonly CProxy_Block block_proxy;

mainchare Main {
  entry Main(...);
};

array [3D] Block {
  entry void recvHalo(...);
  entry void run() {
    for (iter = 0; iter < n_iters; iter++) {
      // Pack and send halo data to neighbors
      serial { packHalos(); sendHalos(); }

      // Asynchronously receive halo data and unpack
      for (count = 0; count < 6; count++) {
        when recvHalo[iter](int r, double* h, int s) {
          serial { unpackHalo(); }
        }
      }

      // Perform Jacobi update on the entire block
      serial { update(); }
    }
  }
};
```

(a) .ci file

```
void Main::Main(...) {
  // Create a 3D indexed array of chares
  block_proxy = ckNew(n_x, n_y, n_z, ...);

  // Start the simulation by invoking an entry method on all block chares (broadcast)
  block_proxy.run();
}

void BlockChare::sendHalos() {
  // Send halos to neighbors by using proxy
  for (int dir = 0; dir < 6; dir++) {
    block_proxy(idx[dir]).recvHalo(iter, halo[dir], size);
  }
}

void BlockChare::packHalos() { ... }
void BlockChare::unpackHalo() { ... }
void BlockChare::update() { ... }
```

(b) .C file

Figure 5.2: Charm++ version of Jacobi3D with automatic overlap. The Charm Interface (CI) file contains user-declared components that relate to parallel execution, including chares, entry methods, and proxies.

performance.

CUDA-aware MPI implements GPU-aware communication for NVIDIA GPUs in MPI, by supporting GPU buffers as inputs to its communication API. This not only eases programming by obviating the need for explicit host-device data transfers, but also improves performance by directly moving data between the GPU and Network Interface Card (NIC). GPUDirect [37, 38] is one of the core technologies that drive GPU-aware communication, providing direct GPU memory access to the NIC.

In Charm++, there are two available mechanisms for GPU-aware communication: GPU Messaging API and Channel API. The GPU Messaging API retains the message driven execution model but requires an additional metadata message to arrive before the receiver is able to post the receive for the incoming GPU buffer. The metadata message also invokes a *post entry method* on the receiver, which is used to inform the runtime system where the destination GPU buffer is located [5]. The Channel API has been recently developed to address the performance issues with this mechanism, which uses two-sided send and receive semantics for efficient data movement [59]. It should be noted that both APIs use the Unified Communication X (UCX) library [36] as a low-level interface. In this work, the Channel API is used to drive GPU-aware communication in Charm++, with its implementation in Jacobi3D discussed in Section 5.2.2.

5.2 DESIGN AND IMPLEMENTATION

We propose the integration of GPU-aware communication in asynchronous tasks created with overdecomposition to improve application performance and scalability. In addition to a detailed discussion on combining these two mechanisms, we describe optimizations to the baseline Jacobi3D proxy application for reducing synchronization and improving concurrency of GPU operations. Furthermore, we explore techniques for fine-grained GPU tasks such as kernel fusion and CUDA Graphs to mitigate potential performance issues with strong scaling. It should be noted that although this work uses terminology from NVIDIA GPUs and CUDA, most discussions also apply to GPUs from other vendors.

5.2.1 Achieving Automatic Overlap on GPU Systems

We use Charm++ as the vehicle to achieve automatic computation-communication overlap in GPU-accelerated execution. Allowing GPU work to progress asynchronously and detecting their completion as early as possible are equally important in creating opportunities for overlap. CUDA Streams [14], which allows GPU operations to execute asynchronously

and concurrently, is the preferred method of offloading work to GPUs in Charm++ applications. A common usage of a CUDA stream involves enqueueing GPU work such as a kernel or memcpy and waiting for it to finish using a synchronization mechanism, e.g., `cudaStreamSynchronize`. Since submitting work to a CUDA stream is asynchronous, other tasks can be performed on the host CPU until the synchronization point. While this may be sufficient for traditional MPI applications where a single process runs on each PE, it can be detrimental to scheduler-driven tasking frameworks such as Charm++; synchronization can prevent the scheduler from processing other available messages and performing useful work. Asynchronous completion frees up the host CPU to perform other tasks while GPU work is being executed, facilitating overlap.

Hybrid API (HAPI) [17] enables asynchronous completion detection of GPU operations in Charm++, using CUDA events to track their status in the scheduler. It allows the user to specify which Charm++ method should be executed when the completion of the tracked GPU work is detected. Meanwhile, the scheduler can perform other useful tasks, increasing opportunities for computation-communication overlap. More implementation details of HAPI can be found in our previous work [4]. In the optimized version of Jacobi3D as described in Section 5.2.3, HAPI is used to ensure that the Jacobi update and packing kernels have been completed before sending halo data to the neighbors.

In addition to asynchronous completion detection, prioritizing communication and related GPU operations (e.g., packing and unpacking kernels) is key to exploiting overlap. Since multiple chares can utilize the same GPU concurrently due to overdecomposition, communication-related operations of one chare can be impeded by computational kernels launched by other chares unless they are given higher priority. Such delays in communication translate directly into performance degradation [4]. In Jacobi3D, host-device transfers and (un)packing kernels are enqueued into high-priority CUDA streams. The Jacobi update kernel utilizes a separate stream with lower priority. These streams are created for every chare object so that independent tasks from different chares can execute concurrently on the GPU when possible.

5.2.2 GPU-Aware Communication in Charm++

Exploiting computation-communication overlap with overdecomposition can be highly effective in weak scaling scenarios where performance improvements from overlap outweigh the overheads from finer-grained tasks. With small problem sizes or with strong scaling, however, overdecomposition can quickly reach its limits as task granularity decreases. One of the main sources of overhead with fine-grained tasks is communication, as the ratio of com-

```
/* C file */
// Create Charm++ callback to be invoked when
// a channel send or recv completes
CkCallback cb = CkCallback(CkIndex_Block::callback(), ...);

// Non-blocking sends and receives of halo data
for (int dir = 0; dir < 6; dir++) {
  channels[dir].send(send_halo[dir], size, cb);
  channels[dir].recv(recv_halo[dir], size, cb);
}

/* .ci file */
// When a Charm++ callback is invoked, check if it means
// completion of a receive and unpack if so
for (count = 0; count < 12; count++) {
  when callback() serial { if (recv) processHalo(); }
}
```

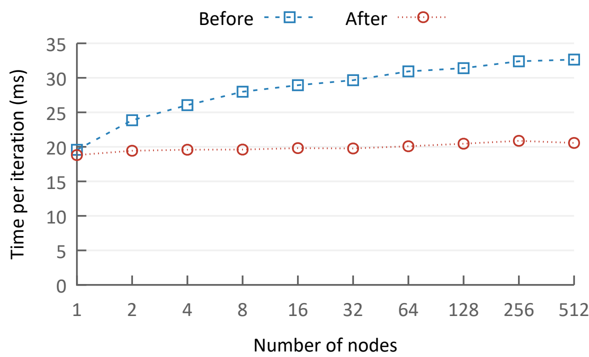
Figure 5.3: Usage of Channel API in Charm++ Jacobi3D.

putation to communication diminishes and subsequently less communication can be hidden behind computation. GPU-aware communication can mitigate such overheads by utilizing the networking hardware more efficiently.

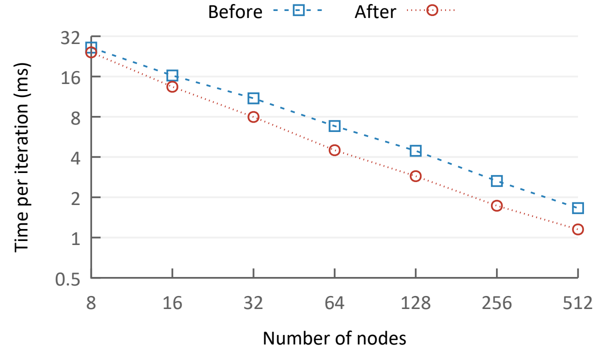
As described in Section 5.1.2, Charm++ offers two mechanisms for GPU-aware communication: GPU Messaging API and Channel API. As the communication pattern in Jacobi3D is regular, the Channel API can be easily used to exchange halo data with two-sided sends and receives. Figure 5.3 demonstrates the usage of the Channel API in Jacobi3D, where a communication channel is established between each pair of neighboring chares. Send and receive calls are made to the channel to transfer halo buffers on the GPU, which are translated into calls to the underlying UCX library. A Charm++ callback is passed to the channel primitives to invoke an entry method upon completion, enabling asynchronous completion detection and facilitating computation-communication overlap.

5.2.3 Optimizations to Baseline Performance

The original implementation of Jacobi3D [4] performed a host-device synchronization right after launching the Jacobi update kernel, to ensure that the update is complete before incrementing the iteration counter and swapping the pointers to the GPU buffers. Note that Jacobi3D maintains two separate buffers in GPU memory to be used as input and output for the Jacobi update kernel. However, this synchronization step is redundant, as the above operations to prepare for the next iteration can instead be performed just before



(a) Weak scaling, with block size of 1536 × 1536 × 1536 per node



(b) Strong scaling, with global grid size of 3072 × 3072 × 3072 (same global grid size as on 8 nodes with weak scaling)

Figure 5.4: Performance comparison of Charm++ Jacobi3D with host-staging communication before and after optimizations on the Summit supercomputer.

the halo exchanges. This optimization reduces the number of host-device synchronizations per iteration from two (after Jacobi update and before halo exchanges) to one (before halo exchanges).

By profiling the performance of Jacobi3D with NVIDIA Nsight Systems, we observe that there is another optimization opportunity to increase the concurrency of independent GPU operations. Instead of enqueueing device-host transfers and (un)packing kernels to the same stream, we create two additional high-priority streams for data transfers, one for device-to-host and another for host-to-device. This allows (un)packing kernels to overlap with the data transfers, as well as the bi-directional transfers to overlap with one another. Unfortunately, this optimization makes enforcing dependencies between the streams more complicated. Figure 5.4 showcases the improvements from the above optimizations in weak and strong scaling performance of Charm++ Jacobi3D, with host-staging communication and a four-times overdecomposition. All the following experiments use this new baseline implementation for various MPI and Charm++ versions of Jacobi3D.

5.2.4 Techniques for Fine-grained GPU Tasks

Strong scaling increases the amount of computational resources, e.g., number of GPUs, while maintaining the same problem size. Consequently, the size of work and data assigned to each resource decreases as the problem is scaled out. In GPU-accelerated environments, this causes the proportion of kernel launch overheads in execution time to grow. Applying overdecomposition, either for computation-communication overlap or runtime adaptivity

(e.g., load balancing), can exacerbate this issue. We explore techniques such as kernel fusion [54] and CUDA Graphs [55] to mitigate this problem in the context of fine-grained GPU execution.

Kernel fusion combines multiple kernels as a single kernel to reduce the aggregate kernel launch latency. CUDA Graphs is a mechanism for NVIDIA GPUs where an executable graph can be constructed from multiple consecutive GPU operations, including kernels and memory copies, to reduce launch overheads. It can also expose opportunities for optimization as all necessary dependencies are presented to the CUDA runtime. These two techniques can be used together; kernel fusion can be applied to reduce the total number of kernels, and CUDA Graphs can capture all such kernel launches and other GPU operations for more efficient repeated execution of the same graph.

Kernel fusion. With Jacobi3D, we explore three different strategies for kernel fusion, with the fused kernels outlined below:

(A) Packing kernels

(B) Packing kernels and unpacking kernels (as two separate kernels)

(C) Unpacking kernels, Jacobi update kernel, and packing kernels (all as a single kernel)

Note that packing kernels can be launched right after the Jacobi update kernel, but each unpacking kernel can only be launched after the corresponding halo data arrives from a neighbor. Thus the fused version of the unpacking kernels can only be launched after all halo data arrive. When fusing the packing/unpacking kernels, the total number of GPU threads is computed as the *maximum* of the different halo sizes. Each thread consecutively looks at the six faces that could be copied out as halo data, and if its index is smaller than the halo size, performs a copy into the respective halo buffer. We have found this implementation to be faster than having the total number of GPU threads to be the *sum* of the halo sizes, which allows all faces to be processed concurrently but suffers from excessive control divergence. Fusing all kernels using Strategy C effectively results in one kernel execution per iteration, a significant reduction in the number of kernel launches. In this work, kernel fusion is only used in concert with GPU-aware communication to avoid complications with host-device transfers and their ensuing dependencies.

CUDA Graphs. We build a CUDA graph in Jacobi3D by capturing the entire flow of kernel launches at initialization time. The graph contains all dependencies and potential concurrency of unpacking kernels, Jacobi update kernel, and packing kernels; this simplifies

each iteration of Jacobi3D to be the halo exchange phase followed by the launch of a CUDA graph. An issue that we encountered when implementing CUDA Graphs in Jacobi3D is the limitation that parameters passed to the GPU operations in a CUDA graph should not change during execution. This is problematic since the two pointers referring to input and output data need to be swapped every iteration. Although nodes in a CUDA graph can be individually updated to use a different parameter, this is infeasible in Jacobi3D since the graph needs to be updated every iteration, nullifying the performance benefits. Our solution was to create two separate CUDA graphs, one with the two pointers reversed to the other, and alternate between them for each iteration. As with kernel fusion, CUDA Graphs is only evaluated with GPU-aware communication.

5.3 PERFORMANCE EVALUATION

In this section, we evaluate the performance and scalability of our approach that incorporates computation-communication overlap with GPU-aware communication. We also explore the performance impact of kernel fusion and CUDA Graphs in strong scaling.

5.3.1 Experimental Setup

We use the Summit supercomputer at Oak Ridge National Laboratory for conducting our experiments. Summit contains 4,608 nodes each with two IBM POWER9 CPUs and six NVIDIA Tesla V100 GPUs. Each CPU has 22 physical cores with support for up to four-way simultaneous multithreading (SMT), contained in a NUMA domain with 256 GB of DDR4 memory, totaling 512 GB of host memory. Each GPU has 16 GB of HBM2 memory, with an aggregate GPU memory of 96 GB per node. Summit compute nodes are connected in a non-blocking fat tree topology with dual-rail EDR Infiniband, which has an injection bandwidth of 23 GB/s. The Bridges-2 supercomputer at Pittsburgh Supercomputing Center and Expanse at San Diego Supercomputer Center have also been used to test and debug GPU acceleration in Charm++.

The performances of the MPI versions of Jacobi3D are obtained using the default MPI and CUDA environments on Summit: IBM Spectrum MPI 10.4.0.3 and CUDA 11.0.3. The Charm++ versions of Jacobi3D use the yet-to-be-released Channel API, with UCX 1.11.1 and CUDA 11.4.2. The more recent version of CUDA used with Charm++ is not compatible with IBM Spectrum MPI, which is why an older version of CUDA is used for the MPI experiments. In our tests, we have not observed any noticeable difference in performance between the two CUDA versions.

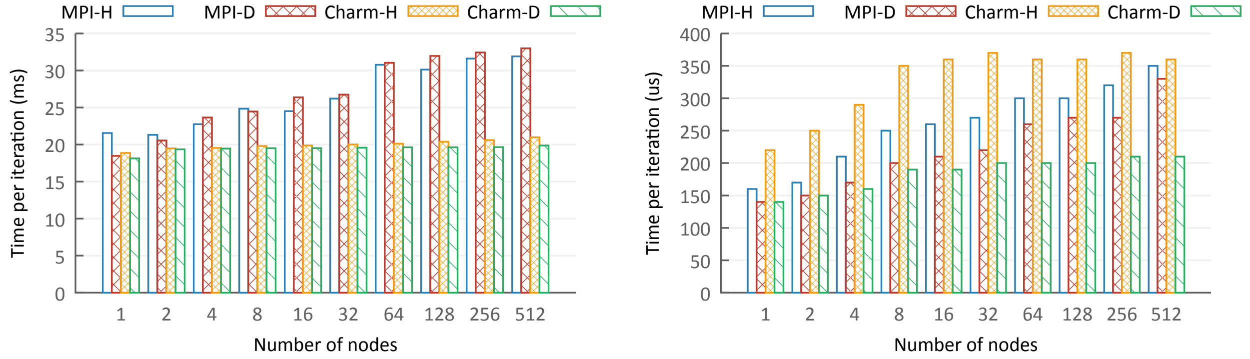
As is the norm with GPU-accelerated MPI applications, each MPI process is mapped to one CPU core and one GPU, and is responsible for a cuboid block of the global simulation grid. For example, when Jacobi3D is run on a single node (six MPI processes and GPUs), the global grid is divided into six equal-sized blocks; the grid is decomposed in a way that minimizes the aggregate surface area, which is tied to communication volume. The Charm++ experiments are also carried out using one CPU core and one GPU per process in non-SMP mode, but with an additional parameter, *Overdecomposition Factor (ODF)*, which determines the number of chares per PE and GPU. With an ODF of one, the decomposition of a Charm++ program is equivalent to MPI, where one chare object is mapped to each PE. A higher ODF creates more chares each with finer granularity, providing more opportunities for computation-communication overlap and runtime adaptivity, albeit with increased fine-grained overheads. We experiment with varying ODFs from one to 16, increased by a factor of two, to observe the impact of overdecomposition on performance.

For the following scalability experiments, we compare the performance of four different versions of Jacobi3D: MPI with host-staging communication (MPI-H), CUDA-aware MPI (MPI-D), Charm++ with host-staging communication (Charm-H) and Charm++ with GPU-aware communication using Channel API (Charm-D). The Charm++ versions of Jacobi3D are run with different ODFs and the one with the best performance is chosen as the representative for each point in scaling. Jacobi3D is run for 10 warm-up iterations and then timed for 100 iterations. Each experiment is repeated three times and averaged to obtain accurate performance results.

5.3.2 Weak Scaling

We evaluate the weak scaling performance of Jacobi3D using two different base problem sizes per node: 1,536 × 1,536 × 1,536 and 192 × 192 × 192. Each element of the grid is a double precision floating point (eight bytes). With weak scaling, the size of each dimension is increased successively by a factor of two, allowing the data size per GPU to remain approximately the same. When decomposed into six GPUs per node, the larger problem size uses roughly 9 GB of GPU memory and the smaller problem uses 18 MB, most of which is for storing two separate copies of the block data from the previous and current iterations. The size of messages being exchanged in the halo exchange phase also differs greatly, with up to 9 MB and 96 KB, respectively.

Figure 5.5a compares the weak scaling performance of the different implementations of Jacobi3D, with a base problem size of $1{,}536^3$. ODF-4 (four chares per GPU) provides the best performance out of all the tested ODFs in Charm-H, whereas ODF-2 performs the



(a) Block size of 1536 × 1536 × 1536 per node (b) Block size of 192 × 192 × 192 per node

Figure 5.5: Weak scaling performance of Jacobi3D.

best in Charm-D. These ODFs strike a good balance between computation-communication overlap and overdecomposition overheads; an excessive ODF creates too many fine-grained chares whose overheads can outweigh the benefits from overlap. Charm-D shows the best performance at a lower ODF than Charm-H, since GPU-aware communication substantially reduces communication overheads and does not require higher degrees of overdecomposition for more aggressive overlap. Charm-D outperforms Charm-H only by up to 5% since the automatic computation-communication overlap employed in Charm-H is able to hide most of the communication overheads. Nevertheless, combining automatic overlap and GPU-aware communication provides a performance improvement of 61% on 512 nodes, compared to the performance without overdecomposition and with host-staging communication.

An interesting observation in Figure 5.5a is that GPU-aware communication in IBM Spectrum MPI (MPI-D) does not improve performance starting from four nodes. By profiling the runs with NVIDIA Nsight Systems, we find that the large message sizes (up to 9 MB) in the halo exchanges cause a protocol change in the underlying communication framework. For such large messages, a pipelined host-staging mechanism that splits each message into smaller chunks is used, rather than GPUDirect [50]. Conversely, this behavior does not appear in UCX-based Charm++ and GPUDirect is always used regardless of the message size. With Charm++, we observe a more gradual, almost flat incline in execution time compared to MPI, owing to computation-communication overlap providing higher tolerance to increasing communication overheads at scale.

For a smaller base problem size of 192 × 192 × 192 (halo size of up to 96 KB), GPU-aware communication provides substantial improvements in performance in both MPI and Charm++ as demonstrated in Figure 5.5b. However, because of the much smaller task granularity, overheads from the Charm++ runtime system including scheduling chares, lo-

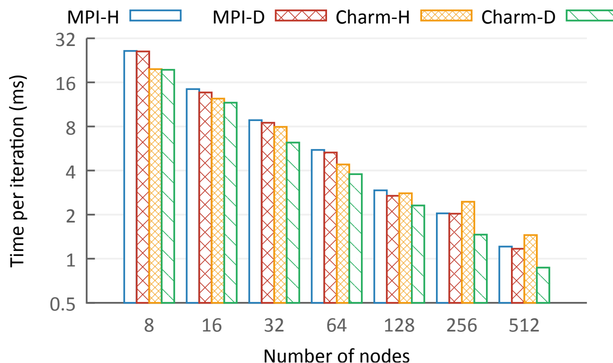
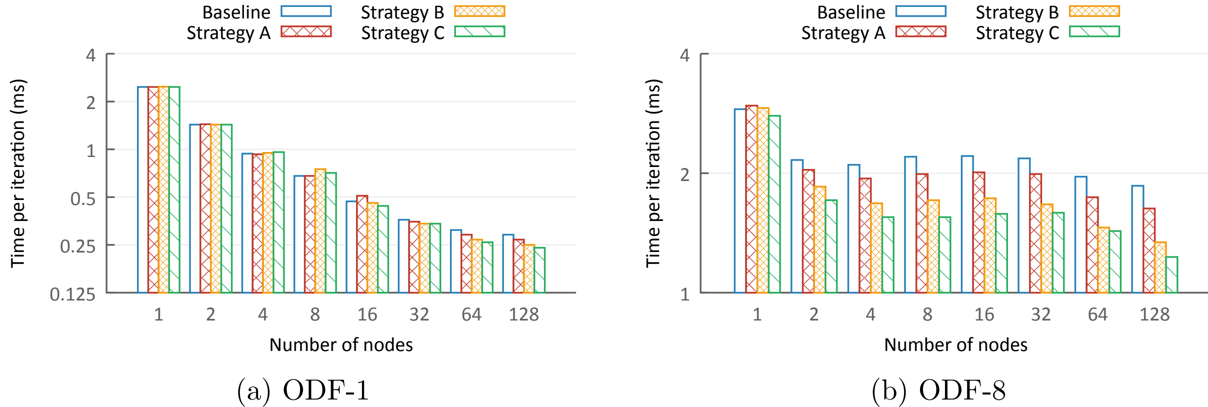


Figure 5.6: Strong scaling performance of Jacobi3D, with global grid size of 3072 × 3072 × 3072 (same global grid size as on eight nodes in Figure 5.5a).

cation management, and packing/unpacking messages become more pronounced. Moreover, overdecomposition only degrades performance, as the potential benefits from overlap pale in comparison to the overheads of finer decomposition; ODF-1 (no overdecomposition) performs the best in both Charm-H and Charm-D. The performance of CUDA-aware Spectrum MPI (MPI-D) becomes unstable on 64 or more nodes, with the time per iteration varying between 300 us and 800 us from run to run. There seems to be a problem with the MPI library as we have been able to reproduce this issue multiple times.

5.3.3 Strong Scaling

For strong scaling, we experiment with a fixed global grid of size 3,072 × 3,072 × 3,072. As we scale out and the number of nodes is doubled, the size of each work unit decreases by a factor of two. With Charm++, this means that the best overdecomposition factor will likely become smaller, as the overheads from high degrees of overdecomposition grow. Figure 5.6 illustrates the strong scaling performance of the different versions of Jacobi3D. The best ODF of Charm-H remains at four until 16 nodes, after which ODF-2 starts to outperform until 512 nodes, where ODF-1 performs the best. For Charm-D, ODF-2 provides the best performance at all scales, demonstrating that the reduction in communication overheads from GPU-aware communication enables a higher degree of overdecomposition to retain its effectiveness. On 512 nodes, ODF-2 in Charm-H is 13% slower than ODF-1, whereas ODF-2 in Charm-D is 13% faster than ODF-1. The performance issue observed with pipelined host-staging communication in MPI with weak scaling becomes less relevant with strong scaling, as GPUDirect is used instead at larger scales with the smaller halo messages. Charm-H, with host-staging communication, outperforms both MPI-H and MPI-D implementations until 128 nodes thanks to overdecomposition-driven overlap. Charm-D,



(a) ODF-1

(b) ODF-8

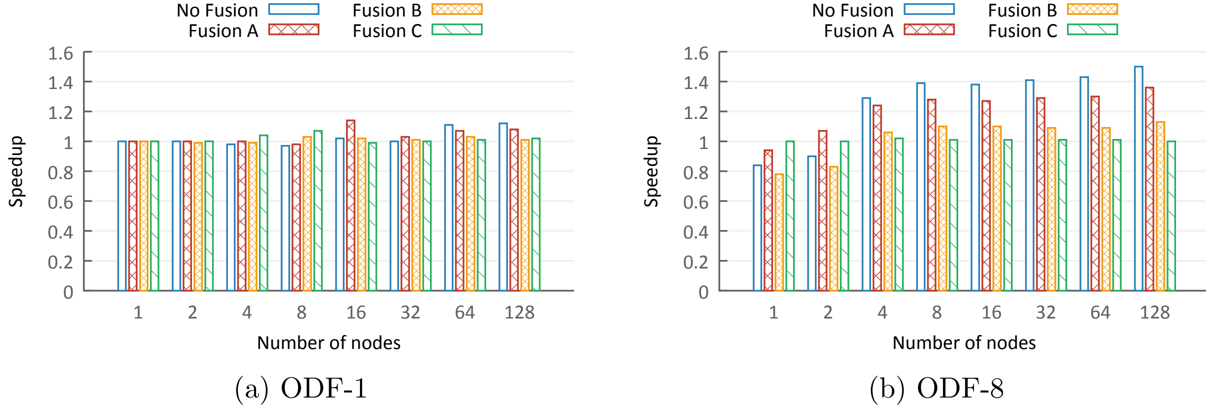
Figure 5.7: Impact of kernel fusion on the strong scaling performance of the Charm++ version of Jacobi3D with GPU-aware communication.

combining automatic computation-communication overlap and GPU-aware communication, substantially outperforms all other versions of Jacobi3D and scales out further, achieving a sub-millisecond average time per iteration on 512 nodes (3,072 GPUs).

We also evaluate the performance impact of kernel fusion and CUDA Graphs, which are techniques that can be used to counter fine-grained overheads in strong scaling[2]. The Charm++ version of Jacobi3D with GPU-aware communication (Charm-D in previous plots) is used as the baseline for this experiment, with a relatively small simulation grid of $768 \times 768 \times 768$ scaled out to 128 nodes. In this case, overdecomposition does not improve performance; nevertheless, we present results both without overdecomposition (ODF-1) and with a high degree of overdecomposition (ODF-8), to consider scenarios where overdecomposition can be used for other adaptive runtime features such as dynamic load balancing rather than for performance.

Kernel fusion. Figure 5.7 illustrates the effectiveness of the kernel fusion strategies described in Section 5.2.4 on strong scaling performance. The baseline results do not employ any type of kernel fusion, and fusion strategies from A to C become increasingly aggressive (fusing more types of kernels). Without overdecomposition (ODF-1), kernel fusion does not noticeably affect performance until 32 nodes. At larger scales, however, more aggressive fusion strategies (C > B > A) improve performance more than the others; Strategy C improves the average time per iteration by 20% on 128 nodes. This demonstrates that kernel fusion is indeed effective at mitigating kernel launch overheads, especially with smaller task granularity at the limits of strong scaling. Greater performance effects from kernel fusion

[2]These techniques can also be helpful in weak scaling with a small base problem size, but we focus on their effects on strong scaling in this work.



(a) ODF-1

(b) ODF-8

Figure 5.8: Speedup from using CUDA Graphs in addition to kernel fusion with GPU-aware Charm++ Jacobi3D.

can be observed with ODF-8, where the already fine-grained work units are further split up with an eight-fold overdecomposition. Fusion strategy C provides up to 51% increase in the overall performance on 128 nodes.

Although higher degrees of overdecomposition can degrade performance with small problem sizes, they may be needed to enable adaptive runtime features such as load balancing and fault tolerance. As such, kernel fusion can be a useful technique for reducing kernel launch overheads to improve strong scaling performance especially with overdecomposition.

CUDA Graphs. Figure 5.8 shows the obtained speedup from using CUDA Graphs, with and without kernel fusion. Without overdecomposition (ODF-1), CUDA Graphs has little impact on the overall performance, with small improvements at larger scales and less aggressive fusion strategies. Such moderate performance improvement when compared to other studies [60] stems from the low CPU utilization in Jacobi3D, where the CPU resources are mostly used only by the Charm++ runtime system. With bulk of the computation offloaded to the GPU, CPUs largely sit idle waiting for GPU work to complete, aside from scheduling chares for execution and managing communication. This causes the reduction in aggregate kernel launch latency from the use of CUDA Graphs to have less impact on the performance of Jacobi3D, when compared to workloads such as deep learning in PyTorch [60] that heavily utilize CPU resources in addition to GPUs.

However, performance improvements are more apparent with ODF-8, where we obtain a speedup of 1.5x on 128 nodes without kernel fusion. This is because CPU utilization rises substantially in accordance with the increase in overdecomposition factor. More fine-grained tasks are created, resulting in more kernel launches and GPU operations that utilize the host CPU. Conversely, the performance impact of CUDA Graphs diminishes as a more aggressive

kernel fusion strategy is used, even with ODF-8. With a higher degree of kernel fusion, the total number of kernels decreases, leaving less room for improvement in the aggregate kernel launch latency. In summary, CUDA Graphs has the potential to provide substantial performance improvements especially for workloads with high CPU utilization and when there are a sufficient number of kernel launches to optimize.

5.4 RELATED WORK

Task-based programming models such as Legion [61] and HPX [28] facilitate automatic computation-communication overlap by extracting parallelism at the level of the runtime system. Castillo et al. [21] discusses the disparity between asynchronous task-based programming models and the underlying messaging layer (MPI) that limits achievable overlap. A study by Danalis et al. [62] applies transformations to the application code to expose more opportunities for overlap. As for GPU-aware communication, many works have discussed the necessary implementations and improvements in performance [32, 33, 49]. This work distinguishes itself from others by illustrating the gains in performance and scalability from combining GPU-aware communication with automatic computation-communication overlap, enabled with overdecomposition.

5.5 CONCLUDING REMARKS

In this work, we explored how automatic computation-communication overlap from overdecomposition and asynchronous execution can be used together with GPU-aware communication to improve performance and scalability on modern GPU-accelerated systems. Using implementations in MPI and Charm++ of a scientific proxy application, Jacobi3D, we evaluated the impact of our approach on both weak and strong scaling performance with various problem sizes. We observed that the Charm++ version of Jacobi3D with overdecomposition-driven overlap and GPU-aware communication is able to achieve the best performance with strong scaling, achieving a sub-millisecond time per iteration on 512 nodes of the Summit supercomputer. With weak scaling, however, we see that the performance impact of combining overdecomposition and GPU-aware communication varies depending on the problem size.

In addition to demonstrating the importance of minimizing host-device synchronizations and increasing concurrency in GPU operations, we evaluated the usage of kernel fusion and CUDA Graphs to mitigate fine-grained execution in strong scaling scenarios. With the most

aggressive kernel fusion strategy, we achieved up to 20% improvement in overall performance with ODF-1 and 51% with ODF-8. CUDA Graphs enabled performance improvements of up to 50% when used without kernel fusion, demonstrating its effectiveness for workloads with high CPU utilization and a large number of kernel launches.

CHAPTER 6: TOWARDS A HETEROGENEOUS MESSAGE-DRIVEN PARALLEL PROGRAMMING SYSTEM

Although computing platforms have been rapidly evolving toward heterogeneous and GPU-centric architectures, such as Frontier [64] at Oak Ridge Leadership Computing Facility (OLCF) illustrated in Figure 6.1, the predominant approach to building applications for such systems has remained a mixture of MPI and a GPU programming model such as CUDA for over a decade, with MPI responsible for distributed-memory communication and CUDA for managing GPU-accelerated workloads. There are good reasons as to why this has been the case, including software availability, incremental adoption of GPUs by existing MPI applications, reliable performance, and continued support from vendors and organizations. Nevertheless, there have been many studies on the limitations of the current status quo of parallel programming, pertinent to programmer productivity and performance, including inefficient interactions between MPI and asynchronous task-based programming models [21, 65], and impedance mismatch between MPI and GPU programming models regarding asynchronous execution [66]. Due to the difficulty of composing MPI communication primitives with asynchronous GPU execution, significant burden is being placed on the programmer to ensure correctness and extract performance on heterogeneous distributed-memory systems with complex memory hierarchies and data movement requirements. It is becoming increasingly challenging for applications to fully realize the performance potential of the available hardware because of the lack of efficient, asynchronous mechanisms to facilitate interactions between the communication framework and GPU execution model. All in all, there is increasing demand for a coherent parallel programming framework that can oversee the heterogeneous and multi-node execution, allowing both users and runtime systems to exercise more fine-grained and adaptive control of the communication stack and computational resources beyond what is currently possible with MPI and a vendor-provided GPU runtime such as CUDA.

A number of works aimed to overcome the limitations of the MPI + X model, either by driving the parallel and distributed-memory execution from inside the GPU (dCUDA [67], Juggler [68]), developing new programming models with runtime-automated data movement (Legion [30], Chapel [69]), or remodeling the communication framework to be compliant with the asynchronous nature of GPU execution (NCCL [70], NVSHMEM [71]). In this chapter, we present CharminG (short for Charm++ in GPUs), a GPU-centric runtime system and parallel programming model with active messages, which takes inspiration from the aforementioned works and the Charm++ parallel programming system to explore the potential benefits as well as the present limitations of GPU-driven parallel execution. Although

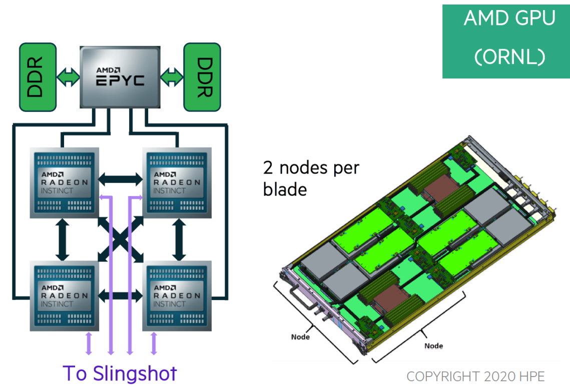


Figure 6.1: Node diagram of OLCF Frontier [63].

this chapter focuses solely on a GPU-resident runtime, CharminG aims to be a stepping stone towards an overarching heterogeneous parallel programming system that manages all available computing resources including CPUs and GPUs. With an object-oriented programming model and asynchronous message-driven execution, CharminG seeks to realize the core principles of the Charm++ parallel programming system on the GPU, including overdecomposition, computaiton-communication overlap, and runtime adaptivity with more control over how tasks utilize GPU resources and engage in data movement. It explores the possibilities of a GPU-resident runtime system with the latest advances in GPU-accelerated platform architecture and related software, with GPU-driven scheduling and communication mechanisms that minimizes interactions and synchronizations with the host CPU. CharminG is not expected to outperform the current host-driven approaches such as MPI + CUDA, as its primary goal is to identify the limitations of the current hardware and software for building a GPU-centric parallel programming system and assist the development of future systems, but it does strive to extract as much performance as possible from the available resources with optimizations across the runtime software stack. To the best of our knowledge, CharminG is the first fully GPU-resident and scalable implementation of a message-driven parallel programming model.

6.1 DESIGN OVERVIEW

6.1.1 Design Principles

Before delving into the implementation details, we first define the core design principles that guide the development of the CharminG parallel programming system as follows.

- Develop a GPU-centric runtime system programmable with an asynchronous message-

driven execution model and explore its potential benefits such as overdecomposition and automatic computation-communication overlap.

- Explore the feasibility of granting autonomy to GPUs with GPU-resident scheduling and messaging mechanisms, limiting explicit interactions with the host to setup and teardown.
- Aim to extract as much performance as possible from the currently available hardware and software functionalities, without expecting superior performance compared to existing heterogeneous parallel programming approaches (e.g., MPI + CUDA, NVSHMEM).
- Identify the limitations and challenges of implementing an asynchronous task-based runtime system on the GPU, and present potential improvements for co-design of future heterogeneous platforms and parallel programming systems.

6.1.2 Object-Oriented, Asynchronous Message-Driven Execution on the GPU

Inspired by the Charm++ parallel programming system described in Section 2.1, CharminG aims to realize an object-oriented, asynchronous message-driven execution model on the GPU. Similarly to Charm++, a CharminG program is constructed using chare objects that encapsulate computational tasks and data, which communicate with one another via asynchronous entry method invocations. In CharminG, however, chare objects reside within the GPU devices with their data stored in GPU memory, and the execution of their entry methods is coordinated by GPU-resident schedulers instead of the host CPU. Entry method invocations are translated into message sends using GPU-initiated communication, supported by messaging mechanisms implemented using CUDA atomics on global memory and NVSHMEM [71] operations for intra-GPU and inter-GPU communication, respectively.

To first give the readers an idea of what a GPU-resident parallel program composed with CharminG resembles, we first explore a simple hello world program where messages are sent from one chare to another in a chain-like fashion. Figure 6.2 shows the code of the reference Charm++ version of the program that uses a 1D chare array. To demonstrate overdecomposition, twice as many chares as the number of processing elements (PEs) are created as an indexed collection of chares, called a *chare array*. The `Main` function of the mainchare on PE 0 is executed first, which creates the 1D chare array and begins by sending a zero to the first chare element using the array proxy. Once each chare receives a number,

```
// Charm++ Interface (CI) file: hello.ci
mainmodule hello {
  readonly CProxy_Hello hello_proxy; // Chare array proxy
  readonly int nelems; // Number of chares in array

  mainchare Main {
    entry Main(CkArgMsg *m);
  };

  // Chare array and its entry methods
  array [1D] Hello {
    entry Hello();
    entry void greet(int num);
  };
};
```

```
// Source file: hello.C
#include "hello.decl.h"

/* readonly */ CProxy_Hello hello_proxy; // Chare proxy
/* readonly */ int nelems;

// Main function of mainchare executed on PE 0
class Main : public CBase_Main {
public:
  Main(CkArgMsg *m) {
    // Create hello chare array with twice as many chares
    // as the number of PEs
    nelems = CkNumPes() * 2;
    hello_proxy = CProxy_Hello::ckNew(nelems);

    // Start by sending 0 to the first chare
    hello_proxy[0].greet(0);
  }
}

class Hello : public CBase_Hello {
public:
  Hello() {}

  void greet(int num) {
    // Terminate if this chare is the last, otherwise send (received + 1) to next
    if (thisIndex == nelems-1) {
      CkExit();
    } else {
      hello_proxy[thisIndex+1].greet(num+1);
    }
  }
}
```

Figure 6.2: Source code of a Charm++ hello world program.

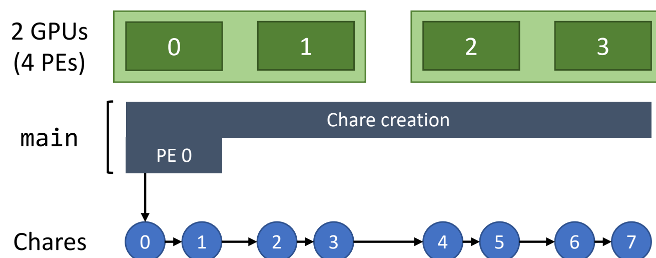


Figure 6.3: Execution flow of a CharminG hello world program with SM-level scheduling on a total of four PEs (two SMs per GPU utilized as PEs). An overdecomposition factor of two is applied, resulting in eight chares spread across the four PEs. Black arrows depict the flow of messages.

```
// Header file: hello.h
#include <charming.h> // Main CharminG header file

// Declaration of chare object
struct Hello : charm::chare {
  __device__ Hello() {}
  __device__ void greet(void* arg);
};

// Wrapper function for entry method, needed to prevent linker warnings
__device__ void entry_greet(Hello& c, void* arg) { c.greet(arg); }
```

Figure 6.4: Header file of a CharminG hello world program.

the number is incremented by one and sent to the next chare until the last chare is reached, at which point the program terminates by calling `CkExit`.

Figure 6.3 illustrates the overall flow of the CharminG version of the program using SM-level scheduling, with the code provided in Figures 6.4 and 6.5. With the SM-level scheduling mode of CharminG, each SM has its own separate multi-threaded scheduler (run with a single thread block) and acts as a PE; in this example each GPU is assumed to have two SMs (PEs), with a total of four PEs on two GPU devices. A total of eight chares are created on the four PEs, also employing a 2x overdecomposition. Comparing the Charm++ and CharminG versions of the hello world program, CharminG appears more complicated mainly because of the branch statements needed to switch between single-threaded and multi-threaded execution, which is a byproduct of the user functions being executed by a thread block instead of a single thread as in Charm++. On the other hand, with the GPU-level scheduling mode of CharminG, the whole GPU is treated as a single PE, allowing the

```
1  // Source file: hello.cu
2  #include "hello.h"
3
4  // Proxy for chare management and entry method invocations
5  __shared__ charm::chare_proxy<Hello>* hello_proxy;
6
7  // CharminG main function is executed on PEs and CEs
8  __device__ void charm::main(int argc, char** argv, size_t* argvs, int pe) {
9    // Registration of chares and their entry methods, must be done on all PEs
10   if (threadIdx.x == 0) {
11     hello_proxy = new charm::chare_proxy<Hello>();
12     hello_proxy->add_entry_method<&entry_greet>(); // Will have 0 as its index
13     hello_proxy->create(charm::n_pes() * 2); // 2x overdecomposition
14   }
15   __syncthreads();
16   barrier(); // Ensure all PEs have completed chare creation
17
18   // The following should be executed only on PE 0
19   if (pe == 0) {
20     int* sent_int_p = nullptr;
21     if (threadIdx.x == 0) {
22       send_int_p = new int;
23       *send_int_p = 0;
24     }
25     __syncthreads();
26
27     // Start by sending 0 to the first chare
28     // Syntax: invoke(dst chare index, entry method index, src buffer, size)
29     hello_proxy->invoke(0, 0, send_int_p, sizeof(int));
30   }
31 }
32
33 // Entry method
34 __device__ void Hello::greet(void* arg) {
35   int recv_int = *(int*)arg;
36   int* send_int_p;
37
38   // Terminate if this chare is the last, otherwise send (received + 1) to next
39   if (charm::chare::i == charm::chare::n-1) {
40     charm::end();
41   } else {
42     if (threadIdx.x == 0) {
43       send_int_p = new int;
44       *send_int_p = recv_int + 1;
45     }
46     __syncthreads();
47
48     hello_proxy->invoke(i+1, 0, send_int_p, sizeof(int));
49   }
50 }
```

Figure 6.5: Source code of a CharminG hello world program, with SM-level scheduling.

scheduler and functions of the user's chare objects to utilize the entirety of the GPU for data-parallel execution. The scheduling mode of CharminG, SM-level or GPU-level, can be selected at compile time according to the characteristics of the application. As a general rule of thumb, applications with relatively larger task granularity are more suited to GPU-level scheduling as each work unit (chare object) can utilize the data parallelism of the entire GPU, whereas SM-level scheduling may be more appropriate for programs with finer granularity and those that would benefit more from adaptive runtime support (e.g., dynamic load balancing). The implementation details of the scheduling modes are discussed in more detail in Sections 6.2.2 and 6.2.3.

Coming back to the CharminG hello world program with SM-level scheduling, a global synchronization follows the creation of chares in the main function to ensure that all PEs have observed it. PE 0 then sends a message containing the number zero to Chare 0 to initiate the chain of messages. When each chare receives a message, the number packed inside the message is incremented by one and sent to the next chare. This process continues until the last chare. Entry method invocations using the `invoke` function of the chare proxy, as in line 48 of `hello.cu` in Figure 6.5, require parameters such as the index of the destination chare object, the index of the target entry method (starting from zero in the order of registration in `main`), as well as the address and size of the source buffer to be sent. Such entry method invocations are asynchronous, meaning that they only initiate the communication and do not wait (block) for completion. This allows the runtime to continue other independent computational work of the current chare or switch to a different chare (possible with overdecomposition), providing the opportunity to automatically exploit computation-communication overlap.

Note that with SM-level scheduling, all user functions (e.g., `Hello::greet`) are executed in parallel by all the threads of a single thread block, requiring the user to switch between single-threaded and parallel executions for different parts of the code. Heap allocation is an example that needs to be performed with a single thread, whereas entry method invocations (e.g., `hello_proxy->invoke`) should be done in parallel to allow the CharminG runtime to leverage multiple threads for its operations. The `main` function is one exception that is simultaneously executed on all SMs instead of one PE, to ensure the creation of chares and registration of their entry methods before the start of the parallel execution.

6.2 IMPLEMENTATION DETAILS

6.2.1 Building Blocks: Chares, Entry Methods, Chare Proxies

```cpp
struct chare {
  int i; // Index in 1D chare array
  int n; // Total number of chares
  __device__ chare() : i(-1), n(0) {}
}
```

Figure 6.6: Chare data structure in CharminG. User chare types should inherit from this.

CharminG is provided to the user as a static C++ library that needs to be included at link time, in addition to other components such as CUDA and NVSHMEM. To use the functionalities of CharminG, the `charming.h` header file should be included in the user application as seen in line 2 of `hello.h` in Figure 6.4. This header file provides the `charm` namespace under which components such as chares (`charm::chare`), chare proxies (`charm::chare_proxy`), and helper functions are defined.

As CharminG currently only supports 1D chare arrays, the `chare` data structure is defined as a simple struct that stores the chare index and the total number of chares in the chare array as shown in Figure 6.6. These values are accessible by the user via `charm::chare::i` and `charm::chare::n`. The user can define their own chare types by inheriting from `charm::chare`, as in line 5 of `hello.h`. Because chares are executed inside the GPU device, their member functions must be decorated with `__device__`. The same applies to any other function that may be called by a chare's member function.

Certain member functions of a chare object are designated as *entry methods*, which are special functions that may be invoked by chares potentially on other PEs, GPU devices or even physical nodes. The invocation of an entry method involves the construction of a message inside the CharminG runtime system that is sent to the PE where the target chare resides, which is picked up by the scheduler running on the destination PE for the execution of the designated entry method. This essentially transfers the flow of execution from the source chare to the destination chare, referred as message-driven execution based on the active messages model [9]. An example of an entry method is `Hello::greet` declared in line 7 of `hello.h`, whose body starts from line 36 of `hello.cu`. As can be seen from the function prototype, an entry method is currently required to take a `void` pointer as its sole argument and have a return type of `void`, to allow the runtime system to keep track of the registered entry methods. This restriction may be lifted in the future with a more aggressive use of C++ templates. The user also needs to define a *wrapper entry method* for each entry method as in line 11 of `hello.h`, to provide a static point of entry to the actual entry method and prevent linker warnings. Since entry methods are not static functions and are instead

```
1  // Base class for storing different types of entry methods
2  template <class C>
3  struct entry_method_base {
4    __device__ virtual void operator()(C& chare, void* arg) = 0;
5  }
6
7  template <class C, void Func(C&, void*)>
8  struct entry_method : entry_method_base<C> {
9    __device__ virtual void operator()(C& chare, void* arg) { Func(chare, arg); }
10 }
```

Figure 6.7: Entry method as a CUDA C++ functor in CharminG. C++ templates are used to cater to different types of chares and their entry methods.

member functions of chare objects created at runtime, using function pointers[1] to register them to the runtime system prevents the linker from determining the stack size of GPU kernel. Entry methods are stored as C++ functors within the CharminG runtime system, using class inheritance and virtual functions to support different types of entry methods in the user application. This is illustrated in Figure 6.7. There is a common base C++ functor class for all entry methods for each chare type, `entry_method_base`, from which the individual entry methods inherit as `entry_method` and take the corresponding wrapper entry method (e.g., `entry_greet`) as an additional template parameter. This allows the runtime system to store the entry methods in an array inside the chare proxy, and the combination of C++ functors and virutal functions allows a pointer to `entry_method_base` to be used to execute the derived `entry_method`, which in turn executes the wrapper entry method and subsequently the corresponding entry method.

The creation of a chare array as well as the registration and invocation of its entry methods are done through a *chare proxy*, which is a locally available data structure that contains information about the corresponding chare array. Like entry methods, chare proxies are also templated on the chare type to support different types of chares. An example is shown in lines 10-17 of `hello.cu` with `hello_proxy`, and the `chare_proxy` data structure is depicted in Figure 6.8.

In the SM-level scheduling mode, chare proxies are stored in shared memory for fast access by the PEs (SMs), whereas they would be stored in global memory with the GPU-level scheduling mechanism. In the `main` function, entry methods can be registered with the `add_entry_method` function once memory for the chare proxy is allocated, with the wrapper entry method (e.g., `entry_greet`) passed as the template parameter. The order of registration determines the indices for the entry methods, starting with zero for the first

[1]A recent addition to CUDA, `nvstd::function`, has made this possible within device code.

```
// Base class for storing chare proxies of different chare arrays
struct chare_proxy_base {
  int id; // ID of corresponding chare array

  __device__ chare_proxy_base() {}
  __device__ virtual bool call(int idx, int ep, void* arg, int ref) = 0;
}

template <class C>
struct chare_proxy : chare_proxy_base {
  C** objects; // Chare objects on this PE (local chares)
  entry_method_base<C>** entry_methods; // Registered entry methods
  ...

  // Register an entry method (wrapper function passed as template parameter)
  template <void Func(C&, void*)>
  __device__ void add_entry_method() { ... }

  // Creation of chares: provide total number of chares and an optional block mapping
  __device__ void create(int n_chares, int* block_map) { ... }

  // Entry method invocation: calls into the runtime system to send a message
  // to the PE where the target chare (with the specified index) resides
  inline __device__ void invoke(int idx, int ep, void* buf, size_t size) { ... }

  // Execute the target entry method on the specified local chare object
  // (Called internally by the runtime system when processing a message)
  __device__ virtual bool call(int idx, int ep, void* arg, int ref) {
    ...
    (*(entry_methods[ep]))(*(objects[local_idx]), arg);
  }
}
```

Figure 6.8: Chare proxy in CharminG. Registration of entry methods and chare creation is done through chare proxies, as well as entry method invocation and execution on a local chare object.

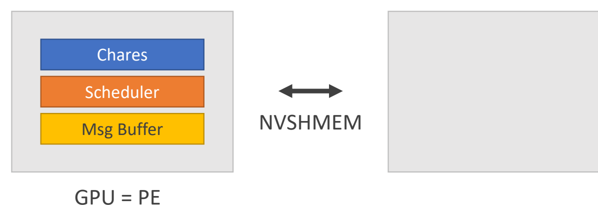


Figure 6.9: GPU-level scheduling in CharminG. A single scheduler is run on each GPU device, with messages exchanged between PEs (GPUs) via NVSHMEM.

entry method. The creation of chares is also performed through a chare proxy via the `create` member function, which requires the total number of chares to be created and an optional block mapping that specifies the number of chares to be created on each PE. Using this information, each PE locally creates its portion of chares and stores them in the `objects` array. Without a specified mapping, chares are mapped to PEs in a block fashion with (almost) equal number of chares per PE. For example, with 16 chares and 4 PEs, chares 0-3 are mapped to PE 0, chares 4-7 to PE 1, and so on. Note that in the current implementation, the creation of a chare proxy, registration of entry methods and creation of chares must be called on all PEs in the `main` function before chares can be used for parallel execution. Once the entry method registration and chare creation process is complete, chare proxies can be used to invoke an entry method of a target chare using the `invoke` function, as shown in line 48 of `hello.cu` of Figure 6.5. The most generic form of the `invoke` function takes the target chare index, entry method index, and the address and size of the source buffer as parameters. This is translated into a message send in the underlying CharminG runtime system, which will be discussed in more detail in the following sections. It is worth noting here that the `call` function of chare proxies is used by the scheduler to execute the target entry method on the target chare (one of its local chares), once such a message arrives.

6.2.2 GPU-Resident Scheduling

Streaming multiprocessors (SMs) available on a GPU can be utilized in different ways to schedule chare objects for execution. The simplest and most coarse-grained method, *GPU-level scheduling*, is to regard a GPU device as a processing element (PE), so that each chare can utilize the entire GPU for its computation. This is illustrated in Figure 6.9. As only one chare should be executed at a time on each PE, a single scheduler instance per GPU is sufficient to drive the parallel execution. At the opposite end of the spectrum is *SM-level scheduling* where each SM can be utilized as a PE, resulting in multiple PEs per GPU device as depicted in Figure 6.10. This would however limit the compute resources available to a chare object to a single thread block on a SM but allow multiple chares to execute concurrently on each device, making it more suitable for fine-grained applications. Grouping a subset of SMs as a PE is also theoretically possible, but only with the most recently released GPU hardware as of this writing, the NVIDIA Hopper architecture [72]. Hopper supports *thread block clusters*, which allows multiple thread blocks on different SMs to synchronize and communicate with one another, an essential feature in implementing a scheduler that spans across a subset of SMs on a GPU. Distributed shared memory (DSMEM), also part of the improvements in the Hopper architecture, allows more efficient data movement between

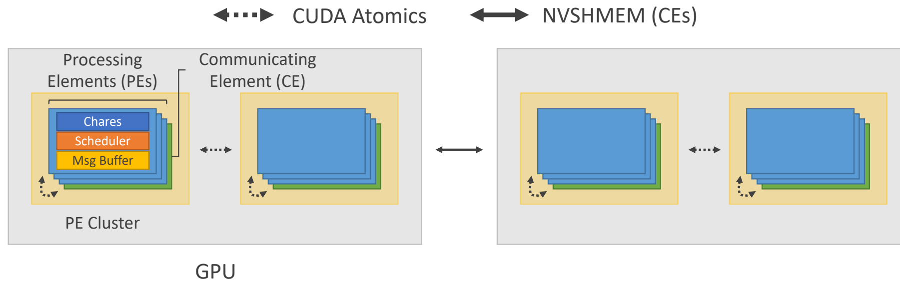


Figure 6.10: SM-level scheduling in CharminG. A GPU device is organized into a number of PE clusters, each of which is a collection of SMs designated as PEs (where chares are executed) or communicating elements (CEs). CEs are exclusively responsible for inter-GPU communication using NVSHMEM, and all PEs and CEs within the same device can communicate using CUDA atomics on global memory. Each chare can only utilize data parallelism of a single thread block.

shared memory buffers of different SMs, which would benefit the performance of a scheduler that utilizes multiple SMs.

Both GPU-level and SM-level modes of scheduling utilize the concept of *persistent thread blocks*, where a single thread block runs persistently on each SM. Having multiple thread blocks per SM for scheduler-driven execution is not feasible as the order of execution of the thread blocks cannot be controlled, and there is no straightforward synchronization/communication method between those thread blocks (e.g., one thread block letting the other blocks know that it has found a message to process). The use of persistent thread blocks also conforms to the requirements of CUDA Cooperative Groups [73] utilized for grid-level synchronization and of NVSHMEM used for inter-GPU communication. CUDA Dynamic Parallelism [74], a feature that allows child kernels to be launched from inside a parent kernel, was also explored as an alternative to achieve scheduler-driven execution; however, this was deemed unsuitable because of the need for device-wide synchronization after the execution of a child kernel and the lack of control over how many SMs are utilized by the child kernel. CUDA Dynamic Parallelism also cannot be used together with CUDA Cooperative Groups, which is a necessary feature for NVSHMEM. In the following sections, we discuss the GPU-level and SM-level scheduling mechanisms in more detail.

GPU-level scheduling. With GPU-level scheduling, an entire GPU device is treated as a PE as illustrated in Figure 6.9, with each chare able to utilize all of the SMs for data-parallel execution. The scheduler runs a while loop on a single SM, performing the following operations:

1. Check the message buffer for incoming messages. If there are any, process the message (can be various types but most common is entry method invocation). This can result in message sends, especially if the executed entry method performs further entry method invocations or the message triggers program termination.

2. Clean up messages that have been sent (and have arrived at the destination) and messages that have been processed.

When an entry method is executed as the result of processing a message, it is executed with as many thread blocks as the number of SMs, allowing the user to perform data-parallel computation. When a message needs to be sent to another PE, NVSHMEM is used as the underlying communication framework as further discussed in Section 6.2.3.

SM-level scheduling. Instead of having a single scheduler instance per GPU device, we could opt for a more fine-grained mechanism where each SM becomes the unit of scheduling chare objects. In this configuration, the SMs of a device are divided into *PE clusters*, each of which consists of PEs and communicating elements (CEs) as illustrated in Figure 6.10. PEs are where the schedulers are run, each now utilizing a single SM (a single thread block), and CEs are responsible for performing inter-GPU communication on behalf of the PEs in the PE cluster. If there are more than one CE in the cluster, PEs are mapped to the CEs in a round-robin fashion. A possible configuration on an NVIDIA Tesla V100 GPU with 80 SMs is 4 PE clusters each with 19 PEs and 1 CE. With the hello world program described in Figures 6.4 and 6.5, this will result in a total of 152 chares as there are a total of 76 PEs.

Since messages may now be exchanged between PEs in the same device, CUDA atomics are used to 'transfer' the message residing in global memory from one PE to another. There is no actual data transfer with such local communication as described in Section 6.2.3; only the address (offset) and size of the source buffer is provided to the destination PE so that the source buffer can be directly accessed by the destination PE. The same occurs when messages or requests need to be exchanged between any pair of PEs or CEs on the same device. When messages need to be sent remotely to a PE (or a CE in some cases) that resides in a different GPU device, CEs are involved in the process, which utilize NVSHMEM atomic signals and one-sided communication to move the data from one GPU to another. Since the functionalities of PEs and CEs vary, their scheduler loops are also different; the CE scheduler loop performs more communication-related operations as CEs have to engage in both device-local and remote communication, but they never execute entry methods as chares only reside on PEs. The scheduler loop on PEs is similar to that of GPU-level

scheduling, checking the local message buffer for incoming messages and processing them, with the addition of possible communication with PEs and CEs on the same device.

6.2.3 Asynchronous Message-Driven Execution

CharminG employs the active messages model [9] to implement asynchronous message-driven execution, where messages are sent from one chare to another to trigger useful work on the receiver. To achieve this inside the GPU, mechanisms for transferring message data within the same device and across different devices are required. For device-local messages, there is no need to copy the message and only the address (offset) and size of the source buffer can be provided to the destination PE. CharminG utilizes CUDA atomics To send a message to another device, CharminG utilizes NVSHMEM [71] as the underlying communication backend. To the best of our knowledge, NVSHMEM is the only openly available communication library that supports inter-GPU communication from within a device kernel. With these messaging mechanisms, CharminG realizes message-driven execution on one or more GPU devices potentially spread across different physical nodes.

Memory management. In CharminG, the user's data are generally required to be copied into a message managed by the runtime system to attach the metadata necessary for message-driven execution. Since the NVSHMEM communication library is used for exchanging messages across GPU devices in both the GPU-level and SM-level scheduling modes, an NVSHMEM buffer with a fixed size is allocated on all participating GPU devices. As an implementation of the OpenSHMEM API, NVSHMEM supports *symmetric* memory allocations distributed across GPU devices connected through NVLink, PCIe or network interconnects such as Infiniband. Its device-side API enables GPU-initiated communication and allows threads of a GPU kernel to efficiently perform one-sided communication and atomic operations on NVSHMEM-allocated buffers. NVSHMEM buffers can also be used as regular GPU buffers by the kernel threads. CharminG builds on the NVSHMEM device-side API to implement mechanisms to transfer messages between distinct GPU devices, which enable scalable execution on clusters of GPU-accelerated compute nodes.

To support the allocation and release of messages of varying sizes in the CharminG runtime system, it would be ideal to integrate a general-purpose dynamic memory allocator such as jemalloc [75] that uses the pre-allocated NVSHMEM buffer as the backing storage. Unfortunately, a GPU-resident implementation of such allocators require efficient data structures such as `std::vector` that are currently not available for use within GPU kernels. As such, CharminG manages the NVSHMEM buffer for messages as a *circular ring*

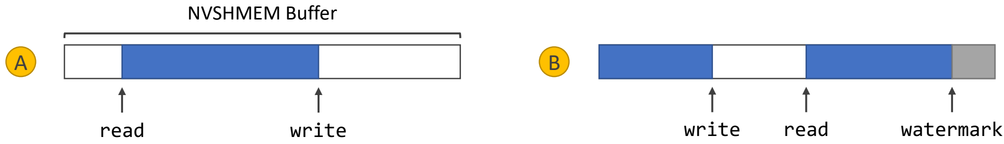


Figure 6.11: NVSHMEM buffer managed as a circular ring buffer in CharminG.

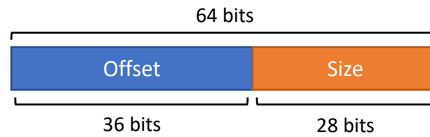


Figure 6.12: A custom 64-bit datatype, composite, used in CharminG. A composite compresses the offset of a message from the start of the NVSHMEM buffer and the size of the message together.

buffer using a custom in-device data structure, with allocations continuously filling up the buffer and de-allocations performed from the other end of the buffer (freeing messages from the lowest address). Two variables, `write` and `read`, keep track of which part of the ring buffer is available for allocation and which is being used for messages by the runtime system; `write` is incremented by the message size upon allocation, and `read` is incremented when a message is freed. An additional variable, `watermark`, marks the end of the buffer when the remainder of the ring buffer is smaller than the size of the message to be allocated and an early wrap-around occurs. To keep the state of the ring buffer consistent, `write` must always be in front of the read (conceptually) and also never catch up to `read`. Figure 6.11 shows two possible states of the ring buffer, first of which where `write` leads (A) and second where `write` follows `read` with an early wrap-around (B). In the case of an exact wrap-around, which happens if there is just enough space at the end of the buffer for a message allocation, `watermark` is not set. Since each PE manages a distinct NVSHMEM buffer[2] and only a single thread from each PE accesses the ring buffer data structure, there is no need to consider concurrent access.

The problem, however, is that messages may not be released by the runtime system in the order of their allocation; for example, after messages A, B, and C are allocated, they may be freed in the order of B, C, and then A. The CharminG runtime system utilizes a *min-heap* to tackle this issue, storing the offsets from the beginning of the NVSHMEM buffer of the to-be-released messages. A min-heap stores values in a binary heap in a way that the value at the root is always the minimum among all the values, and the value in a parent node is

[2]With SM-level scheduling, a single NVSHMEM buffer per device is partitioned to the device-local PEs.

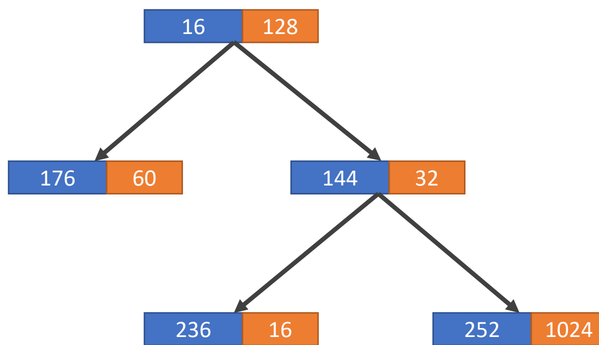


Figure 6.13: Min-heap of composites used in CharminG to track candidate messages for de-allocation. Offset from the start of the NVSHMEM buffer is in blue, with message size in orange. The composite with the lowest offset can be retrieved in constant time.

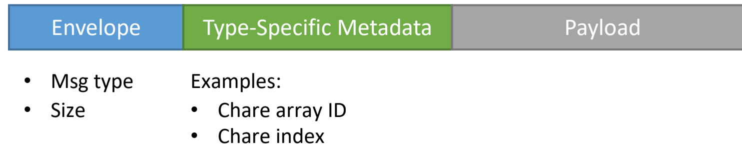


Figure 6.14: Message structure used in CharminG.

smaller than the values of its children nodes. This allows the runtime to query the lowest offset of the candidates for release in constant time and compare it against the current `read` value of the ring buffer, and actually free the corresponding message (increment the `read` value of the ring buffer) if they match. Instead of storing only the offset of a message in the min-heap, however, its size also needs to be stored, as it determines by how much the `read` value of the ring buffer is incremented. To achieve this, a custom 64-bit datatype, *composite*, is introduced. As described in Figure 6.12, it splits the 64-bit space into two, with the first 36 bits reserved for the byte offset of the message and remaining 28 bits for its size. While configurable, this supports an underlying NVSHMEM buffer size of 64 GB and message sizes of up to 256 MB. The C++ binary comparison operator (`operator<`) is overloaded to allow composites to be stored in the min-heap with only the offsets used for the ordering within the heap. An example min-heap storing composites are shown in Figure 6.13.

Message structure and types. Figure 6.14 depicts the structure of messages used in the CharminG runtime system. The first part of the message, the *envelope*, is common to all types of messages and contains the type and size of the message. Type-specific metadata

follows; for regular messages created from entry method invocations, information such as the chare array ID, target chare index, and entry method index is included. The remainder of the message is reserved for the payload that depends on the message type (for some types the payload could be empty). For regular messages, which would be a copy of the user's buffer supplied to the entry method invocation for regular messages. The following are the different message types for message-driven execution in CharminG:

- Regular (REG): Used for entry method invocations. Contains information such as the target chare array ID, chare index, entry method index and reference number.
- Request (REQ): Used by a PE to request a remote message transfer to a cluster-local CE in the SM-level scheduling mode.
- Forward (FWD): Used by a CE to send a message requested by a PE to a remote CE, and by the receiver CE to forward the message to the target PE. Also exclusively used in the SM-level scheduling mode.
- Begin terminate (BTM): Message sent to PE 0 (CE 0 with SM-level scheduling) to trigger termination of the CharminG program.
- Do terminate (DTM): Message sent from PE 0 (CE 0 with SM-level scheduling) to all other PEs for termination.

Detailed descriptions of their usage are provided in the following sections.

Messaging with GPU-level scheduling. With GPU-level scheduling, each GPU device is regarded as a PE with multi-threading capabilities. The scheduler can take advantage of the computing resources of the entire GPU, albeit needing to have the same number of thread blocks as the number of SMs for grid synchronization with CUDA Cooperative Groups. Thus messages are only exchanged between distinct GPU devices, which is realized in CharminG with a one-sided get-based messaging mechanism built on NVSHMEM. The CharminG runtime system is developed with a modular code structure, where the data structures and logic for conducting communication are contained within a *communication module*, which resides in global memory of the device and hence accessible by all threads of the persistent thread blocks.

At the heart of the GPU-level messaging mechanism are the following three data structures:

- `send_status_remote`: An array of `uint64_t` NVSHMEM signals symmetrically allocated with NVSHMEM that describe the status of the messages sent by the PE. Each signal is one of `free`, `used`, or `cleanup`.
- `recv_comp_remote`: An array of composites also symmetrically allocated with NVSHMEM, as described in Figure 6.12, which may arrive from other PEs using NVSHMEM's atomic signaling methods.
- `send_comp_remote`: An array of composites used to store composites sent to other PEs for lazy cleanup.

These are suffixed with `remote` as there are additional `local` versions used in the SM-level scheduling mode. All three arrays have the same number of elements, `REMOTE_MSG_MAX * N_PES`, as communication is tracked on a per device-pair basis. `REMOTE_MSG_MAX` denotes the maximum number of messages allowed to be in flight between a pair of PEs. The meaning of their indices are different according to the array, however; an index in the `send_status_remote` and `send_comp_remote` arrays (`send` arrays) implies the destination PE, whereas an index in the `recv_comp_remote` array (a `recv` array) implies the source PE. For example with two PEs and `REMOTE_MSG_MAX = 4`, each PE will have 8 elements for all three arrays, with four consecutive elements used for communication with one other PE (including itself). An element with index 5 in the `send` arrays on PE 0 refers to a message sent to PE 1 (destination), as it is part of the second batch of four consecutive elements reserved for each PE. Conversely, the same element in a `recv` array on PE 0 indicates that a message will arrive from PE 1 (source).

It would be best to illustrate the messaging mechanism by describing how a message is sent from one PE to another. As an example, let us assume that a chare on PE 0 invoked an entry method of a chare on PE 1, which triggers a message send in the runtime system as shown in Figure 6.15. There are a total of two PEs and `REMOTE_MSG_MAX` is set to four. The runtime on PE 0 would first allocate space for a REG message from the NVSHMEM buffer (managed as a circular ring buffer) and fill in the necessary metadata such as the destination chare array ID, chare index, and entry method index. Then the user's data passed to the entry method invocation needs to be copied into the message, which is done using an in-device memcpy function that is adapted from the NVSHMEM library. The original memcpy function from NVSHMEM is intended for use with a single thread block, but it is very efficient as it uses vectorized copies that takes advantage of the memory alignment. We extend this to use all the available thread blocks for the GPU-level scheduling mode of CharminG, which improves performance when the user data to be copied is sufficiently large. Once the memcpy

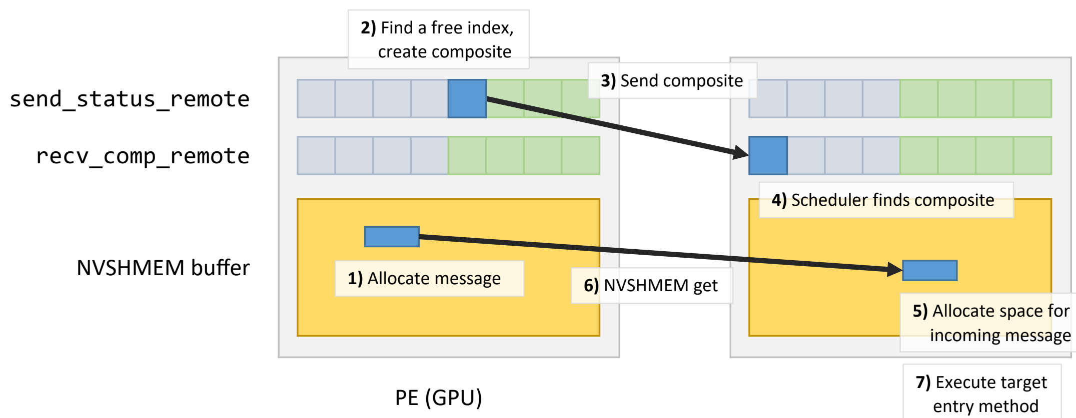


Figure 6.15: Messaging mechanism with GPU-level scheduling. A chare on PE 0 is invoking an entry method on a chare on PE 1, which triggers a message send in the runtime system. There are a total of two PEs, and four messages are allowed to be in flight at a time in one direction between the pair of PEs with `REMOTE_MSG_MAX = 4`.

is complete, the message is ready to be sent. The sender PE looks for a `free` element in the `send_status_remote` array with an index between `REMOTE_MSG_MAX` and `REMOTE_MSG_MAX + 3` as the destination PE is 1. Let us say that the first element (with index `REMOTE_MSG_MAX`) is free and chosen by the runtime. The corresponding element in `send_status_remote` is then set to `used` atomically with the NVSHMEM signal mechanism, and a composite is created using the offset value returned from the message allocation and the size of the message. This 64-bit composite is sent atomically to the destination PE (GPU) also using an NVSHMEM signal, which updates the element with index 0 in the `recv_remote_comp` array. The index is 0 instead of `REMOTE_MSG_MAX` on the receiver since the sender PE is 0 and the first element was chosen from the `REMOTE_MSG_MAX` number of elements reserved for each pair of devices. The composite is also stored on the sender PE in the `send_comp_remote` array, which is used for cleaning up the message once it arrives at the destination.

The scheduler continuously runs on a loop on each PE as described in Section 6.2.2, checking for incoming messages and performing cleanup. Incoming messages can be identified as non-zero composites that arrive in the `recv_comp_remote` array, which is polled atomically using the `nvshmem_uint64_test_some` routine of NVSHMEM. The original implementation of the routine, however, only supports a single GPU thread, which degrades performance when the tested array contains more than few elements. As this routine is called in every scheduler loop, its performance is critical; to mitigate this issue, we modify the mechanism to allow a thread block to check the provided array in parallel, substantially improving

the polling performance. Once the NVSHMEM routine determines which indices of the composite array are valid, the offset and size of the message is extracted from each composite and space for the incoming message is allocated in the destination PE's NVSHMEM buffer. After the allocation, an NVSHMEM get operation is performed to transfer the message from the source PE's NVSHMEM buffer to the allocated portion of the destination PE's NVSHMEM buffer. This is currently done in with a blocking get, which makes the message send asynchronous on the sender side but not on the receiver side. To achieve full asynchrony, this routine is planned to be replaced by a non-blocking get in the future, which would require additional data structures to keep track of the get operations in progress. When the get completes and the corresponding message becomes available, the destination PE processes the message in parallel. In most cases it would be a REG message, which triggers the execution of the entry method of the target chare object once the necessary information such as the chare array ID, chare index and entry method index are obtained. The user's entry method is executed in parallel by all the available thread blocks, which enables data parallel computation in the user application. It is worth noting that the one-sided get operation is utilized as the underlying communication primitive instead of the put because the sender has no knowledge of the status of the NVSHMEM buffer on the receiver. The get-based messaging mechanism also allows the receiver to exclusively manage the allocations and de-allocations on its own NVSHMEM buffer. Once the message is processed, the receiver PE cleans up the locally allocated message by pushing the composite to the min-heap, and sends a cleanup signal to the source PE's `send_status_local` array. The cleanup signal, when picked up by the scheduler, causes the corresponding composite to be pushed to the min-heap. When the offset part of the composite at the root of the min-heap matches the circular ring buffer's `read` value, the message is freed from the NVSHMEM buffer.

The termination mechanism of a CharminG application is also achieved with messages. As briefly introduced previously, the begin terminate (BTM) and do terminate (DTM) message types are involved in the termination process. Any chare object can trigger termination by calling `charm::end()`, which instructs the runtime system to send a BTM message to PE 0. Once PE 0 receives a BTM message, it sends DTM messages to all PEs (including itself). On the receipt of the DTM message, each PE will complete the current scheduler loop and proceed to end the GPU kernel.

Messaging with SM-level scheduling. With SM-level scheduling, each SM can be utilized as a PE rather than the entire GPU as illustrated in Figure 6.10. Communication between SMs in the same device may occur in addition to inter-GPU communication using NVSHMEM, which is implemented using CUDA atomics on global memory. Each SM is des-

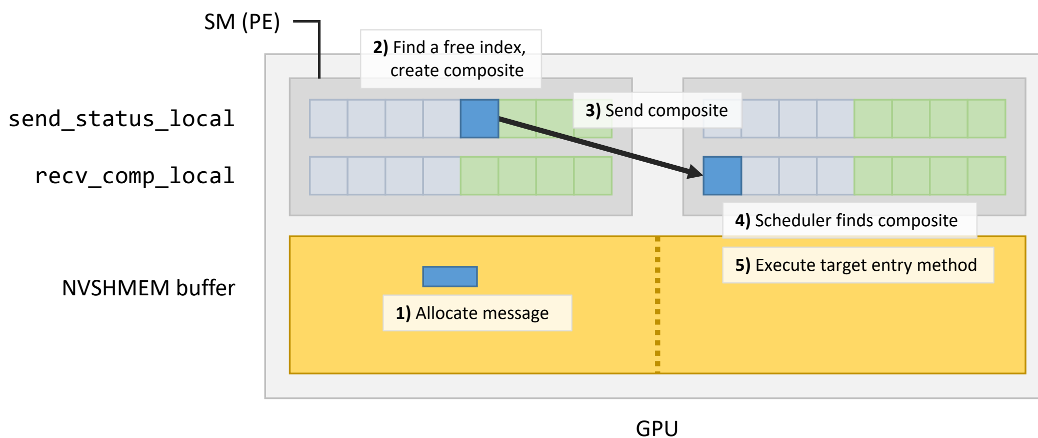


Figure 6.16: Intra-GPU messaging with SM-level scheduling. A chare on PE 0 is invoking an entry method on a chare on PE 1, where two PEs are part of the same GPU device. Four messages are allowed to be in flight at a time in one direction between the pair of PEs with `LOCAL_MSG_MAX = 4`.

ignated as either a processing element (PE) or a communicating element (CE), and messages may be exchanged between PEs and CEs on the same device. Communication between PEs on different devices always go through CEs to limit the number of NVSHMEM calls made per device, as they cause more traffic to and from global memory than device-local communication. There is a scheduler for each PE and CE, which behaves slightly differently; the scheduler on a PE only processes device-local messages by checking the `recv_comp_local` array, whereas the scheduler on a CE processes both device-local messages (from PEs or CEs on the same device) and remote messages (from CEs on a different device) by polling both the `recv_comp_local` and `recv_comp_remote` arrays. PEs receive messages of types REG, FWD and DTM, and CEs receive messages of type REQ, FWD, BTM, and DTM. To explore the messaging mechanism used with SM-level scheduling in more depth, we look at the two following scenarios: (1) a PE sends a message to another PE on the same device, and (2) a PE sends a message to a PE on a different device.

Figure 6.16 describes how a message is sent from one PE to another on the same GPU device for entry method invocation (REG message). The same mechanism applies when communication occurs between a PE and a CE, or between CEs, and with other message types, except that entry method execution would not occur unless the target element is a PE and the message type is REG. All PEs and CEs on a device share the same NVSHMEM buffer, although it is partitioned such that each element has exclusive access to its portion of the buffer. To walk through the messaging mechanism, let us assume that PE 0 is sending

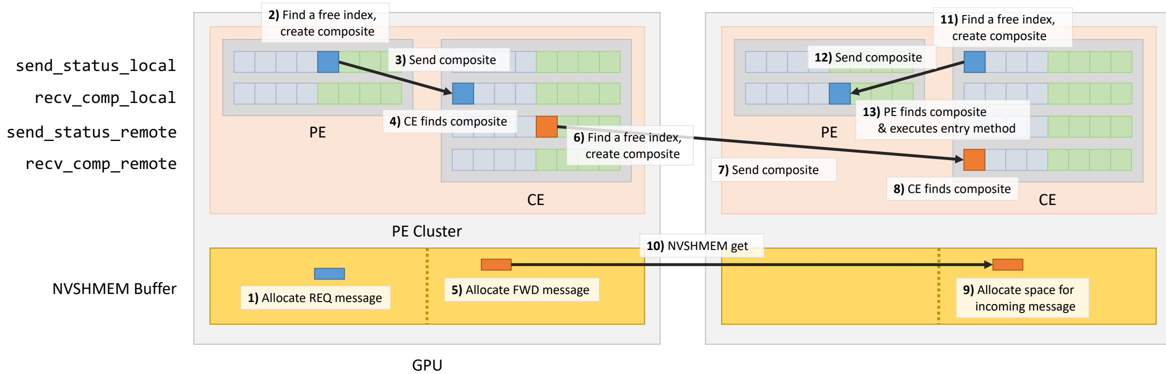


Figure 6.17: Inter-GPU messaging with SM-level scheduling. A chare on PE 0 is invoking an entry method on a chare on PE 1, which is on a different device. CEs in the same PE cluster as the involved PEs are responsible for inter-GPU communication.

a message to PE 1 on the same device. PE 0 first allocates space for the message in the NVSHMEM buffer, and fills it in with the necessary metadata. Then PE 0 looks for a free index in the PE 1 portion of the `send_status_local` array and atomically sets it to `used`, and creates a composite to be sent to (atomically stored at) the corresponding index of the destination PE's `recv_comp_local` array. Note that as these two arrays are regular CUDA arrays and there is no explicit atomic fetch in CUDA, they are accessed using `volatile` pointers to avoid the data being cached. When the destination PE's scheduler finds this composite, it directly accesses the message allocated by the source PE and executes the target chare's entry method. Once the message is processed, the corresponding element in the `send_status_local` array of the source PE is set to `cleanup` so that it can be pushed to the min-heap for eventual cleanup by the source PE. Note that there is no data movement involved with the actual message that resides in the NVSHMEM buffer, only CUDA atomics are used to send and check for composites.

The mechanism for messaging between different GPU devices is more complicated as CEs are involved in the process, in order to reduce the number of elements making NVSHMEM calls and avoid heavy traffic in global memory. Figure 6.17 illustrates the process where one PE is sending a message to another PE on a different device for entry method invocation. The source PE first allocates a REQ message, which contains the necessary information for creating the actual message that contains the user data (such as the address of the user buffer and its size). The REQ message does not contain any payload, as messages sent across GPU boundaries are only created by the CEs. The source PE creates and sends a composite that contains the offset and size of the REQ message, and when picked up by the CE in the same PE cluster, a FWD message is allocated and populated using the

information in the REQ message. The user data is also copied into the FWD message, which is then 'forwarded' to the CE in the PE cluster of the destination PE. When the destination CE finds the composite sent by the source CE, it allocates space in its NVSHMEM buffer for the incoming message and fetches it using an NVSHMEM get operation. This retrieved message is again forwarded to the destination PE, using the intra-GPU messaging mechanism described in Figure 6.16. The target entry method is executed when the destination PE finds the corresponding composite and accesses the FWD message allocated by the destination CE.

Termination can be triggered by any of the chares and occurs in a hierarchical fashion, where a BTM message is first sent to CE 0. Once CE 0 receives the BTM message, it sends DTM messages to all the CEs (in the same device and in other devices). The CEs then each send DTM messages to the PEs in the same PE cluster, ultimately signaling all PEs and CEs to terminate their scheduler loop.

Alternative mechanisms: support for direct one-sided communication, persistent messaging. While message-driven execution is made possible with the previously described messaging mechanisms, its performance is not optimal on current GPU-accelerated platforms due to limitations including the lack of efficient data structures for use within GPU kernels and reliance on one-sided communication primitives. One-sided operations such as put and get are highly performant for pure data movement, but require additional coordination and synchronization between the sender and receiver that render them less suitable for implementing message-driven execution. For some applications, however, one-sided communication can be used to extract more performance where message driven execution is not strictly required; an example is a stencil application that performs halo exchanges, where the exchanges between blocks can be done more efficiently using one-sided communication and signal-based synchronization as in an NVSHMEM implementation [76]. With CharminG, the simulation grid can be (over)decomposed into chare objects each responsible for a block of the grid, and the message-driven execution model can still be used to trigger the chares to start iterating. The halo exchanges can be performed using NVSHMEM one-sided operations and subsequent signals used for notifying the receiver that the halo data have arrived.

However, there is an issue with using the original NVSHMEM signal-based synchronization routines for overdecomposed applications; calls such as `nvshmem_TYPENAME_wait_until_all` used by each block to wait for the arrival of halo data from its neighbors are synchronous, which causes a deadlock with overdecomposition as other chares on the same PE are not able to progress. To enable NVSHMEM one-sided primitives and signals to be used together

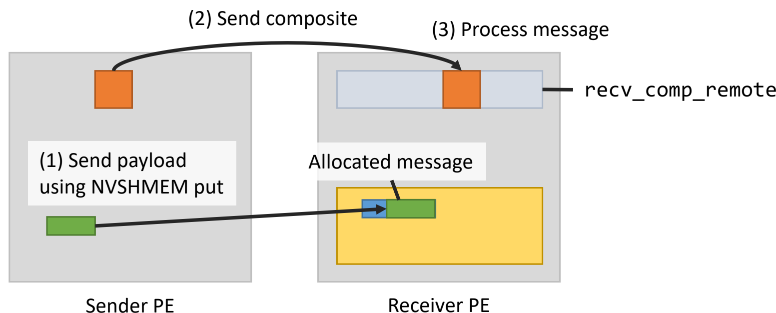


Figure 6.18: Potential persistent messaging mechanism in CharminG.

with overdecomposition, we introduce an asynchronous completion mechanism named *asynchronous wait* (`async_wait`). When a chare needs to wait for NVSHMEM signals before proceeding, such as in the halo exchange above, it can call the `async_wait` function of its proxy, which has the following signature:

```
__device__ void async_wait(uint64_t* ivars, size_t nelems, int cmp, uint64_t cmp_value,
    int idx, int ep);
```

The asynchronous wait API takes the same input as the NVSHMEM wait call including the NVSHMEM signal array (`ivars`), the number of signal elements (`nelems`), a comparison operator (`cmp`), and a comparison value (`cmp_value`). Additional parameters such as the chare index and the entry method index enable the provided entry method to be executed on the target chare once the runtime system determines that the NVSHMEM signals have reached the desired values. The `async_wait` call registers a data structure, `async_wait_t`, which contains all the necessary information for checking the signal array and invoking the continuation entry method. The scheduler checks the status of the registered NVSHMEM signal arrays in every loop with the NVSHMEM test API, and once all the signals of a `async_wait_t` are found to be complete, the corresponding entry method is executed on the target chare object.

Persistent messaging is another mechanism that has the potential to provide improve performance for applications with a persistent communication pattern. When the same entry method has to be executed (potentially with different data) between the same pair of chares over multiple iterations, performance can be substantially improved by avoiding message creation and handshakes between the sender and receiver PEs for every message send. Although not currently implemented in CharminG, the persistent messaging API would work as follows (between PEs on different devices):

1. Coordinate between the sender and receiver PEs of the participating chares to create

space for a REG message on the receiver PE and create a composite on the sender PE with the message's offset and size.

2. Whenever the sender chare wants to send a persistent message to the receiver chare, it sends the user data directly into the payload of the pre-allocated message on the receiver using an NVSHMEM put operation.

3. The sender chare then sends the composite to the receiver (to a `recv_comp_remote` array), which will allow the scheduler on the receiver PE to process the persistent message after the sender's data arrives.

4. After the receiver processes the message, the corresponding composite in the `recv_comp_remote` array is cleared in preparation for the next persistent message.

This is also illustrated in Figure 6.18. The above process can be repeated as many times as needed to efficiently exchange messages between a pair of chares in a persistent manner. Note that each send of a persistent message only involves one NVSHMEM put operation and one send of an NVSHMEM signal (composite), avoids having to copy the user's data into the message allocated on the sender side, and the put operation can be performed proactively by the sender rather than the receiver having to use a get operation to fetch the message.

6.3 PERFORMANCE EVALUATION

6.3.1 Experimental Setup

The Summit supercomputer at Oak Ridge National Laboratory was used to evaluate the performance of CharminG. Each compute node of Summit contains two 22-core IBM POWER9 CPUs and six NVIDIA Tesla V100 GPUs split across two NUMA domains, with a total of 4,608 nodes. The experiments in this section use up to 16 nodes of Summit, or 96 GPU devices. As for the software, CUDA v11.4.2 and NVSHMEM v2.4.1 were used for both native NVSHMEM and CharminG runs, with modified NVSHMEM point-to-point synchronization calls used in CharminG to improve the scheduler's performance.

We first evaluate the communication performance of CharminG, with its message-driven execution model, with a pingpong micro-benchmark. Two chares are created, one on each PE, with each chare sending a message to the other in each iteration. Both the GPU-level and SM-level scheduling modes of Charm++ are evaluated; an additional intra-GPU pingpong experiment is carried out with SM-level scheduling as messages can be exchanged within the same GPU device. The performance of CharminG is compared against a put-based

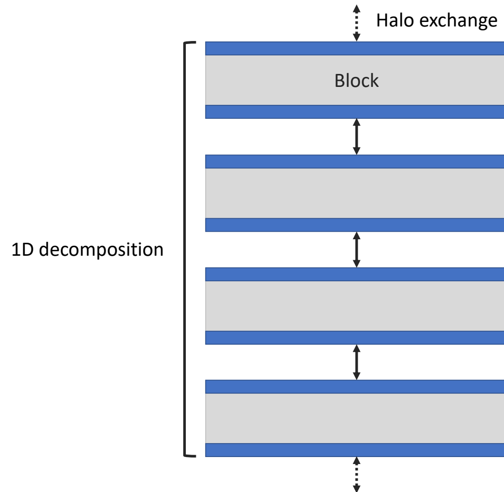


Figure 6.19: Domain decomposition and halo exchange in Jacobi2D proxy application.

pingpong benchmark of NVSHMEM, although CharminG is expected to perform worse due to the handshake and get-based mechanism used to realize the active messages model, as well as additional scheduler operations and memory copy from the user buffer to the runtime message required in CharminG.

A proxy application for two-dimensional stencil applications, Jacobi2D, is also used to benchmark the performance of CharminG. The simulation domain is decomposed across a single dimension (1D decomposition) as shown in Figure 6.19, with a set of rows (block) mapped to each work unit. Each block performs the Jacobi update in parallel and subsequently exchanges the halo data with its neighbors. In the reference NVSHMEM implementation [76], each 2D block is maintained within an NVSHMEM rank, and NVSHMEM put operations are used to transfer the halo data to the two neighbors. After the put operations, NVSHMEM signals are sent to the appropriate signal array elements to notify the arrival of halo data to the neighbors. Each block utilizes the NVSHMEM synchronous wait operation to wait for the neighbors' signals before proceeding to the next iteration. Three different versions of Jacobi2D implemented with CharminG are compared against the reference NVSHMEM version: (1) using NVSHMEM put and synchronous wait in the same way as the reference implementation, (2) using NVSHMEM put and asynchronous wait (supported by CharminG), and (3) using NVSHMEM put and asynchronous wait with 2x overdecomposition. In the CharminG versions, the domain is decomposed into chare objects and the GPU-level scheduling mode is used, allowing each chare to fully utilize data parallelism of the entire GPU for the stencil computation. The asynchronous wait mechanism of CharminG enables overdecomposition as the scheduler can switch between chare objects without

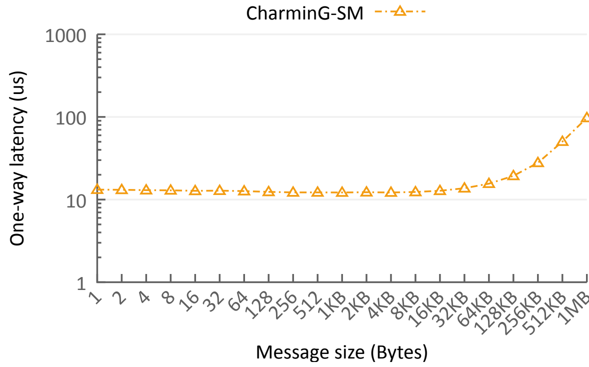


Figure 6.20: Intra-GPU pingpong performance of CharminG (SM-level scheduling only).

being blocked by synchronization.

6.3.2 Experimental Results

Pingpong micro-benchmark. With the pingpong micro-benchmark, we measure the messaging performance of CharminG between two chare objects, each running on a separate PE, and increasing the message size from one byte to one megabyte by a factor of two. The latency for each message size is averaged over a total of 1000 iterations with 10 additional warmup iterations. The performance of raw NVSHMEM gathered using the `shmem_put_ping_pong_latency` benchmark is also provided as reference for intra-node (inter-GPU) and inter-node configurations.

Figure 6.20 illustrates the measured one-way latency of intra-GPU messaging in CharminG with SM-level scheduling; there are no results for CharminG with GPU-level scheduling and raw NVSHMEM as both require two GPU devices to run the pingpong benchmark. The latency for small messages is about 13 us, which includes time for the sender PE to allocate a message, copying the user buffer into the message, calling a CUDA atomic operation to notify the receiver PE, and the receiver PE polling the atomic signal and processing the message. This latency also depends on the duration of each scheduler loop as the array of CUDA atomics used to signal message arrival is checked once per iteration, and the scheduler performs other operations such as checking the min-heap for message cleanup in each loop. As the message size increases, the time to copy the user's buffer into the runtime-managed message starts to dominate as the other factors stay relatively constant.

The results of pingpong between two GPU devices in the same node are depicted in Figure 6.21a. As the measured latency of CharminG includes additional factors on top of the underlying NVSHMEM get operation, both CharminG with GPU-level scheduling

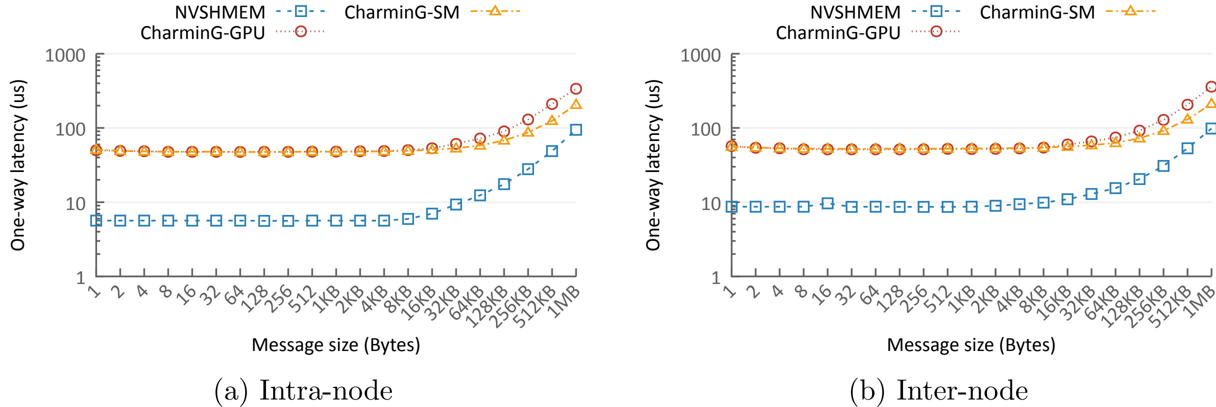


Figure 6.21: Inter-GPU pingpong performance of CharminG (with GPU-level and SM-level scheduling), compared with NVSHMEM.

(CharminG-GPU) and SM-level scheduling (CharminG-SM) perform slower than NVSHMEM which solely involves an NVSHMEM put operation, a signal send, and wait for the arrival of the peer's signal. The performance of CharminG includes time to allocate space for the message using the circular ring buffer data structure, using CUDA atomics or NVSHMEM atomic signals to search for a free message index, storing metadata such as the target chare index inside the message, performing a memory copy of the user's buffer into the allocated message, and other scheduler overheads such as polling for the arrival of a composite needed to perform the NVSHMEM get operation. CharminG-SM suffers additional latency from the involvement of CEs and message forwarding in the inter-GPU messaging process, but ultimately performs similarly to CharminG-GPU for small messages due to faster thread-block-wide synchronization compared to device-wide synchronization needed in the GPU-level scheduler. With larger messages, however, CharminG-GPU performs worse than CharminG-SM due to overheads related to memory management; because there is only one PE per device with GPU-level scheduling, each PE has to manage a much larger memory space when compared to SM-level scheduling. The performance results for two PEs across the node boundary as illustrated in Figure 6.21b tell a similar story, with higher latency times caused by the inter-node data transfers.

Jacobi2D proxy application. Figure 6.22 shows the performance of the Jacobi2D proxy application with various implementations in NVSHMEM and CharminG. The global grid is comprised of 8,192 x 8,192 floating point numbers and the performance of Jacobi2D is averaged over 1000 iterations. As the CUDA cooperative kernel launch feature is used in both NVSHMEM and CharminG, the same number of thread blocks as the number of SMs are used, with each thread block containing 512 threads. As described in Section 6.3.1, the

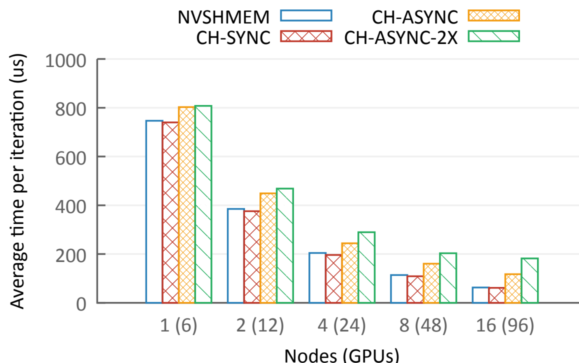


Figure 6.22: Strong scaling performance of Jacobi2D in different versions of NVSHMEM and CharminG.

CharminG implementation with synchronous wait (CH-SYNC) exhibits the same communication pattern as the reference NVSHMEM implementation, with each chare performing the Jacobi stencil operation in parallel and exchanging halo data with its neighbors using NVSHMEM put and signal-based synchronization. However, CH-SYNC performs slightly better than NVSHMEM across the board by a few microseconds, as it avoids the GPU kernel launch overheads exhibited in the NVSHMEM version. CharminG is able to utilize data-parallel device function calls instead of kernel launches with its persistent thread blocks approach. CharminG with asynchronous wait (CH-ASYNC) adds a constant amount of overhead of about 60 us per iteration, due to the relatively inefficient implementation of the asynchronous mechanism that uses array-based data structures. Nevertheless, the asynchronous wait feature enables overdecomposition which can be potentially used for adaptive runtime features such as automatic computation-communication overlap and dynamic load balancing. Unfortunately, the performance of the asynchronous wait feature in CharminG is currently too lacking to observe any performance benefits with overdecomposition, as can be seen with a 2x overdecomposition (2 chares per PE) in CH-ASYNC-2X. With more efficient mechanisms for asynchronous execution in the future, we should be able to improve performance with overdecomposition by exploiting automatic computation-communication overlap, as demonstrated in Chapters 3 and 5.

6.4 CHALLENGES AND LIMITATIONS

As one of the earliest runtime systems designed to be GPU-resident, the development of CharminG has exposed many challenges and limitations imposed by the currently available set of software features and the functionalities of the underlying GPU hardware.

6.4.1 Challenges

We first discuss the challenges that arose in the implementation of the runtime system, which are issues that we were able to overcome despite the difficulties they caused. First is the weakly-ordered memory model of CUDA, where the order of writes performed by one thread is not guaranteed to be the same as the order observed by other threads. This is especially problematic with the SM-level scheduling mechanism in CharminG, as thread blocks on running on separate SMs communicate with one another by writing to and reading from global memory. Memory fences (e.g., `__threadfence()`) can be used to ensure the ordering of groups of memory accesses, but do not guarantee that they are visible to other threads; the `volatile` quantifier must also be used to prevent the variable accessed by multiple threads from being cached or optimized by the compiler. These mechanisms are used in CharminG to enable proper communication between the PEs (implemented using persistent thread blocks) on the same GPU device.

Memory alignment is also an issue that needs to be considered, especially with data structures for use in a GPU-resident runtime system. A salient example in CharminG is related to the allocation of messages; because the NVSHMEM buffer is managed in terms of bytes, each message allocation needs to be properly aligned to avoid errors caused by misaligned memory accesses. For instance, if a message of size 15 which includes a 4-byte header and 11 bytes of the user's payload is allocated, the address of the next allocated message will not have its address aligned to 4 bytes and accessing its header will cause a misaligned memory access error. Such alignment problems are prevented in CharminG by forcing all message allocations to be 16-byte aligned.

Other challenges arise from the slow performance of `malloc` and `memcpy` calls from inside device code. As both the CharminG runtime system and user application are composed using device functions (decorated with `__device__`), memory allocations and copies may be done using in-device calls such as `malloc` and `memcpy`. Although these calls are supported in CUDA, their performance is far from ideal; the time consumed in a `malloc` call by a single thread tends to increase substantially as more thread blocks are executed (most likely to enforce memory consistency), and the `memcpy` call is implemented as a single-threaded assignment loop performed by the calling thread which exhibits poor performance especially when used to copy a large user payload into a runtime-managed message. As such, memory allocations using `malloc` should only be used in places that are not performance-critical such as the beginning of the application, and the CharminG runtime system makes extensive use of pre-allocated buffers to avoid `malloc` calls during execution. As for memory copies, since they are required by the CharminG runtime to copy the user's buffer into a runtime-managed

message, we adopt the `nvshmemi_memcpy` function used inside the NVSHMEM library that utilizes all the threads of a thread block and vectorized memory accesses [77] to perform efficient data-parallel memory copies. While this mechanism can be used directly by each scheduler thread block in the SM-level scheduling mechanism, we also extend it to work with the entire thread grid for the GPU-level scheduling mechanism.

6.4.2 Limitations

Here we discuss the limitations that are imposed by the currently available software and hardware features and their impact on the efficiency and usability of our GPU-resident parallel programming system. A major hurdle to realizing an efficient GPU-driven parallel programming model is the lack of standard library support for device code. Although there is an effort to recreate the C++ standard library for GPU code, libcu++ [78], it currently supports only a small subset of the functionalities. Data structures such as vectors and queues are valuable when developing functionalities such as memory management and message passing between PEs, but they need to be manually implemented for use in device kernels using fixed-size arrays as was done for CharminG. Although most such data structures in CharminG are currently designed for single-threaded access, implementations that utilize all the threads of a thread block or even the entire grid can greatly improve the performance of the scheduler, notwithstanding potential complications from the weakly-ordered memory model of CUDA and concurrent access to the data structures.

Another fundamental limitation or difference from existing host-based runtime systems and applications is the fact that the scheduler and all user-side functions are executed in a multi-threaded fashion. This is because CharminG adopts the persistent thread blocks approach for its schedulers and has to utilize the data-parallelism of the GPU (either on the level of a SM or the entire GPU device), unlike CPU-based Charm++ where each PE is generally single-threaded. This makes both the runtime and user code unwieldy, as there are frequent switches between single-threaded and multi-threaded execution. A potential solution to this problem is CUDA dynamic parallelism, which enables children kernels to be launched from within a device kernel, as it would allow the scheduler and the user code to run single-threaded for the most part, launching a child kernel only when data-parallel execution is required. However, as it currently stands, CUDA dynamic parallelism offers little control over how a child kernel is launched (e.g., how many and which SMs it utilizes) and requires a device-wide synchronization after the launch of a child kernel, which make it insufficient for use as the driving functionality of a GPU-resident runtime system.

As CharminG takes the approach of persistent thread blocks to realize scheduler-driven

execution, the number of thread blocks is limited to the number of SMs in the GPU to have a one-to-one mapping between thread blocks and SMs. While this is currently required to enable thread-block-wide and grid-wide synchronization as well as NVSHMEM collective operations and synchronization, having less thread blocks than regular CUDA programs results in reduced potential of latency hiding than having multiple thread blocks per SM available for switching. Other limitations that were discovered in the design and implementation of CharminG are the unavailability of runtime type information (RTTI) which would have been useful for supporting user chare and entry method types, and lack of profiling tools for breaking down the performance of a device kernel. Due to the lack of RTTI in CUDA, the user's entry methods are confined to a single function type that takes a `void` pointer as the sole parameter and returns `void`. Profiling tools such as `nvprof` and Nsight Systems only analyze kernels at the kernel granularity and does not provide more detailed information such as which device functions are dominating the kernel execution time, which would be useful in determining what parts of the runtime system should be optimized further.

6.5 CONCLUDING REMARKS

We have shown that a fully GPU-resident approach to message-driven execution is feasible through the design and implementation of the CharminG runtime system. However, this is only the first step in the exploration of parallel programming systems suitable for heterogeneous and data-parallel execution, and there are many more studies that could be developed. Firstly, although we have demonstrated that overdecomposition can be achieved, its benefits pertinent to performance were not observed due to the inefficiency of the asynchronous wait mechanism. With improvements to the mechanism and further optimizations in the runtime system, it should be possible to demonstrate speedup with overdecomposition-driven computation-communication overlap as was shown with the CPU-based Charm++ parallel programming system (Chapters 3 and 5). Secondly, we have developed two different modes of scheduling and messaging that can be configured to either treat the entire GPU or each SM as a PE. We have evaluated and compared their communication performance with the pingpong benchmark, but it would be worth carrying out a more detailed comparison between the two mechanisms using other micro-benchmarks and proxy applications. More adaptive features such as dynamic load balancing and fault tolerance, in addition to overdecomposition-driven computation-communication overlap, can be implemented using both scheduling mechanisms to compare their behavior and impact on the overall performance.

As CharminG currently only supports point-to-point messages between a pair of chares,

collective communication routines such as broadcasts and reductions should be implemented to support real-world applications. More efficient mechanisms for applications with certain communication patterns such as persistent messaging can also be developed to mitigate communication overheads and improve performance. This is especially effective for message-driven execution as it bypasses the overheads of message preparation and coordination between the sender and receiver, only exposing the time needed for the underlying data transfer and triggering of the receiver's work. Building on the experiences from the development of CharminG as a GPU-resident runtime system, we should ultimately work towards a comprehensive parallel programming system with schedulers executing on all available heterogeneous computing resources (including CPUs and GPUs), each suited to the characteristics of the underlying hardware and able to dynamically adapt to the properties and behaviors of the user application.

As new software features and hardware functionalities are introduced, the design and implementation of CharminG can be improved for better performance and more adaptive runtime features for heterogeneous systems. With the recently announced Hopper architecture [72] of NVIDIA GPUs, a new concept named *thread block clusters* has been added in the GPU programming hierarchy, which allows multiple thread blocks to be grouped together for concurrent execution on a group of SMs. A dedicated network between SMs in a GPC, which is a group of SMs that are always physically close together, allows efficient data sharing within a thread block cluster. This would enable a new mode of scheduling in CharminG, placed in between SM-level and GPU-level, where multiple SMs can be grouped together to operate as a PE using a thread block cluster. The flexibility in choosing the granularity of scheduling would greatly increase, significantly improving the adaptivity of the GPU-resident runtime. Not only can the SMs of a GPU be partitioned to best fit the task granularity of the application, the grouping can also be modified mid-execution to adapt to any changes in the application behavior. An example would be adaptive mesh refinement where the granularity of the mesh units change during the execution; the configuration of thread block clusters can dynamically change to better exploit locality in each work unit. Distributed shared memory (DSMEM) is another feature that would create synergy with thread block clusters by enabling SMs in the same thread block cluster to directly access data in one another's shared memory. This would increase the efficiency of synchronization and communication required between thread blocks in a thread block cluster required to implement a data-parallel scheduler in runtime systems such as CharminG.

CHAPTER 7: CONCLUSION

In this dissertation, the asynchronous message-driven execution model is used as the driving factor to improve the performance and scalability of HPC applications on modern heterogeneous systems, and to explore the possibilities of a GPU-driven parallel execution model with a GPU-resident runtime system enabled by in-device task scheduling and GPU-initiated communication.

The contributions made in this thesis include:

- Analysis of issues pertinent to realizing the benefits of overdecomposition and asynchronous message-driven execution on GPU-accelerated systems.

- New capabilities in the Hybrid API (HAPI) module of the Charm++ runtime system, such as support for mappings of PEs to GPU devices to improve GPU utilization and mechanisms for asynchronous completion detection to enable asynchrony in GPU-accelerated execution of fine-grained work units and minimize synchronizations between the host and device.

- Analysis of challenges with regard to achieving automatic computation-communication overlap on GPU systems, and discussion of solutions including providing higher priority to communication-related operations and support for asynchronous progress of GPU workloads by the runtime system.

- Evaluation of performance improvements from automatic computation-communication overlap with proxy applications, Jacobi3D and MiniMD, on two state-of-the-art supercomputers Summit and Lassen. We observe up to 50% improvement in weak scaling performance and 35% in strong scaling performance.

- Discussion of challenges in integrating support for GPU-aware communication in message-driven execution models, including the need for metadata used to realize the active messages model.

- Design of two different APIs, GPU Messaging API and Channel API, to support GPU-aware communication in Charm++ and other parallel programming models built on top of the Charm++ runtime system such as Adaptive MPI (AMPI) and Charm4py.

- Performance evaluation of GPU-aware communication mechanisms in Charm++ with latency and bandwidth micro-benchmarks and Jacobi3D proxy application on the Summit supercomputer. Latency improvements of up to 10.1x is demonstrated with

Charm++, 11.7x with AMPI, and 17.4x with Charm4py, as well as bandwidth improvements of up to 10.1x with Charm++, 10x with AMPI, and 10.5 with Charm4py. The overall execution time of Jacobi3D improves up to 82% with Charm++, 41% with AMPI, and 630% with Charm4py.

- Combination of automatic computation-communication overlap and GPU-aware communication to improve both weak and strong scaling performance on GPU-accelerated systems. On the Summit supercomputer, we observe up to 61% performance improvement.
- Evaluation of techniques such as kernel fusion and CUDA Graphs to combat overheads with fine-grained GPU workloads, applied to the Jacobi3D proxy application. Kernel fusion provides up to 51% increase in the overall performance on 128 nodes of Summit, and CUDA Graphs provides 50% performance improvement also on 128 nodes when used without kernel fusion and with ODF-8.
- Exploration of a GPU-driven parallel programming system, CharminG, which performs task scheduling and communication from within a device kernel. Two scheduling mechanisms, GPU-level and SM-level, which treat the entire device or each SM as a processing element, are developed and discussed as well as memory management schemes for message allocation/deallocation and messaging capabilities using CUDA atomics and the NVSHMEM communication library.
- Performance evaluation of the CharminG GPU-resident runtime system with the Jacobi2D proxy application, and assessment of limitations imposed by the currently available hardware and software capabilities as well as potential improvements to GPU-driven parallel execution.

7.1 FUTURE DIRECTIONS

Although many modern supercomputing platforms use GPU devices to accelerate computation, the driver of parallel execution is predominantly the host CPU. The first part of this dissertation approached the problem of mitigating data movement costs by improving the efficiency of such host-driven parallel execution schemes, with automatic computation-communication overlap and GPU-aware communication. Although this approach has been demonstrated to be effective with micro-benchmarks and proxy applications, there may be unforeseen obstacles when it comes to realizing the same benefits in real-world applications.

With more components that affect the parallel execution, such as complex task dependencies, need for collective communication, and load imbalance, supporting large-scale applications would require a more complete and robust set of capabilities from the underlying adaptive runtime system. Two immediate objectives for the Charm++ parallel programming system are adding support for GPU-aware communication in collective communication routines such as broadcasts and reductions, and supporting dynamic load balancing with GPU-resident data taken into calculation. As Charm++ relies on overdecomposition and resulting fine-grained work units to provide its adaptive runtime features, a deeper dive into task granularity may be an interesting research topic; finding out the composition of the execution time such as GPU kernel time, communication overhead, runtime system overhead, and synchronization, and how it changes with different grain sizes could provide an useful insight into what affects the runtime of GPU-accelerated applications at different scales of execution and what needs to be improved to obtain high levels of resource utilization.

In the second part of the dissertation, the CharminG runtime system was used as the vehicle to explore the possibilities and limitations of GPU-driven parallel execution on modern heterogeneous systems. Although it was found to be feasible to realize asynchronous message-driven execution entirely inside the GPU with GPU-resident scheduling and messaging mechanisms, and data structures, there still remains many challenges with the current implementation of the runtime system and capabilities of the currently available software libraries and hardware. In addition to obtaining performance improvements from overdecomposition-driven computation-communication overlap, being able to dynamically configure and group the SMs on the GPU to build adaptive, data-parallel schedulers is a potential direction of future research to better utilize the increasingly GPU-centric computing systems. Ultimately, follow-up research from this dissertation would aim towards an adaptive runtime system that is able to oversee the parallel execution on the entire heterogeneous computing platform, including CPUs, GPUs, and potentially other types of computing resources, with efficient interactions between the different programming and runtime components as well as mitigation of data movement costs that prevent the full potential of the machine from being realized.

REFERENCES

[1] "Programming models and runtimes - exascale computing project," 2022. [Online]. Available: https://www.exascaleproject.org/research-group/programming-models-runtimes/

[2] A. Gürsoy and L. V. Kale, "Performance and modularity benefits of message-driven execution," *Journal of Parallel and Distributed Computing*, vol. 64, no. 4, pp. 461–480, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731504000486

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

[4] J. Choi, D. F. Richards, and L. V. Kale, "Achieving computation-communication overlap with overdecomposition on gpu systems," in *2020 IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 1–10.

[5] J. Choi, Z. Fink, S. White, N. Bhat, D. F. Richards, and L. V. Kale, "Gpu-aware communication with ucx in parallel programming models: Charm++, mpi, and python," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2021. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPSW52791.2021.00079 pp. 479–488.

[6] J. Choi, D. F. Richards, and L. V. Kale, "Improving scalability with gpu-aware asynchronous tasks," 2022. [Online]. Available: https://arxiv.org/abs/2202.11819

[7] J. C. Phillips, D. J. Hardy, J. D. C. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Hénin, W. Jiang, R. McGreevy, M. C. R. Melo, B. K. Radak, R. D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L. V. Kalé, K. Schulten, C. Chipot, and E. Tajkhorshid, "Scalable molecular dynamics on cpu and gpu architectures with namd," *The Journal of Chemical Physics*, vol. 153, no. 4, p. 044130, 2020. [Online]. Available: https://doi.org/10.1063/5.0014475

[8] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, "Massively parallel cosmological simulations with changa," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–12.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: Association for Computing Machinery, 1992. [Online]. Available: https://doi.org/10.1145/139669.140382 p. 256–266.

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

[11] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive mpi," in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322.

[12] J. J. Galvez, K. Senthil, and L. Kale, "Charmpy: A python parallel programming model," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 423–433.

[13] S. Kumar, Y. Sun, and L. V. Kalé, "Acceleration of an asynchronous message driven programming paradigm on ibm blue gene/q," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* IEEE, 2013, pp. 689–699.

[14] "Gpu pro tip: Cuda 7 streams simplify concurrency," 2022. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[15] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in charm++," in *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2. Piscataway, NJ, USA: IEEE Press, 2016. [Online]. Available: https://doi.org/10.1109/ESPM2.2016.7 pp. 40–43.

[16] D. Kunzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012, http://charm.cs.uiuc.edu/media/12-45/.

[17] "Charm++ documentation - gpu support," 2022. [Online]. Available: https://charm.readthedocs.io/en/latest/charm%2B%2B/manual.html#gpu-support

[18] "Nvidia topology-aware gpu selection (nvtags)," 2022. [Online]. Available: https://developer.nvidia.com/nvidia-nvtags

[19] "Cuda callback," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html#group__CUDART__EVENT

[20] "Cuda events," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1g74aa9f4b1c2f12d994bf13876a5a2498

[21] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, and A. Bhatele, "Optimizing computation-communication overlap in asynchronous task-based programs," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3330345.3330379 p. 380–391.

[22] C. Zimmer, S. Atchley, R. Pankajakshan, B. E. Smith, I. Karlin, M. L. Leininger, A. Bertsch, B. S. Ryujin, J. Burmark, A. Walker-Loud, M. A. Clark, and O. Pearce, "An evaluation of the coral interconnects," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356166

[23] "Mantevo/minimd," 2020. [Online]. Available: https://github.com/Mantevo/miniMD

[24] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002199918571039X

[25] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3202–3216, Dec. 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.07.003

[26] "Kokkos lectures module 5: Simd, streams and tasking," 2020. [Online]. Available: https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/KokkosTutorial_05_SIMDStreamsTasking.pdf

[27] "Kokkos github issue #2545: Undesired fence-like behavior without calling a fence," 2019. [Online]. Available: https://github.com/kokkos/kokkos/issues/2545#issuecomment-555143767

[28] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2676870.2676883

[29] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, pp. 173–193, 06 2011.

[30] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 863–874.

[32] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012, pp. 1848–1857.

[33] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 80–89.

[34] D. Bonachea and P. H. Hargrove, "Gasnet-ex: A high-performance, portable communication library for exascale," in *Languages and Compilers for Parallel Computing*, M. Hall and H. Sundar, Eds. Cham: Springer International Publishing, 2019, pp. 138–158.

[35] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 34–39.

[36] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[37] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia gpudirect over infiniband–a new model for gpu to gpu communications," *Comput. Sci.*, vol. 26, no. 3–4, p. 267–273, June 2011. [Online]. Available: https://doi.org/10.1007/s00450-011-0157-1

[38] "Gpudirect rdma :: Cuda toolkit documentation," 2021. [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[39] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. D. K. Panda, "Designing efficient small message transfer mechanism for inter-node mpi communication on infiniband gpu clusters," in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10.

[40] "Charm++ zero copy messaging api," 2021. [Online]. Available: https://charm.readthedocs.io/en/v6.10.2/charm++/manual.html#zero-copy-messaging-api

[41] J. J. Galvez, K. Senthil, and L. Kale, "Charmpy: A python parallel programming model," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 423–433.

[42] "Charm4py channels api," 2021. [Online]. Available: https://charm4py.readthedocs.io/en/latest/introduction.html#channels

[43] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt, "Pmix: Process management for exascale environments," *Parallel Computing*, vol. 79, pp. 9 – 29, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118302424

[44] "Charm++ futures," 2021. [Online]. Available: https://charm.readthedocs.io/en/latest/charm++/manual.html#futures

[45] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[46] "Ucx-py," 2021. [Online]. Available: https://github.com/rapidsai/ucx-py

[47] "Charm4py futures api," 2021. [Online]. Available: https://charm4py.readthedocs.io/en/latest/introduction.html#futures

[48] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, "Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters," in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 110–120.

[49] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "Mvapich2-gpu: Optimized gpu to gpu communication for infiniband clusters," *Comput. Sci.*, vol. 26, no. 3–4, p. 257–266, June 2011. [Online]. Available: https://doi.org/10.1007/s00450-011-0171-3

[50] N. Hanford, R. Pankajakshan, E. A. León, and I. Karlin, "Challenges of gpu-aware communication in mpi," in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 1–10.

[51] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, "Performance evaluation of mpi libraries on gpu-enabled openpower architectures: Early experiences," in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, Eds. Cham: Springer International Publishing, 2019, pp. 361–378.

[52] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified parallel c for gpu clusters: Language extensions and compiler implementation," in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 151–165.

[53] M. P. Robson, "Techniques for communication optimization of parallel programs in an adaptive runtime system," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2020. [Online]. Available: {http://hdl.handle.net/2142/108622}

[54] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, 2010, pp. 344–350.

[55] "Getting started with cuda graphs — nvidia developer blog," 2022. [Online]. Available: https://developer.nvidia.com/blog/cuda-graphs/

[56] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–10.

[57] R. V. Aaron Becker and L. V. Kale, "Patterns for Overlapping Communication and Computation," in *Workshop on Parallel Programming Patterns (ParaPLOP 2009)*, June 2009.

[58] "Structured control flow: Structured dagger," 2022. [Online]. Available: https://charm.readthedocs.io/en/latest/charm++/manual.html#structured-control-flow-structured-dagger

[59] J. Choi, Z. Fink, S. White, N. Bhat, D. F. Richards, and L. V. Kale, "Accelerating communication for parallel programming models on gpu systems," 2022. [Online]. Available: https://arxiv.org/abs/2102.12416

[60] "Accelerating pytorch with cuda graphs," 2022. [Online]. Available: https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

[61] M. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," 2014.

[62] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 58–58.

[63] "Olcf frontier node diagram," 2022. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2020/02/frontier_node_diagram_lr.png

[64] "Frontier," 2022. [Online]. Available: https://www.olcf.ornl.gov/frontier

[65] J. Schuchart, P. Samfass, C. Niethammer, J. Gracia, and G. Bosilca, "Callback-based completion notification using mpi continuations," *Parallel Computing*, vol. 106, p. 102793, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819121000466

[66] S. Jones, "Lazy gpu programming," 2021. [Online]. Available: https://github.com/mpiwg-hybrid/hybrid-issues/blob/master/slides/2021-03-10%20--%20Lazy%20GPU%20Programming%20(Stephen%20Jones).pdf

[67] T. Gysi, J. Bär, and T. Hoefler, "dcuda: Hardware supported overlap of computation and communication," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 609–620.

[68] M. E. Belviranli, S. Lee, J. S. Vetter, and L. N. Bhuyan, "Juggler: A dependence-aware task-based execution framework for gpus," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3178487.3178492 p. 54–67.

[69] "Chapel," in *Programming Models for Parallel Computing.* The MIT Press, 11 2015. [Online]. Available: https://doi.org/10.7551/mitpress/9486.003.0008

[70] "Nvidia collective communications library (nccl)," 2022. [Online]. Available: https://developer.nvidia.com/nccl

[71] "Nvshmem," 2022. [Online]. Available: https://developer.nvidia.com/nvshmem

[72] M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy, "Nvidia hopper architecture in-depth," 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[73] M. Harris and K. Perelygin, "Cooperative groups: Flexible cuda thread programming," 2017. [Online]. Available: https://devblogs.nvidia.com/cooperative-groups/

[74] A. Adinets, "Cuda dynamic parallelism api and principles," 2014. [Online]. Available: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/

[75] "jemalloc," 2022. [Online]. Available: https://jemalloc.net/

[76] "Nvshmem jacobi," 2022. [Online]. Available: https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/nvshmem_opt

[77] J. Luitjens, "Cuda pro tip: Increase performance with vectorized memory access," 2013. [Online]. Available: https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

[78] "libcu++: The c++ standard library for your entire system," 2022. [Online]. Available: https://nvidia.github.io/libcudacxx/