MOVEMENT AND PLACEMENT OF NON-CONTIGUOUS DATA IN DISTRIBUTED GPU COMPUTING

BY

CARL PEARSON

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Abstract

Steady increase in accelerator performance has driven demand for faster interconnects to avert the memory bandwidth wall. This has resulted in wide adoption of heterogeneous systems with varying underlying interconnects, and has delegated the task of understanding and copying data to the system or application developer. Data transfer performance on these systems is now impacted by many factors including data transfer modality, system interconnect hardware details, CPU caching state, CPU power management state, driver policies, virtual memory paging efficiency, and data placement.

This work finds that empirical communication measurements can be used to automatically schedule and execute intra- and inter-node communication in a modern heterogeneous system, providing "hand-tuned" performance without the need for complex or error-prone communication development at the application level. Empirical measurements are provided by a set of microbenchmarks designed for system and application developers to understand memory transfer behavior across different data placement and exchange scenarios. These benchmarks are the first comprehensive evaluation of all GPU communication primitives. For communication-heavy applications, optimally using communication capabilities is challenging and essential for performance. Two different approaches are examined. The first is a high-level 3D stencil communication library, which can automatically create a static communication plan based on the stencil and system parameters. This library is able to reduce iteration time of a state-of-the-art stencil code by 1.45× at 3072 GPUs and 512 nodes. The second is a more general MPI interposer library, with novel non-contiguous data handling and runtime implementation selection for MPI communication primitives. A portable pure-MPI halo exchange is brought to within half the speed of the stencil-specific library, supported by a five order-of-magnitude improvement in MPI communication latency for non-contiguous data.

To my wife, for her support, guidance, and cool head.

Acknowledgments

I would like to thank my committee for their valuable feedback and guidance. This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy Contract No. DE-AC05-00OR22725. This work utilizes resources supported by the National Science Foundation's Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign. I would also like to thank Dawei Mu, Omer Anjum, I-Hsin Chung, and Mert Hidayetoglu.

Table of Contents

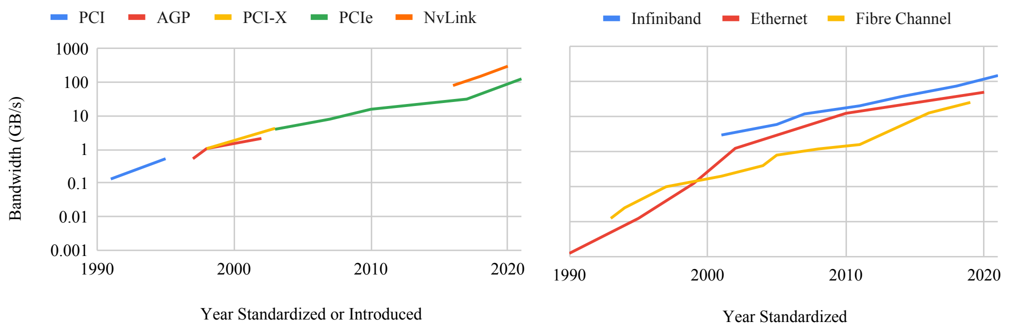
Chapter 1

Introduction

With the end of Dennard scaling, computer architects have sought to satisfy demand for increasing performance by providing specialized hardware accelerators tuned to computation with particular characteristics. Perhaps the most successful example of this trend is the widespread adoption of graphics processing units (GPUs) for more general data-parallel compute tasks. With the success of GPUs as a template, architects are moving forward with a wide variety of accelerators, such as SIMD extensions [1, 2, 3], AI accelerators (Google tensor processing unit [4], Huawei Neural Processing Unit [5], IBM neuromorphic chips [6], Intel Nervana [7]), motion coprocessors (Apple M-series [8]), field-programmable gate arrays (Xilinx Virtex [9], Intel Stratix [10]), network processors (Netronome Agilio [11]), digital signal processors (Qualcomm Hexagon [12], NXP DSP56xx Family [13]), vision processing units (Eyeriss [14], Movidius VPU [15], Mobileye EyeQ [16], Microsoft Holographic Processing Unit [17]) and many others. These heterogeneous systems have become the dominant system architecture.

The enormous compute capability accelerators demands high-bandwidth data access to "feed the beast." Without this bandwidth, the performance potential of the accelerator is largely wasted waiting for data. The trend of integration (also motivated by reduction of total system cost) where semiconductor die-size or power limits allow has provided one approach to solving this problem. By integrating an accelerator onto the same die as the CPU, the accelerator more easily gets high-bandwidth low-power access to data shared with the CPU. For accelerators with high memory demands, however, the system memory DRAM bandwidth may ultimately limit performance.

The second approach is to provide accelerators with their own high-performance memory. Unfortunately, managing this memory then falls upon runtime systems or the application developer, and moving data into accelerator memory to support high-performance execution is a first-order design consideration for any accelerated application. The data-placement and data-movement challenge is exacerbated by the growing demand for data-driven applications. Analytics and neural-network applications ingest huge amounts of data, and even if the computation per data element is small, the aggregate computation can be commensurately large. That motivates developers to use accelerators for these applications. To achieve high performance on accelerators, developers must marshal and coordinate their data movement and computation in heterogeneous systems.



(a) GPU Bandwidth (b) Network Bandwidth

Figure 1.1: Growth of interconnect bandwidth over time. In all cases, the fastest configuration is used (e.g., 16 lanes of PCIe 3.0).

For accelerators with their own limited high-performance memory, the effect of the interconnect on the overall system performance has not escaped notice. Figure 1.1 shows the rapid growth of GPU interconnect bandwidth over time, by year introduced (NVLink) or standardized (PCI, AGP, PCI-X, PCIe) [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]. As the importance of these interconnects grows, they also are used as the backbone for various software- and hardware-backed coherency schemes between accelerators and host components. Performance of the interconnects that tie accelerators together is the foundational motivation for this work.

This work finds that empirical communication measurements can be used to automatically schedule and execute intra- and inter-node communication in a modern heterogeneous system, providing "hand-tuned" performance without the need for complex or error-prone communication development at the application level. This is demonstrated through the development and evaluation of a high-level communication library for distributed stencil codes. Since the communication library automatically discovers and applies relevant techniques, substantial performance improvement is realized for an existing application, which was limited by implementation complexity and lack of optimization for the evaluation platform. The primary downside of implementing the techniques in a high-level library is that existing applications would need to be modified to use it. To that end, this work also demonstrates how a widely-used communication interface can use the same empirical measurements.

The rest of this document is organized as follows:

- Chapter 2 describes background information on heterogeneous computers and the CUDA programming system, Linux NUMA system, and MPI.

- Chapter 3 describes the design, implementation, and evaluation of the CUDA communication microbenchmarks. These microbenchmarks are the first comprehensive measurement of point-to-point bulk CUDA communication methods, and provide the empirical measurements.

- Chapter 4 describes and evaluates how insights from the benchmarks can be used to automatically plan and execute communication without system knowledge from the application developer. The evaluation is conducted by developing, deploying, and analyzing a high-level stencil library and includes analysis of techniques that influence performance in different circumstances.

- Chapter 5 describes how techniques developed in Chapters 3 and 4 can be integrated transparently with existing MPI applications. This includes a novel low-overhead and general strategy for handling non-contiguous GPU data, low-latency use of empirical performance information at runtime to inform communication strategy, and an implementation that can transparently improve communication methods without modifying applications.

- Chapter 6 discusses related work.

- Chapter 7 offers conclusions and future directions for this work.

Chapter 2

Background

2.1 CUDA System

For the purposes of this document, a CUDA-enabled computer comprises three pieces: first, one or more CUDA "devices" – general-purpose graphics processing units (GPGPUS or usually GPUs). Second, the "host" – the CPU(s) and associated memory. Third, the "CUDA system" – the combination of the CUDA runtime library (accessible to the program through the CUDA runtime library), the CUDA driver (not directly controllable from the program), and the interconnect hardware (accessible through the operating system).

The host and devices are where the traditional focus of high-performance computing has been, e.g. loop optimization, vectorized operations, branch prediction, and low-cost software abstractions. While these two components are crucial to application performance, so is the CUDA system itself. Within the CUDA system, the runtime library, driver, and interconnect hardware all contribute to the observable performance.

2.2 CUDA Runtime API vs. CUDA Operation

The performance of CUDA operations comprises two pieces: the time it takes the application thread to initialize the operation, typically by making CUDA runtime library calls, and the time it takes the CUDA system to execute the operation.

Figure 2.1 shows four scenarios: 2.1a, a synchronous operation, where control is not return to the program until the GPU activity is completed; 2.1b, an asynchronous operation, where the calling thread is blocked for long

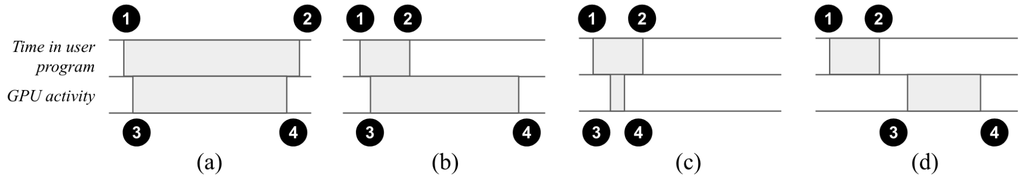


Figure 2.1: Examples of time taken in the CUDA runtime and corresponding GPU activity. (a) A synchronous operation, (b) an asynchronous long operation, (c) an asynchronous short operation, and (d) a delayed operations.

enough to initialize the operations; 2.1c, an asynchronous operation where the operation is so short it completes before control returns to the calling thread; and Figure 2.1d , an asynchronous operation that is substantially delayed from the point of initiation. The interval ❶-❷ represents how long before control returns to the CPU (possibly to initiate another GPU operation), and ❸-❹ represents actual execution using the GPU resources. Depending on the API call and the operation, these intervals may be very different, though both intervals are relevant for understanding GPU performance.

2.3 CUDA Streams and Events

A CUDA stream is a queue of device work. With some exceptions for the default stream, each stream represents an independent queue whose contents is to be consumed by a GPU sequentially. Contents in different queues can be consumed by one or more GPUs in parallel. Within a stream, no operation may begin until the previous operation has completed. Between streams, there is no implicit ordering.

Streams are the main vehicle for task-level concurrency in single-GPU and multi-GPU systems. A single GPU can improve its utilization by pulling independent tasks from multiple streams. Multiple GPUs can have parallel tasks executing from multiple streams.

Kernels can be enqueued into a specific stream via their launch parameters. The cudaMemcpyAsync* family of functions enqueues data transfer operations in a stream explicitly. A CUDA event can be inserted into a stream, and serves as a no-op that still maintains its position in the queue. Streams can be synchronized with the host through cudaStreamSynchronize, or with other streams independent of the host with cudaStreamWaitEvent. Finally, streams can have higher or lower priority. When a GPU can take an operation from a stream, it will pick an operation from a higher priority stream over a lower one.

2.4 Data Movement in CUDA Systems

This section describes the three classes of user-facing data transfers in CUDA systems: explicit transfers, zero-copy/mapped transfers, and managed memory.

2.4.1 Explicit Block and Strided Transfers

Explicit transfers are initiated through the cudaMemcpy* class of API routines. A contiguous buffer referenced by a pointer and a size is transferred from one address to another. GPUs include copy engines, which are able to handle these transfers without invoking the GPU SMs or the CPU. CUDA also provides similar APIs that allow transfers of non-contiguous memory regions through the cudaMemcpy2D* and cudaMemcpy3D* functions. Depending on the GPU hardware, these non-contiguous transfers place various restrictions on the size and alignment of the individual contiguous blocks of the non-contiguous object. Otherwise, these functions are similar to the explicit transfers described above.

2.4.2 Zero-copy and Direct Access

"Zero-copy" is a common name for the ability of different devices to directly access memory which is physically present on another device. These accesses are served by a transaction over the interconnect, without changing the location of the backing data. Memory accesses originating from the CPU or GPU that reference data on another device are converted to requests that cross the interconnect (e.g. PCIe or NVLink). Data is retrieved from the owning device, and returned to the source device over the interconnect. These accesses are particularly high latency, and require very careful attention to alignment and coalescing to achieve full link utilization [29]. When this mechanism is used for accesses between the CPU and GPU, it is commonly called a "zero-copy" access, or an access to "zero-copy" memory. There is not a broadly accepted term for a GPU kernel accessing data on another GPU; in this document, it is referred to as "direct access" or "zero-copy".

2.4.3 Unified Memory

CUDA optionally provides a unified memory abstraction, where the CUDA system is responsible for presenting a coherent image of memory to all devices, including atomic operations across the entire system. Such memory can be allocated with the cudaMallocManaged function. In order to accelerate performance, CUDA will try to move data at the page granularity to the device that most recently accessed it (the "demand" mechanism). It may also rely on the direct-access mechanism when thrashing access patterns are detected. The user can provide usage hints through the cudaMemAdvise API, including prefetching data to the device (the "prefetch" mechanism).

2.5 Synchronous and Asynchronous CUDA Operations

"Synchronous" CUDA operations are those which block progress on the calling CPU thread until they have completed. They are commonly used due to simpler integration with host code during the initial development process. For example, a compute-intensive CPU function (which is naturally blocking, as the caller does not proceed until the function returns) can be directly replaced with a synchronous GPU operation without changing the semantics of the application.

"Asynchronous" operations are those which do not block the calling thread. In this case, the calling thread will dispatch work to the GPU, and then proceed, and the GPU will execute that work at some point in the future. An example of such an operation is the `cudaMemcpyAsync` function, which starts a data transfer but may return before that transfer is complete. Use of these functions typically allows better utilization of GPU resources, but requires more complex coordination of CPU and GPU execution. Even when the primary interaction between the CPU and GPU is asynchronous, synchronous operations are always eventually used to make sure the host does not attempt to retrieve incomplete results from the GPU.

2.6 MPI

MPI is a specification for a library that implements the message-passing parallel programming model [30]. MPI implementations have become the dominant choice for distributed-memory high-performance computing on supercomputers. Several implementations are widely used, including Spectrum MPI [31], Open MPI [32], MPICH [33], and MVAPICH [34]. MPI functions operate on untyped buffers, which are typically "source" or "destination" buffers, or both. MPI includes a variety of point-to-point transfers and collective operations, and synchronous and asynchronous versions of most functions.

2.6.1 CUDA-Aware MPI

CUDA does not directly provide a mechanism to move data between GPUs on different nodes. A common programming pattern on GPU-accelerated distributed computing is to use CUDA to manage data movement between GPUs and from CPU to GPU, and to use MPI to move data between CPUs. Some MPI implementations optionally support passing device pointers directly to MPI calls, leaving it up to the MPI implementation to handle moving data between GPUs in different ranks. Such implementations are said to be CUDA-aware MPI implementations.

Chapter 3

Measurements

The foundation for improving multi-socket and multi-GPU data transfer performance is measurement of the properties of that transfer. Comm|Scope [35] is a tool primarily developed by the author to address some of the pitfalls and gaps of previous measurement work. For a more detailed discussion of related work see Chapter 6. Comm|Scope contributes low-overhead bandwidth measurement implementations coupled with careful system performance controls. With detailed measurements, it is possible for users to adjust the design of their applications to maximize performance, and for system developers to observe minute effects that may point to performance bugs. This chapter describes the design and implementation of Comm|Scope, shows how CUDA achieves dramatically different bandwidth under different configurations, and describes guidelines for high-performance CUDA data transfer.

3.1 Comm|Scope Design

Comm|Scope [35] is a CUDA C++ microbenchmark program developed by the author that measures the performance of CUDA data transfers and associated API calls. It uses the libscope system benchmarking library, also developed by the author and described in Section 3.2.

3.1.1 Low-overhead Bandwidth Measurement

Section 2.2 describes the contributions of the CUDA runtime and the rest of the CUDA system to the total execution time of CUDA operations. When measuring the raw bandwidth achievable over the link, Comm|Scope's microbenchmarks minimize the unintended measurement of CUDA runtime overhead. This section describes the measurement approach.

Asynchronous operations are best measured with CUDA events, which minimize the overhead of the measurement. Figure 3.1 shows an example timeline of measurement. An event is recorded at the beginning and end of one or more CUDA operations within a stream, and then the cudaEvent-GetElapsedTime function provides the time between the two events.

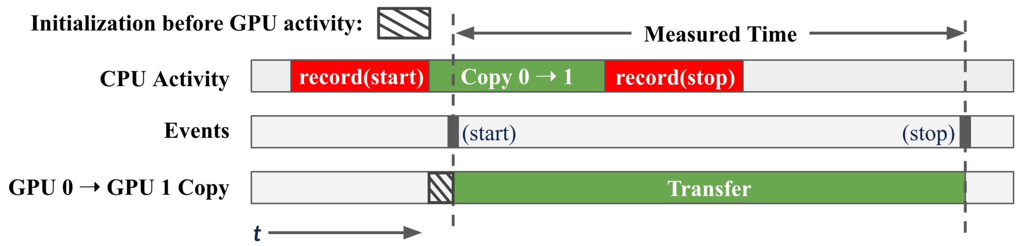


Figure 3.1: Example timeline of correctly measuring an asynchronous data transfer between GPU 0 and GPU 1. The CPU records a *start* and *stop* event on either end of the transfer. The CUDA system records when those events trigger in the stream, and `cudaEventGetElapsedTime` is used to measure the transfer time, without including initialization time on the CPU before the GPU activity begins.

A common but less-accurate method is to use host wall-clock time with synchronous CUDA operations, or asynchronous CUDA operations followed by `cudaDeviceSynchronize` or `cudaStreamSynchronize`. Figure 3.2 shows an example of this method. It incorrectly includes two unknown times: the time between the function call and the start of the operation (❶), and the time between the end of the operation and the end of the synchronization with the host (❷). Even widely referenced benchmarks like SHOC [36] use this method. When Comm|Scope measures the time of asynchronous CUDA operations, it uses the low-overhead method in Figure 3.1.

3.1.2 Bidirectional Transfer Measurements

The bidirectional bandwidth of a link is the amount of data that can be transmitted simultaneously in both directions in a specified amount of time. When the two transfers begin and end at the same time, the bandwidth is the total data amount divided by the elapsed time. In practice, using this approach to measure bidirectional bandwidth is challenging due to skew between the transfer start and stop times. It is also not possible to accurately

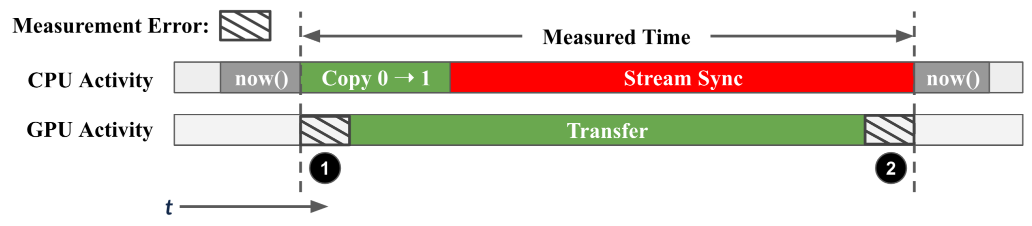


Figure 3.2: Example timeline of a less-accurate measure of asynchronous operation time. The host wall time (`now`) is recorded before and after the operation is launched. ❶ (❷) marks a duration when the CPU initiates (waits for) the operation and the operation actually begins (ends). In contrast to the procedure shown in Figure 3.1, these durations are incorrectly included in the measured time.

determine *only* the overlapping portion of the two transfers: CUDA events cannot be queried for an absolute start and end time, and elapsed time between events in different streams cannot be compared. An obvious but less-accurate approach is to record the wall time, initiate the asynchronous events, and then record the wall time again once both events have completed. Figure 3.3 shows a timeline of such a measurement.

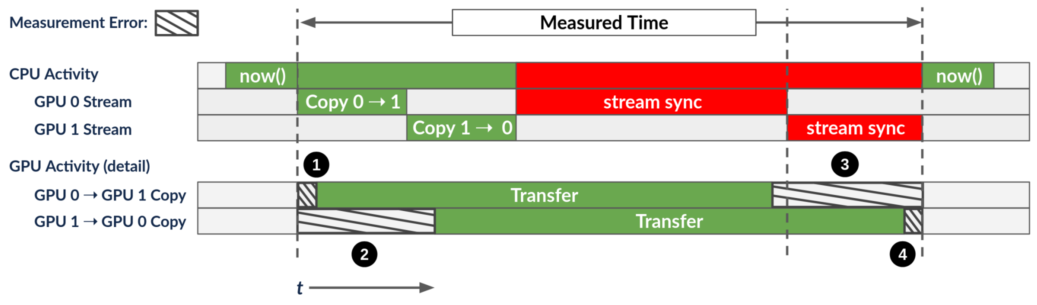


Figure 3.3: Example of improper measurement of a bidirectional transfer. Before both transfers are initiated and after both transfers complete, the host wall time is used to infer the achieved bandwidth. ❶ is skew between the CPU entering the CUDA runtime call and the beginning of the operation. ❷ highlights how the second transfer can be delayed due to the CPU cost of initializing the first transfer. ❷ shows how one transfer may end before the other. ❸ is skew between the end of the operation and the CUDA runtime call returning. These measurement errors make the bidirectional bandwidth appear lower than its true value.

Two streams are used to allow the transfers to execute at the same time, with one in each stream. The CPU thread records the starting wall-time, initiates both transfers, synchronizes both streams, and then records the ending wall time. The weaknesses of this approach is that it introduces skew in the start (❶), stop (❷), and includes time between the end of the operation and return of control to the host thread (❸ and ❹). All these errors improperly reduce the estimate of the bidirectional bandwidth, as the links are not fully active during the measured time.

Comm|Scope minimizes the effect of the start-time skew and synchronization overhead (❶ and ❸ in Figure 3.3) through a corrected measurement implementation, shown in Figure 3.4.

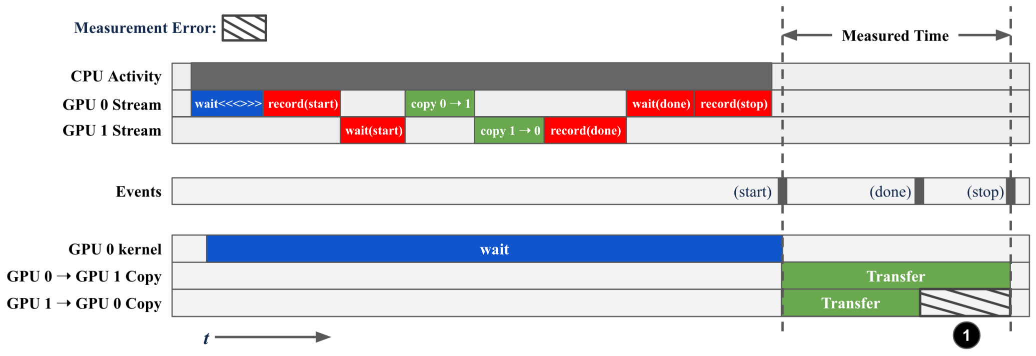


Figure 3.4: Example of an accurate measure of a bidirectional transfer. One direction is associated with each stream. First, a busy-wait kernel is launched to block operations from beginning during initialization. The "start" event is used to synchronize the beginning of the operations in each stream, and a "done" event is used to mark when both operations have completed. In this manner, the "start" and "stop" events bookend both transfer options with minimal overhead. ❶ is skew when the actual transfers do not take the same amount of time, and therefore are not fully overlapped.

Before the measurement begins, a busy-wait kernel is launched in the first stream. This kernel occupies the GPU, and the CUDA event *start* inserted afterwards is used to block the execution of the two data transfers until the kernel completes. The run-time of the wait kernel is sufficient to allow the CPU to set up all asynchronous transfers and events, thereby removing the start-time skew. This is ensured by querying whether the *start* event has been triggered after all communications are initialized. If so, the kernel was no long enough, and the process is restarted with a longer wait kernel. This is ensured by progressively lengthening the kernel until it has not completed after the CPU Since CUDA events in different streams cannot be compared for elapsed time, the *stop* event in the first stream waits for the *done* event in the second stream. The result is that *start* marks the time the transfers begin, and *stop* is only recorded once both transfers have ended. The synchronization overhead is removed by using `cudaEventGetElapsedTime` to measure the transfer time. ❶ may still occur if one transfer is slower than the other.

3.1.3 Measuring Synchronous Operations

Synchronous CUDA operations do not return control to the calling thread until they are complete. Comm|Scope measures synchronous operations by using the wall-time before and after the operation. Figure 3.5 summarizes the technique.

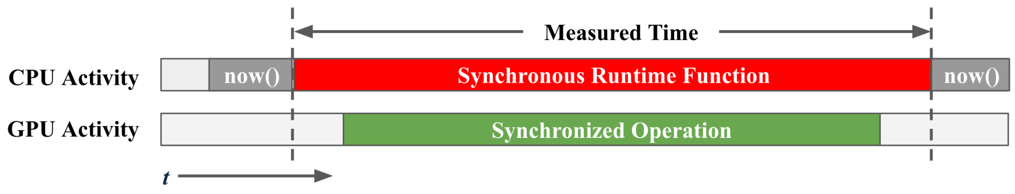


Figure 3.5: Example of an accurate measure of a synchronous operation. The purpose of the synchronous operation is to block the calling thread, so the measured time is simply the length of time the calling thread is blocked.

3.2 Libscope Design

Libscope is a C++ system benchmarking library, developed by the author, which brings a variety of pre-existing techniques under one umbrella to ease the implementation of CUDA microbenchmarks that are sensitive to system configuration. This section describes the techniques implemented in libscope.

Variable CPU Clock Speeds

Many computers feature dynamic CPU frequency scaling to conserve power when idle and boost performance for transient tasks. This presents a challenge when measuring performance, as CPU frequency may not be the same from run to run. In the context of this work, the CPU performance could have a substantial impact on performance of the CUDA unified memory system and CUDA driver operations. On Linux, libscope automatically sets the CPU governor to “performance” and can disable CPU boosting through the ACPI or Intel P-State [37] interface. The original CPU scaling behavior is restored when the benchmarks exit or are interrupted. Prior CUDA communication benchmarks make no report of whether or how this variable is controlled.

CPU Data Caching

CPU caches have a measurable effect on CPU/GPU data transfer performance. Libscope provides an interface for flushing CPU caches through the `dcbf` [38, p. 773] on POWER and `clflush` [39, p. 139] on AMD64. These instructions invalidate and flush the cache lines associated with a particular virtual address from all CPU data caches. This is done in some Comm|Scope benchmarks before the transfer is initiated. Prior works that measure CUDA transfers do not address this consideration.

3.2.1 NUMA Pinning

In ordinary program execution, the operating system may move the program between CPU cores or sockets. This introduces execution time variability by causing cache misses and changing which interconnects are required to move data between the CPU and GPU. Libscope uses libnuma [40] to control the execution and memory allocation pinning to specific sockets or CPUs in order to control which interconnects are measured.

3.2.2 Compiler Side-Effects

When measuring zero-copy or unified memory host-to-GPU performance, the GPU kernel should be as close to pure reads as possible to ensure that host-to-GPU data delivery is measured with as little overhead as possible.

Listing 3.1: Minimal GPU read kernel (not implemented).

```
1  template <unsigned GD, unsigned BD, typename read_t>
2  __global__ void gpu_read(const read_t *ptr,
3                           const size_t bytes) {
4    const size_t gx = blockIdx.x * BD + threadIdx.x;
5    const size_t num_elems = bytes / sizeof(read_t);
6
7    for (size_t i = gx; i < num_elems; i += GD * BD) {
8      read_t t;
9      t = ptr[i];
10   }
11 }
```

Listing 3.1 shows such a minimal kernel. The grid of threads (grid size GD and block size BD) loops over `bytes` bytes pointed to by `ptr`, loading them with `read_t` accesses. When the benchmarks are compiled with optimizations turned on, the compiler observes that the load implied by line 9 has no effect, and the entire code is eliminated, preventing the load performance from being measured.

To correct this, the code is modified in two ways, shown in Listing 3.2. First, the `do_not_optimize` function is called on the result of the load. `do_not_optimize` is a wrapper that inserts PTX code which puts a fake data dependency and memory side-effect on the argument. This does not insert any instructions, but prevents the compiler from removing the unused load in the generated PTX code.

Unfortunately for benchmarking (but fortunately when generating optimized application code), the second phase of the compilation which transforms the PTX code into SASS will also do some simple optimization. In this case, it will observe that the virtual register corresponding to `t` is unused, and remove it (along with the load that generated it, which is the target of the measurement). To defeat the second optimization, the `flag` parameter is added, and a dummy store is hidden behind a conditional predicated on the value of `flag`. This prevents the compiler from statically removing the load. At run-time, this flag is given a null pointer, so the dummy store is not executed. However, the compiler can still observe that the load only has an effect if the store occurs (if `flag` is true), and lowers the load into the conditional body guarded by `flag`. The `do_not_optimize` function also prevents this from occurring.

Listing 3.2: gpu_read kernel

```
template <unsigned GD, unsigned BD, typename read_t>
__global__ void gpu_read(const read_t *ptr, read_t *flag,
                         const size_t bytes) {
  const size_t gx = blockIdx.x * BD + threadIdx.x;
  const size_t num_elems = bytes / sizeof(read_t);

  for (size_t i = gx; i < num_elems; i += GD * BD) {
    read_t t;
    do_not_optimize(t = ptr[i]);
    if (flag) {
      *flag = t;
    }
  }
}
```

Listing 3.3 shows two implementations of `do_not_optimize` for different argument types. Since they are template __device__ functions, they are inlined and there is no function call overhead.

Listing 3.3: device do_not_optimize

```
template<> __device__
void do_not_optimize<int32_t>(const int32_t& t) {
  asm volatile("" ::"r"(t) : "memory");
}

template<> __device__
void do_not_optimize<int64_t>(const int64_t& t) {
  asm volatile("" ::"l"(t) : "memory");
}
```

3.3 Observations and Guidelines

Systems-oriented microbenchmarking can accurately quantify data transfer performance, reveal subtleties of performance not previously described in the literature, and reveal surprising behavior that might suggest performance bugs. This section highlights some examples from the leadership-class Summit system at Oak Ridge National Lab.

In particular, the measurements highlight:

- The large difference in observed data transfer performance depending on which data transfer method is used: 2× for GPU-GPU, 8× for GPU-CPU (Section 3.3.2).
- How locality improves bandwidth bandwidth (Section 3.3.3).
- How bidirectional transfers improve link utilization (Section 3.3.4).
- How multiple threads cannot hide CPU cost (Section 3.3.7).
- Using the CUDA Graph API to reduce CPU cost (Section 3.3.7).

3.3.1 Experimental System

All experiments for this chapter are carried out on Summit [41], a leadership-class computing system at Oak Ridge National Labs. Summit comprises 4,600 compute nodes, each summarized in Table 3.1 and Figure 3.6. Triplets of GPUs are associated with each CPU: GPUs 0-2 with socket 0 and GPUs 3-5 with socket 1. Within a triplet, components are fully connected by NVLink 2.0 x2 links, for 100 GB/s bidirectional bandwidth. Between triplets, the sockets are connected with a 64 GB/s x-bus SMP interconnect. This means that communication localized to one triplet should be at much higher bandwidth than communication between triplets. The network is a non-blocking fat tree of EDR InfiniBand with 23 GB/s node injection bandwidth [42].

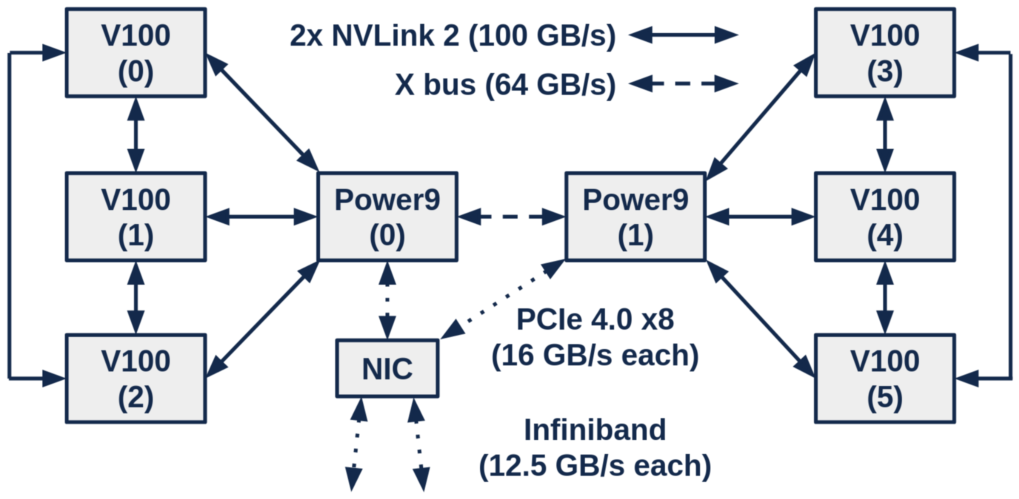


Figure 3.6: Diagram of interconnect bandwidths of a Summit compute node.

Table 3.1: Summit node hardware summary

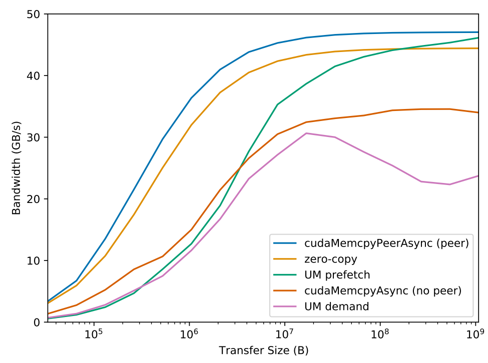
CPU	OS	Kernel	GPUs	CUDA Driver	MPI	nvcc	cc
22-core POWER9	RHEL 7.6	4.14.0-115.21.2.el7a.ppc64le	V100-SXM2-16GB	418.116	Spectrum 10.3.1.2	10.1.243	g++ 6.5.0

3.3.2 Bandwidth Utilization

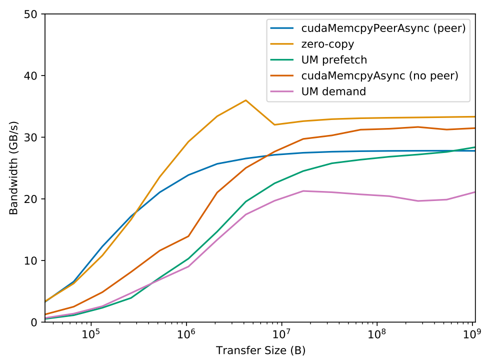
CUDA provides a variety of methods for moving data between participating components, and not all provide the same performance. For example, five methods for moving data (explicit transfers with and without peer access, zero-copy access, and unified memory through the demand or prefetch mechanism) between components are compared in Figure 3.7. All transfers are unidirectional. Figures 3.7a, 3.7c, and 3.7e show "near"-component bandwidth (directly connected CPUs and GPUs), and Figures 3.7b, 3.7d, and 3.7f show "far"-component bandwidth (CPUs and GPUs associated with different sockets). Several observations are apparent:

- For "small" sizes ($< 10^6$), elapsed time is dominated by a fixed overhead.
- For "large" sizes ($> 10^8$), elapsed time is dominated by the transfer size.
- Larger transfers typically have greater-or-equal bandwidth to smaller transfers of the same type.
- No transfer reaches the 50 GB/s theoretical "near" limit (the fastest is 94%).
- The x-bus is nominally 64 GB/s bidirectional [43], but certain "far" transfers are able to exceed 50% of that speed, reaching 40 GB/s.
- zero-copy transfers can reliably match explicit transfers for favorable access patterns. The gap in Figures 3.7a - 3.7e is probably due to a small amount of overhead introduced by interaction with the cache. For the slower "far" transfers, the interconnect is slow enough to mask the effect.
- Section 3.3.3 highlights the locality effects.

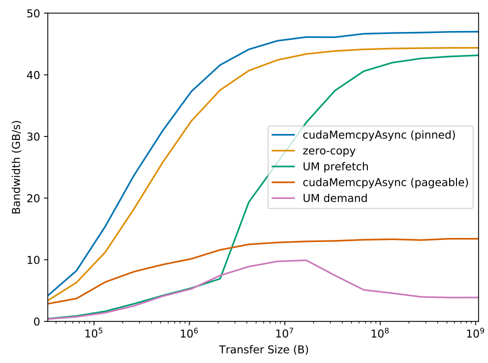
It is clear that the achievable bandwidth depends strongly on the modality, i.e. the method used. The variation of achievable bandwidth can be up to 2x for GPU-GPU transfer.



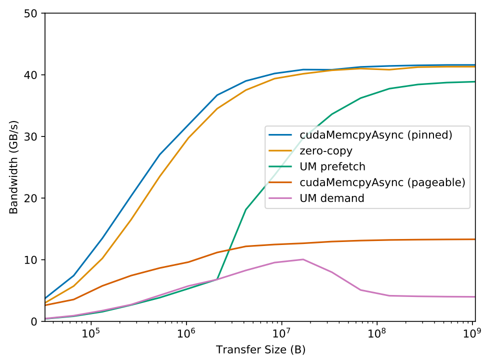
(a) GPU-GPU Bandwidth (0-1)



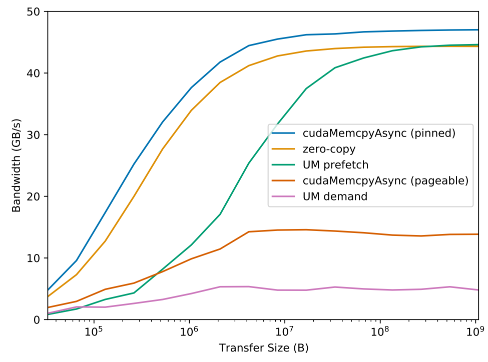
(b) GPU-GPU Bandwidth (0-3)



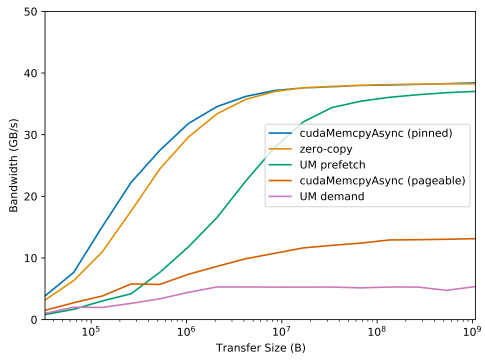
(c) CPU to GPU Bandwidth (0-0)



(d) CPU to GPU Bandwidth (0-3)



(e) GPU to CPU Bandwidth (0-0)



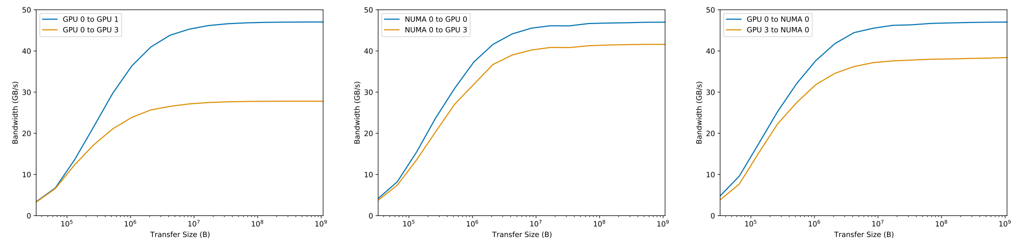
(f) GPU to CPU Bandwidth (0-3)

Figure 3.7: GPU-GPU and GPU-CPU bandwidth for different CUDA transfer methods. Numbers in parentheses, e.g. (0-1), refer to the participating CPU or GPU ids. For each row, all transfers occur over the same links, but the CUDA communication method can strongly affect performance.

3.3.3 Locality

Section 3.3.1 described how different components have different theoretical bandwidth between them. These bandwidth differences have a strong effect on bandwidth measurable at the application level. Figure 3.8 highlights a specific transfer method from Figure 3.7 to demonstrate the locality effect between a pair of GPUs (3.8a), CPU-to-GPU (3.8b), and GPU-to-CPU (3.8c). For all transfers over NVLink, the system achieves 47.0 GB/s out of the theoretical 50 GB/s provided by the interconnect. For other transfers over x-bus, bandwidth drops. GPU-GPU transfers achieve 27.8 GB/s, while GPU $\rightarrow$ CPU achieves 38.4 and CPU $\rightarrow$ GPU achieves 41.6. The specifications for the x-bus are 64 GB/s bidirectional, but the CPU-GPU transfers achieve more than 50% of that capacity. This suggests the X-bus boosts its transfer rate when only a single direction is used (e.g., by increasing clock speed).

Similar effects can be seen for zero-copy and unified-memory prefetch transfers, the other two "fast" transfer methods.



(a) cudaMemcpyPeerAsync GPU-GPU Bandwidth

(b) cudaMemcpyAsync CPU to GPU Bandwidth

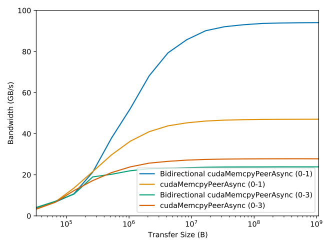
(c) cudaMemcpyAsync GPU to CPU Bandwidth

Figure 3.8: GPU-GPU and GPU-CPU bandwidth. Data transfers over multiple interconnects (Section 3.3.1) exhibit lower bandwidth due to the lower x-bus bandwidth.

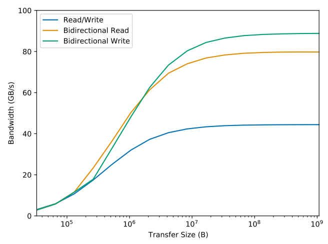
In summary, for CPU-GPU transfers, the performance effect of locality is less than expected since the unidirectional transfers achieve more than 50% of the bidirectional bandwidth. At most, CPU-GPU locality only affords a 23% bandwidth improvement. For GPU-GPU transfers it is important to place data so that larger communication occurs between directly connected GPUs, with directly connected GPUs having a 69% bandwidth improvement.

3.3.4 Bidirectional Transfers

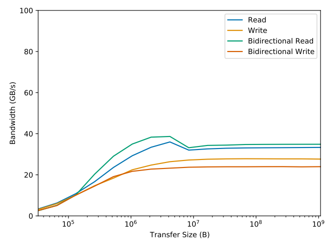
Typical interconnects have a higher bidirectional bandwidth than single-directional. This is due to the physical construction of these bidirectional interconnects, which comprise pairs of single-directional physical links. Figure 3.9 shows how utilizing both directions of the link simultaneously typically improves aggregate bandwidth. Figure 3.9a shows the performance of cudaMemcpyPeerAsync. Figure 3.9b shows the performance of direct accesses between GPUs 0 and 1. Figure 3.9c shows the performance of direct accesses between GPUs 0 and 3, across the X-bus between two sockets.



(a) cudaMemcpyPeerAsync bandwidth. Bidirectional bandwidth drops for “far” transfers, but doubles for “near” ones.



(b) Zero-copy (GPUs 0 to 1). Minor bandwidth variability in bidirectional transfers is observable between near and far accesses, but both greatly increased the total bandwidth.

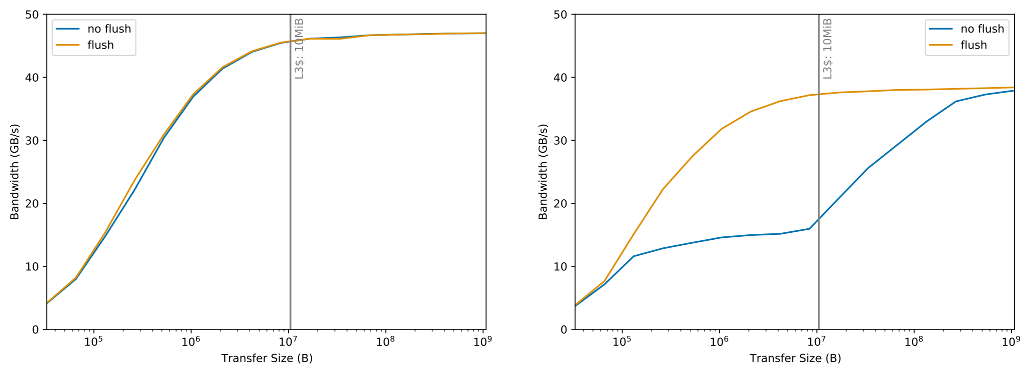


(c) Zero-Copy (GPU 0 to 3). Bidirectional write accesses are slower than any unidirectional transfer. Bidirectional read offers some small improvement.

Figure 3.9: Bidirectional transfer bandwidth for cudaMemcpyPeer, and zero-copy transfers between GPUs. For “far” transfers, bidirectional transfers cause a large bandwidth regression for cudaMemcpyPeerAsync and zero-copy transfers. For “near” transfers, performance nearly doubles, as expected. It is possible that some performance bug affects inter-socket transfers, or perhaps the X-bus boosts its transfer rate when only a single direction is used (e.g., by increasing clock speed).

3.3.5 Cache Effects

The state of the CPU cache affects the performance of GPU-to-CPU transfers. Figure 3.10 shows how remote GPU-to-CPU bandwidth varies when the L3 cache is flushed. When the cache is flushed, bandwidth for data coming to the CPU is greatly increased around the L3 cache size. This is probably because the cache does not need to be flushed on-demand. In a small unflushed transfer, each line in the destination buffer maps to a unique line in the cache. For a larger transfer, multiple addresses in the destination



(a) GPU 0 to NUMA 0 (single-hop) (b) GPU 3 to NUMA 0 (multi-hop)

Figure 3.10: GPU-to-CPU transfer bandwidth on a Summit node using `cudaMemcpyAsync`. When the CPU caches are flushed before the transfer, bandwidth is much higher for transfers around the L3 cache size (10MiB per core-pair).

buffer correspond to each cache line, which only needs to be invalidated once. As the transfer grows, the cost of each invalidation is amortized over many transferred bytes, and the bandwidth climbs back towards the maximum [1] .

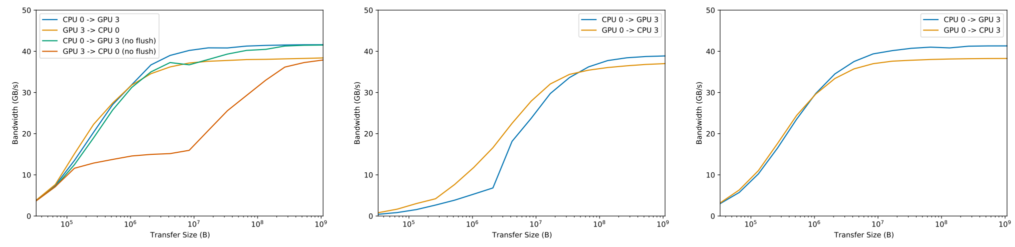
3.3.6 Anisotropy

Anisotropy is the property of exhibiting different properties in different directions. In this context, it means that the bandwidth between two components is different in different directions.

This is observed on Summit for some transfers that cross the CPU socket boundary (e.g., between GPUs 0 and 3). This observation is not directly actionable for the user, as applications do not typically offer any flexibility in which direction data must move between the CPU and the GPU. System developers, however, may use these observations as a starting point to investigate performance bugs.

Figure 3.11 shows this effect in three scenarios. Figure 3.11a shows that CPU 0 to GPU 3 is several GB/s faster than the reverse direction. The effect is greatly magnified if the CPU cache is not flushed before receiving the data. Figure 3.11b shows that during unified-memory prefetch, smaller messages are faster in the GPU-to-CPU direction, while the opposite is true

[1]The locality effect (Section 3.3.3) is also visible.



(a) cudaMemcpyAsync (b) Unified memory prefetch (c) zero-copy

Figure 3.11: Examples of anisotropy in multi-hop intra-node bandwidth. Similar effects are not observed for directly connected components.

for messages larger than 4×10^7 bytes. Figure 3.11c shows that for zero-copy transfers, the CPU-to-GPU direction is faster for messages above several dozen kilobytes. The GPU-to-CPU direction is stores instead of loads, which may incur some overhead for cache coherency.

3.3.7 CUDA Runtime

Until now, this chapter was concerned with the achievable bandwidth across component links under various conditions. Comm|Scope can also be used to measure the CPU cost of invoking the runtime operations themselves. The high theoretical interconnect bandwidths mean that even a relatively fast runtime operation represents a large amount of data movement. Table 3.2 shows the cost of select CUDA runtime operations, and the maximum of data that could be moved in that time between two GPUs on a Summit node. For smaller transfers, the cost of initiating the transfer may dwarf the time of the transfer itself. To hide this cost, large transfers should be initiated before small ones to keep the CUDA system busy.

Comm|Scope also shows that using multiple threads to overcome this CUDA runtime cost is not effective. Figure 3.12 demonstrates this effect for various CUDA runtime operations. As the number of threads increases, the aggregate throughput does not substantially improve (and even degrades). Multiple threads controlling a GPU context introduce mutual exclusion locks, which causes the run-time cost of each call to grow. These measurements suggest that using multiple threads to control the GPU is not generally an appropriate method to issue CUDA runtime operations faster. Multiple threads

Table 3.2: Cost of select CUDA runtime operations on CPU 0, and how much data could be moved at 50GB/s during that time.

CUDA Runtime Call	**Time (s)**	**Bytes (50 GB/s)**
cudaMemcpyAsync	5.169×10^{-6}	2.58×10^{5}
cudaMemcpy3DPeerAsync	6.188×10^{-6}	3.09×10^{5}
kernel launch (0B)	5.887×10^{-6}	2.94×10^{5}
kernel launch (1B)	6.093×10^{-6}	3.05×10^{5}
kernel launch (256B)	6.064×10^{-6}	3.03×10^{5}
kernel launch (4096B)	6.595×10^{-6}	3.30×10^{5}

may be useful if other CPU work is the bottleneck instead of CUDA runtime throughput.

Another way to reduce the CPU cost of some CUDA operations is the CUDA graph API. CUDA Graph permits a two-step process, where a sequence of calls are “instantiated” (recorded), and then “launched” (replayed) later with reduced overhead. Any captured kernels are configured once, allowing future execution of the same kernel with the same arguments to be faster. This is especially useful if the same sequence of operations will be repeated over and over again. Figure 3.13 shows the run-time cost of the two CUDA Graph operations with a variable number and type of CUDA runtime calls. Using cudaGraphLaunch for kernels provides substantial speedup (7.6× at 20 kernels, compared with Table 3.2).

3.4 Conclusion

This chapter describes the design and implementation of Comm|Scope, a point-to-point communication microbenchmark for multi-GPU multi-socket systems. Two trends have combined to make intra-node bandwidth increasingly difficult to understand. First, improving performance of the underlying interconnect hardware causes other overheads (e.g. cache effects) to become apparent. Second, as systems become more heterogeneous, the interconnects between components become more non-uniform.

This chapter began by introducing the techniques that allow detailed measurements (Section 3.1). Synchronous and asynchronous CUDA operations are handled differently, as are unidirectional and bidirectional transfers. Once detailed measurement methodology is established, small sources of variabil-

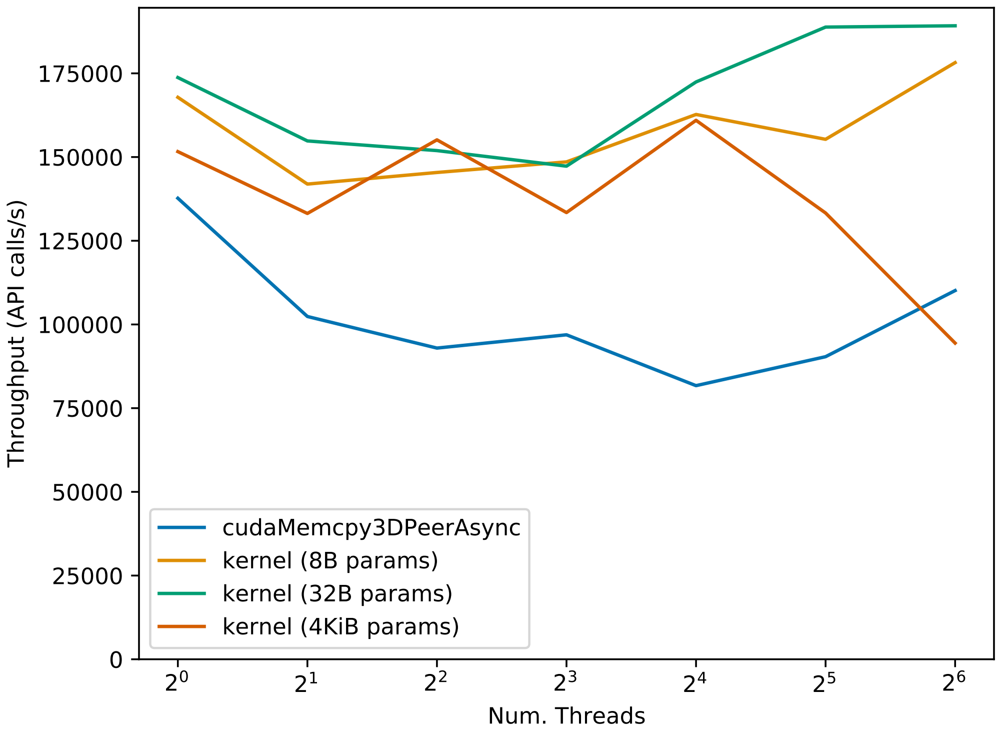


Figure 3.12: Aggregate throughput of selected CUDA runtime calls with various numbers of calling threads (higher is faster). “params” refers to the total number of bytes in the CUDA kernel arguments. A large jump in cost is observed from one to two threads. As the number of threads further increases, some calls show performance rising back to the single-thread case, while other calls degrade further.

ity become visible. A separate library, LibScope, brings techniques together to manage variable CPU clock speeds, CPU data caching, NUMA pinning, and selectively defeat some compiler optimizations that inhibit benchmarking (Section 3.2).

Finally, the chapter concludes by presenting quantitative results, and some corresponding qualitative guidelines (Section 3.3). First, on fast interconnects, the choice of communication method is especially important. For GPU-GPU transfers, the fastest method (explicit) is more than double the speed of the slowest (unified memory demand accesses) due to the page-ownership mechanism required by the latter. For CPU-GPU transfers, that difference grows to roughly 9× due to the CPU’s inability to generate enough demand accesses to saturate the interconnect. On heterogeneous systems, GPU locality is important, with “nearby” GPUs featuring a 69% bandwidth

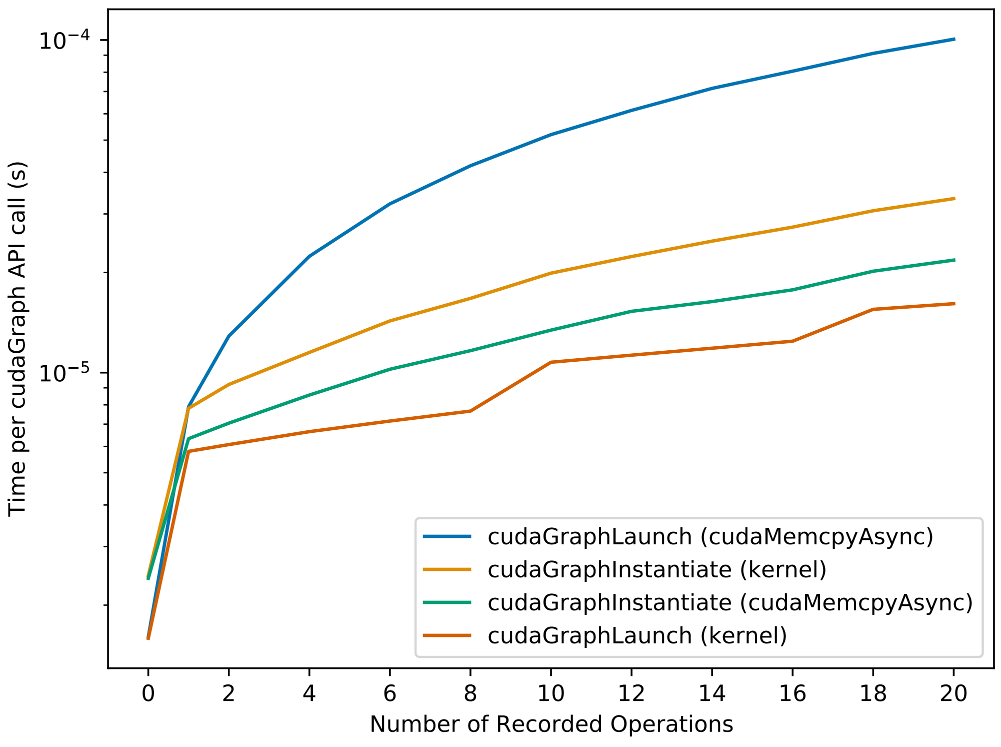


Figure 3.13: Cost of cudaGraphInstantiate (record) and cudaGraphLaunch (replay) when various numbers of runtime calls are captured. cudaGraphInstantiate is a one-time cost, and cudaGraphLaunch is incurred each time the sequence of operations is executed. cudaGraphLaunch provides no speedup for most runtime calls, but a 7.6× speedup for launching 20 kernels.

improvement over “far” GPUs.

Some detailed measurements are more relevant for system integrators than application developers. When the CPU cache is flushed, GPU-to-CPU transfers can be doubled for sizes around the L3 cache size. Also measurable is “anisotropy,” where the same transfers in different directions have different performance. This observation is typically not relevant for applications, but may allow system developers to identify unexpected performance behavior.

Comm|Scope can also be used to measure some aspects of the CUDA runtime performance. Specifically, this chapter addresses the cost of initiating certain device operations, as well as attempts to amortize that cost with the cudaGraph API (successful), and OS threads (unsuccessful).

This chapter focused on the performance of the OLCF Summit platform specifically. Discussion of additional platforms can be found in Pearson et al. [35]. The Appendix describes how to retrieve the Comm|Scope code (and libscope) source code. In Chapter 4, the lessons from Comm|Scope are integrated into a 3D stencil library. Chapter 5 attempts to generalize to arbitrary MPI applications, and also shows how some of the quantitative results can be used in an MPI implementation. Section 7.2.1 discusses extending Comm|Scope to intra-node communication.

Chapter 4

3D Stencil Halo Exchange Library

This chapter describes how the results obtained in Chapter 3 inform the design of HPC stencil codes for heterogeneous computers. It also evaluates the effect of the design and explores how the design decisions can be automated, so application developers do not need to be experts in system configuration to achieve high performance.

- Fastest communication: heuristically select a fast communication method based on participating GPUs.
- Minimize run-time: use CUDA graph API to minimize CUDA kernel launch cost.
- Minimize run-time: use a single thread per rank to control GPU.
- High link utilization: all communication happens asynchronously.
- High link utilization: longer transfers before shorter, to overlap transfer with initiation.

In addition, the code makes the following algorithm-level optimizations that synergize with the system-level communication optimizations above:

- Hierarchical spatial decomposition to minimize communication.
- Elision of unneeded halo exchanges based on stencil kernel "shape".

Furthermore, the library automatically handles indexing to simplify

- Accessing memory from GPU kernels through grid coordinate.
- Overlapping GPU kernel execution with distributed data transfer.

This chapter motivates, describes, and evaluates a patch-based distributed stencil library developed by this author and first introduced in Pearson et al. [44]. Consideration is restricted to stencil codes on homogeneous systems, i.e., each group of individual resources has the same characteristics. This is consistent with a typical execution on current high-performance computing platforms, and sidesteps any complications from externally-imposed resource contention or a changing execution environment. Given these preconditions, the system properties can be measured once, and an effective static communication strategy can be created. It is possible that a static environment with different architectures and/or endianness on sending and receiving nodes could be created, while still maintaining similar performance characteristics. While this chapter and the next do not explicitly address that case, the discussion and findings still apply.

The rest of this chapter is organized as follows. Section 4.1 describes a general CUDA+MPI distributed stencil code. Section 4.2 describes challenges of using CUDA+MPI directly to implement the stencil halo exchange. Sections 4.3, 4.4, and 4.5 describe how the stencil library implements a fast CUDA+MPI halo exchange. Section 4.6 evaluates the library in the context of the Astaroth stencil code. Finally, Section 4.7 concludes.

4.1 Distributed Stencil Overview

Stencil computation is a fundamental formulation for solving differential equations using finite difference, finite volume, and finite element methods, which are used widely in high-performance computing (HPC) applications such as simulating fluid dynamics, magnetohydrodynamics (MHD), space weather predictions, seismic wave propagation, and others. The application domain is represented as a discrete grid; stencil codes iteratively update each gridpoint based on some function of its local neighborhood. The stencil *kernel* (distinct from GPU kernels) describes the weights that each quantity from the neighboring grid points contributes to the new value of the produced gridpoint.

Each gridpoint may have several *quantities* associated with it (e.g. temperature, pressure, partial derivatives, etc.). Each quantity is typically stored in an "structure-of-arrays" style, rather than interleaving the quantities for each gridpoint in a “array-of-structures” style. This can assist with efficient memory access during vectorization, as the same quantities for multiple gridpoints are contiguous in memory and can be accessed in a single large load or store. Furthermore, separate allocations also ensure that alignment requirements for different datatypes are met.

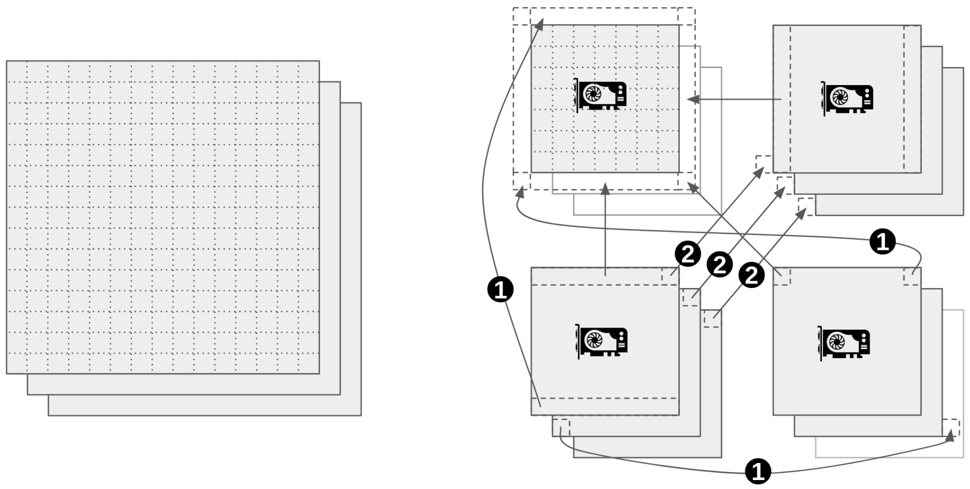
Modeling phenomena with high spatial and/or temporal resolution leads to enormous stencil grids. Current large-scale CPU simulations use up to 10^{10} grid points and 10^5 CPUs [45, 46], and are still orders of magnitude too small to capture phenomena of interest in available time and energy budgets. This has led to interest in using GPUs for stencil applications.

GPUs excel when there is limited data exchange, structured data reuse, and massive parallelism. Stencils exhibit all of these properties [47]. Once the stencil data is initialized on the GPU, it remains there without further exchange with the host. The data-reuse between neighboring gridpoints is (relatively) easy to leverage through shared memory and register queues in GPU kernels, and the grid points can be updated in parallel.

For large-scale stencil applications, the grid data may be much larger than a single GPU’s memory. Recent stencil codes use 1-8 quantities, a typical stencil radius of 3, and subdomains per GPU of 512^3, with a total domain size of around 10^{10} at most [47, 48, 49, 50].

Typically, the stencil grid is spatially decomposed into *subgrids*, which are placed in different memories. In each iteration, the exterior “shell” of these subgrids needs gridpoint values that are located in different memories. An explicit halo-exchange is used, where each subgrid includes a perimeter of *ghost points* representing grid points from neighboring regions. During each iteration, these ghost cells are updated with the new value from the corresponding neighboring subgrid. This update is called the *halo exchange*, and is the main focus of this chapter. These ghost gridpoints for the neighboring subgrid are stored in the same allocation as the real gridpoints, preserving locality for the stencil computation and keeping memory access regular during local computation.

Figure 4.1 shows an example of a distributed stencil. Figure 4.1a shows a full stencil grid with three quantities. In Figure 4.1b the grid is split among four GPUs, with each GPU holding all three quantities of a subgrid. Gridpoints near the edges in one subgrid are reflected as the ghost points of neighboring subgrids. Each quantity only exchanges with the corresponding



(a) Full stencil grid.

(b) Decomposed stencil grid with ghost points.

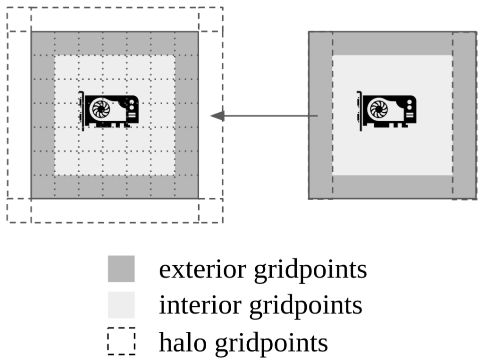
Figure 4.1: A stencil grid with three quantities distributed among four GPUs. Data from one subgrid is sent to the ghost region of the neighboring subgrid. Some exchanges reflect periodic boundary conditions (❶). Exchanges are done on a per-quantity basis (❷).

quantity in neighboring subgrids (❷). Communications may "wrap" around the grid perimeter (❶) for periodic boundary conditions.

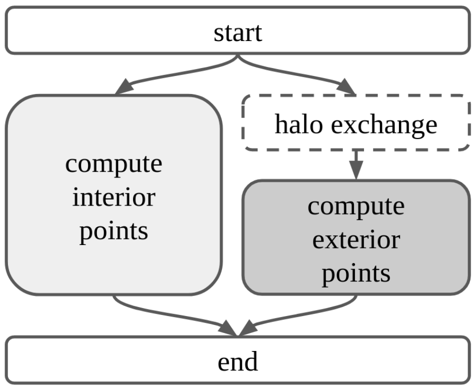
In this construction, there is a large amount of parallelism available. At a high level, the gridpoints allocated in each subgrid can be divided into three groups. The largest are interior gridpoints. The values needed to produce these points are entirely owned by the subgrid, and are not among the ghost gridpoints. An outer shell of exterior gridpoints is also owned by the subgrid, but cannot produce new values until the values from the neighbors arrive in the ghost points. The thickness of this shell is defined by the order of the stencil function.

The values for interior gridpoints may be computed immediately when the iteration starts, as the subgrid already contains all values needed for them after completion of the previous iteration. Since the interior points do not need the ghost points, the halo exchange can also immediately begin in parallel. Once the halo exchange has completed, the exterior points can be computed. Figure 4.2 summarizes this.

Fine-grained parallelism is available in the halo exchange. Each halo exchange can be broken up into $N_{quant} \times N_{dir}$ independent and parallel messages, where N_{quant} is the number of quantities and N_{dir} is the number of directions.



(a) Diagram of exterior, interior, and ghost gridpoints.



(b) Dependency graph for halo operations. Interior gridpoints may be operated on while halo exchange or exterior gridpoint kernels are running.

Figure 4.2: Dependency graph for stencil operations. Interior gridpoints only have a dependency on gridpoints already in the local subgrid. Exterior gridpoints require values from the ghost points to produce new values. The ghost gridpoints are provided by neighboring subgrids during halo exchange.

Likewise, the exterior gridpoints can be correspondingly divided into groups according to which ghost points they need from which messages. These exterior gridpoints could be launched immediately when the corresponding ghost points are received, without waiting for the entire halo exchange to complete.

4.2 Challenges with CUDA+MPI Stencil Codes

Emerging distributed HPC clusters feature nodes of multi-socket CPU and multiple GPUs, with CUDA and MPI libraries to exploit the hardware. These libraries are relatively low-level, featuring fine-grained control of the underlying platform and many options for communication and data allocation. Thus, implementing high-level data placement and communication strategies for large-scale stencil computations on such clusters is a challenging task.

Computational parallelism is straightforward to capture through GPU kernels (though much work is devoted to optimal implementations for various cases). The challenge from a systems perspective comes from high-performance combination of GPU and MPI communication primitives to facilitate data movement through the heterogeneous system. This design is informed by careful measurement of the primitives (Chapter 3).

In the last decade, CUDA-aware MPI implementations have allowed GPU-resident data to be passed to MPI operations. This simplifies CUDA+MPI applications, as the developer no longer needs to manage CPU-GPU data transfers. GPUDirect [51] has promised to accelerate these operations by allowing GPUs and NICs to interact directly without staging data through the CPU. Despite that, careful use of user-facing functions can surpass the performance of these abstractions.

MPI does not feature a primitive that directly maps to stencil communication, though it does offer some building blocks. MPI datatypes can be used to describe the (mostly) non-contiguous data that needs to be exchanged between subgrids (this is discussed further in Section 5.1.3). This allows application code to operate above the abstraction of messaging with individual bytes, which simplifies the code and allows the MPI implementation to provide high-performance handling of non-contiguous types. There would be at least one datatype per equivalent halo region. For example, depending on how the MPI communication routines are invoked, the +x and -x face (the subgrid surfaces whose normal vectos are the positive and negative directions of the x-axis) may be able to share the same MPI derived datatype but operate with different starting addresses.

MPI collectives allow all participating ranks to send at most a single message to all other ranks. The simplest form (MPI_Alltoall) restricts all ranks to send/receive the same count and type of data to all other ranks. MPI_Alltoallv allows each rank to send/receive a different count from each rank, and MPI_Alltoallw further relaxes each rank to send/receive a different type from each other rank. When combined with derived datatypes, MPI_Alltoallw is the most natural collective to use, as each rank can exchange the corresponding halo region datatype with the corresponding neighboring rank. MPI_Alltoallv can be used if MPI_Pack/Unpack is first used, explicitly transforming each non-contiguous datatype into a flat buffer of MPI_TYPE_PACKED. MPI_Alltoall can only be used if a separate MPI_Alltoall call is used for each different size message.

Unfortunately, all collectives only allow data to be sent/received from a single source/destination buffer, meaning that if quantities are stored in separate allocations, then multiple collectives must be used. Furthermore, in stencils with periodic boundary conditions, a pair of ranks may need to exchange data along more than one "direction". This is distinct from the common case, where two ranks would exchange a single halo region in a single direction. This can be handled by using MPI_Pack and MPI_Unpack to place multiple datatypes into a single buffer, or by building an MPI_Struct type to combine the two types.

Furthermore, stencil is not a good fit for collectives because most ranks will not exchange data. For example, in a 3D stencil, each subgrid will have at most 26 neighbors regardless of how many ranks are present. For very large decompositions, substantial time in the collective call can occur iterating over ranks that exchange no data. MPI introduced "topologies" to handle this case. First, the MPI Cartesian topology simplifies determination of neighboring ranks in a regular grid. It only operates directly on coordinate directions, but through multiple directional shifts (e.g. up, then left), diagonal ranks can also be determined. MPI graph topologies likewise allow the construction and query of arbitrary neighbor relationships. For each topology there is a collective operation corresponding to the ones described above, where only neighbors participate. Theoretically, this fixes the sparsity in the collectives.

These obstacles have driven many attempts to create distributed stencil communication frameworks. Fundamentally, the library described in this work represents a comprehensive effort to automate, combine, and evaluate partially realized communication techniques used in previous stencil works. Compared to prior work, it introduces automatic communication specialization, flexibility under mappings of GPUs to MPI ranks, and evaluation of the communication performance specifically thanks to these techniques. It combines those techniques with benchmark-driven design, node-aware data placement and communication overlapping. See Chapter 6 for a more thorough discussion of related work.

4.3 Grid Partitioning

The stencil library uses a two-level hierarchical recursive bisection algorithm common for this type of problem [52, 53]. Consider a system with N nodes and P GPUs per node. First, the stencil grid is evenly partitioned into N

node-level subgrids to minimize the total off-node communication volume. The algorithm ensures these subgrids are as cubical as possible, minimizing the exterior-to-interior (Section 4.1) volume, and therefore the required inter-node communication per gridpoint.

The same algorithm is applied to each node-level subgrid to further subdivide it into P GPU-level subgrids, again minimizing the exterior-to-interior volume ratio and the required *inter-GPU* communication (subject to the already minimized inter-node communication).

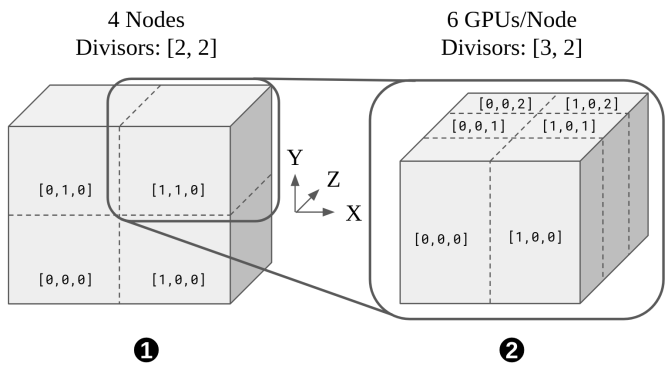


Figure 4.3: Hierarchical partitioning of the stencil grid into node-level subgrids (❶) and further into GPU-level subgrids (❷). In this example, four nodes are partitioned into 2x2 along the X and Y dimensions. The resulting subgrid is partitioned among six GPUs by 3 in the z dimension and 2 in the x dimension. Each partition has a three-dimensional node and GPU index.

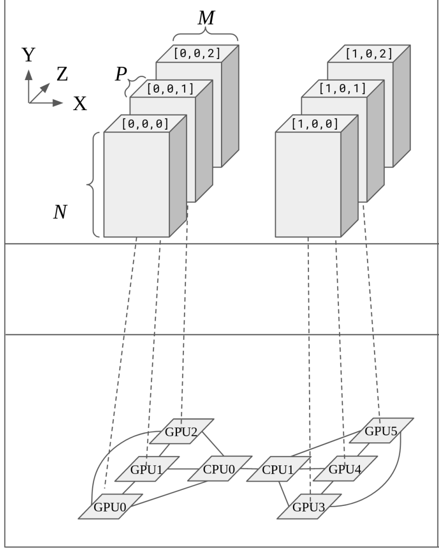
Figure 4.3 shows an example of the hierarchical decomposition of a stencil grid for four nodes with six GPUs per node ($N = 4$, $P = 6$). First, the recursive bisection scheme is applied to the whole grid at the node level (❶). The grid is largest in the x-dimension, so the grid is divided by the largest prime factor of 4, which is 2. After that division, y is the largest dimension, so the grid is further divided by the next prime factor, again 2. This yields four subgrids, each with a three-dimensional node index. The exterior volume of these subgrids is minimized given the requirement of four equally sized subgrids.

Then the recursive bisection is applied to each node subgrid at the GPU level (❷). For six GPUs, the prime factors are three and two. The longest dimension of the node subgrid is z then x, so the node subgrid is divided along those axes by three and two, respectively, to yield the subgrids that will be assigned to each GPU (Section 4.4).

Each subgrid therefore has a 3D node and 3D GPU index. These indices are unique for each subgrid, and can be used to determine which subgrids need to communicate with which others. For example, the -x neighbor of subgrid might be [[1,1,0],[1,0,0]] is [[1,1,0],[0,0,0] (same node, -1 in x dimension of GPU index), where the first triplet is the X,Y,Z coordinate in the 3D node space, and the second is the coordinate in the 3D GPU space.

4.4 Subgrid Placement

After partitioning, the stencil library assigns each subgrid to a GPU. One node-level subgrid is assigned to each node. The stencil library does not attempt to evaluate node proximity, so node subgrids are assigned to nodes arbitrarily. This is because the OLCF summit system has a full-bandwidth fat-tree network, so the bandwidth between any pair of nodes is equal.



Communication Matrix	$$\begin{array}{c|cccccc} & [0,0,0] & [1,0,0] & [0,0,1] & [1,0,1] & [0,0,2] & [1,0,2] \\ \hline [0,0,0] & - & rNP & rNM & rrN & 0 & 0 \\ [1,0,0] & rNP & - & rrN & rNM & 0 & 0 \\ [0,0,1] & rNM & rrN & - & rNP & rNM & rrN \\ [1,0,1] & rrN & rNM & rNP & - & rrN & rNM \\ [0,0,2] & 0 & 0 & rNM & rrN & - & rNP \\ [1,0,2] & 0 & 0 & rrN & rNM & rNP & - \end{array}$$
Mapping	[0, 3, 1, 4, 2, 5]
Bandwidth Matrix	$$\begin{array}{c|cccccc} & GPU0 & GPU1 & GPU2 & GPU3 & GPU4 & GPU5 \\ \hline GPU0 & - & 100 & 100 & 64 & 64 & 64 \\ GPU1 & 100 & - & 100 & 64 & 64 & 64 \\ GPU2 & 100 & 100 & - & 64 & 64 & 64 \\ GPU3 & 64 & 64 & 64 & - & 100 & 100 \\ GPU4 & 64 & 64 & 64 & 100 & - & 100 \\ GPU5 & 64 & 64 & 64 & 100 & 100 & - \end{array}$$

Figure 4.4: Example communication matrix (w in quadratic assignment problem) and bandwidth matrix (element-wise reciprocal of d). The result is the mapping (f), where the subgrid for row (or column) i of the communication matrix is mapped to GPU $f(i)$. The entries in the communication matrix are given in terms of the subgrid size as well as the stencil radius r.

In contrast to the nodes within the system, the GPUs within a node do not have uniform bandwidth. Therefore, it may be desirable to place neighboring subgrids on GPUs that have fast interconnects between them. Within each node, this is modeled as a quadratic assignment problem (QAP). The quadratic assignment problem is concerned with assigning a set of P facilities to P locations, according to the *flow* between the facilities and the *distance* between the locations, with the goal of placing facilities with high flow close to one another. This is analogous to placing subgrids with high exchange volume on GPUs that have high communication bandwidth. The assignment is a bijection f between facilities and locations. Let real-valued square matrices w and d represent the flow between facilities i and j, and the distance between locations i and j, respectively. Then, the QAP minimizes the cost function

$$\sum_{i,j<P} w_{i,j} d_{f(i),f(j)}$$

the sum of the flow-distance products under f.

The flow matrix entries are the number of bytes of data exchanged between GPU subgrids, and the distance matrix entries are the element-wise reciprocal of a matrix B which captures the bandwidth of GPUs i and j in $B_{i,j}$. Figure 4.4 summarizes the construction of the matrices, and gives an example mapping. The CUDA driver provides the Nvidia Management Library `libnvidia-ml`, which can be used to infer the connection and bandwidth between GPUs in a system The quadratic assignment problem is NP-hard. In this work, we simply check all possible subdomain-GPU mappings on each node. Since the number of GPUs in a node is typically small, the cost of exhaustively searching all combinations is acceptable.

Figure 4.4 summarizes the scheme. On each OLCF Summit node, six subgrids are assigned. Their communication requirements (QAP flow) depend on their logical position within the grid. The QAP distance is determined by the GPU bandwidth. On Summit, each GPU triplet is connected by 2x NVLink2, for 100 GB/s bidirectional bandwidth. Any connection across the x-bus is limited to 64 GB/s. The element-wise inverse of the bandwidth is the distance, and the mapping is the bijection delivered by the QAP formulation.

4.5 Specialization

Once the compute region has been partitioned (Sec. 4.3) and assigned to GPUs according to the theoretical communication performance (Sec. 4.4),

Table 4.1: Summary of requirements for communication methods. "✓" means the communicating subregions must share the corresponding topology for the communication method to work. "Preference" refers to the heuristic preference for that communication method, if all requirements are met (1 is highest). "Async" refers to how the library implementation allows multiple communications to be overlapped.

		Same...				
Method	**Preference**	**...GPU**	**...rank**	**...node**	**Async**	**Notes**
Kernel	1	✓	✓	✓	native	peer access
Memcpy	2	×	✓	✓	native	peer access
ColocatedMemcpy	3	×	×	✓	state machine	peer access
CudaAwareMPI	4	×	×	×	native	CUDA-Aware MPI
Staged	4	×	×	×	state machine	

a fast communication method is selected based on the physical (node) and logical (rank) location of the two GPUs.

In general, the exchange operation consists of taking the (possibly) non-contiguous boundary gridpoints from the interior (non-ghost points) of the source subgrid, packing it into a contiguous buffer, sending that buffer to the destination GPU, and unpacking that buffer into the appropriate exterior of the destination subgrid. Figure 4.5 shows an example of a pack operation on a 3D region. In this example, we consider an XYZ storage order, yielding a non-contiguous storage for the 3D region shown. The result of the pack operation is to copy that data into a contiguous buffer.

In order to support high-performance exchanges in a variety of system configurations, the library implements five communication methods. The methods are selected appropriately for each sending and receiving pair of subgrids. All methods are asynchronous, allowing them to be freely overlapped, even within a single process. Table 4.1 shows the communication methods and when they apply.

4.5.1 Baseline CUDA-aware MPI Communication

The baseline for the stencil library performance evaluation is a state-of-the art approach where every halo exchange uses a single CUDA-aware MPI_-Isend/MPI_Irecv with each other rank it needs to communicate with. This places the burden of overlapping and optimized communication on the MPI implementation instead of the application code.

This "CUDA-aware" method shares the same general structure as all CUDA-

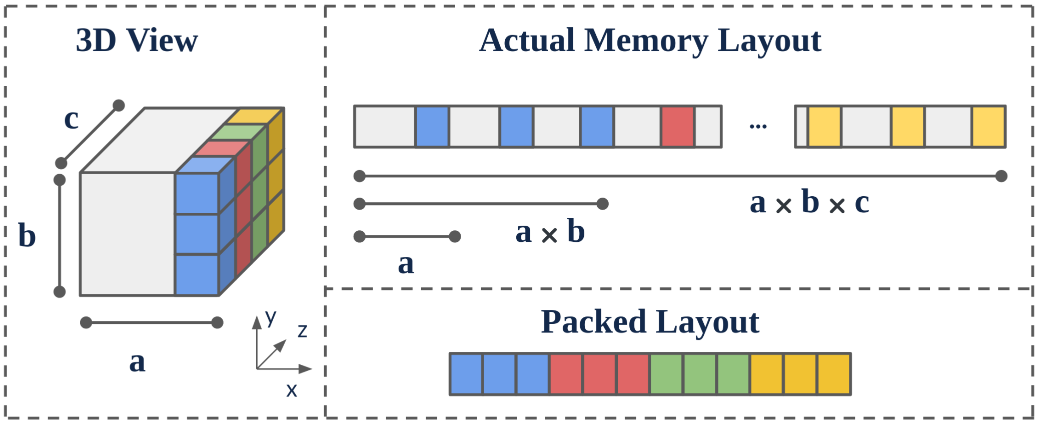


Figure 4.5: Example of packing for a 3D region. In general, the linear storage order of the subdomain in memory causes the elements of the 3D region to be strided. The pack operation places only those elements in a dense buffer with some predetermined order.

+MPI communicators in this work. Paired *sender* and *receiver* objects are created on the source and destination ranks to manage both ends of the communication. Therefore, a pair may handle the communication of more than one direction if the grid decomposition and boundary conditions cause a pair of ranks to be neighbors in multiple directions. This maximizes the size of the messages when the grid decomposition does not give each subgrid 26 unique neighbors, but also reduces the number of concurrent messages.

Figure 4.6b shows the paired sender and receiver objects for the baseline CUDA-aware method. Data from each quantity is packed (A) into a single buffer on the source GPU. A single kernel is invoked to pack all quantities. Since the shape of this halo region is the same in all quantities, the same kernel launch parameters offer good performance for all quantities. This also means that only the kernel launch latency is accrued once for each direction, instead of once for each combination of of quantities and directions. MPI_-Isend (B) is initiated after the data is packed.

The receiver does the reverse, first initiating an MPI_Irecv (C) and then unpacking the data into the corresponding quantity arrays on the destination GPU (D). Figure 4.6a shows the sender and receiver object decomposed into two states. Section 4.5.5 describes how these states contribute to CUDA+-MPI communication overlap.

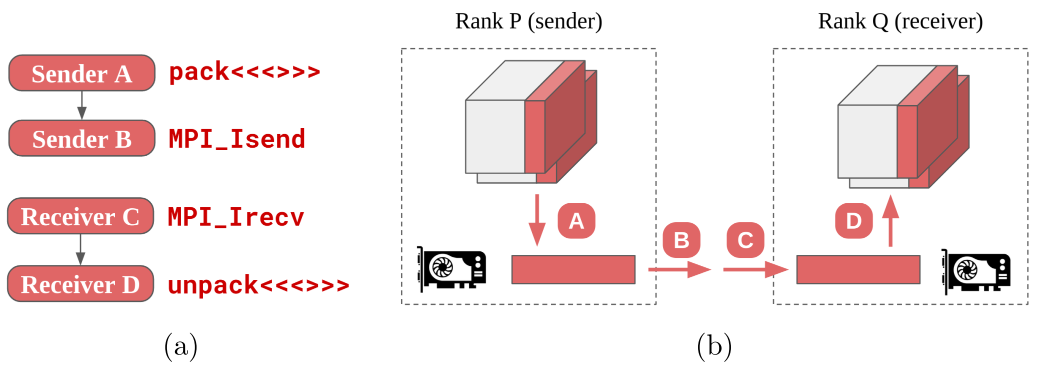


Figure 4.6: CUDA-aware MPI communicator, showing state transitions (a) and data flow (b). Here, the CUDA-aware MPI implementation is responsible for moving data between GPUs. Data from each quantity is packed (A) into a buffer on the source GPU and an MPI_Isend (B) is initiated. The receiver starts by initiating an MPI_Irecv (C) and then unpacks the data into the corresponding quantity arrays on the destination GPU (D).

4.5.2 "Staged" Communication

The "staged" communication method is the foundation upon which further specializations are applied. Instead of relying on the CUDA-aware MPI implementation to manage device data, the staged method uses CUDA APIs to explicitly transfer data between the GPU and host, and uses MPI to move data between ranks. This applies both to intra- and inter-node transfers.

Figure 4.7b shows an outline of this method. Data is packed just like the CUDA-aware communicator (Section 4.5.1). Once all data has been packed, the contiguous buffer is copied (A_2) to pinned memory on the source CPU, and then an MPI_Isend (B) is initiated. This is in contrast to the CUDA-aware method, where the MPI_Isend was directly invoked on the packed GPU buffer. The receiver starts by initiating an MPI_Irecv (C), copies the received data to a buffer on the destination GPU (D_1), and unpacks the data into the right location in GPU memory (D_2). One would expect staged communication to perform worse than the baseline unless CUDA-aware MPI is poorly implemented, hence "specializations" instead of "optimizations." In practice, the staged method is sometimes superior, as discussed in Section 4.6.

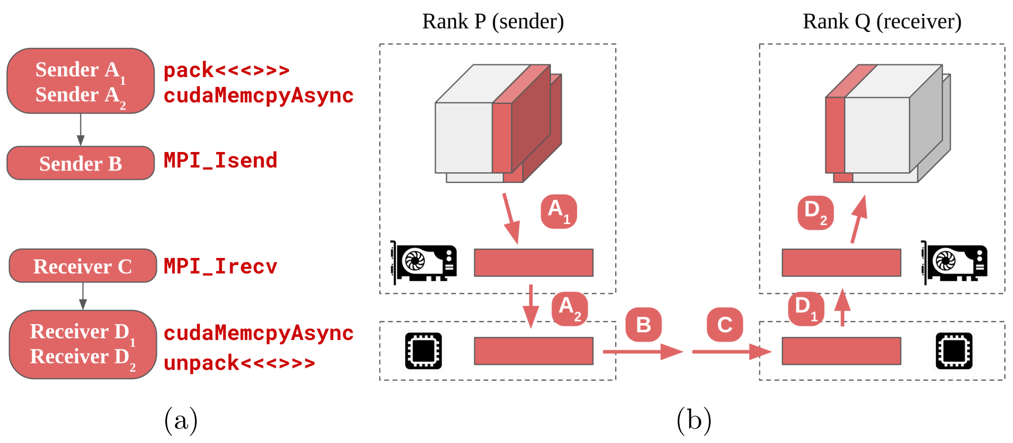


Figure 4.7: Staged CUDA+MPI communicator, showing state transitions (a) and data flow (b). Data from each quantity is packed (A_1) into a buffer on the source GPU, copied (A_2) to the source CPU, and then an MPI_Isend (B) is initiated. The receiver starts by initiating an MPI_Irecv (B), copies the received data to a buffer on the destination GPU (D_1), and unpacks the data into the right location in GPU memory (D_2).

4.5.3 "Colocated" Communication

When two ranks are on the same node, data can be transferred directly between GPUs in different address spaces without passing that data through MPI. Figure 4.8 shows a diagram of the transfer method.

In the staged transfer method, MPI provides two roles: first, moving data between the source and destination address spaces, and second, blocking the receiver until data arrives. In the colocated receiver, data is moved directly between GPUs through cudaMemcpyPeerAsync. The source requires a pointer to the destination buffer that is valid in the source address space. The receiver uses cudaIpcGetMemHandle to get an opaque handle to its GPU buffer, and sends that to the host through MPI. Synchronization between sender and receiver is achieved through a single CUDA event, which is shared between ranks in a similar manner, through cudaIpcGetEventHandle. MPI is still used to ensure the receiver does not query the CUDA event before the sender has recorded the event, though only a single-byte message is sent, instead of all the data.

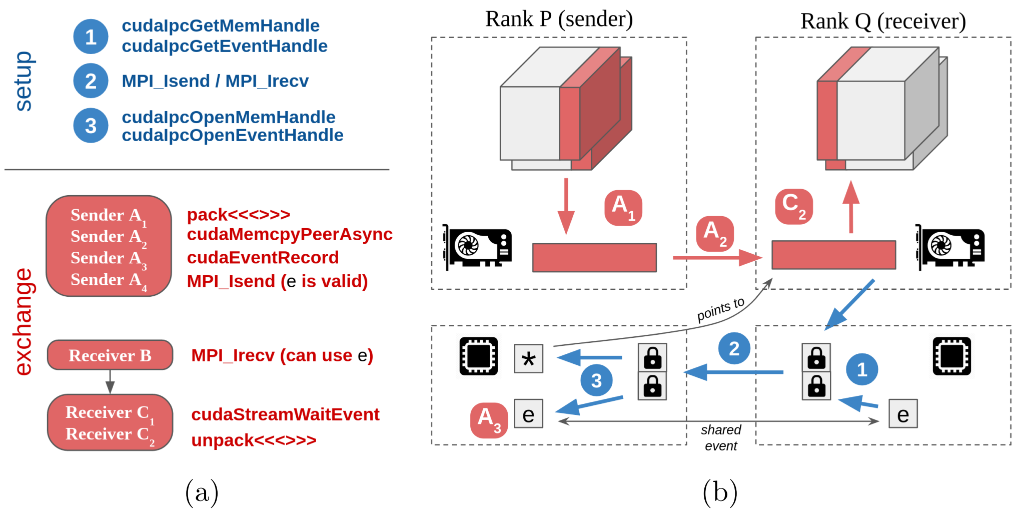


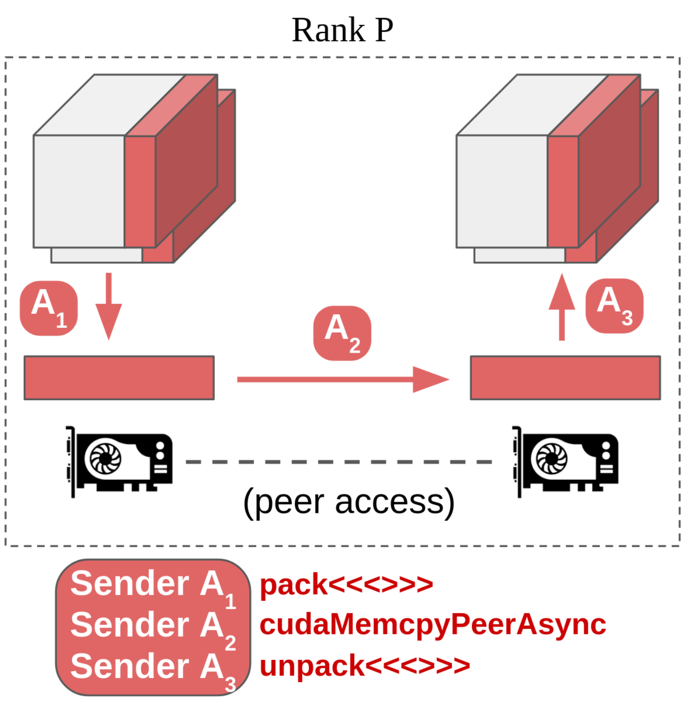
Figure 4.8: Colocated CUDA+MPI communicator, showing setup and receiver state transitions (a) and data flow (b). During application initialization, a pointer to the buffer on the destination GPU is passed to the source rank through MPI and the cudaIpc* family of functions. Likewise, a single CUDA event gets a handle in each address space. During each exchange, data is packed (A_1) into a buffer on the source GPU then copied (A_2) directly to the destination GPU using CUDA. The source rank records in the event that the copy has been issued (A_3), and then sends a 1-byte MPI message to the destination rank (A_4), letting it know the event is valid. The receiver starts by initiating an MPI_Irecv (B), where it waits for the source to start the transfer. Once the signal is received, it blocks execution of the unpack kernel (C_2) until the event fires, which means the copy is done (C_1).

4.5.4 “Peer” and “Kernel” Communication

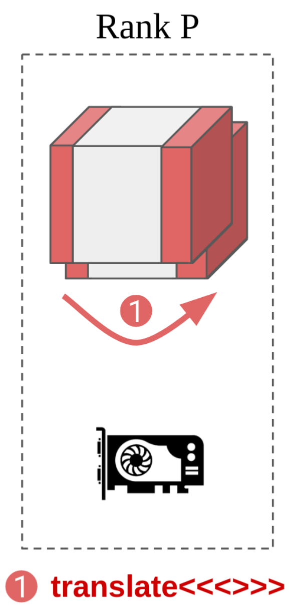
When two GPUs are in the same rank, they share an address space and many complexities of the *colocated* method are avoided. Figure 4.9 summarizes the two methods. For two different subdomains managed by the same MPI rank, data is transferred between GPUs with cudaMemcpyPeerAsync. When one subdomain is its own neighbor, a GPU kernel is used, keeping the data in-memory.

4.5.5 Overlapping and State Transition Engine

Overlapping communication (Section 3.3.4) is crucial for achieving good performance. Overlapping communication is achieved by implementing all trans-



(a)



(b)

Figure 4.9: Peer CUDA communicator (a) and Kernel CUDA communicator (b). For the *peer* communicator, both GPUs are in the same MPI rank, so that rank is both the sender and receiver, and so sender/receiver synchronization is required. The data is packed into a buffer on the source GPU (A_1), copied to the destination GPU (A_2), and then unpacked (A_3). These operations are inserted into the same stream to order them. The *kernel* communicator only applies when a GPU is both the source and destination. A single kernel is used to move the data directly within the memory of that GPU.

fers asynchronously, even when the CUDA and MPI APIs that make up those methods have a synchronous relationship. Each *sender* and *receiver* object is implemented as a finite state machine, where necessary, as part of the library code running on the CPU. On the send side, this allows each sender to initiate the asynchronous packing operation on the GPU, then yield so that the next sender may begin. Once all sends have been initiated, the library repeatedly polls all the senders in turn, checking if their GPU operations have completed by querying the corresponding stream. When a GPU operation has completed, the second asynchronous operation (MPI_Isend for CUDA-aware method, cudaMemcpyAsync for staged method, and cudaMemcpyPeerAsync for colocated method) is initiated. This process repeats until all senders have initiated their GPU and MPI operations, and the send-side work is finished. During this time, the sender process is fully occupied with the repeated checking of all remaining unfinished senders. In this way, each send operation can execute concurrently with maximal overlap of all operations. Each sender and receiver object maintains its own high-priority CUDA stream (Section 2.3) to prevent spurious scheduling delays that increase synchronization wait time.

Furthermore, transfers that are expected to be slower are initiated before transfers that are expected to be faster. The transfers are initiated in reverse order of preference from Table 4.1, and within each method, from largest size to smallest size. This allows the CPU cost of initiating the faster transfers to be incurred while the slower transfers are progressing.

4.5.6 CUDA Graph API

Each iteration of the distributed stencil grid requires a halo exchange. Each of these halo exchanges involves the same amount of data moving between the same memories to and from the same allocations – i.e., their packing and unpacking CUDA kernels are launched with the same arguments.

The stencil library uses the cudaGraph* API family to accelerate these repeated operations. Section 3.3.7 shows microbenchmarks of how the cudaGraph API can accelerate CUDA runtime operations. Before the first halo exchange, the relevant kernel operations for each sender and receiver object are recorded using cudaStreamBeginCapture, cudaStreamEndCapture, and cudaGraphInstantiate. This recording operation does some of the necessary kernel launch work ahead of time, so that future invocations can be faster. Then, when the time comes to actually invoke those kernels, cudaGraphLaunch is used to start the actual pack operations with lower latency.

4.6 Astaroth Evaluation

Astaroth [49] is a 3D stencil code which simulates stellar dynamics. Each grid point has eight double-precision quantities, and each GPU is responsible for a 256^3 cubical subgrid. It uses a three-step Runge-Kutta integration scheme, where a full stencil iteration consists of three full halo exchanges and three integration kernel invocations. This section uses the stencil library to implement the communication for the Astaroth grid.

Astaroth maintains its own implementation of the stencil halo exchange code. Each quantity is maintained as a pair of "in" and "out" buffers, where the stencil kernels are applied to the "in" buffer to produced values in the "out" buffers. These "vertex buffers" are not generally exposed to the application code directly, as Astaroth provides a domain-specific language (DSL) in which the user can describe their stencil kernel, and Astaroth will generate the appropriate CPU/GPU kernel code which accesses those buffers, as well as the communication code to handle halo exchange. The stencil communication library described in this chapter manages the gridpoint data itself to facilitate optimized communication. In the modified Astaroth code, the stencil communication library reads the stencil kernel and grid parameters from the Astaroth configuration data. The Astaroth data allocation code is then replaced with a command to the stencil library to allocate the gridpoint data appropriately. The Astaroth vertex buffer objects are then overwritten with pointers to the corresponding internal buffers of the stencil communication library. Then, the application can use the existing Astaroth interface to apply the kernels to the gridpoint data, and use the stencil communication library to handle the communication.

In all experiments in this section, each GPU handles a 256^3 cube of gridpoints, for a total of 2 GB of gridpoint data (current and next values). This ensures that there is a bijection between ranks in all experiments, and that the communication volume between neighbors is identical. Including the stencil order of three, each quantity allocation logically becomes 262^3 8-byte words, or $2096 \times 262 \times 262$ bytes. The 3D allocation has a pitch of 512 bytes, so each quantity allocation is actually $2560 \times 262 \times 262$ bytes, or 167.6 MiB, for a total of 2.6 GiB, including halo space and unused space for row pitch. A 512^3 cube would be approximately eight times larger, and would exceed the 16 GiB of GPU memory capacity.

Astaroth uses Morton ordering [54] to assign 3D subregions to ranks and only supports powers-of-two numbers of ranks. For this reason, the Astaroth code cannot fully utilize the resources of the Summit system, which has six GPUs per node. Furthermore, this decomposition strategy is different from the one used in the stencil library, which explicitly groups subgrids onto nodes for locality.

To maintain a like-for-like comparison, both the unmodified Astaroth implementation as well as the implementation modified to use the stencil communication library maintain 256^3 gridpoints per GPU. The unmodified Astaroth distributed grid extents are chosen to maintain as cubical a shape as possible. Due to the hierarchical nature of the stencil communication library, a modified approach is required to maintain a cube of gridpoints on each GPU that is 256 in each dimension. To understand why, consider the 2-node, 6-rank-per-node scenario. A natural cube would be approximately $256 \times 12^{\frac{1}{3}} = 586$ gridpoints on each side. This is first split in the X dimension among two nodes, yielding $293 \times 586 \times 586$ each. Then, it is split within the node by 3 and 2 in the longest dimensions, yielding a final shape of $293 \times 195 \times 293$. While the one, two, and four ranks-per-node configurations could have been identical to the corresponding unmodified Astaroth configuration, that would prevent easy comparison of different configurations within the stencil library itself, as a largely different decomposition would exist for the six-rank-per-node configuration. As a consequence of the different grid extents, the configurations with a small number of nodes will necessarily have different proportions of on-node and off-node communication between the two implementations. However, once eight nodes is reached, that difference disappears.

4.6.1 Flaws in Spectrum MPI CUDA-aware Implementation

The Spectrum MPI 10.3.1.2 implementation on the Summit system does not provide a good platform for CUDA-aware MPI application optimization. Details about the design of the implementation are not available but some information can be gleaned from using a CUDA profiler like Nvidia Nsight Systems [55]. Spectrum MPI routes CUDA operations from CUDA-aware MPI transfers into several different CUDA streams. Some on-node transfers are turned into device-to-device transfers using cudaMemcpyAsync. These transfers are either placed into the default stream, or a second created stream. Some off-node transfers are implemented using device-to-host and host-to-device transfers, presumably with an intervening CPU-to-CPU MPI operation. Two additional streams are dedicated to host-to-device and device-to-host transfers respectively. These transfers are never placed into the default stream.

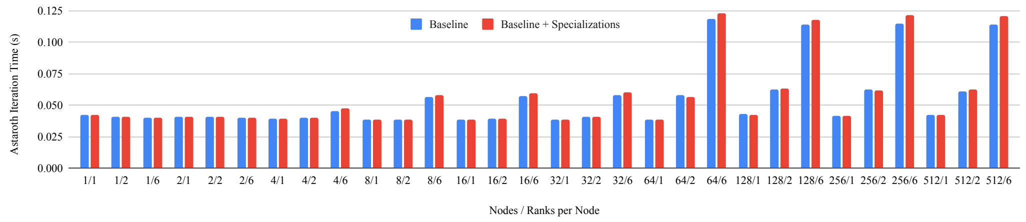


Figure 4.10: Performance of the Colocated and Kernel methods on top of the baseline Spectrum MPI CUDA-aware communication method. Due to spurious synchronization introduced by the Spectrum MPI implementation, the optimizations provide no benefit.

Each device-to-device transfer that Spectrum MPI places in the default stream is followed by a cudaDeviceSynchronize. This has two effects. First, the default stream has special synchronization semantics with otherwise asynchronous CUDA operations, requiring explicit programmer effort to avoid, e.g. cudaStreamCreateWithFlags and cudaStreamNonBlocking. Second, the cudaDeviceSynchronize calls spuriously block other unrelated CUDA operations (from Spectrum MPI or the application) from occurring in parallel. This prevents application use of the GPUs from overlapping with Spectrum MPI's movement of data to and from the GPU. Figure 4.10 shows the Astaroth iteration time (both computation and halo exchange) for the baseline Spectrum MPI CUDA-aware transfer, and with the communication specialization described in Section 4.5. Applying the stencil library specializations on top of the CUDA-aware MPI method does not improve the performance, since the communications that do involve the CUDA-aware MPI still introduce synchronizations. The staged method avoids this problem by avoiding Spectrum MPI's CUDA-aware operations. Section 4.6.5 shows the effect of the staged method.

4.6.2 Node-Level and GPU-Level Data Placement

Sections 4.3 and 4.4 describe how node-level subgrids are created for each node, further subdivided into GPU-level subgrids for each GPU. Figure 4.11 shows the normalized Astaroth halo exchange (no compute) latency for three different placement schemes. "Baseline" refers to a linear assignment of subgrids to ranks. No node-level subgrids are created, so any neighboring subgrids appearing in the same node are purely by chance. “Intra-node Random” allows the creation of node-level subgrids, so all GPU subgrids within a node are guaranteed to be from the same node-level subgrid, and therefore highly localized. Within the node, however, the subgrid positions are randomized, so neighboring subgrids may have a slow link between them. “Optimized” allows the full placement scheme described in Section 4.4.

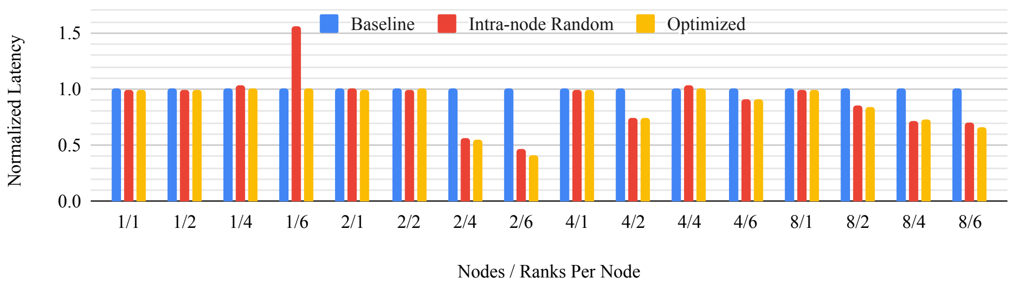


Figure 4.11: Normalized Astaroth halo exchange latency for baseline subgrid placement, intra-node random, and optimized placement. “Baseline” assigned each subgrid to the rank corresponding to the linearization of its index. “Intra-node” random creates node-level subgrids, but randomizes GPU placement within the node. “Optimized” solves the QAP to place nodes within the subgrid. For multiple nodes, the vast majority of the benefit comes from inter-node grouping rather than intra-node placement. To demonstrate the best-case scenario effect, communication specialization is enabled.

The single-node configurations (1/1, 1/2, 1/4, 1/6) show two effects. The first is that the default rank-linearized subgrid placement scheme happens to be the optimal one for this decomposition. Second, it shows that the effect of randomizing the placement only has negative effects when most GPUs on the system are utilized. For the 1/1 and 1/2 cases, there is no difference from default and random. For the 1/4 case, three GPUs in one triplet and a fourth GPU in another triplet are occupied. In this case, it does not matter which corner of the 2x2 arrangement of GPU subgrids is placed “far” away. For the six-GPU configuration (1/6), randomizing the intra-node placement has a 56% slowdown as neighbors are placed across slower interconnects.

The multi-node configurations show a different set of effects. For the one-rank-per-node configurations (X/1), all placements are identical. In the 2/2 configuration, the default arrangement happens to be the optimal one. For the other multi-rank configurations, most of the benefit comes from moving neighbors onto the same node, rather than arranging neighbors within the node. This is in contrast to the one-node case, where the intra-node placement had a large effect at six subgrids. For the multi-node case, most of the communication time is consumed by MPI, so there is little relative benefit for careful intra-node arrangement.

In summary, the vast majority of the benefit comes from inter-node grouping rather than intra-node placement.

4.6.3 Data Placement and Communication Method

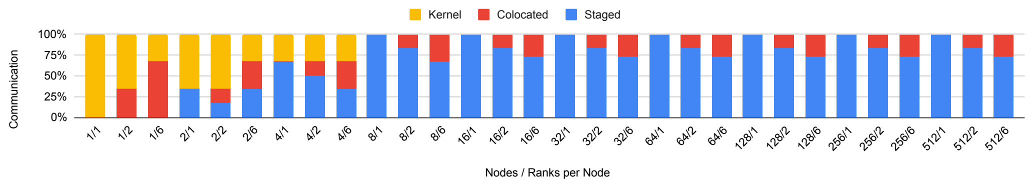
Section 4.6.2 showed that node-level data placement has a substantial effect on halo exchange time. This section examines why. Figure 4.12a shows the share of communication that goes through the staged, colocated, and kernel methods for different node/rank counts with baseline rank placement (Figure 4.12a) and optimized rank placement (Figure 4.12b).

For a single node and a single rank, all communication occurs through the kernel method. Each of the 26 directions is a periodic boundary condition that "sends" the halo region to the other side of the grid on the same GPU. When multiple subgrids are on the same node, a larger and larger share of the communication happens through the colocated method, as more data is exchanged with on-node neighbor subgrids. Since there is only one node to place the subgrids on, placement has no effect on the communication breakdown.

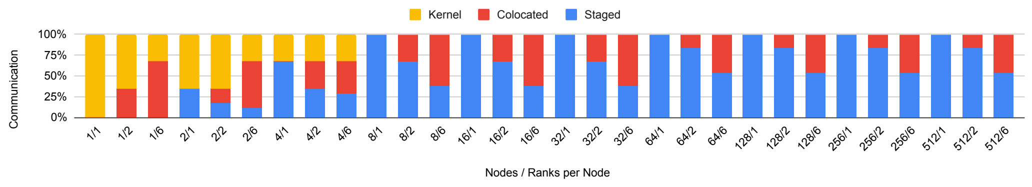
For two nodes, the grid is first decomposed into [2x1x1] node-level subgrids. When there is only one rank per node, data either goes off-node through the staged method, or stays in the same subgrid through the kernel method. Once multiple subgrids are placed on each node, some data stays within the node but moves to a different subgrid through the colocated method.

For two nodes with less than six ranks, the default placement and library-optimized placement are identical. This is because both methods yield the same placement for these particular grid sizes. Some of the communication always occurs over the staged method, as not all neighbors are on the same node. With six ranks, library-optimized placement shifts most of the staged bytes into colocated, showing that the subgrids have been rearranged to keep more communication on-node.

A similar shift from staged to colocated can be seen at all larger sizes where more than one rank is used per node. At 512 nodes and 6 ranks per node, the default placement has 27% of communication occurring through the faster colocated method, while the library-optimized placement increases that to 47% of the communication.



(a) Baseline placement.



(b) Proposed stencil communication library placement.

Figure 4.12: Communication amount by method for the Astaroth halo exchange with baseline or stencil library optimized placement. When more than one rank is on the node, optimized placement has a larger share of the communication to occur on-node, opening up the opportunity to optimize the communication further.

4.6.4 Data Placement and Iteration Time

Section 4.6.3 demonstrated that placing neighboring subgrids on the same node can allow for faster communication methods. This section further examines the performance effect of that shift. In contrast to Section 4.6.2, this section does not attempt to split the difference between inter-node and intra-node placement effects. All experiments in this section enable both types of placement, and are restricted to six ranks per node as that fully utilizes each node.

Astaroth iteration (three integration steps) time is determined by both the performance of the halo exchange and the GPU kernel time, which can be overlapped (Section 4.1) to some extent. Figures 4.13, 4.14, and 4.15 show the effect of data placement on the Astaroth iteration time. The figures show the effect when using the baseline CUDA-aware communication method, the staged method, and the optimized method (enabling the colocated and kernel communication shortcuts). All results are normalized to the iteration time when the baseline CUDA-aware MPI implementation is used.

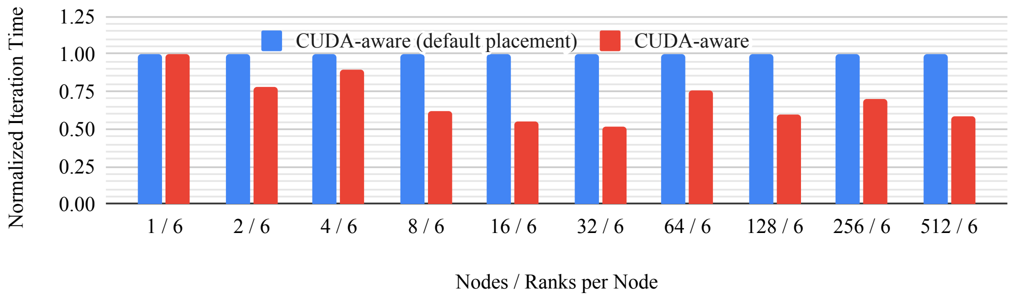


Figure 4.13: Normalized Astaroth iteration time with the default placement and the stencil library placement using the CUDA-aware communication method.

Figure 4.13 compares the baseline CUDA-aware method under the default rank mapping and when the stencil library placement is enabled. For a single node, there is no effect as there are not multiple nodes to map subregions to, and the CUDA-aware synchronization hides any effect from the intra-node placement. When multiple nodes are introduced, more neighbors are on the same node under the library placement, improving communication performance. At 512 nodes, this yields a 1.72× speedup.

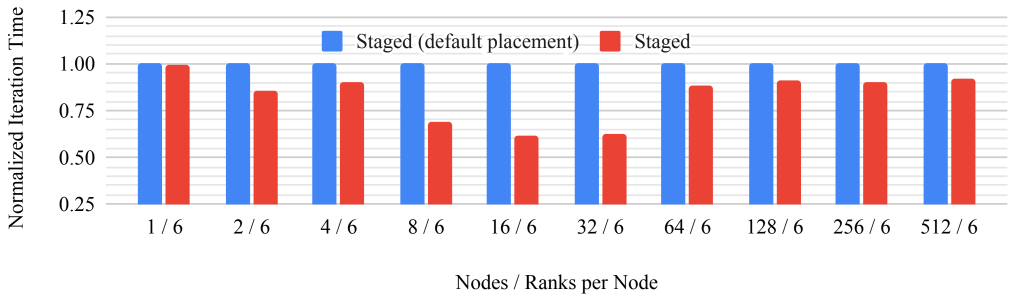


Figure 4.14: Normalized Astaroth iteration time, comparing staged with default placement and staged with library placement.

Figure 4.14 compares the staged method with both default and improved placement (normalized to the default). Note that each subgrid has 26 neighboring directions, and when there are fewer than 26 ranks, due to the periodic boundary conditions several of those directions will refer to the same neighboring rank. As described in Section 4.5.2, the staged communication method packs all gridpoints required by a single neighbor into a single message to that neighbor. Therefore, when there are fewer neighbors, there is smaller message concurrency.

Three different performance regimes are visible. For a “small” number of nodes (1-4), enabling placement has a small effect since much of the traffic is on-node in both cases. For a “medium” number of nodes (8-32), enabling placement has a relatively large effect. Enough of the communication is off-node that there is substantial opportunity for placement to bring it back on. In these scenarios, there are proportionally fewer off-node messages, so reducing them further with placement has a large effect.

When more than 64 nodes are used, enabling placement produces only a small speedup again. To understand why, consider the 32-node 6-rank case with a grid size of [3072 $\times$ 2048 $\times$ 512] (Table 4.2). This will be divided by [4x4x2] (32 nodes) to yield [768 x 512 x 256] for each node, which will be divided by [3 x 2 x 1] (6 GPUs) to yield the expected [256 x 256 x 256] per GPU. The nodes make a [4x4x2] cuboid, and since the z-extent is 2, each node’s +z and -z neighbors are the same. At 64 nodes, the node-level cuboid is instead [4x4x4] and each GPU has a unique neighbor in each of the 26 directions. Despite bringing much of the communication on-node, there is no placement which can avoid communication with 26 other nodes.

The staged scheme for the 1/6 through 4/6 configurations is limited more by the lack of parallelism than by the communication bandwidth. As a result, the placement optimization produces only a modest performance improvement for the staged method. For larger configurations with less than 64 nodes, there is more concurrency and the communication is more limited by the slow link within each node. Thus, the performance benefit of placement optimization is more pronounced. For the largest configurations, the performance is limited by the off-node bandwidth. At the 512/6 configuration, placement yields a 1.09$\times$ speedup for staged communication.

Figure 4.15 shows results similar to those of Figure 4.14, except with communication specialization enabled, instead of restricting to the staged method. At the 512/6 configuration, placement yields a 1.22$\times$ speedup. This is more than double the speedup for the staged method, since on-node

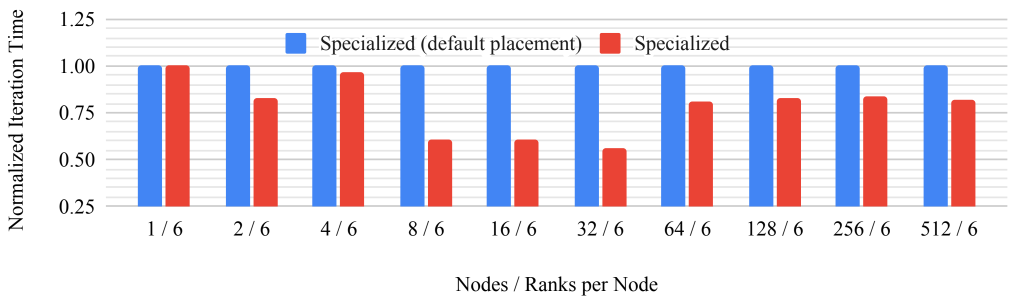


Figure 4.15: Normalized Astaroth iteration time, comparing CUDA-aware with default placement, specialized communication with default placement, and specialized communication with library placement.

communication is faster with specialization enabled.

4.6.5 Specialization and Iteration Time

This section considers the incremental effects of communication specialization once data placement is enabled. Figure 4.16 shows how the various optimized communication methods influence iteration time for different node and ranks-per-node configurations.

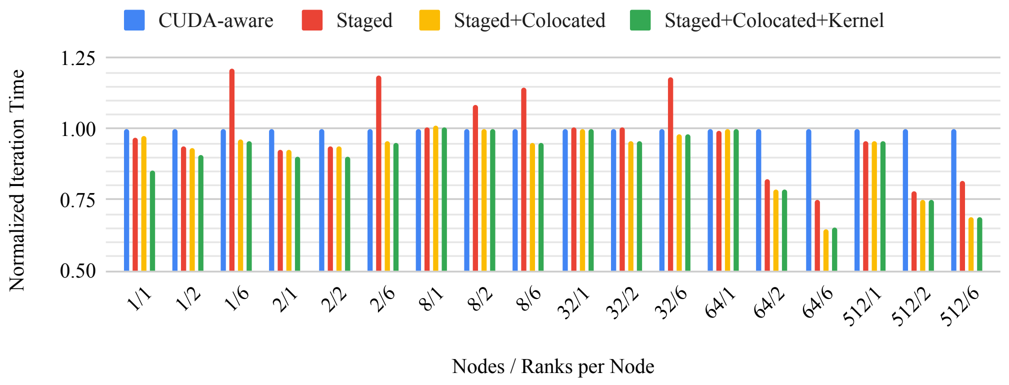


Figure 4.16: Astaroth iteration time normalized to the baseline CUDA-aware implementation with data placement for various node/rank configurations. Results for 4 nodes (not shown) are substantially similar to 2 nodes, 16 nodes (not shown) are similar to 32 nodes, and 128 and 256 nodes (not shown) are similar to 512 nodes.

First, switching away from the baseline CUDA-aware implementation to the staged method has varying effects. When the 26 neighbor directions are distributed among fewer than 26 neighbor ranks, switching to the staged method has the smallest improvement (or largest deficit). The baseline implementation introduces spurious synchronization, so when there are fewer messages, the effect of removing this synchronization is smaller. Once each subgrid communicates with 26 unique neighbors, the staged implementation is better able to exploit concurrency across messages.

The kernel optimization only has an effect when the number of nodes is small. This corresponds to decompositions where subgrids have self-communication.

The colocated specialization only has an effect when there is more than one rank per node, and in that scenario, it brings most of the further performance improvement.

There is an intermediate regime of 8-32 nodes where specialization does not bring any improvement. This seems to be the window where there are few enough messages that the baseline CUDA-aware's concurrency limitations do not negatively affect performance.

In the 512/6 node configuration, full specialization yields a 1.44× speedup after placement is enabled.

4.6.6 Overall Improvement

Figure 4.17 shows the results of combining placement and optimization. The observations made in Sections 4.6.4 and 4.6.5 are all visible here. In all cases, data placement combined with specialization yields speedup over the baseline CUDA-aware implementation. At the 512/6 configuration, total speedup is 2.5×.

Figures 4.18 and 4.19 show the effect of replacing the Astaroth communication code with the stencil library. Astaroth uses a Morton-indexed subgrid placement, capturing some locality among nodes, and uses cudaIpc* functions for intra-node communication. The stencil library manages to improve on the baseline with a more general placement algorithm that supports non-power-of-two process counts, and achieves better overlapping of intra-node and inter-node communication. In the 512/4 configuration, the halo exchange speedup is 1.48×.

The overall iteration time improvement vs. Astaroth is lower. For smaller

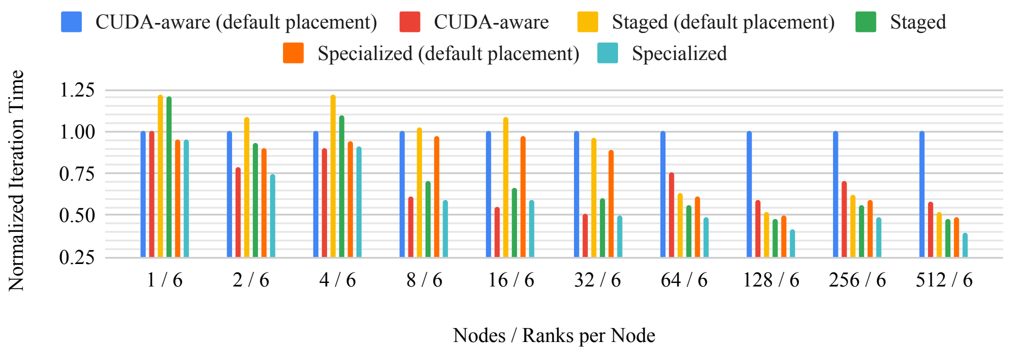


Figure 4.17: Joint effect of placement and specialization on Astaroth iteration time, normalized to the CUDA-aware communication with default subgrid placement. "Specialized" refers to enabling the staged colocated and kernel methods.

grids, the computation time dominates. At scale, the communication is more relevant, and the stencil library achieves 1.45× speedup in the 512/4 configuration, almost identical to the pure halo-exchange speedup. Geometric mean halo exchange speedup is 1.3×, and overall iteration time is 1.17×.

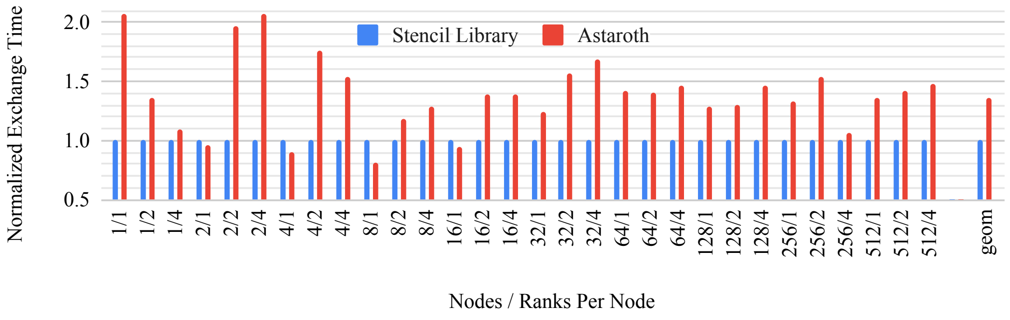


Figure 4.18: Astaroth halo exchange time, normalized to the stencil library. "geom" is the geometric mean of the corresponding series.

4.6.7 Test Simulation

Integration between the Astaroth integration kernels and the stencil library communication routines was verified through a rudimentary simulation. It allows for verification of correct physical properties, without introducing enough complex features to obfuscate the correctness of the exchange. In this way, it serves as an additional layer of verification to complement vari-

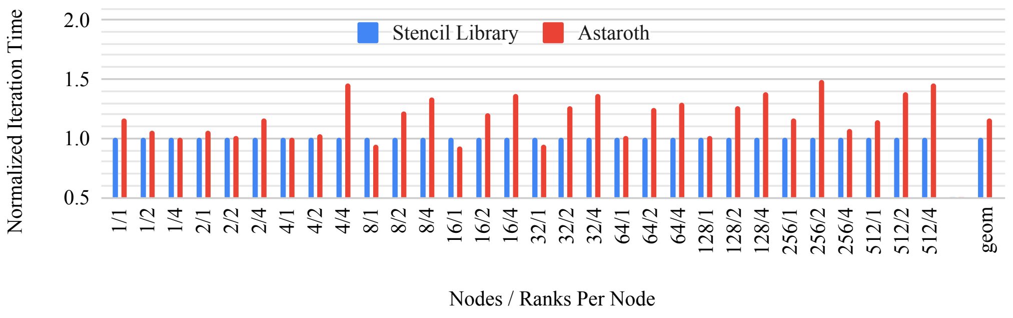


Figure 4.19: Astaroth iteration time, normalized to the stencil library. "geom" is the geometric mean of the corresponding series.

ous individual unit and functionality tests. A domain of 64^3 gridpoints per GPU was constructed. The origin is set at the center of the lower half (Z direction) of the domain in Figure 4.20. The entropy and magnetic field quantities of each grid point are initialized with a uniform random distribution in the range $[0, 1)$. The density quantity is initialized to a constant 0.5, and the three velocity components (X,Y,Z) are initialized to a "Gaussian explosion."

The velocity vector u is stored in three quantities u_x, u_y, and u_z, representing the X, Y, and Z components respectively. The velocity vector quantities u_x, u_y, and u_z at coordinate x,y,z are initialized in terms of polar coordinates into a pattern to facilitate visualization as follows:

The radius r is given by

$$r = \sqrt{x^2 + y^2 + z^2}$$

the polar angle θ is given by

$$\theta = \begin{cases} acos(z/r) & z \geq 0 \\ \pi - acos(-z/r) & z < 0 \end{cases}$$

and the azimuthal angle ϕ is given by

$$\phi = \begin{cases} atan(y/x) & x > 0, y > 0 \\ \pi - atan(-y/x) & x < 0, y \geq 0 \\ 2\pi - atan(-y/x) & x > 0, y < 0 \\ \pi + atan(y/x) & x < 0, y < 0 \\ \pi/2 & x = 0, y > 0 \\ 3\pi/2 & x = 0, y < 0 \\ 0 & x = 0, y = 0 \end{cases}$$

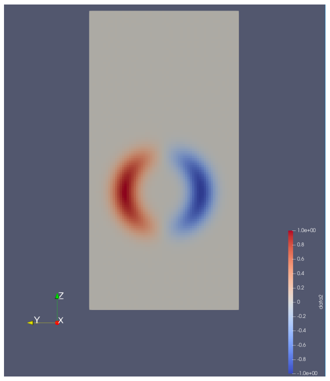
The radial magnitude $\bar{u}$ is given by

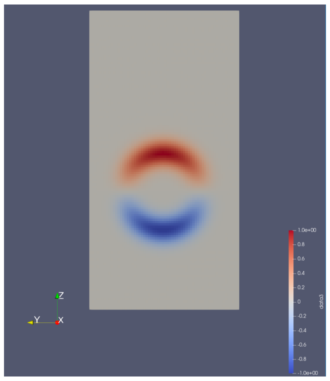
$$\bar{u} = A \times e^{-1 \times \frac{(r-R)^2}{2 \times W^2}}$$

where A controls the amplitude of the velocity, R controls the shell radius, and W the shell width. Finally, the initial condition values of the velocity vector components are given by

$$\begin{aligned} u_x &= \bar{u} \times \sin\theta \times \cos\phi \\ u_y &= \bar{u} \times \sin\theta \times \sin\phi \\ u_z &= \bar{u} \times \cos\theta \end{aligned}$$

Figure 4.20 shows a visualization of a small test case. Two GPUs participate, yielding a $64 \times 64 \times 128$ grid, with the largest extent in the Z direction. The explosion is centered on the -X face so that effects are visible on the surface; effectively, only the +X half of the explosion is in the simulated region. The explosion is positioned so that it crosses the boundary region covered by the halo exchange. Any errors in the halo exchange code will cause visible artifacts. Figures 4.20a and 4.20b show the Y and Z components of the velocity vector initialization. Figure 4.20c shows the result on the density after a single iteration. No artifacts are visible, demonstrating a successful integration of the halo exchange library code. Furthermore, the simulation results are qualitatively sound, and match the unmodified Astaroth results. The leading edge of the explosion where the stellar medium is compressed features higher density. The trailing edge features lower density.


(a) Velocity vector Y component ($t = 0$s).


(b) Velocity vector Z component ($t = 0$s).

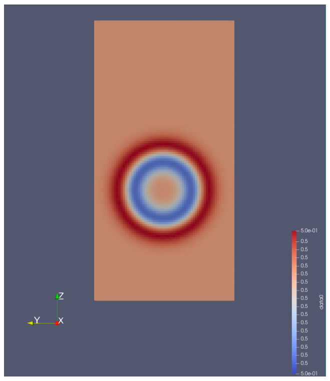

(c) Resulting density distribution ($t = 1 \times 10^{-8}$s)

Figure 4.20: Visualization of the -X face of the simulated region, showing velocity vector Y (a) and Z (b) components at simulation time $t = 0$, and density (c) after one iteration, at $t = 1 \times 10^{-8}$. The initial conditions represent a "Gaussian explosion" centered on the -X face of the simulated region. The X component of the velocity vector is not shown as it is only non-zero "inside" the simulated region.

4.7 Conclusion

This chapter described how some of the lessons from detailed multi-GPU communication measurement can be applied to an actual scientific application.

Specifically, the lessons were applied to a distributed stencil library, where the user provided the desired compute domain, data quantities, and stencil shape, and the library derives the highest-performance communication strategy based on the properties of the system

First, information about the stencil grid was used to partition the subgrid (Section 4.3). Then, an intra-node placement strategy was applied to maximize interconnect bandwidth utilization (Section 4.4). Once neighboring subgrids were placed on the same node, specialized communication shortcuts were implemented to help realize the theoretical bandwidth that drove the placements (Section 4.5).

The chapter then examined the extent to which those optimizations affected the actual application performance. The test-case was a large GPU-accelerated distributed 3D stencil code, Astaroth. At scale of 512 nodes with 3072 GPUs, subgrid placement was found to contribute to a speedup of 1.72× compared to the baseline CUDA-aware implementation, 1.09× for the staged method, and 1.22× with specialized communication enabled. With data-placement enabled, communication specialization was found to contribute a 1.43× speedup. Overall, the iteration time was improved by 2.5× over the baseline CUDA-aware implementation. Over a variety of configurations, the stencil library was able to improve existing Astaroth halo exchange time by geometric mean of 1.3×, and the iteration time by 1.17×.

In summary, there are two complementary pathways to speeding up performance. For the first pathway, it is crucial to minimize off-node communication. This was seen to be important in relieving the poorly performing CUDA-aware Spectrum MPI implementation of as much work as possible. It was also important when the on-node communication is fast, in the case of specialization. Using the staged method resulted in only a minor speedup.

The second approach is to provide fast on-node GPU-GPU communication methods. Thanks to the iterative nature of the stencil, direct GPU-GPU communication channels can be quickly configured at the beginning of the application, and then re-used each iteration.

Table 4.2: Stencil grid dimensions for Astaroth node / ranks per node configurations. These dimensions ensure that each GPU has exactly 256^3 grid points under the different grid decomposition strategies.

	Ranks	**Astaroth w. Stencil Library**			**Unmodified Astaroth**		
Nodes	**Per Node**	**X**	**Y**	**Z**	**X**	**Y**	**Z**
1	1	256	256	256	256	256	256
	2	512	256	256	256	256	512
	4	512	512	256	256	512	512
	6	768	512	256	–	–	–
2	1	512	256	256	256	256	512
	2	1024	256	256	256	512	512
	4	1024	512	256	512	512	512
	6	1536	512	256	–	–	–
4	1	512	512	256	256	512	512
	2	1024	512	256	512	512	512
	4	1024	1024	256	512	512	1024
	6	1536	1024	256	–	–	–
8	1	512	512	512	512	512	512
	2	1024	512	512	512	512	1024
	4	1024	1024	512	512	1024	1024
	6	1536	1024	512	–	–	–
16	1	1024	512	512	512	512	1024
	2	2048	512	512	512	1024	1024
	4	2048	1024	512	1024	1024	1024
	6	3072	1024	512	–	–	–
32	1	1024	1024	512	512	1024	1024
	2	2048	1024	512	1024	1024	1024
	2	2048	2048	512	1024	1024	2048
	6	3072	2048	512	–	–	–
64	1	1024	1024	1024	1024	1024	1024
	2	2048	1024	1024	1024	1024	2048
	4	2048	2048	1024	1024	2048	2048
	6	3072	2048	1024	–	–	–
128	1	2048	1024	1024	1024	1024	2048
	2	4096	1024	1024	1024	2048	2048
	4	4096	2048	1024	2048	2048	2048
	6	6144	2048	1024	–	–	–
256	1	2048	2048	1024	1024	2048	2048
	2	4096	2048	1024	2048	2048	2048
	4	4096	4096	1024	2048	2048	4096
	6	6144	4096	1024	–	–	–
512	1	2048	2048	2048	2048	2048	2048
	2	4096	2048	2048	2048	2048	4096
	4	4096	4096	2048	2048	4096	4096
	6	6144	4096	2048	–	–	–

Chapter 5

Non-contiguous Data Optimization for MPI

The main strength of the library-based approach in Chapter 4 is also its main weakness; with complete flexibility to define the halo-exchange interface comes the requirement for existing applications to be rewritten to use it. Due to the challenges and opportunities identified in the previous chapter, the Astaroth code took a similar route, implementing their own specialized communication layer to improve performance. In Section 4.6, the Astaroth [49] communication layer was replaced with the stencil library, effectively a re-write of a major application component. Even though the results show that much performance improvement can be gained with this re-write, its is unlikely that the developers of existing applications will spend the effort to rewrite and retest a significant portion of their applications.

A different approach is to determine how to fit the high-performance aspects of the stencil library into MPI itself. Any application that uses the same parts of the MPI interface would readily benefit. The disadvantage is that the MPI interface places some constraints on the capability and this the achievable performance improvement of the implementation.

Instead of creating a full MPI implementation to examine some of these questions, this chapter uses the Topology Experiments for MPI (TEMPI) library developed by the author. TEMPI is an interposer library inserted into the application link order before the system MPI library. MPI symbols will be resolved in the TEMPI library, allowing new functionality to be executed instead of (or in addition to) the system MPI implementation. TEMPI is intended as a vehicle for adding experimental modifications to MPI without the burden of creating a new MPI implementation or modifying an existing one. If a concept proves to be useful in TEMPI, it could then be integrated into an existing MPI. This is similar to the MPI Profiling Interface (PMPI functions), except we do not rely on the profiling interface existing, and TEMPI can be chained with other PMPI-based tools.

This chapter describes how lessons from Chapters 3 and 4 can be generalized into MPI. TEMPI provides several transparent transformation layers between the application and the system MPI. First, TEMPI transforms non-contiguous application data to contiguous data presented to the underlying system MPI. Second, TEMPI re-numbers system MPI ranks to improve locality. Third, TEMPI transparently switches data-packing operations based on empirical performance measurements. These operations are evaluated through microbenchmarks and the Astaroth halo exchange.

The experiments are carried out on three MPI implementations spanning two hardware platforms summarized in Table 5.1. All multi-node performance is evaluated using Spectrum MPI on OLCF Summit. For intra-node operations, performance is also evaluated on the single-node *openmpi* and *mvapich* platforms. The *mvapich* platform does not use MVAPICH-GDR, which integrates some prior work by Chu et al. [56, 57], but requires [34] Mellanox networking hardware and drivers despite their irrelevance to datatype handling.

Table 5.1: Experimental platform summaries

Name	OLCF Summit	openmpi	mvapich
MPI	Spectrum MPI 10.3.1.2	OpenMPI 4.0.5	MVAPICH 2.3.4
CPU	IBM POWER 9	AMD Ryzen 7 3700x	
GPU	Nvidia V100	Nvidia GTX 1070	
nvcc	11.0.221	11.1.105	
gcc	9.3.0	10.2	
GPU Driver	418.116.00	455.32.00	

First, Section 5.1 describes two ways the Astaroth communication pattern can be implemented in pure MPI. Section 5.2 describes how MPI Derived Datatypes can encode the information required for packing/unpacking. It also describes how derived datatype support is limited on GPUs, and proposes a robust approach for handling regular strided types. Next, Sections 5.3 through 5.6 describe how datatype handling is integrated with MPI, through the MPI_Type_commit, MPI_Pack, MPI_Send, and MPI_Isend functions, and evaluate the integration with some microbenchmarks. Section 5.7 describes how data placement is implemented through MPI_Dist_graph_create_adjacent. Section 5.8 describes the design of the TEMPI library which implements the experimental code. Finally, Section 5.9 presents an evaluation using the Astaroth communication pattern. Section 5.10 concludes.

5.1 Astaroth Communication in MPI

In Chapter 4, a custom library was responsible for data placement, fast handling of non-contiguous data, and overlapping of independent communications. Astaroth chose a custom implementation to avoid limited MPI performance for this scenario, same as the replacement stencil library from that chapter. Instead of modifying the Astaroth code again, this chapter restricts its evaluation to MPI implementations of the communication pattern *only*, as Chapter 4 found that at scale, the iteration time was dominated by the communication time. To that end, two custom pure-MPI implementations of the GPU stencil grid data and the halo-exchange communication were created, without any corresponding computation kernels.

5.1.1 Data Placement

The abstraction in the stencil library did not expose communication in terms of MPI ranks, and therefore allowed the library to choose which MPI ranks should take which subgrids based on their proximity in the machine. In an MPI implementation there is no such abstraction, and it is typical for logical position among the work decomposition to be determined by rank. Unfortunately, there is no *a priori* relationship between an MPI rank number and position in the machine (though this can be adjusted at launch time outside the application code).

MPI_Dist_graph_create_adjacent is one standardized way to solve this problem within the application code. That function creates a new communicator with attached topology information, i.e., which ranks are logically neighbors. The caller can optionally indicate that they want MPI to *reorder* the ranks. This allows the MPI implementation to change the rank numbering to place neighboring ranks closer together than they would otherwise be.

Figure 5.1 shows an example. Consider a system with two nodes and two processes per node (A/B and C/D respectively). When MPI_Init is called, ranks are assigned to the processes in some arbitrary way. In this example, this creates the MPI_COMM_WORLD communicator with rank 0 on process A, through rank 3 on process D. The application can analyze the problem domain and decide that ranks 0 and 2 will communicate heavily, while ranks 0 and 1 will not. The application will provide that information

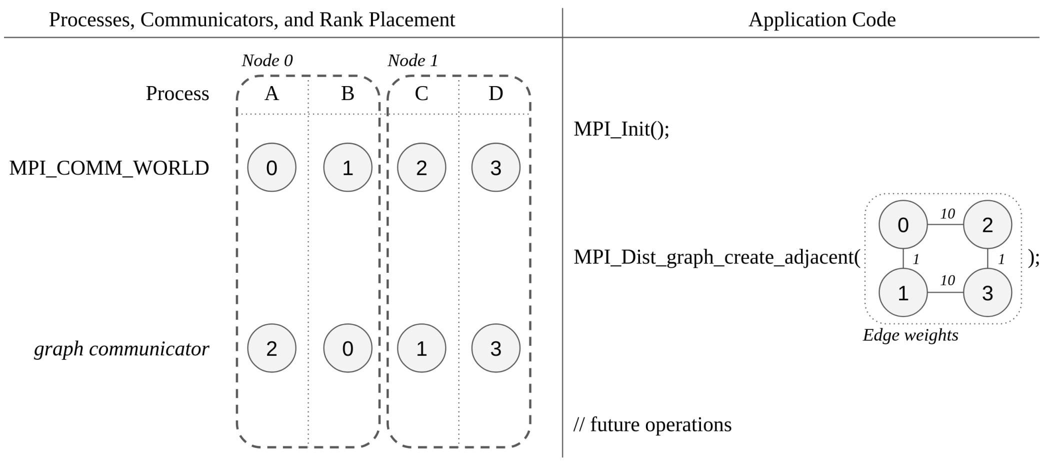


Figure 5.1: Example of MPI_Dist_graph_create_adjacent being called with rank edge weights, and producing a reordered communicator. The function produces a new communicator (*graph communicator*) where MPI_COMM_WORLD ranks 0 and 2 are on the same node.

to MPI_Dist_graph_create_adjacent in the form of edge weights, and allow the function to reorder ranks. This creates a new communicator with (possibly) reordered ranks - i.e., process A is rank 2 in the new communicator (and still rank 0 in MPI_COMM_WORLD). As requested, ranks 0 and 2 in the new communicator are close together in Processes A and B, as are ranks 1 and 3 in Processes C and D. In the original arbitrary placement, ranks 0 and 2 were on separate nodes. There is no standardized way for MPI ranks to query their distance from one another, so this reordering process is the only standardized way to programmatically take advantage of machine topology in pure MPI. Note that the state of the process (A) which was rank 0 in MPI_COMM_WORLD is present in rank 2 in the new communicator - actual process data is not moved automatically.

MPI_Dist_graph_create_adjacent has a second benefit - the topology information enables sparse collectives called "neighborhood" collectives. In these collectives, each rank only exchanges data with its neighbors in the graph, instead of all ranks in the communicator.

5.1.2 MPI_Neighbor_alltoallv and MPI_Isend

Once communicating ranks are placed within the same node, the next step is the communication implementation. In general, the stencil communication is quite sparse, with each rank communicating with at most 26 others. At the largest scales evaluated in this chapter (3072 ranks), each rank communicates with less than 1% of the other ranks. The MPI collectives are not a good fit for this operation, as most of the send/receive “counts” describing how much data is exchanged will be 0.

The “alltoallv” method takes advantage of the sparse communicator created by MPI_Dist_graph_create_adjacent, and splits each halo exchange into three phases. First, repeated calls to MPI_Pack are used to fill a send buffer with the packed halo data to send to each rank. Second, MPI_Neighbor_alltoallv collective is used to exchange data with all neighbors. Third, repeated calls to MPI_Unpack move the data from the receive buffer into the grid.

The “Isend” method uses a paired Isend/Irecv in each direction to move the halo data. Each Isend/Irecv pair therefore handles a different datatype, with communications hopefully overlapped. The communication sparsity is captured by only creating a send/receive pair between communicating ranks.

5.1.3 Non-Contiguous Data

The final piece of the puzzle is handling the non-contiguous data for the halo exchange. MPI Derived Datatypes [30] are an abstraction for describing the layout of non-contiguous data in memory. They allow MPI functions to operate on such data without intermediate handling by the user application, especially packing the data into a contiguous buffer before transfer. As GPUs have become a dominant high-performance computing accelerator, MPI implementations such as OpenMPI [32], MVAPICH [58], Spectrum MPI [31] and MPICH [33] have become “CUDA-aware”. In such implementations MPI can directly operate on CUDA device allocations to streamline application development and potentially accelerate inter-rank transfers of GPU-resident data.

MPI datatypes can be composed to describe multi-dimensional strided objects. This work consideres the following “strided” datatypes due to their applicability to stencil codes:

- “Predefined” or “named”[30, §3.2.2]: these are the base MPI datatypes (MPI_BYTE, MPI_FLOAT, etc.) that correspond to various C or FORTRAN types.

- "Contiguous"[30, §4.1.2]: these describe "replication of a datatype in contiguous locations." MPI_Type_contiguous(n, *oldtype*, *newtype*): *newtype* is n contiguous repetitions of *oldtype*.

- "Vector/Hvector"[30, §4.1.2]: these describe "replication of a datatype into...equally spaced blocks." MPI_Type_vector(c, l, s, *oldtype*, *newtype*): *newtype* is a vector of c blocks, each block is l contiguous repetitions of *oldtype* and the beginning of each block is separated by s contiguous repetitions of *oldtype*. For hvector, s is given in bytes instead.

- "Subarray"[30, §4.1.3]: these describe "n-dimensional subarray of an n-dimensional array." MPI_Type_create_subarray(n, $\{$*sizes*$\}$, $\{$*subsizes*$\}$, $\{$*offsets*$\}$ *order*, *oldtype*, *newtype*): *newtype* is an n-dimensional subarray of an *oldtype* array with extent *sizes*. The subarray is of extent *subsizes* at offset *offsets*. *Order* controls C or FORTRAN ordering.

These types may be composed in many ways to describe the same non-contiguous bytes. For example, consider the 3D object in Figure 5.2, which can be visualized as a three-dimensional sub-object of an enclosing three-dimensional object, where the sub-object shares an origin with the enclosing object and each element of the object is a single-precision floating-point number (an MPI_FLOAT), consuming four bytes.

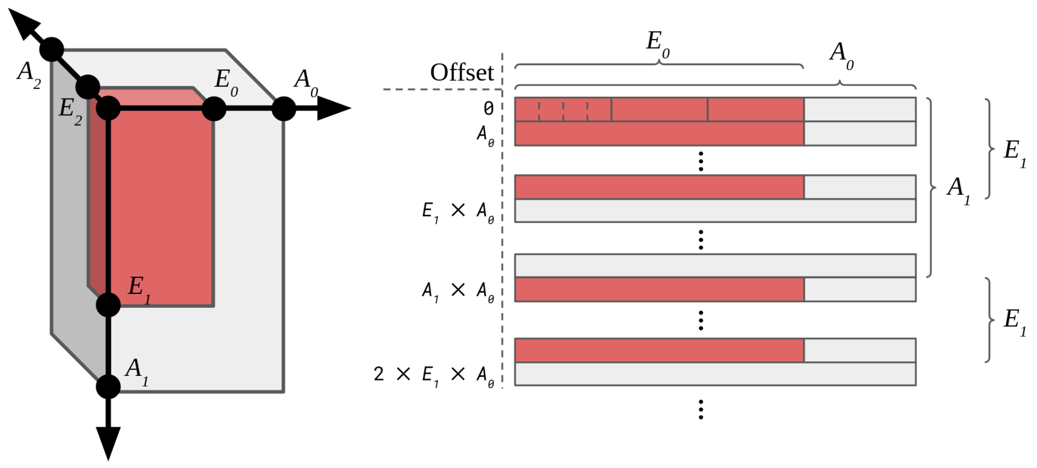


Figure 5.2: A 3D object with extent $E_0 \times E_1 \times E_2$ floats in an allocation $A_0 \times A_1 \times A_2$ bytes, and the corresponding linearized memory layout.

Each 1D row of the object ($E_0 \times 4$ contiguous bytes) to be described in many ways; a non-exhaustive list follows (meanings of function parameters described in the bulleted list above):

- MPI_Type_contiguous(E_0, MPI_FLOAT, &row): "row" comprises a contiguous replication of E_0 single-precision floating-point (4-byte) elements.
- MPI_Type_contiguous($E_0 \times 4$, MPI_BYTE, &row): "row" is $E_0 \times 4$ 1-byte elements.
- MPI_Type_vector(1, E_0, 1, MPI_FLOAT, &row)
- MPI_Type_vector(E_0, 4, 4, MPI_BYTE, &row)
- MPI_Type_create_hvector($E_0 \times 4$, 1, 1, MPI_BYTE, &row)
- MPI_Type_create_subarray(1, $\{A_0\}$, $\{E_0\}$, $\{0\}$, MPI_ORDER_C, MPI_FLOAT, &row)
- MPI_Type_create_subarray(1, $\{A_0 \times 4\}$, $\{E_0 \times 4\}$, $\{0\}$, MPI_ORDER_C, MPI_BYTE, &row)

These are equivalent for describing a single row, but are not entirely interchangeable since their extents vary. This distinction is relevant for certain compositions of these types (e.g., below), or when multiple types are manipulated at once.

A 2D plane (E_1 rows, offset by A_0 bytes between the beginning of each row) can be constructed directly from named types:

- MPI_Type_vector(E_1, E_0, A_0, MPI_FLOAT, &plane)
- MPI_Type_vector(E_1, $E_0 \times 4$, A_0, MPI_BYTE, &plane)
- MPI_Type_create_subarray(2, $\{A_0, A_1\}$, $\{E_0, E_1\}$, $\{0, 0\}$, MPI_ORDER_C, MPI_FLOAT, &plane)
- MPI_Type_create_subarray(2, $\{A_0 \times 4, A_1\}$, $\{E_0 \times 4, E_1\}$, $\{0, 0\}$, MPI_ORDER_C, MPI_BYTE, &plane)

or alternatively, as an hvector of rows:

- MPI_Type_create_hvector(E_1, 1, A_0, row, &plane)

or for the subarray row types:

- MPI_Type_vector(E_1, 1, 1, row, &plane)
- MPI_Type_create_subarray(1, A_1, E_1, 0, MPI_ORDER_C, row, &plane)

Similarly planes comprise a cuboid (E_2 planes, offset by $A_0 \times A_1$ bytes between the beginning of each plane). For example,

- MPI_Type_create_hvector(E_2, 1, $A_0 \times A_1$, plane, &cuboid)
- MPI_Type_create_subarray(2, $\{A_0, A_1, A_2\}$, $\{E_0, E_1, E_2\}$, $\{0, 0, 0\}$, MPI_ORDER_C, MPI_FLOAT, &cuboid)
- MPI_Type_create_subarray(2, $\{A_0 \times 4, A_1, A_2\}$, $\{E_0 \times 4, E_1, E_2\}$, $\{0, 0, 0\}$, MPI_ORDER_C, MPI_BYTE, &cuboid)

Such a three-dimensional datatype can be used to describe the halo regions in a three-dimensional stencil code.

5.2 MPI Strided Datatype Handling

MPI provides a facility designed for operating on the non-contiguous data of the stencil halo exchange region: MPI derived datatypes [30] ("datatypes"). A datatype can be used to tell MPI which bytes make up a particular non-contiguous region of memory. More detailed information about the semantics of relevant datatypes is described in Section 5.1.3.

Each datatype can be considered as a list of contiguous blocks, where each block is defined by an offset and a size. Many prior works start with such a representation as the foundation [59, 60, 61, 56, 62], occasionally with additional optimization [63]. The weakness of this approach is that representing datatype may consume as much GPU memory as the datatype itself.

Consider such a type describing N non-contiguous blocks of M MPI_-FLOATs. To support objects dispersed across large address ranges, the block offset and size would be 8 bytes each, yielding at least $16 \times N$ bytes to represent $M \times N \times 4$ bytes of data. If M is relatively small (common for any higher-dimension object) the representation will consume about the same amount of memory as the data itself. This has two effects. First, any operation must access as much metadata as object data, necessarily slowing the operation. For example, if the operation is memory-bandwidth limited (MPI_Pack), the memory bandwidth must be split between metadata and data access. Second, the metadata may consume as much space as the object itself, limiting GPU memory available to other applications. For a stencil code, this effect will be minor since the exchanged data is much smaller than the total subgrid data.

Further datatype optimizations include specialization for types with certain kinds of regularity [59, 60, 61]. These naturally lend themselves to specific compact representations, e.g. an MPI vector of any size as only a block length, block count, and stride. The combinatorial explosion of equivalent representations renders the strategy of specialized kernels infeasible in general. Figure 5.3 shows MVAPICH exhibiting this behavior. MVAPICH is fast for vectors, but slow for equivalent subarray types. Similarly, MVAPICH's fast vector handling is disabled when multiple objects are packed at once. TEMPI features equivalent high performance in all scenarios.

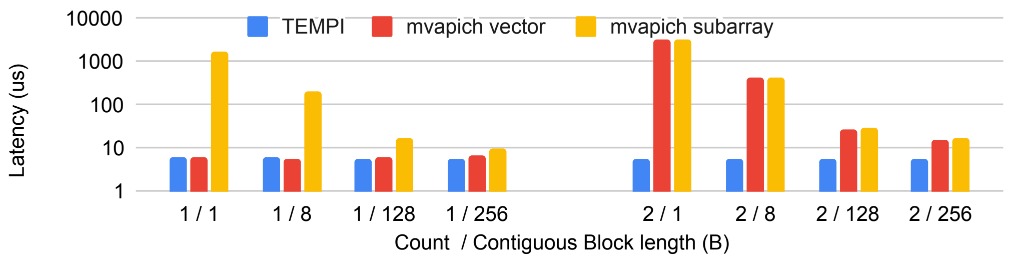


Figure 5.3: TEMPI and MVAPICH latency for MPI_Pack on one and two 1 KiB 2D objects in GPU memory. MVAPICH has specialized handling for a single vector but slows for subarrays or multiple vectors. TEMPI's transformation phase causes all equivalent descriptions to be treated equally quickly.

Other works present sophisticated approaches for handling arbitrary datatypes on the GPU [63, 56]. This section presents a middle ground between the block-list form and more specific or elaborate constructions through the observation that that compositions of contiguous, vector, hvector, and subarray types are all special cases of the same object, and that object is suitable for compact representation. A translation phase converts the datatype into an in-memory representation (Section 5.2.1), a canonicalization phase generates a simplified representation (Section 5.2.2), and a parameterized kernel is selected to pack and unpack the data transparently (Section 5.2.3). This affords wide coverage of structured non-contiguous data without performance fragility of specialized kernels or large metadata sizes for arbitrary datatype handling.

5.2.1 Type Translation

The first phase of the datatype handling process is to convert a fully specified MPI derived datatype into a *Type* hierarchy, which represents a (possibly non-contiguous) set of bytes from a memory region. Each *Type* level has a field *data* of *TypeData*, which represents information about the level. Each *Type* also tracks zero or one child *Type* levels. The *Type* hierarchy and its children describe the MPI datatype, where the order of the hierarchy matches the hierarchy of the constructed MPI datatype.

The IR currently includes two kinds of *TypeData*: *DenseData* for contiguous bytes, and *StreamData* for strided patterns of a single child Type. *DenseData* plays the same role as a named type in MPI: it represents a sequence of contiguous bytes and has no children.

1. *DenseData*
   (a) integer *offset*, the number of bytes between the lower bound and the first byte of the Type
   (b) integer *extent*, the number of contiguous bytes in the Type
2. *StreamData*, a strided sequence of elements of the child type
   (a) integer *offset*, as DenseData
   (b) integer *stride*, the number of bytes between elements
   (c) integer *count*, the number of elements in the stream

This is done by converting each MPI datatype to a corresponding *DenseData* or *StreamData* node, and then recursively doing the same to its child before attaching them to the converted node. The recursive base case is when an MPI Named type is reached, which by definition has no children.

An MPI named type (MPI_INT, etc.) is translated into a *DenseData* with the *extent* field equal to the extent of the named type, and offset 0. A named type is not a derived type, so it has no children.

An MPI contiguous type (MPI_Type_contiguous) is a special case of *StreamData* where the stride matches the size of the element. It is not *DenseData* as *oldtype* may not be dense. *Offset* is 0, *stride* equal to the extent of the *oldtype* argument, and *count* equal to the *count* argument.

An MPI vector (MPI_Type_vector) or hvector (MPI_Type_create_hvector) are translated into two nested *StreamData*, a "parent" and "child". The parent represents the repeated blocks, and the child the repeated elements within each block. Both offsets are 0. The child count is the vector blocklength, and the child stride is the extent of *oldtype*. The parent count is the vector count, and the parent stride is the child *stride* times the vector stride. For hvector the parent *stride* is given directly in the hvector *stride* argument and does not need to be computed.

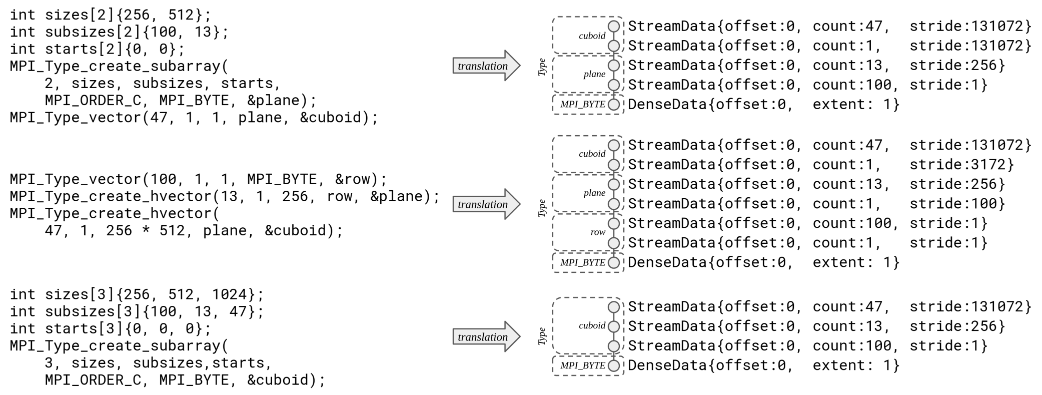
An MPI subarray (MPI_Type_create_subarray) is a set of nested *StreamData* equivalent to the dimension of the subarray. MPI subarray arguments are provided inner-to-outer, which corresponds to a descendant-ancestor relationship in the *Type* tree. The count of dimension i is provided by the corresponding subarray *subsize*. The stride of dimension i is the product of the MPI extent of the subarray *oldtype* and the $i-1$ preceding subarray sizes. The offset of each dimension is given in terms of elements and is converted to bytes for the *TypeData*.

Figure 5.4 shows three different MPI C snippets to create the 3D object described in Figure 5.2.

5.2.2 Type Canonicalization

The construction of the Type tree described in Section 5.2.1 yields a hierarchy of *StreamData* with a base of *DenseData*. Since each level of the datatype has a direct correspondence in the *Type* hierarchy, semantically equivalent datatypes may have different *Type*s. In order to provide fast handling of equivalent types, these various representations are canonicalized.

Four transformations are used to canonicalize the Type tree. "Dense folding" collapses *DenseData* into a parent *StreamData*. "Stream elision" re-


```
int sizes[2]{256, 512};
int subsizes[2]{100, 13};
int starts[2]{0, 0};
MPI_Type_create_subarray(
    2, sizes, subsizes, starts,
    MPI_ORDER_C, MPI_BYTE, &plane);
MPI_Type_vector(47, 1, 1, plane, &cuboid);
```

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:1,   stride:131072}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:100, stride:1}
DenseData{offset:0,  extent: 1}
```

```
MPI_Type_vector(100, 1, 1, MPI_BYTE, &row);
MPI_Type_create_hvector(13, 1, 256, row, &plane);
MPI_Type_create_hvector(
    47, 1, 256 * 512, plane, &cuboid);
```

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:1,   stride:3172}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:1,   stride:100}
StreamData{offset:0, count:100, stride:1}
StreamData{offset:0, count:1,   stride:1}
DenseData{offset:0,  extent: 1}
```

```
int sizes[3]{256, 512, 1024};
int subsizes[3]{100, 13, 47};
int starts[3]{0, 0, 0};
MPI_Type_create_subarray(
    3, sizes, subsizes,starts,
    MPI_ORDER_C, MPI_BYTE, &cuboid);
```

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:100, stride:1}
DenseData{offset:0,  extent: 1}
```

Figure 5.4: Three different MPI C fragments to generate the 3D object from Figure 5.2 with $A_0 = 256$, $A_1 = 512$, $A_2 = 1024$, $E_0 = 100$, $E_1 = 13$, and $E_2 = 47$. The right-hand side shows the corresponding *Type* IR after translation, with parent *TypeData* above child *TypeData*. Equivalent objects can be represented differently and require a later transformation pass.

moves a *StreamData* representing a stream of one element. "Stream flattening" combines two *StreamData* that could be represented as one. "Sorting" ensures the *StreamData* have a unique order. The optimizations are applied repeatedly in turn, only terminating when neither optimization would modify the *Type* hierarchy. Algorithm 1 summarizes the overall simplification process.

Algorithm 1: simplify

```
Function simplify(ty):
    simplified ← ty
    changed ← TRUE
    while changed do
        changed ← FALSE ∨ dense_folding (simplified)          ▷ in-place
        changed ← changed ∨ stream_elision (simplified)       ▷ in-place
        changed ← changed ∨ stream_flatten (simplified)       ▷ in-place
        changed ← changed ∨ sort (simplified)                 ▷ in-place
    end
    return ty
```

Dense folding is driven by the observation that stride of a *StreamData* may match the extent of a child *DenseData*. Such a configuration represents a stream of repeated contiguous dense elements. In that case, the *DenseData* extent can be "folded" up into the *StreamData*, and the pair can be represented as a single *DenseData* node. This scenario may arise when an MPI vector, subarray, or contiguous type is used to describe a contiguous region larger than any MPI named type.

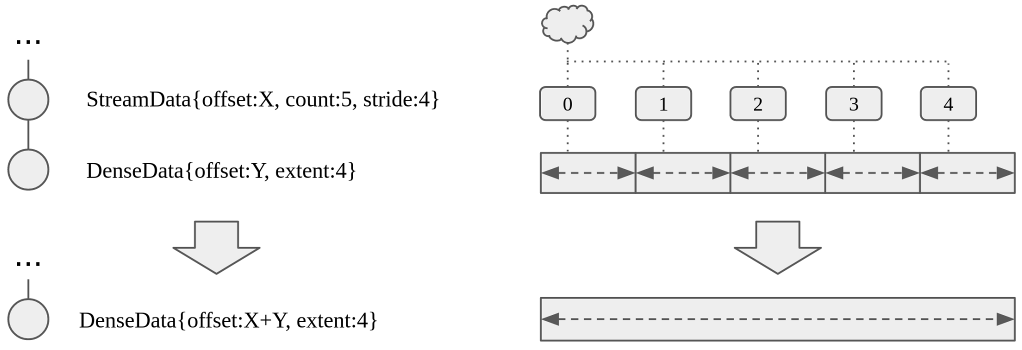

...
StreamData{offset:X, count:5, stride:4}
DenseData{offset:Y, extent:4}
...
DenseData{offset:X+Y, extent:4}
0
1
2
3
4

Figure 5.5: Example of dense folding. When the extent of a *DenseData* matches the stride of a parent *StreamData*, the parent/child combination can be replaced with a single larger *DenseData*.

Algorithm 2 shows how the transformation is applied to a *Type*, and Figure 5.5 shows an example. The transformation is applied to each Type node of the Type tree in a depth-first order. At each node, the transformation only applies if the node (*ty*) is a *StreamData* kind and the node's child (*child*) is a *DenseData*. If the parent's *stride* matches the child's *extent*, the parent is replaced with a larger *DenseData* node that represents the entire contiguous stream. The child's offset is increased to include any offset the parent had.

Stream elision canonicalizes a case where a stream has only a single element. Consider *ty*, a *StreamData* with a child *StreamData* whose count *count* is one. In such a case, *child* is a single element and can be elided. This construction arises in the case of an MPI vector with *blocklength* 1 dimension with *subsize* 1.

Algorithm 3 shows how the transformation is applied to a *Type*, and Figure 5.6 shows an example. Like with dense folding, stream elision is applied separately to each *Type* node in a depth-first order. After that, if both the type *ty* and its child *child* are *StreamData*, then if the child has count of 1, the child is replaced with its own children.

Stream flattening canonicalizes the case where a pair of nested streams could be represented as a single stream. Consider a child *StreamData* with a count A and a stride B. If the parent *StreamData* has a stride that is a product of A and B, that means multiple children are separated by the child's stride. In such a case, the parent and child can be flattened into a single *StreamData* with a larger count.

Algorithm 2: dense_folding from Alg. 1

```
Function dense_folding(ty):
    changed ← FALSE
    for child of ty do
        changed ∨ dense_folding(child)              ▷ fold from bottom up
    end
    if ty.data is not StreamData then
        return changed
    end
    Type child = ty.children[0]
    if child.data is not DenseData then
        return changed
    end
    StreamData cData ← child.data
    StreamData pData ← ty.data
    if cData.extent == pData.stride then
        changed ← TRUE
        cData.off ← cData.off + pData.off
        cData.extent ← pData.count × pData.stride
        ty ← child                                  ▷ replace ty with child
    end
    return changed
```

Algorithm 4 shows how the transformation is applied to a *Type*, and Figure 5.7 shows an example. This operation has some overlap with stream elision. Stream elision handles the specific case when the child's count is 1, which lifts the restriction on the child and parent stride relationship in stream flattening.

Sorting canonicalizes the ordering of a pair of nested streams is arbitrary. For example, consider a 2D non-contiguous object. That object could be constructed as columns of rows of blocks, or rows of columns of blocks. To canonicalize this case, the *StreamData* hierarchy is sorted by stride, with the largest strides first in the hierarchy and the smaller strides last.

This IR can be extended with additional types and transformations in the future to handle MPI's indexed types. However, in its current form, it always provides a common canonicalization of two equivalent regular datatypes. Canonicalization requires that any two equivalent descriptions would end in a form that cannot be further reduced, and that there is only one form to represent a description that cannot be further reduced.

Note that transformations maintain the fundamental structure of the IR: The base of the IR structure is a *DenseData*, and above it are zero or more *StreamData*. Dense folding either replaces the base *DenseData* and parent *StreamData* with an equivalent *DenseData*, or makes no changes. Stream

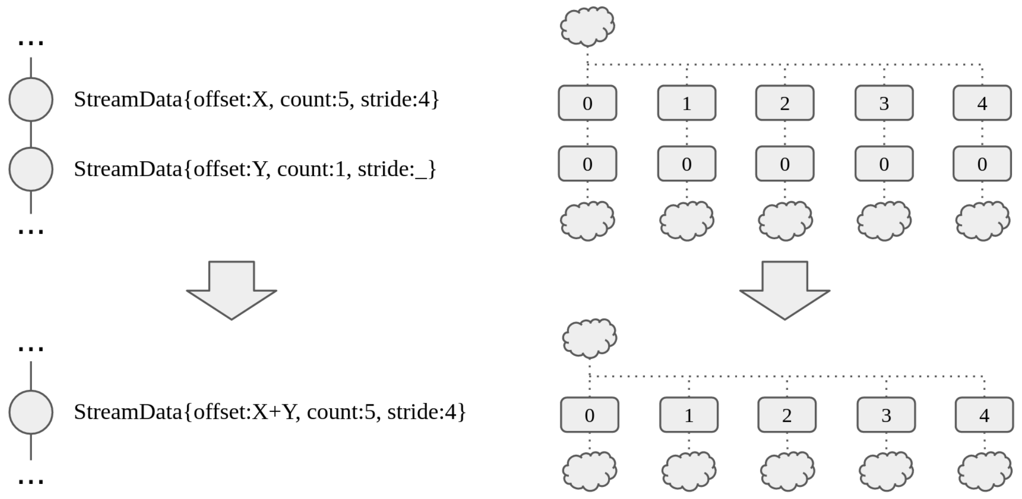

...
StreamData{offset:X, count:5, stride:4}
StreamData{offset:Y, count:1, stride:_}
...
...
StreamData{offset:X+Y, count:5, stride:4}
...
0
1
2
3
4
0
0
0
0
0
0
1
2
3
4

Figure 5.6: Example of stream elision. When a child *StreamData* has only a single element, it can be removed from the *Type* tree.

elision and stream flattening either replace two *StreamData* with a single equivalent *StreamData*, or make no changes. Sorting does not change the number of *StreamData*, only their order.

A characteristic of MPI datatypes is that additional components cannot *remove* any existing non-contiguous bytes from the object. They either leave it the same (effectively representing a single instance of its child) or add more bytes (multiple child instances). This introduces a redundancy challenge for canonicalization, i.e., a *StreamData* may contribute no new bytes to the object. In fact, any such object has an infinite number of possible descriptions since an infinite number of redundant MPI datatypes (and therefore *StreamData*) can be added. Each non-sorting transformation can be viewed as removing a single such addition that did not actually expand the representation. Dense folding removes all *StreamData* that were used to create larger *DenseData* regions that do not correspond to a single MPI named type. Stream elision removes all *StreamData* that represent a single child element. Stream flattening combines *StreamData* when they could together have been represented by a single *StreamData*. After the transformations, the only possible hierarchy is the simplest one where each *StreamData* adds some new bytes to the object.

The final challenge is that MPI datatypes impose no particular order on the construction of the object. This is another way in which two identical objects can have different representations. The sorting transformation chooses an

Algorithm 3: stream_elision from Alg. 1

```
Function stream_elision(ty):
    changed ← FALSE
    for child of ty do
        changed ← changed ∨ stream_elision(child)          ▷ bottom up
    end
    if ty.data is not StreamData then
        return changed
    end
    Type child = ty.child
    if child.data is not StreamData then
        return changed
    end
    StreamData cData ← child.data
    if 1 == cData.count then
        changed ← TRUE
        ty.child ← child.children                          ▷ delete child
    end
    return changed
```

arbitrary canonicalization of this case, ordered by stride.

The properties of redundancy and arbitrary ordering are what fundamentally make the canonicalization necessary. Dense folding, stream elision, and stream flattening all remove any redundancy that is added to the datatype. With redundancy removed, sorting removes any remaining arbitrary ordering of the datatype. This yields an IR with a form that cannot be further transformed, and causes all cases of redundant information and arbitrary ordering to be canonicalized to the same form.

5.2.3 Kernel Selection

Once the type is canonicalized, it is converted into a *StridedBlock* structure. The *StridedBlock* structure is semantically similar to an MPI subarray and is used only to select the kernel implementation.

- *StridedBlock*
    - integer *start*: byte offset between the lower bound and the first element
    - integer list *counts*: number of elements in the dimension
    - integer list *strides*: bytes between the start of each element in the dimension

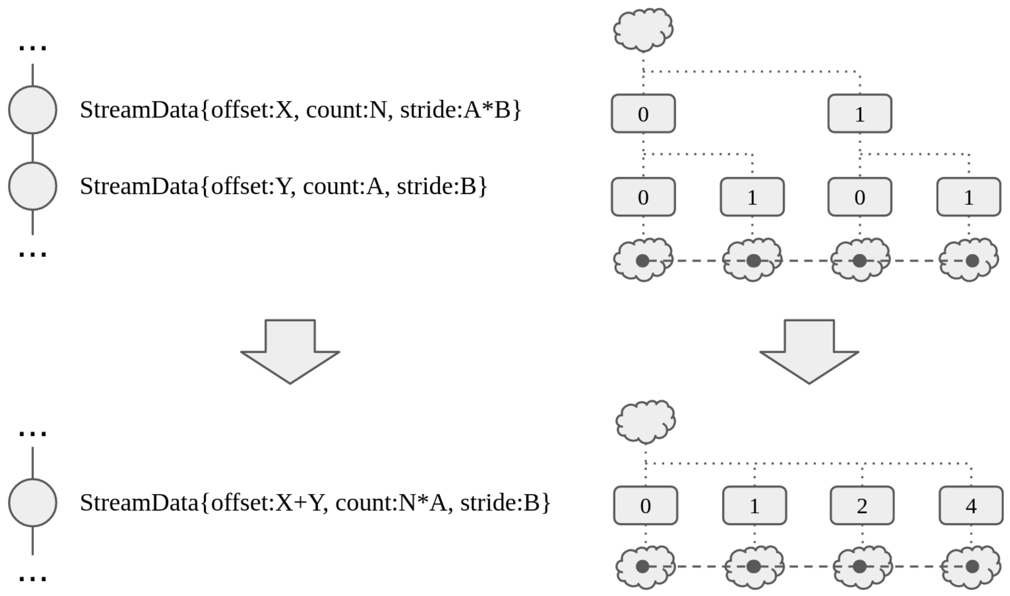

...
StreamData{offset:X, count:N, stride:A*B}
StreamData{offset:Y, count:A, stride:B}
...
...
StreamData{offset:X+Y, count:N*A, stride:B}
...
0
1
0
1
0
1
0
1
2
4

Figure 5.7: Example of stream flattening. When the parent stride allows repeated children to maintain a fixed stride between their elements, the parent and child can be flattened into a single stream.

The *start* field describes the offset of the first byte in the object from the beginning of the allocation. The ith entry of *counts* and *strides* describes the number of repetitions of the previous dimension and the number of bytes separating each repetition, respectively. Section 5.2.4 gives a concrete example of the StridedBlock structure for part of a halo exchange.

Algorithm 5 describes the conversion from *Type* to *StridedBlock*. This is only possible if the bottom is a DenseData and every other object is a StreamData. The process in Section 5.2.2 will apply the conversion if it is possible. The DenseData describes the first dimension, which will have stride 1 and count equal to the extent of the DenseData. Each higher dimension directly corresponds to the StreamData. The offset of each dimension is accumulated into the single offset of the StridedBlock.

Once the *Type* is converted into a *StridedBlock*, the next task is to choose a method for fast packing and unpacking on the GPU. If the StridedBlock is 1D (contiguous), we issue a single cudaMemcpyAsync to move the data into the destination buffer, followed by a cudaStreamSynchronize. This is similar to the implementation in MVAPICH, OpenMPI, and Spectrum MPI. If the StridedBlock is 2D we select a kernel that maps the X dimension of the thread index into the count[0] and the Y dimension to count[1]. If the

Algorithm 4: stream_flatten from Alg. 1

```
Function stream_flatten(ty):
    changed ← FALSE
    for child of ty do
        changed ← changed ∨ stream_flatten(child)          ▷ bottom up
    end
    if ty.data is not StreamData then
        return changed
    end
    Type child = ty.child
    if child.data is not StreamData then
        return changed
    end
    StreamData pData ← ty.data
    StreamData cData ← child.data
    if pData.stride == cData.count × cData.stride then
        changed ← TRUE
        pData.count ← pData.count × cData.count
        pData.stride ← cData.stride
        pData.off ← pData.off + cData.off
        ty.child ← child.children                          ▷ delete child
    end
    return changed
```

StridedBlock is 3D, we map the X dimension to the count[0], Y dimension to the count[1], and Z dimension to the count[2]. Higher dimensional objects can follow the same general pattern, with additional outer loops for each dimension.

Each kernel dimension is filled from X to Z by the smallest power of two that encompasses the corresponding extent, ultimately limited by a block limit of 1024 threads. The grid is then sized to cover the entire input object once the block size is determined.

Each kernel is specialized to a word size W, which is the largest GPU-native type that is both aligned to the object and is a factor of count[0]. The X dimension collaboratively loads count[0] contiguous bytes that make up each block using elements of size W.

Many MPI functions that operate on datatypes accept a *count*, *incount*, or *outcount* parameter, describing how many objects are to be operated on in the buffer. Unlike other properties of the type, this value is not known until the MPI function is called and therefore is not included in the type optimization. The kernels handle this value dynamically either by increasing the grid Z dimension (for 2D), or by applying the entire kernel grid to each

Algorithm 5: conversion of *Type* to *StridedBlock*

```
Function strided_block(ty):
    datas ← []
    cur ← ty                                    ▷ Add all TypeData to an array
    while true do
        datas.append(cur)
        if cur.child == {} then
            break                               ▷ no children left
        else
            cur ← cur.child
        end
    end
    StridedBlock sb                             ▷ to be returned
    for i = 0 to datas.size() do
        if i == 0 then
            if data is DenseData then
                sb.off ← data.off
                sb.counts.append(data.extent)
                sb.strides.append(1)            ▷ DenseData stride is 1
            else
                return NULL                     ▷ Not strided
            end
        else
            if data is StreamData then
                sb.off ← sb.off + data.off
                sb.counts.append(data.count)
                sb.strides.append(data.stride)
            else
                return NULL                     ▷ Not strided
            end
        end
    end
    return sb
```

object in turn (3D).

By the end of this whole process, each MPI datatype has a corresponding kernel implementation with a specific W instantiation. No metadata is consumed in GPU memory - all object parameters are either encoded into the kernel binary (W) or passed as a scalar kernel argument.

5.2.4 Example

Consider a type representing the "interior" data for a send in the -X direction, for the same 256^3 subgrid with stencil radius 3 and 8 byte quantities. This data is a $3 \times 256 \times 256$ region of a single quantity, offset from the origin by 3 gridpoints each in the X, Y, and Z direction to account for the radius-3 shell of ghost points. This data would be received into a corresponding $3 \times 256 \times 256$ ghost region in the +X side of the receiving subgrid. The total allocation will contain space for $262 \times 262 \times 262$ gridpoints to accommodate the ghost points. Figure 5.8 shows the region with various sizes annotated and Listing 5.1 shows one way to construct the object with MPI datatypes.

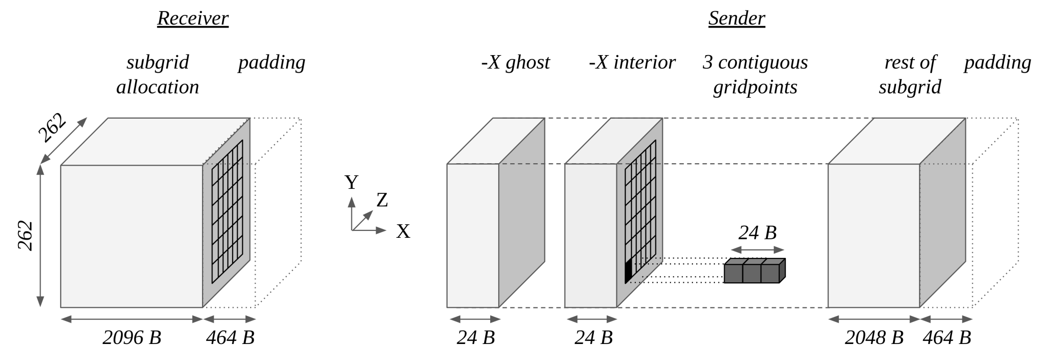


Figure 5.8: Example of the interior gridpoint region involved in a -X direction of a 256^3 subgrid halo exchange with an 8 byte quantity. On the send side, the interior is offset from the origin due to the shell of ghost cells.

Listing 5.1: MPI subarray description of interior gridpoints for -X send

```
MPI_Datatype interior;
int ndims{3};
int array_of_sizes[3]{262, 262, 2560};
int array_of_subsuzes[3]{256, 256, 24};
int array_of_starts[3]{3, 3, 24};
MPI_Type_create_subarray(ndims, array_of_sizes,
                         array_of_subsizes,
                         array_of_starts, MPI_ORDER_C,
                         MPI_BYTE, &interior);
```

Figure 5.9 shows an example of what the TEMPI IR for such a non-contiguous region would look like, directly translated from MPI_Type_create_subarray, without any canonicalization. The bottom *DenseData* (❹) corresponds to the MPI_BYTE type used as the base element for MPI_Type_-create_subarray. The lowest *StreamData* (❸) represents blocks of count = 24 consecutive (stride = 1) child *DenseData*s (❹). Since the data is stored in row-major form, the X dimension is contiguous in memory. These are three gridpoints along the X direction, each an 8-byte quantity ($3 \times 8 = 24$). The 24-byte offset accounts for ghost gridpoints in the X dimension.

```
❶ StreamData{off:2012160, count:256, stride:670720}
❷ StreamData{off:7680, count:256, stride:2560}
❸ StreamData{off:24, count:24, stride:1}
❹ DenseData{off:0, extent:1}
```

Figure 5.9: TEMPI IR, after translation and before canonicalization, of the interior region for the -X direction halo exchange of a 256^3 subgrid with an 8 byte quantity.

The Y dimension of the object is represented by the next *StreamData* (❷). This represents the 256 (count = 256) repeated contiguous X blocks that make up a row of the plane that will be sent. The stride is derived by recalling that radius of 3 causes the subgrid to be 262 gridpoints in the X direction (256 interior points + 6 ghost points). For an 8-byte quantity, this translates to $262 \times 8 = 2096$ bytes. However, the allocation has a 512 byte pitch, so the X dimension is padded out to 2560 bytes, the next multiple of 512 (512×5). Like the X dimension, the Y dimension offset is 3 gridpoints. This means that the offset is 3 times the size of the X dimension in bytes, or $3 \times 2560 = 7680$ bytes.

The final *DenseData* (❶) represents the Z extent (count = 256). The stride of 670720 matches the amount of linear memory consumed by the Y extent, which is the pitch of the allocation (previously found to be 2560 bytes) times the Y extent of the grid (262 points). The offset is 3 gridpoints in the Z dimension, or $670720 \times 3 = 2012160$ bytes.

```
❺ StreamData{off:2012160, count:256, stride:670720}
❻ StreamData{off:7680, count:256, stride:2560}
❼ DenseData{off:24, extent:24}
```

Figure 5.10: TEMPI IR, after translation and canonicalization, of the interior region for the -X direction halo exchange of a 256^3 subgrid with an 8 byte quantity. Figure 5.9 shows the original IR.

After transformation, ❸ and ❹ are collapsed into a single consecutive region (❼). The transformed IR is shown in Figure 5.10. The transformed IR can also be used to understand the performance consequences. The bottom *DenseData* (❼) represents the blocks of 24 contiguous bytes. The *StreamDatas* on top of that (❺ and ❻) represent $256 \times 256 = 65{,}536$ of those contiguous regions, with either 2560 or 670,720 bytes between them. The baseline implementation would issue 65,536 24-byte cudaMemcpyAsync calls to pack that non-contiguous data, while TEMPI will issue a single GPU kernel.

Finally, the corresponding *StridedBlock* is emitted:

```
StridedBlock{ start: 2019864,
              counts: [24 256 256],
              strides: [1 2560 670720] }
```

The strided block captures the relevant information from the transformed IR: namely, where the first byte of the object begins relative to the allocation it is in, and for each dimension, how many elements make up that dimension, and the stride between each element. Implicitly, the base element for the first dimension is a byte, i.e., the first dimension is 24 bytes.

5.3 MPI_Type_commit

The MPI_Type_commit function delineates the boundary between when an application constructs a datatype and when that type may be used with the rest of the MPI functions. The MPI standard advises that "the system may compile at commit time an internal representation for the datatype ... and select the most convenient transfer mechanism." [30]. In line with that advice, the translation, canonicalization, and kernel selection phases occur when the MPI_Type_commit function is called and are cached for use when later MPI functions are called. At that time, the provided datatype is used to look up the corresponding pack/unpack GPU kernel if such an operation is required. When MPI_Type_free() is called, the cached kernel selection is freed.

Figure 5.11 shows the run-time impact of creating MPI derived types, broken down into two phases. Creation refers to using the MPI_Type* and MPI_Type_create* functions to assemble the type description. Commit refers to calling MPI_Type_commit on that description. TEMPI does the same operations in each instance; however, it relies on the performance of the MPI_Type_get_envelope, MPI_Type_get_extent, MPI_Type_size, and MPI_-Type_get_contents functions. The different implementations will have different performance for those routines, and therefore the "commit" component takes variable amounts of time. Within a particular implementation, differ-

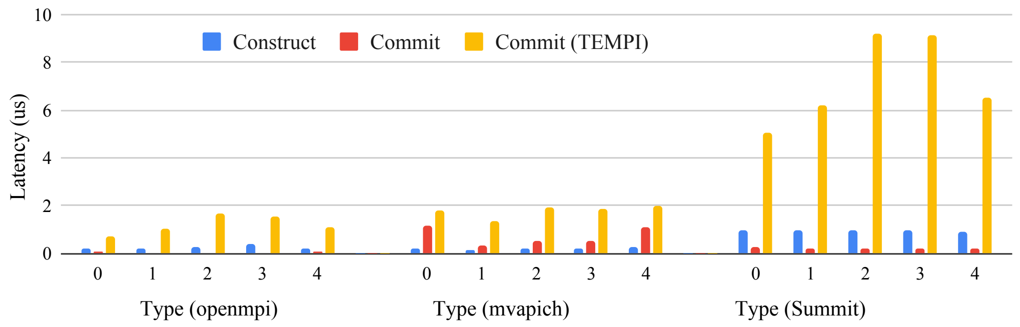


Figure 5.11: Time for MPI derived datatype creation and commit time for equivalent 3D objects described with subarray (1), hvector of vector (2), hvector of hvector of vector (2,3), and subarray of vector (4). The “create” component uses MPI_Type* and MPI_Type_create* family of MPI APIs to describe the type. The “commit” component is how much time is consumed in MPI_Type_commit. The trimean of 30000 executions of each phase is reported. Construction time is unchanged (TEMPI does nothing) and is reported for comparison. In MPI_Type_commit, TEMPI does the same operations regardless of the MPI implementation, but performance varies due to performance of the calls that provide information about MPI types. TEMPI slows commit time substantially, but it still has a negligible impact on application run time.

ent type configurations have different commit times as a different sequence of optimizations is run to arrive at the canonical form. Overall, the transformation and kernel selection process slows down the create+commit process by 2.1× to 5.5× vs. mvapich, 3.5× to 6.8× vs. openmpi, and 4.2× to 11.6× vs. Summit. This slowdown is a one-time cost during program startup and is small in magnitude (a few microseconds).

5.4 MPI_Pack and MPI_Unpack

MPI_Pack takes one or more objects described by a datatype and packs them into a contiguous buffer on the same rank. As such, it is the simplest MPI operation to require datatype handling. When MPI_Pack is called, the pack kernel is looked up and invoked on the input and output buffers directly. The GPU must be synchronized before MPI_Pack returns, as the input and output buffers can be used immediately. Figure 5.12 shows the pack bandwidth achieved for various 2D objects, described as a vector or subarray datatype.

The time elapsed between the call and return of MPI_Pack is reported.

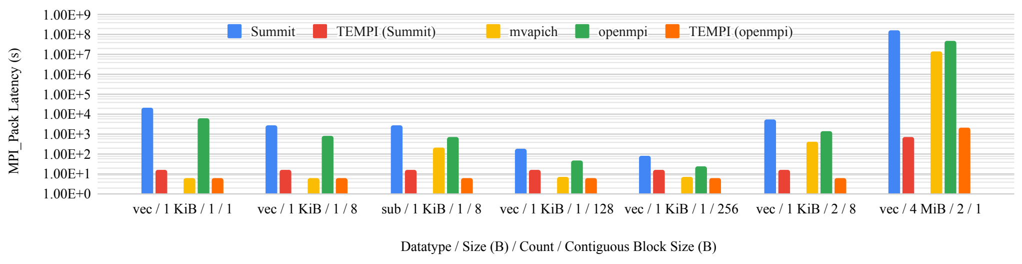


Figure 5.12: MPI_Pack performance of a variety of 2D objects described as a vector or subarray datatype. "Size" is the total object size, "count" is the number of objects packed, and "contiguous block size" is the number of contiguous bytes in each block of the object. The pitch of each contiguous block is 512 B. Comparing "vec / 1 KiB / 1 / 8" with "sub / 1 KiB / 1 / 8" and "sub / 1 KiB / 2 / 8" shows MVAPICH's accelerated vector handling does not generalize to equivalent objects or multiple objects. TEMPI matches MVAPICH's vector performance, and greatly exceeds the datatype packing performance for all other implementations.

Spectrum MPI 10.3.1.2, MVAPICH 2.3.4 and OpenMPI 4.0.5 all support a baseline derived datatype handling approach where each contiguous portion of the derived datatype is copied into a single contiguous buffer through cudaMemcpyAsync (or similar function). The packing throughput is therefore faster as the contiguous block is larger (amortizing overhead), and slower when more contiguous blocks comprise the datatype. MVAPICH also features optimized handling through specialized packing kernels for certain datatypes. TEMPI achieves a speedup of over 242,000 on Summit for the largest datatype. TEMPI nearly matches MVAPICH's performance for the single vector type ("vec / 1 KiB / 1 / 8"), but generalizes that high performance to an equivalent subarray ("sub / 1 KiB / 1 / 8") and multiple vector types ("vec / 1 KiB / 2 / 8"). Across the experiment speedup varies from 0.98× to 242,000×. Generally TEMPI performs better when the contiguous regions are smaller or the total data is larger. In the first case, more memory copies are replaced by a single kernel, and in the second case, the GPU resources are better utilized by the kernel.

5.5 MPI_Send, MPI_Recv, and Performance Modeling

MPI_Send and MPI_Recv are the prototypical MPI point-to-point communication primitives. Like other MPI communication functions, they can operate on datatypes instead of contiguous data. The interposer design requires that interprocess communication is handled by the underlying system MPI. Therefore, integration of datatype handling with underlying communication is restricted to packing and unpacking non-contiguous data into contiguous buffers, upon which system MPI primitives are invoked. The datatype kernel selected during MPI_Type_commit() is executed to covert the object into MPI_PACKED, which is then provided to the system MPI as contiguous data.

Unlike MPI_Pack, the result of the packing operation is not visible to the caller of MPI_Send - it is an intermediate buffer. This means that the location of that buffer is not specified by MPI or by the application code, and it can be chosen by the implementation. TEMPI allows two options - a "device" buffer, where the packed data is resident on the GPU when it is provided to the CUDA-aware system MPI_Send, or a pinned host buffer.

In the "device" packing method (T_{device}, Equation 5.1), the strided object is packed from the original GPU buffer into an intermediate GPU buffer ($T_{gpu-pack}$), then transferred to an intermediate buffer on the destination GPU with CUDA-aware MPI_Send ($T_{gpu-gpu}$), then unpacked into the strided destination object ($T_{gpu-unpack}$).

$$T_{device} = T_{gpu-pack} + T_{gpu-gpu} + T_{gpu-unpack} \tag{5.1}$$

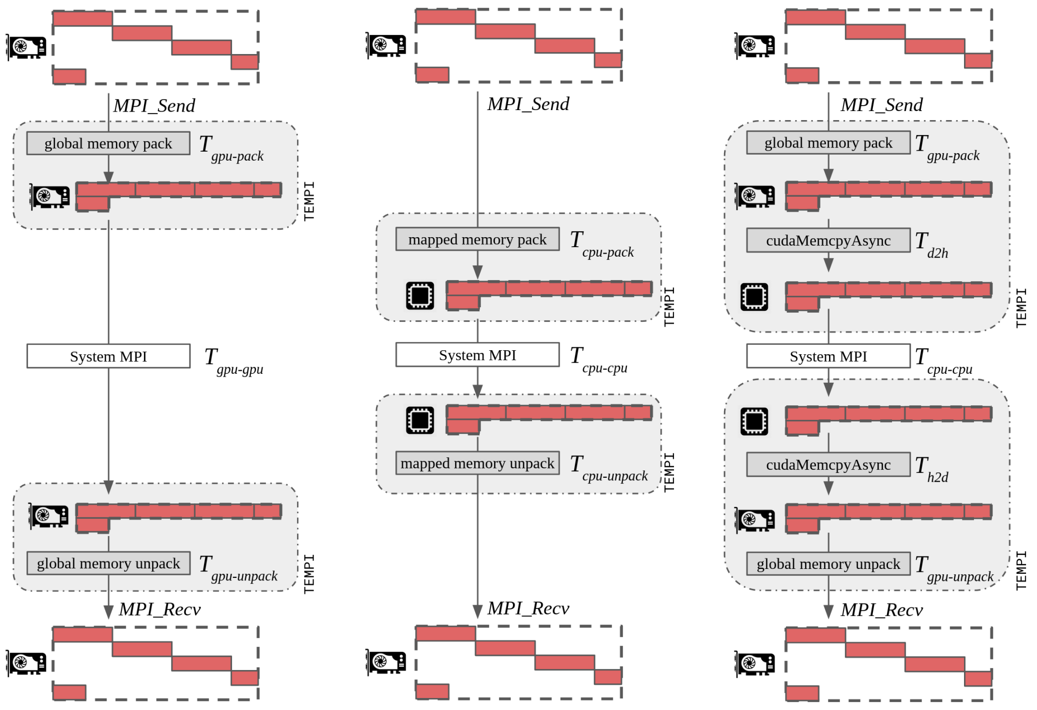
In the "one-shot" packing method ($T_{oneshot}$, Equation 5.2), the strided object is packed from the original GPU buffer into intermediate mapped CPU buffer ($T_{host-pack}$), transferred to an intermediate mapped buffer at the destination ($T_{cpu-cpu}$), then unpacked directly into GPU memory ($T_{host-unpack}$).

$$T_{oneshot} = T_{host-pack} + T_{cpu-cpu} + T_{host-unpack} \tag{5.2}$$

Finally, the "staged" method (T_{staged}, Equation 5.3) matches the device method, except the intermediate GPU buffer is transferred to a pinned buffer on the host (T_{h2d}), where it is transferred to the destination rank's CPU before being copied to the destination GPU (T_{h2d}). This method would only be faster than the device method if $T_{cpu-cpu} + T_{h2d} + T_{d2h} < T_{gpu-gpu}$.

$$T_{staged} = T_{gpu-pack} + T_{d2h} + T_{cpu-cpu} + T_{h2d} + T_{gpu-unpack} \tag{5.3}$$

Figure 5.13 summarizes the three methods.



(a) TEMPI device method (T_{device})

(b) TEMPI one-shot method ($T_{oneshot}$)

(c) TEMPI staged method (T_{staged})

Figure 5.13: Diagrams of TEMPI's device, one-shot, and staged MPI_Send / MPI_Recv methods annotated with quantities from Equations 5.1, 5.2, and 5.3, respectively.

Wang et al. [59] introduce the one-shot and staged methods (using cudaMemcpy2DAsync instead of GPU kernels). They find that the staged method is preferable to one-shot. In contrast, the other works described in Chapter 6 prefer the one-shot method with various GPU kernels.

Modeling the performance of each is a challenge in its own right. Internode message latency ($T_{cpu-cpu}$) is commonly modeled as a latency term plus a bandwidth term [64], possibly refined into short, eager, and rendezvous regimes. Inter-node GPU message latency ($T_{gpu-gpu}$) further complicates the model with GPU-CPU bandwidth, GPU control latency, direct communication between GPU and NIC (Nvidia's "GPUDirect"), and pipelining of large messages [65]. When datatypes are involved, there is additional complexity regarding efficiency of non-contiguous memory accesses served through device memory ($T_{gpu-pack}$) or over the CPU-GPU interconnect ($T_{cpu-pack}$). The interposer design places TEMPI at the mercy of the performance characteristics of the underlying system, so this work sidesteps these concerns by measuring the relevant performance directly and using them at run-time to choose the packing method.

To determine which method offers the best performance in practice, the quantities can be measured for a variety of object kinds and sizes.

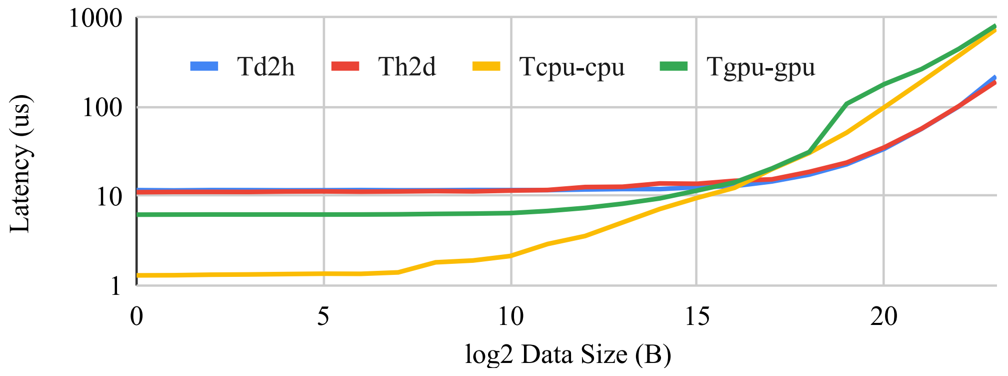
- $T_{cpu-cpu}$: MPI_Send/MPI_Recv on CPU buffer
- $T_{gpu-gpu}$: MPI_Send/MPI_Recv on GPU buffer
- T_{d2h}: cudaMemcpyAsync from device (GPU) to host (CPU) and cudaStreamSynchronize
- T_{h2d}: cudaMemcpyAsync from host to device and cudaStreamSynchronize

The MPI operations are measured through a ping-pong between two ranks, and the reported time is half of the total ping-pong time. The two ranks are on separate nodes. The CUDA operations are recorded using wall-time around the first and last calls, which reflect when control leaves and returns to the application.

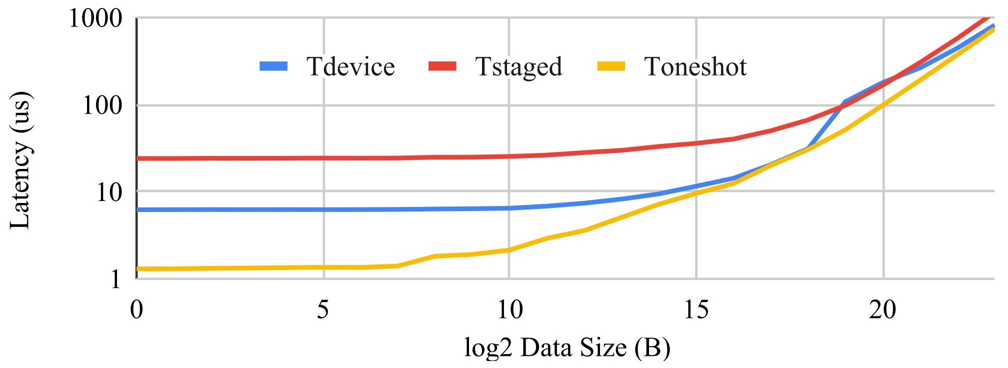
Figure 5.14a shows the results of the four operations for various data sizes. CUDA-aware MPI transfers show a latency floor of approximately 6 µs, compared to 1.3 µs transfers from pinned system memory.

Figure 5.14b shows the measurements in Equations 5.1, 5.2, and 5.3 while holding $T_{gpu-pack/unpack}$ and $T_{cpu-pack/unpack}$ to zero (i.e., $T_{oneshot} = T_{cpu-cpu}$ and $T_{device} = T_{gpu-gpu}$). There is no region where T_{staged} is faster than T_{device} and it will be disregarded in further discussion. Whether T_{device} or $T_{oneshot}$ is faster will depend on the relative pack/unpack performance of the two methods. As T_{device} has pack/unpack occur in the faster device memory, it may be faster than $T_{oneshot}$ for various transfer sizes.

To complete the model, Figure 5.15 shows the measured latency of pack and unpack operations for one-shot ($T_{cpu-pack}$, $T_{cpu-unpack}$) and device



(a) Measurements of T_{d2h}, T_{h2d}, $T_{cpu-cpu}$, and $T_{gpu-gpu}$ on Summit.



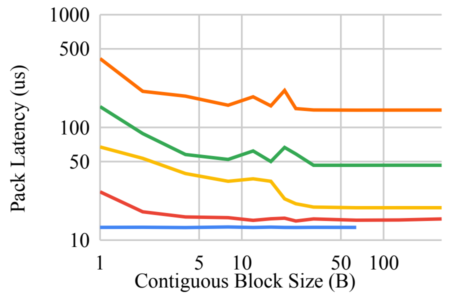
(b) Partial values of T_{device}, $T_{oneshot}$, and T_{staged}, (excluding pack time), using the values from (a). T_{staged} is never lower than the other methods, and is excluded from further discussion.

Figure 5.14: Raw measurements and partial performance models (omit pack/unpack) for various data transfer methods on OLCF Summit.

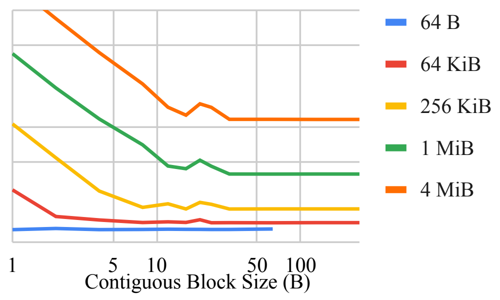
($T_{gpu-pack}$, $T_{gpu-unpack}$). The recorded time includes all of the operations described in Section 5.2.3, i.e. selecting appropriate grid dimensions, executing the kernel, and synchronizing after execution.

Pack/unpack latency depends on both the object size and the size of the contiguous blocks in the object. Larger objects are faster as GPU resources are more fully utilized. Larger contiguous blocks tend to be faster as accesses become more coalesced and make better use of memory and interconnect transactions. One-shot performance is maximized at 32 B contiguous blocks and in-device performance at 128 B. The unpack operation is slower than the pack due to non-contiguous writes (instead of non-contiguous reads in the pack operation).

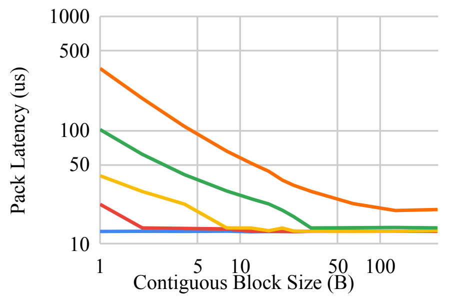
Therefore, whether $T_{oneshot}$ or T_{device} is faster depends on both the ob-



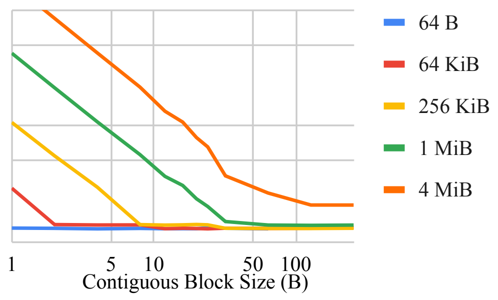
(a) One-shot pack.



(b) One-shot unpack.



(c) Device pack.



(d) Device unpack.

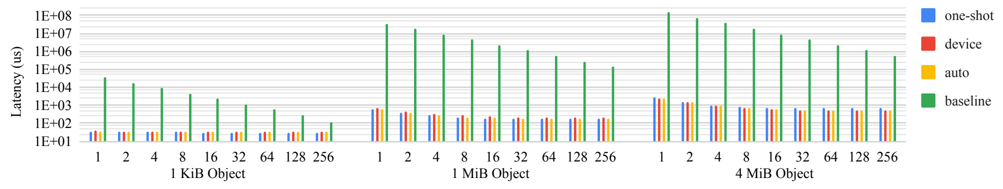
Figure 5.15: Pack/unpack latency using the one-shot and device strategies for 64 B - 4 MiB objects. For smaller contiguous regions, performance is reduced due to low memory or interconnect efficiency for non-coalesced accesses. For larger objects, performance increases as GPU resources are better utilized.

ject size and the length of the contiguous blocks that make up that object. Qualitatively, the one-shot method is faster when objects are smaller, as the packing kernels are limited by launch and synchronization overhead and the CPU-CPU transfers are faster than GPU-GPU. It is also faster when objects are more contiguous, where the zero-copy accesses over the interconnect make good use of the interconnect bandwidth.

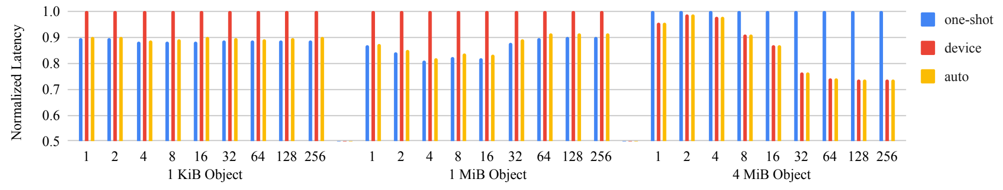
When MPI_Send is called, TEMPI uses the object size and parameters to query the performance model. TEMPI provides a binary that records system performance parameters to the file system. This binary should be run once before TEMPI is used in an application. Performance measurements are sparse by necessity. $T_{cpu-cpu}$ and $T_{gpu-gpu}$ are estimated through 1D interpolation of the object size, while $T_{cpu-pack}$, $T_{cpu-unpack}$, $T_{gpu-pack}$, and $T_{gpu-unpack}$ are estimated through a 2D interpolation of the stride and block length of the datatype. These modeling functions are “pure”, and their results are cached so that future invocations using the same parameters do not require a redundant expensive interpolation.

In each iteration of the Isend/Irecv implementation of the Astaroth communication pattern at 512 nodes with 6 ranks per node, 480 messages are sent using the one-shot method, and 144 using the device method. Generally, smaller messages use the one-shot method, and larger methods use the device method, according to the performance model.

Figure 5.16 shows the application-visible performance of MPI_Send/MPI_Recv compared to the baseline Spectrum MPI 10.3.1.2. Figure 5.16a shows that the vast majority of the speedup comes from the datatype handling ("baseline" vs. one-shot/device). Since $T_{oneshot}$ or T_{device} may be faster depending on the arguments passed to MPI_Send, TEMPI uses the performance model and system measurements to estimate which method will have lower latency. Figure 5.16b shows that the automatic model-based selection is accurate enough to reliably choose the faster of the one-shot or device methods. In the 1 KiB object some small model slowdown is observed as TEMPI must dynamically query the performance model to make its method selection.



(a) MPI_Send / MPI_Recv latency for the one-shot, device, model-based automatic selection, and Summit MPI baseline. The vast majority of the performance improvement comes from the datatype handling, before the one-shot or device method is selected.



(b) Normalized latency of the one-shot, device, and model-based selection based on the measured system parameters. The model-based automatic selection reliably chooses the faster method with minimal overhead.

Figure 5.16: Time for an MPI_Send/MPI_Recv pair for 1KiB, 1MiB, and 4MiB 2D objects with contiguous blocks of various sizes. "Baseline" is the Summit platform without TEMPI. Each group of bars is labeled with the contiguous block size.

This slowdown is present at all sizes, but not as visible at the larger object sizes. Over these tests, model selection added 277 ns of latency. The latency floor is around 30 µs, of which 26 µs can be directly attributed to the pack/unpack kernels on the sending and receiving side. The rest of the time is consumed by looking up the cached datatype handler and checking to see if the user-provided buffers are GPU-resident. Speedup between the baseline and TEMPI's automatic selection is up to 59000× for large objects with small blocks.

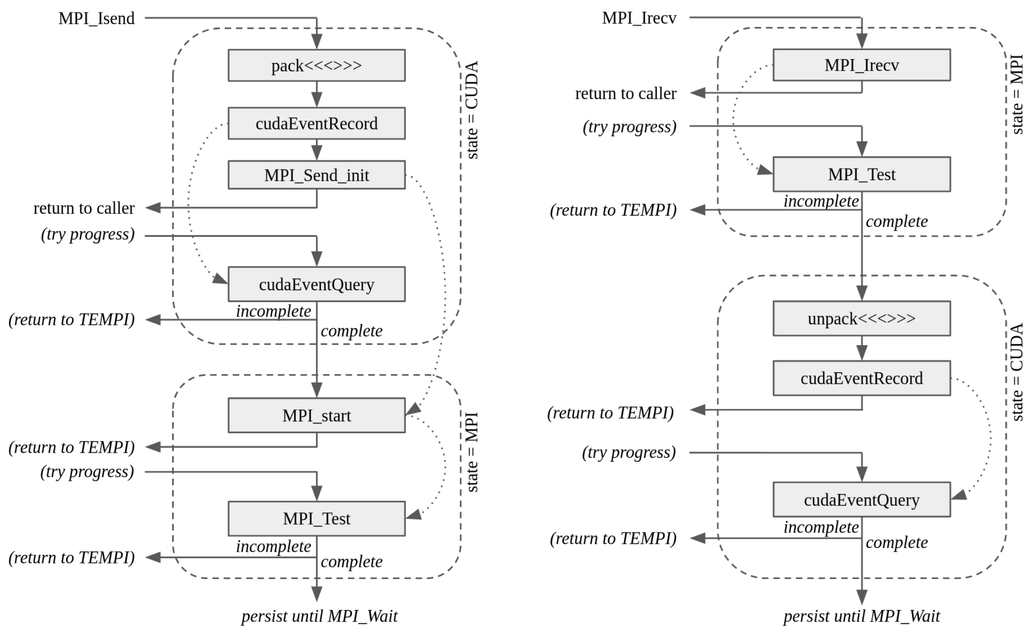
5.6 MPI_Isend/Irecv

MPI_Isend and MPI_Irecv are asynchronous versions of MPI_Send and MPI_-Recv. They should return as quickly as possible to allow the application to overlap as many communications as possible. Consider the structure of MPI_Isend. An asynchronous packing kernel can be invoked immediately and control can return to the application. Then at some indeterminate future time, that kernel will finish and the underlying MPI communication operation should begin.

TEMPI is designed to work on systems with MPI_THREAD_SINGLE, so only one thread may make MPI calls. Since the application expects to make MPI calls, that thread must be an application thread. This prevents TEMPI from assigning the sequential CUDA and MPI operations to another thread that can run concurrently with the application thread.

The selected implementation introduces an object which manages all active asynchronous operations. When MPI_Isend or MPI_Irecv is called, an active operation is recorded inside the management object. Internally, this operation may contain handles to its component asynchronous operations (e.g. a cudaEvent marking completion of a CUDA kernel, or an MPI_Request referring to an MPI operation). Consistent with the MPI interface, TEMPI provides a fake MPI_Request object back to the application. Future MPI_Wait* functions called on these fake MPI_Requests are routed through the manager for handling instead of to the underling MPI implementation.

Whenever control enters TEMPI (at the interface of any overloaded MPI call), TEMPI will attempt to make progress on any active asynchronous operations. This involves querying the state of all active operations, and initiating the next component of any ready operation.



(a) MPI_Isend (b) MPI_Irecv

Figure 5.17: Diagrams of TEMPI's MPI_Isend and MPI_Irecv interaction with CUDA and MPI, and how control enters and leaves. Upon the application call, TEMPI starts the operation and returns as quickly as possible. TEMPI allows each operation to progress opportunistically as control re-enters the TEMPI library. TEMPI tracks the state of the operations until MPI_Wait is called.

More concretely, Figure 5.17a summarizes TEMPI's MPI_Isend implementation. When MPI_Isend is called, the packing operation selected using the procedure in Section 5.2 is issued and the Isend record is put into the *CUDA* state. A cudaEvent is used to record the state of the stream when the packing operation is completed. Finally, an MPI_Request object is created using MPI_Send_init. This request is stored internally, and a separate placeholder request is delivered to the application when control returns.

When control later re-enters TEMPI again (e.g. when the application makes an MPI call), TEMPI checks if the pack operation is complete. If not, TEMPI will proceed with the requested function immediately. If so, TEMPI will start the MPI operation and put the Isend record in the *MPI* state. This is an example of "opportunistic progress:" when execution control enter TEMPI, TEMPI attempts to progress the state of all in-flight asynchronous operations. Now that the Isend record is in the *MPI* state, opportunistic progress involves checking the MPI operation for completeness with MPI_-Test. When complete, this will NULL the internal MPI_Request (which is why a placeholder was provided to the application). Later, when the application calls MPI_Wait on the placeholder, it can be reconciled with the completed Isend record. Figure 5.17b shows a similar, reversed sequence for MPI_Irecv.

5.7 Graph Partitioning for Data Placement

Section 5.1.1 described how MPI_Dist_graph_create_adjacent function can be used to create a reordered topology where heavily communicating ranks are placed on the same node. TEMPI implements this functionality by using a modified version of the process_mapping interface from the KaHIP [66, 67] graph partitioning library. KaHIP provides the process_mapping function, but that function does not guarantee that the partitions will be of equal size. TEMPI uses a slight modification that *does* enforce that restriction. For a system with N nodes and P ranks per node, TEMPI will request that KaHIP partition the edge-weight graph into N partitions of P vertices. TEMPI will then arbitrarily assign processes within each group of P to ranks in the old communicator. Distinct from the stencil library, TEMPI does not attempt to place ranks according to the intra-node bandwidth. Section 4.6.2 demonstrated that this has a minimal effect on the performance for the stencil code.

The MPI specification allows the implementation to ignore the reorder request. All MPI platforms tested do not implement the reorder. TEMPI calls the system MPI MPI_Dist_graph_create_adjacent to create the new communicator, but since that communicator has the same ranks as the parent, TEMPI maintains an internal mapping between those ranks and the reordered rank presented to the application. Any MPI operation that uses that communicator has its ranks relabeled by TEMPI before being passed on to the non-reordered system MPI implementation.

Figure 5.18 shows the effect of the data placement strategy on the Astaroth communication time. Like in the stencil library, placement has the largest effect at six ranks per node, since that provides the most opportunities to move communication from off-node to on-node. Unlike the stencil

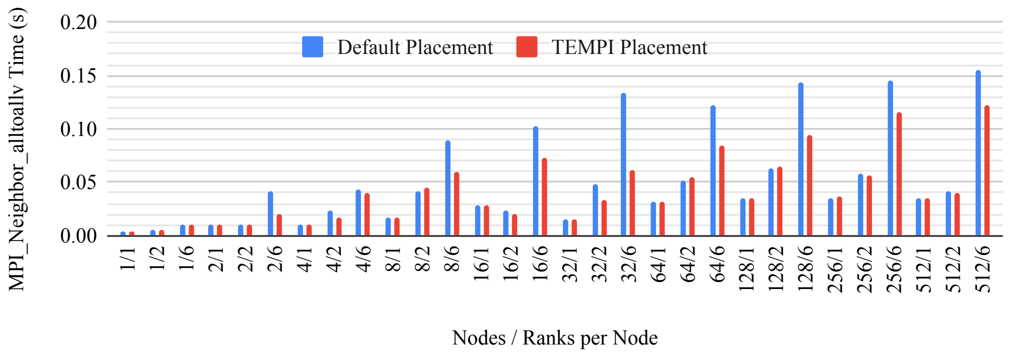


Figure 5.18: MPI_Neighbor_alltoallv time consumed in each Astaroth iteration with default MPI placement or TEMPI automatic placement. Placement only has a substantial effect when six ranks are used on each node, as it offers the most opportunity to reduce off-node communication.

library, TEMPI uses MPI for communication regardless of the location of the two ranks. This further confirms that the benefits from optimized on-node communication (i.e., avoiding the system MPI) are limited compared to the benefits of moving communication on-node. At 512 nodes with 6 ranks per node, placement yields a 1.25× speedup.

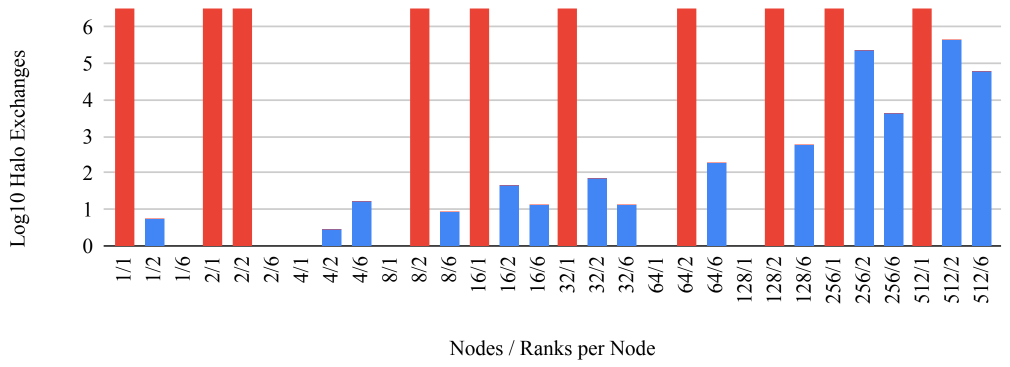


Figure 5.19: Number of Astaroth halo exchanges needed to recoup initial data placement cost. Values above 6 are clipped for display purposes.

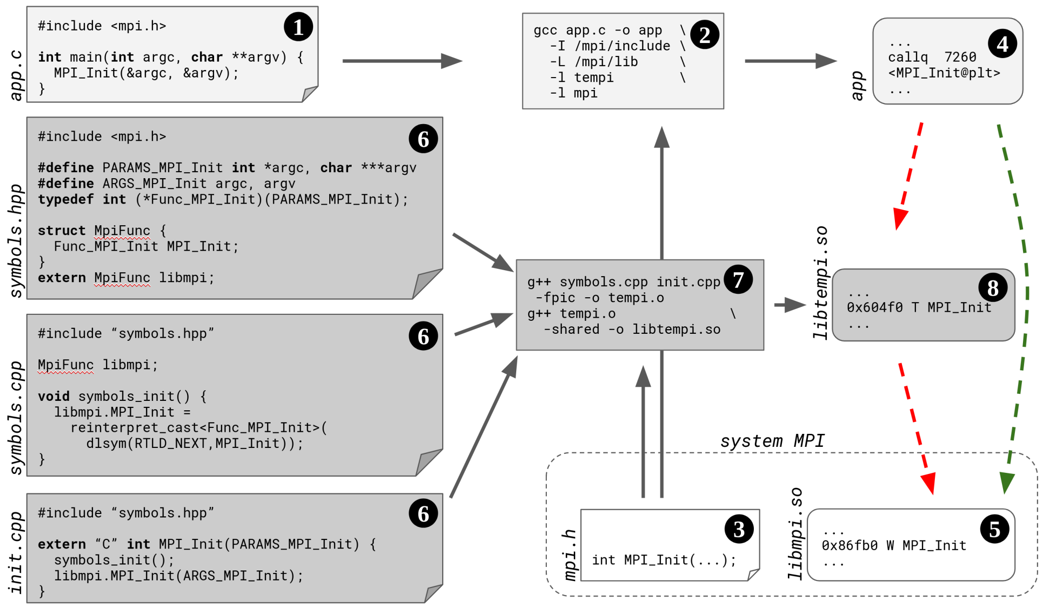
Figure 5.19 shows the cost in iterations to recoup the initial placement cost. First, no effort was made to optimize the performance of the placement algorithm. This is a challenging problem in its own right, and the balance between placement time and improvement of application performance is one that is ripe for future work. Recall that the main effect of placement is to convert inter-node communication to intra-node communication. Configurations with one rank per node (*/1 configurations) will never recoup any placement cost, as there is no way to convert off-node to on-node communication. Figure 5.19 shows these as an infinite cost, as any runtime overhead, even an infinitesimal one that disables placement when there is only one rank per node, can never be recovered through improvement. Following the same line of reasoning, the two rank-per-node cost (*/2 configurations) is higher than the 6 rank-per-node cost (*/6 configurations). This is because there is less benefit from placement when only two ranks can be on each node – there is less opportunity to turn off-node communication into on-node communication. This outweighs the reduced placement cost due to the smaller number of nodes.

Generally, as the number of ranks grows, so does the placement cost. The placement algorithm is executed serially on the root node and results are distributed to all nodes. The cost of the serial placement algorithm grows faster than the improved performance that results. Furthermore, there is a cap on the improvement from placement, as stencil communication is local. Therefore, after each rank has 26 unique neighbors, no further improvement from placement can be achieved (Figure 4.12). Finally, as more nodes are involved, the share of the time devoted to off-node communication grows, even when the off-node communication amount does not increase on a per-node basis. This is due to increased contention in the network and greater distances for the data to travel. All of this contributes to a generally increasing number of iterations to recover the placement time.

5.8 Interposer Library

Despite broad GPU deployment in distributed computing and substantial work in datatype handling (Chapter 6), high-performance handling of GPU datatypes remains a rare feature. The methods of Wei et al. [61] were implemented as a non-public OpenMPI fork that was never merged. MVAPICH-GDR features some fast handling of GPU datatypes [56, 57], but requires [34] the unrelated Mellanox OpenFabrics Enterprise Distribution, which is tied to specific network hardware not present on all systems. Therefore the challenge of deployment seems to be a fundamental one. Consequently, one contribution of this work is to demonstrate that an interposed library can transparently deliver large derived-datatype performance improvements without application modification. The Topology Experiments for MPI (TEMPI) library implements this work and has been tested with OpenMPI 4.0.5, MVAPICH 2.3.4, and Spectrum MPI 10.3.1.2.

5.8.1 Interposer Architecture



```
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
}
```

```
gcc app.c -o app   \
  -I /mpi/include \
  -L /mpi/lib     \
  -l tempi        \
  -l mpi
```

```
...
callq  7260
<MPI_Init@plt>
...
```

```
#include <mpi.h>

#define PARAMS_MPI_Init int *argc, char ***argv
#define ARGS_MPI_Init argc, argv
typedef int (*Func_MPI_Init)(PARAMS_MPI_Init);

struct MpiFunc {
  Func_MPI_Init MPI_Init;
}
extern MpiFunc libmpi;
```

```
g++ symbols.cpp init.cpp
 -fpic -o tempi.o
g++ tempi.o                \
  -shared -o libtempi.so
```

```
...
0x604f0 T MPI_Init
...
```

```
#include “symbols.hpp”

MpiFunc libmpi;

void symbols_init() {
  libmpi.MPI_Init =
    reinterpret_cast<Func_MPI_Init>(
      dlsym(RTLD_NEXT,MPI_Init));
}
```

```
#include “symbols.hpp”

extern “C” int MPI_Init(PARAMS_MPI_Init) {
  symbols_init();
  libmpi.MPI_Init(ARGS_MPI_Init);
}
```

```
int MPI_Init(...);
```

```
...
0x86fb0 W MPI_Init
...
```

Figure 5.20: The application source file (❶) includes the MPI header file provided by the system (❸). It is compiled (❷) and linked with the system MPI implementation. When the binary (❹) is executed, symbols are resolved and the MPI code from the system MPI library is executed. The TEMPI source files (❻) are compiled (❼) into a dynamic library (❽) using the system MPI header. The application is compiled as normal except for TEMPI being inserted into the link order (❷), or an unmodified application can be used with the LD_PRELOAD mechanism. When the application is executed, any symbols defined by the TEMPI library will be resolved there (❽), allowing the TEMPI code to be executed. Any others will be resolved in the system implementation.

The Topology Experiments for MPI library is designed to make MPI modifications available to research and production code without relying on updates to the system MPI implementation. For reference, Figure 5.20 shows a compiled MPI application (❶-❺) and the TEMPI interposer (❻-❽). The application source (❶) includes the system MPI headers (❹) and is compiled (❷) to produce a binary (❸). At run time, the operating system will resolve the symbols in the application binary according to the order of linked libraries, and MPI_Init is found in the system MPI implementation (❺).

TEMPI provides new MPI functionality for unmodified applications by exporting a partial implementation for the MPI interface. For example, init.cpp (❻) implements the MPI_Init function. The TEMPI source includes the system MPI header, and must be compiled (❼) with the same MPI as the target application so that the ABI matches. If the original application can be recompiled, the TEMPI library (❽) may be inserted into the link order before the system MPI library (❷). If not, the TEMPI library can be injected using LD_PRELOAD or similar mechanism (not shown).

Either way, the operating system will search for the MPI_Init symbol in the TEMPI library. As it is found there, that function will be called instead of the system MPI. Internally, TEMPI may ultimately call some system MPI function after introducing its own functionality. This is achieved through the dlsym function. Any parts of the MPI interface that TEMPI does not cover will fall back to the system MPI library automatically.

MPI provides a similar facility, the MPI Profiling Interface. The MPI standard suggests that the "real" MPI should be implemented in the PMPI_* family of functions, and the standard MPI_* interface is just a wrapper around them. Like with TEMPI, someone wanting to implement a profiler could then re-link with a program that implements the PMPI_* family. Unlink PMPI_*, TEMPI is designed to be chained with additional downstream PMPI overloads.

5.8.2 Temporary Buffers

The device and one-shot communication methods pack non-contiguous application data into intermediate buffers on which TEMPI calls system MPI operations. In some cases, those buffers involve pinned host memory, which is slow to allocate on-demand as the system must eagerly back the virtual pages with physical ones from the system memory. Device allocations are also slow since they must interact with the accelerator.

TEMPI uses pool allocators for objects of various sizes. A separate pool of allocations is maintained for each size 2^N bytes. When an intermediate buffer of size M is requested, TEMPI checks pool i where $2^{i+1} > M >= 2^i$. If all allocations in the pool are used, or the pool is empty, TEMPI makes the corresponding cudaMalloc or new/cudaHostRegister call to get a device or pinned host allocation. When the operation that required the allocation is complete, the allocation is released to the pool, but not deallocated. In this manner, the first time an application reaches its maximal intermediate buffer footprint, allocations will be slow, but after that time allocations will not interact with the OS or CUDA driver and will be served very quickly.

Table 5.2: Intermediate allocator statistics

Allocator	Host	Device
Alloc/Dealloc	336	80
Req/Rels	100800	24000
Max Usage (MiB)	38.03125	160

Table 5.2 summarizes the allocator statistics for 100 iterations of the Isend/Irecv Astaroth communication pattern using 512 nodes with 6 ranks per node. As the performance model splits the communications across the device and one-shot method, the host and device allocators are both used. Since the stencil communication has repeated communications over each iteration, only in the first iteration do requests result in using the CUDA or operating system allocators. When those allocations are released, TEMPI does not deallocate them, and uses them to serve future requests. In this manner, each allocation is made to serve 300 separate requests (100 iterations times three halo exchanges per iteration). The trade-off is that each rank consumes an additional 160 MiB of GPU memory and 38 MiB of pinned host memory for the duration of the execution.

5.8.3 Performance Model Cache

Querying the system performance model on each communication can add substantial latency. To determine whether the one-shot or device method is better, the performance model must be evaluated for each method. Since not every possible object contiguous block size or total packed data size is measured, looking up each quantity involves a 1D or 2D interpolation. To evaluate both models, two 1D interpolations ($T_{cpu-cpu}$, $T_{gpu-gpu}$: independent variable is total data size) and four 2D interpolations ($T_{cpu-pack}$, $T_{cpu-unpack}$,

$T_{gpu-pack}$, $T_{gpu-unpack}$: independent variables are total data size and contiguous block size) are required. During a given execution the performance model is static, and therefore the modeling function can be considered to be pure. Once the performance models are evaluated, the choice of device or one-shot method can cached for the next identical invocation. TEMPI uses a C++ std::unordered_map (a hash map) keyed on a tuple of ⟨contiguous block size, object size, colocated⟩ (the last element referring to whether the sending/receiving pair are colocated). The value of the map is whether to use the one-shot or device method. In an iterative application, each iteration may repeatedly send the same quantity of data, and the cache will prevent repeated expensive model queries.

Each unique key tuple will result in a new entry in this cache. Figure 5.21 measures the time it takes to retrieve the cache entry based on the cache size. The hash map provides a constant 5 ns cost on the Summit machine, a negligible addition to the microsecond latency of small messages.

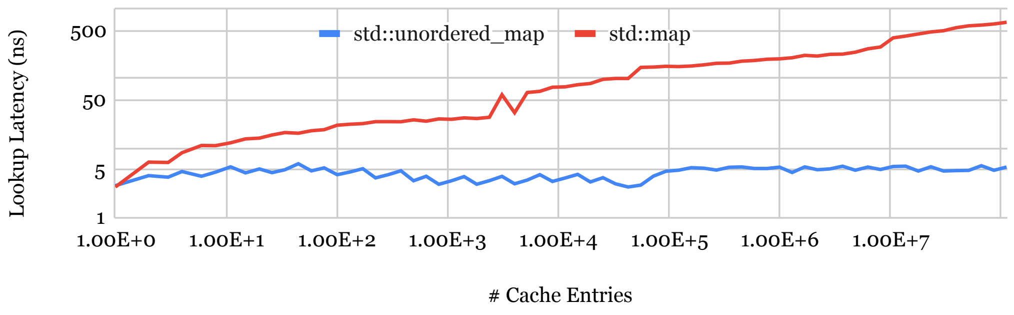


Figure 5.21: Time to retrieve the one-shot vs. device MPI_Send method from the cache, based on the number of unique cache entries. The C++ std::unordered_map (a hash map, amortized constant-time lookup) provides a constant 5 ns latency, while std::map (a binary tree, amortized log-time lookup) becomes more expensive as the cache grows.

For 100 iterations of the Isend/Irecv implementation of the Astaroth communication pattern at 512 nodes with 6 ranks per node, the cache is queried 124,800 times and 5 of those queries are misses, which require an actual evaluation of the performance model. For the stencil case, the number of distinct datatypes is small, however, std::unordered_map should provide equivalent performance regardless of the number of datatypes used in other applications.

5.8.4 IID Testing

TEMPI provides a system measurement binary which determines the constant values for $T_{cpu-cpu}$ and $T_{gpu-gpu}$ (inter- and intra-node) and the pack-/unpack times ($T_{cpu-pack}$, $T_{cpu-unpack}$, $T_{gpu-pack}$, $T_{gpu-unpack}$). $T_{cpu-cpu}$ and $T_{gpu-gpu}$ and measured for 2^i for $0 < i < 23$ bytes, covering both the latency- and bandwidth-bound regimes. The pack/unpack quantities are measured on a 2D sweep of object sizes and contiguous block sizes, with a fixed pitch of 512 bytes. Object size is 2^{2i+6} for $0 <= i < 9$ (64 bytes to 4 MiB). Block size is 2^i for $0 <= i < 9$.

Each benchmark is divided into trials and samples. Each sample measures enough operations to consume at least 200 µs, and represents the mean time measures. At least 7 samples are taken for each trial, and no more than 1 s of time or 500 samples are taken.

Each trial is evaluated using the SP 800-90B [68] statistical tests for randomness. In a true performance measurement, it is expected that each sample will be drawn from the same independent and identically distributed population. If the trial fails, another is repeated, up to 10 trials. If a trial succeeds, the measurement is reported. This strategy for attempting IID measurements is presented for completeness, without evaluation.

5.9 3D Stencil Evaluation

Section 5.1 described how the Astaroth communication pattern was implemented in MPI. Section 5.7 described how placement yielded a 1.25× reduction in halo exchange latency at scale. This section examines in more detail the components that contribute to the overall halo exchange time once placement is applied.

The enormous datatype handling improvement is the primary driver of performance. For reference, Figure 5.22 shows the breakdown of MPI_Pack, MPI_Neighbor_alltoallv, and MPI_Unpack operations for the "alltoallv" halo exchange. Evenly split and consuming effectively all of the time are the MPI_Pack and MPI_Unpack operations. Each phase is timed separately, with MPI_Barrier inserted between them. The longest time consumed by any rank is reported. The average of three runs is reported.

Figure 5.23 shows the same test case when TEMPI's transparent datatype

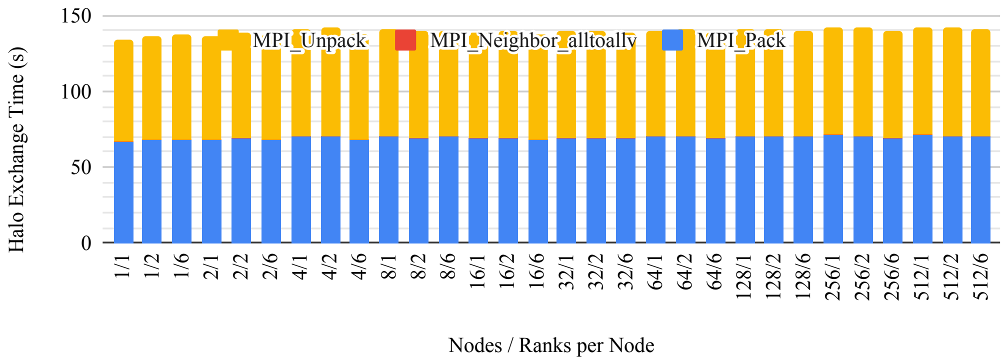


Figure 5.22: Breakdown of MPI_Pack, MPI_Unpack, and MPI_Neighbor_alltoallv time for the Astaroth halo exchange using the baseline Spectrum MPI implementation. The exchange time is dominated by the pack and unpack time, and the alltoallv time is not visible in the chart.

handling is enabled. Though Section 5.4 shows MPI_Pack speedups as large as 242,000×, speedup in the 512/6 configuration is ≈ 970. First, many pack operations will exhibit smaller individual speedup, as the different boundary regions may be more or less contiguous in memory. Second, the time consumed by the alltoallv establishes a maximum speedup of 1147× (if non-contiguous type handling was "free"). Figure 5.24 summarizes the "alltoallv" halo exchange speedup achieved using TEMPI for various MPI configurations for running Astaroth.

Figure 5.25 shows the contribution of various sub-components to the overall halo exchange time for the Isend/Irecv halo exchange implementation. Each component represents the amount of wall time spent by rank 0 in that operation (while the total is the exchange time observed by all ranks).

First, this implementation is slower at all configurations (1.68× at 1/1, $1,45\times$ at 512/6). cudaEventSync, cudaEventQuery, and cudaEventRecord are necessary for tracking completion of asynchronous operations and are not present in the alltoallv method. As such, they can be interpreted as overhead from this implementation choice, but together, they do not contribute enough time to create the overall slowdown except for the smallest of cases.

Second, at scale, the vast majority (86%) of the time is elapsed outside of the TEMPI library. This is not precisely true, as the TEMPI library adds some additional overhead to orchestrate the measured components. However, this does suggest that the performance of the system communication itself is

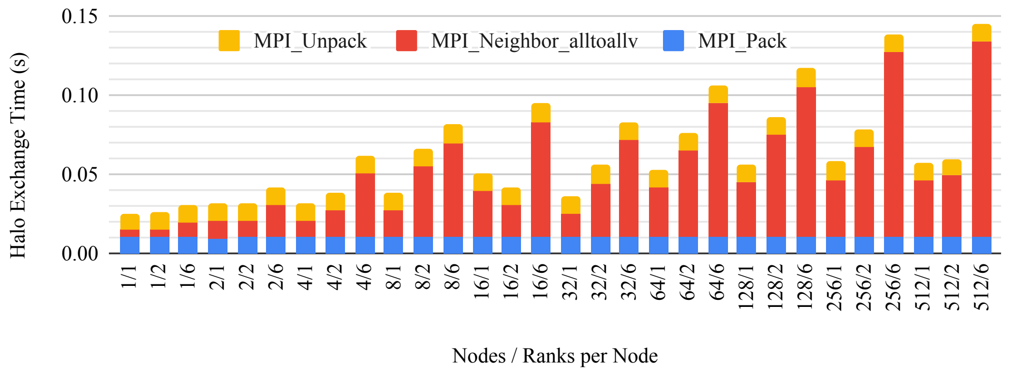


Figure 5.23: Breakdown of MPI_Pack, MPI_Unpack, and MPI_Neighbor_alltoallv in the MPI Astaroth halo exchange using TEMPI (including TEMPI placement). For small numbers of nodes, pack/unpack time dominates, whereas packed data exchange is dominant for larger number of nodes. The Astaroth halo exchange time is measured without concurrent computation kernels, i.e., no contention for GPU execution resources with TEMPI packing/unpacking kernels.

to blame.

5.10 Conclusion

This chapter examined techniques for improving the performance of certain stencil-related MPI operations on the GPU. Chapter 3 introduced how the actual performance of CUDA communication primitives can vary considerably from their theoretical maximums. Chapter 4 followed by describing how to design a stencil halo exchange strategy which minimizes communication, minimizes latency, maximizes overlap, and maximizes bandwidth. This chapter built on both by examining how those lessons could be integrated directly into existing MPI implementations.

Minimizing communication was examined through MPI_Dist_graph_create_adjacent. This MPI function creates a new communicator with attached topology information, associating ranks and their neighbors for future collective operations. This routine also allows ranks to be *reordered*, or renumbered so that communicating ranks are placed on the same node. For a large distributed 3D stencil, this was found to improve halo exchange latency by 1.25×.

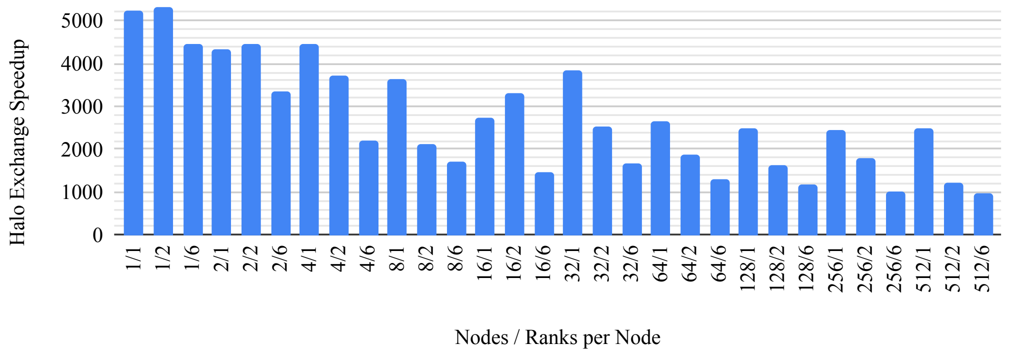


Figure 5.24: Speedup of the alltoallv halo exchange using TEMPI (Figure 5.23) vs. Spectrum MPI (Figure 5.22). Speedup is lower for larger number of ranks as datatype handling is a smaller portion of the total time. At 512 nodes and 6 ranks per node, speedup is 971

Maximizing bandwidth proved to be a core contribution. The baseline MPI primitives did not effectively handle non-contiguous data. A novel MPI derived datatype scheme for GPUs was introduced, with benefits of generality and minimal metadata size while maintaining performance. This scheme reduced baseline MPI_Pack latency by up to 242,000×. Similarly large improvements were observed in MPI_Send.

Overlap between data-packing and communication in the halo exchange pattern was investigated through an implementation using MPI_Isend and MPI_Irecv instead of MPI_Alltoallv. Performance in the point-to-point implementation was observed to be $\approx 1.5\times$ slower than the collective-based implementation. Performance instrumentation in TEMPI ascribed the deficit to MPI primitives themselves, suggesting that previously observed concurrency issues in the Spectrum MPI implementation during unstructured communication are to blame. While TEMPI can be used to bring dramatic performance improvements in some ways, it ultimately still relies on the underlying implementation for inter-process communication. Flawed implementations can inhibit a more thorough understanding of all aspects of the performance.

All of these investigations were carried out through TEMPI, an interposer library that transparently modifies existing MPI applications without requiring a recompilation. TEMPI is publicly available (see Appendix).

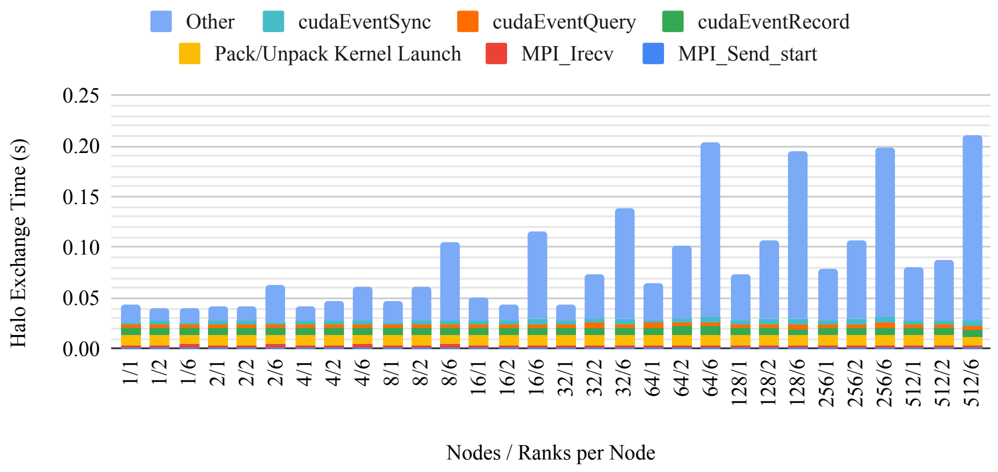


Figure 5.25: Breakdown of Isend/Irecv-based Astaroth halo exchange implementation. Since each rank does a fixed amount of communication, the contribution of various runtime components is relatively constant. Only "other", which includes actual communication time, increases as more ranks are included.

Chapter 6

Related Work

6.1 CUDA Communication Benchmarks

Table 6.1 shows how prior works have overlapped with the microbenchmarks in Comm|Scope. Though some of the specific measurements made by Comm|Scope have been made previously, we believe Comm|Scope represents the most comprehensive coverage of CPU-CPU and CPU-GPU point-to-point communication performance to date.

Li et al. [71, 75] created Tartan, a benchmark suite for evaluating GPU interconnects in the context of machine-learning workloads. Tartan includes microbenchmarks for point-to-point and collective GPU-GPU communication within and across nodes. To that end, Tartan measures bandwidth, latency, and efficiency of GPU-GPU explicit memory copies and the Nvidia Collective Communications Library (NCCL) on PCIe, NVLink 1.0, NVLink 2.0, and Infiniband systems with GPUDirect RDMA. Unlike Tartan, Comm|-Scope includes CPU/GPU transfers, but only measures point-to-point transfer bandwidth within a single node. Tartan also includes 14 larger application benchmarks. Those benchmarks are categorized by what communication pattern they exhibit. Some of those benchmarks are categorized into the CPU-GPU communication pattern, but Tartan does not include corresponding CPU-GPU microbenchmarks.

Tallent et al. [69] evaluate the effect of multi-GPU systems on deep-learning workloads. Along the way, they measure point-to-point explicit GPU-GPU copy bandwidth with and without peer access available, which are two of the microbenchmarks included in Comm|Scope. They also have some GPU/GPU latency and collective communication bandwidth measurements, similar to Tartan.

Ben-nun et al. [76, 70] present Groute and MGBench. Groute is a multi-

Table 6.1: Comm|Scope data transfer microbenchmark coverage, and summary of where related work overlaps. Comm|Scope defines bandwidth benchmarks for all unidirectional and bidirectional primitive CUDA point-to-point transfers.

Host Allocation	Device Allocation	Transfer	Kind	Our	[69]	[70]	[71]	[36]	[72]	[73]	[74]
NUMA / pageable	cudaMalloc	explicit	H2D	✓							
			D2H	✓							
			bi	✓							
pageable	cudaMalloc	explicit	H2D	✓				✓	✓		✓
			D2H	✓				✓	✓		
			bi	✓							
NUMA / pinned	cudaMalloc	explicit	H2D	✓		✓					
			D2H	✓						✓	
			bi	✓							
pinned	cudaMalloc	explicit	H2D	✓		✓		✓		✓	✓
			D2H	✓		✓		✓		✓	
			bi	✓						✓	
mapped	–	implicit (zero-copy)	H2D	✓		✓					
–	cudaMalloc	implicit (zero-copy)	D2D	✓		✓					
			D/D bi	✓		✓					
–	cudaMalloc	explicit / peer	D2D	✓	✓	✓	✓			✓	
			bi	✓		✓	✓			✓	
–	cudaMalloc	explicit / no peer	D2D	✓	✓		✓			✓	
			bi	✓			✓				
cudaMallocManaged		demand page migration	H2D	✓					✓		
			D2H	✓					✓		
			H/D bi	✓							
			D2D	✓							
			D/D bi	✓							
cudaMallocManaged		prefetch	H2D	✓							✓
			D2H	✓							
			H/D bi	✓							
			D2D	✓							
			D/D bi	✓							
pinned	cudaMalloc	cudaMemcpy2DAsync	D2H	✓							
			H2D	✓							
–	cudaMalloc	cudaMemcpy2DAsync	D2D	✓							
pinned	cudaMalloc	cudaMemcpy3DAsync	D2H	✓							
			H2D	✓							
–	cudaMalloc	cudaMemcpy3DPeerAsync	D2D	✓							
pinned	cudaMalloc	custom kernel	H2D	✓							
			D2H	✓							
–	cudaMalloc	custom kernel	D2D (pull)	✓							
			D2D (push)	✓							

GPU programming model, and MGBench was developed partially to understand multi-GPU communication patterns before developing Groute. MGBench includes some host/GPU and GPU/GPU zero-copy benchmarks for coalesced and random accesses. MGBench includes device synchronization time in their bandwidth measurements.

The SHOC benchmark suite [36] is meant to measure the performance of heterogeneous systems running OpenCL and CUDA workloads. SHOC includes unidirectional bandwidth measurements of point-to-point transfers between CPU and GPU.

Landaverde, Zhang, Coskun, and Herbordt [72] investigate the effect of porting several benchmarks to use CUDA's unified memory system on PCIe systems. They present microbenchmarks that correspond to the unidirectional host/device coherence measurements in Comm|Scope. They also have the corresponding implementation using explicit point-to-point copies. They do not account for NUMA topology in their measurements and only consider PCIe systems.

Spafford, Meredith, and Vetter [77] measure NUMA and contention in multi-GPU systems. They overlap with Comm|Scope by measuring point-to-point NUMA-aware CPU/GPU bandwidth on PCIe systems. Though they do not specify, the results suggest that they are measuring bandwidth from pinned host allocations.

Though the CUDA SDK Samples [73] are not intended as a performance measurement tool, they provide demonstration code that measures host/device bandwidth for pageable and pinned allocations and bandwidth and latency of unidirectional and bidirectional GPU/GPU transfers with and without peer access enabled. None of these samples consider NUMA effects, and there are no comparable unified memory performance measurement programs.

Mukherjee et al. [78] describes a microbenchmark that measures data transfer performance in the HSA 1.0 unified memory system and the OpenCL 2.0 shared virtual memory. This work includes similar demand migration bandwidth measurements in CUDA.

Roberts, Ramanna, and Walthour [79] investigate data movement on the AC922 platform, the same that is used on Summit. It measures RDMA bandwidth for a variety of transfer sizes inter-node, whereas this work considers intra-node.

Gu et al. [80] reimplement benchmarks from other benchmark suites using unified memory, but do not include any data transfer microbenchmarks. Chien, Peng, and Markidis [81] do something similar, with an emphasis on advanced managed memory features. Comm|Scope examines prefetching as one of the point-to-point bulk modalities.

Li et al. [82], published concurrently with Comm|Scope, measures latency and bandwidth on systems similar to those in this work, as well as the NvSwitch technology not available at the time of writing.

The cuda-benches software [74] includes a variety of CUDA-related microbenchmarks, including unified memory streaming and cudaMemcpyAsync.

Pai [83] develops some benchmarks for unified memory in CUDA 6.0. The benchmarks are developed to understand which situations incur repeated accesses, instead of measuring bandwidth. An associated (refereed) paper by Pai et al. [84] motivates the creation of the microbenchmarks.

6.2 3D Stencil

Chapter 4 described a stencil library that incorporated automatic partitioning, data placement, and communication specialization. Prior work has focused on kernel performance and largely neglected communication considerations. Some prior work also was unable to consider the multi-GPU or heterogeneous communication case, as multi-node GPUs did not exist or were not common at the time. Thibault and Senocak [85] use a single-node multi-GPU platform, but do not overlap multiple communications on a single GPU. Jacobsen, Thibault, and Senocak [86] use the staged method, and only consider one GPU per rank. Yongpeng and Frank [87], Steuwer et al. [88], Shimokawabe, Aoki, and Onodera [89], and Sourouri, Baden, and Cai [90] all focus on abstraction, code generation and/or autotuning, but use staged communication. Sourouri et al. [91] present a distributed stencil implementation, but all communication is *staged*.

Several works stand out as paying more attention to communication in the context of stencil. Maruyama et al. [92] implement a kernel-based pack-/unpack through zero-copy memory for non-unit-stride transfers, similar to the pack/unpack scheme used for many transfers in this work. Lutz, Fensch, and Cole [93] describe a code generation and autotuning framework. They observe that multi-GPU PCIe heterogeneity causes some slowdown, and include data placement in their autotuning framework. However, instead of a performance-model-based approach, they simply treat the placement space as a black box and apply various optimization heuristics to it. Furthermore, they do not do any communication specialization depending on the node topology. Sourouri et al. [94] investigate multi-GPU performance in the context of a 3D stencil. They use multiple communication streams, and multiple threads within a single address space to control multiple GPUs, and overlap communications. This also allows them to bypass MPI communication for intra-node exchanges. They do not analyze the communication performance, and do not consider node topology. Faraji, Mirsadeghi, and Afsahi [95] characterize the multi-GPU communication problem as a topology-detection, communication evaluation, and mapping problem, like this work. They do not use communication specialization, and instead use the *staged* method.

6.3 MPI Datatype Handling

The datatype handling work presented in Chapter 5 distinguishes itself from related work in three ways. First, TEMPI can be used today without waiting for MPI implementers or HPC system administrators. Second, while prior work uses GPU kernels to accelerate datatype operations, TEMPI is the first work that shows transformations on structured datatypes for canonicalization (as opposed to handling specific cases, or reducing everything to a list of offsets and lengths). This provides wide datatype coverage, tiny GPU memory consumption for metadata, and fast generic kernels. Third, prior work examines how to integrate data type handling into MPI more deeply. As TEMPI uses a library-interposer interface on top of MPI, it is not able to make those low-level changes. Despite that, enormous performance improvement is obtained.

The MPITypes library [96] is one of the first attempts to generalize datatype handling outside of MPI. It provides several functions for flattening and copying datatypes, and a framework for extending those operations. TEMPI tries to maintain the structured information of types so the MPITypes operations are not directly applicable.

Wang et al. [59] describe an early approach in MVAPICH2. Several options are considered, ultimately selecting a pipelined version of the "staged" method that uses cudaMemcpy2D instead of a kernel. Since MVAPICH 1.8a2, MVAPICH had accelerated support for derived types created through MPI_type_vector or MPI_Type_create_hindexed, as long as the base type is a named type. Since MVAPICH 2.2, it has used kernels to accelerate some operations [97], which is still the case as of 2.3.4. Different kernels for different named datatypes exist, but no optimizations are present for nested datatypes, or operations on more than one datatype.

Jenkins et al. [63] provide fast handling of arbitrary MPI datatypes on the GPU. Nested types are represented by a tree structure that must be traversed by each GPU thread using division, modulo, and binary search operations. They restrict inter-node communication to the one-shot method. Section 5.5 described how that is not always preferable due to the relative cost of zero-copy pack operations vs. explicit data transfer.

Shi et al. [60] introduce Hand, an approach for non-contiguous data movement in MPI. Hand also also explicitly breaks the problem into transformation and kernel selection phases. Hand defines specific kernels for handling vector, hvector, subarray, and indexed block types. For other datatypes, it transforms a variety of datatypes into a blocklist, for which it has a specific kernel implementation. Instead, TEMPI recognizes that nested, strided types reduce to (essentially) a subarray, and explicitly designs a transformation and optimal packing kernel to cover all of those scenarios.

Wu et al. [61] describe a fork of OpenMPI that integrates derived datatype handling both on the GPU itself as well as communication between nodes. The datatype is ultimately represented as a list of blocks, and blocks are partitioned among separate kernels with pipelined communication. It also identifies that full GPU resources for handling non-contiguous data are not needed to saturate the communication links. This fork has remained unmerged and not publicly available.

Chu et al. [56] recognize that one of the challenges of all prior work is the latency of kernel launches. Like prior work, they also represents the datatypes as a list of displacements and lengths. Similar to this work, they define extraction, conversion, and caching steps, and use one-shot packing and unpacking. Unlike TEMPI, they do not recognize when the one-shot packing to the host is slower, due to inefficiency of packing and unpacking over the interconnect. Chu et al. [57] identify that a major cost of transfer is the launch of the packing kernels. They develop an engine that is able to merge various packing requests into a single kernel launch. TEMPI addresses the packing kernel launch cost by issuing a single kernel for multiple copies of the same MPI datatype, but cannot fuse further than that. These two works appear to have been integrated into MVAPICH2-GDR. MVAPICH does not include handling for generic datatypes, but does have more primitive handling for specific datatypes.

Hashmi et al. [62] describe a zero-copy-based data movement system for MPI datatypes. They include kernels where a warp is responsible for a contiguous block in a block list. They also describe a variety of integrations with the underlying communication library, which TEMPI is unable to address due to its interposer model.

6.4 Scientific Libraries

A variety of projects attempt to bring a broad suite of distributed computing techniques under a single umbrella to accelerate the development of distributed codes. Broadly, these libraries all recognize that developing parallel high-performance code is especially challenging due to the many relevant aspects of system performance. They try to provide fast primitives and parallel algorithms upon which applications can be developed.

CMSSL [98] is a scientific library for the CM-2, CM-200, and CM-5 computer systems that provides routines for data distribution and some operations that are independent of underlying data distribution. Of particular note is automatic algorithm selection at runtime, similar to the stencil communication library and TEMPI. For on-node computation CMSSL transforms the loop iteration space of various operations according to the size of different levels of the memory hierarchy. The stencil communication library takes a similar style for stencil communication specifically, where implementation decisions are made according to the theoretical properties of the system. TEMPI takes this a step further by trying to optimize communication according to measured performance instead of theoretical properties. TEMPI's approach should allow it to automatically support machines which do not yet exist, but have performance characteristics that are predictable from current trends. For distributed communication, CMSSL attempts to adjust the algorithm to minimize data movement based on properties of the inputs (e.g., matrix shapes in matrix multiplication). The stencil communication library take a similar approach when partitioning the distributed grid to minimize communication.

Other similar efforts include PetSC [99, 100, 101], FleCSI [102], Zoltan [103], Hypre [104], and Trilinos [105]. These libraries target different applications, including partial differential equations (PetSC), dynamic load balancing and graph applications (Zoltan), multi-physics (FleCSI), linear solvers (HYPRE), and general solvers (Trilinos). They share an approach, which provides data structures and interfaces to abstract away the details of distributed memory, allowing the application code to focus on computation instead of communication and data placement. The work described herein shares a similar goal, but generally distinguishes itself by creating a communication plan derived from measured system characteristics.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The work described herein shows that empirical communication measurements can be used to automatically schedule and execute intra- and inter-node communication in a modern heterogeneous system, providing "hand-tuned" performance without the need for complex or error-prone communication development at the application level. The rise of accelerators has changed the high-performance computing paradigm, making movement of data a first-class concern, and performance improvements can be realized from measurement of hardware interconnects

Chapter 3 establishes that the effective bandwidth of the interconnect cannot be derived from the technical specifications. The source and destination memory types, API choice, cache, and underlying hardware all contribute to the ultimate performance, so making the proper selection is crucial. That chapter also described procedures for accurately doing the measurements on the CUDA platform. The software is open-source and freely available.

Chapter 4 applies the insights from Chapter 3 to a stencil library. The library is designed around qualitative observations, especially the significant cost of CUDA runtime APIs, the lack of latency improvement provided by multiple CUDA threads, the improved interconnect utilization with simultaneous bidirectional transfers, and the measurable bandwidth impact of "near" and "far" GPUs. CUDA runtime API performance is mitigated by using the CUDA graph API to capture and replay communication operations. Interconnect utilization was achieved by designing the library to overlap CUDA and MPI operations. GPU locality was captured with a simple data placement scheme. Ultimately, the library was able to reduce the iteration time of a state-of-the-art stencil code by a geometric mean of 1.17×, or

1.45× at 3072 GPUs.

Chapter 5 extends the work in two further ways. First, it uses the MPI implementation on the Summit supercomputer as a platform for integrating some of the stencil communication techniques directly into MPI. A novel MPI derived datatype handling strategy is introduced, which allows fast GPU performance on a variety of regular MPI datatypes, with a compact representation. Quantitative system performance modeling supported by measurements from Chapter 3 are used to improve the derived datatype handling of MPI communication primitives. Additionally, a low-overhead caching scheme is used to allow the implementation to be tuned at runtime based on the nature of the moved data.

Ultimately, this work finds that it is possible to take advantage of careful measurement of data-movement performance to improve high-level libraries and underlying communication primitives. The code for each of the three main chapters is available under a liberal license. At the time of writing, all projects are developed on Github, with additional copies of the source code uploaded to Zenodo. The Appendix describes in more detail how to retrieve the source code.

7.2 Future Work

7.2.1 Comm|Scope

Despite Comm|Scope's comprehensive evaluation of CUDA point-to-point bulk transfer primitives, zero-copy and managed memory are both modalities where bulk transfer bandwidth provides only superficial insight. Those modalities allow flexibility of access granularity, density, and alignment, all of which impact performance independently. Furthermore, they enable atomic operations, which can be used for fine-grained coordination between CPUs and GPUs. None of these advanced considerations are measured.

An additional challenge comes in the face of multiple communicating GPUs. Depending on the interconnect topology, contention may appear on some of the links, further complicating evaluation. This scenario is of particular interest in shared or virtualized environments, and in multi-GPU applications. Similarly, MPI collectives allow multiple off-node GPUs to communicate simultaneously, layering the complexity of contention and MPI primitives.

Further complexity is introduced when GPUs communicate off-node. Chapter 5 demonstrated how even on-node bandwidth measurements can be used to improve the performance of individual MPI primitives. Just as Comm|Scope evaluates the various CUDA communication schemes, it should be extended to measure the various MPI methods. These are analogous to the intra-node communication primitives and face similar problems with variability, advanced features, parallel communication, and contention.

In order to fully support performance-driven automation for distributed systems, Comm|Scope should be expanded to include evaluations of MPI collectives as well. Using the Comm|Scope measurements to influence application performance was hampered by the interaction of multiple simultaneous primitives. These collectives layer the challenges of multi-GPU communication with off-node primitives, but once the simpler operations are understood, it may be possible to reliably measure and understand the aggregate behavior.

7.2.2 Stencil Communication Library

The pack/unpack performance was largely unconsidered in the stencil library design and evaluation. Work on TEMPI showed that packing data into the host may sometimes be preferable to packing on the device, and the stencil communication library could incorporate that result. The stencil communication library's partitioning strategy assumes that all bytes have equal communication cost. However, each "face" of the 3D subgrid has a different stride, and TEMPI shows that the communication performance is influenced by the strategy for handling the non-contiguous data. A performance model for each face should be developed. The same holds true for the actual cost of communicating that face used instead of the number of bytes communicated.

Sophisticated stencil codes feature dynamic mesh refinement, where regions of interest are re-discretized at higher resolution to capture smaller-scale behavior more accurately. Since it is not known ahead of time where in the grid these regions will occur, they are dynamically detected, and grid adjusted accordingly. This provides a challenge for static communication planning, as the requirements may change from iteration to iteration, and neighboring data needs to be up- and down-sampled across the refinement boundary. While the up- and down-sampling is related to communication, it is likely that some input would be required from the application, as the strategy would affect the accuracy of the results. As long as the dynamic refinement does not change frequently, it would be possible to create a new static plan after each refinement and accumulate the results over many iterations. In this case, the cost of the communication planning would become a primary concern.

Certain implementation decisions of the library were not evaluated. The stencil library choses to implement the non-contiguous data handling through GPU kernels. This design allows full flexibility for arbitrary block sizes and alignments. However, it competes with the application code for GPU compute resources. Certain transfers may be compatible with the GPU's DMA engine, allowing the GPU compute elements to remain uninvolved. The stencil library does not take advantage of MPI persistent communications (MPI_Send_init, etc.). These operations promise to remove some of the overhead of repeated communication, for example, ensuring that the receiver has buffer space allocation to receive a large message. With a comprehensive benchmark like Comm|Scope extended to measure the actual impact of persistent communications, the appropriate implementation could be selected in an automatic way. The performance effect of the cudaGraphAPI was never quantitatively examined. It is expected to be most relevant when communication is fastest, as it reduces the overhead for each communication. The final consideration would be to redesign the API to allow exterior gridpoint computations to begin as soon as the corresponding part of the halo exchange is complete. In certain circumstances, this could allow the exterior gridpoint computations to overlap with the communication.

7.2.3 TEMPI

Chapter 5 examined how to generalize lessons from the CUDA+MPI stencil code into MPI in general. The amount of work that remains to be done in this area is vast. MPI collectives provide an opportunity to batch GPU operations effectively. Previous work in MPI derived types could be integrated to MPI's system to support indexed types. Persistent communications provide an opportunity to remove GPU kernel overhead.

While the data placement approach provides a large improvement in iteration time, it often takes tens or hundreds of thousands of iterations to recoup the initial cost of the placement. Fundamentally, data placement algorithms will trade off speed with quality, and this is an area not investigated in this work.

TEMPI could also be used to introduce and evaluate experimental extensions to MPI's interface. Such extensions could bring new high-level operations into MPI or allow experimentation with various implementation strategies for existing functions. A "custom collective" might be of particular interest, as it would allow an application to describe a communication pattern (like how an application can describe non-contiguous data). This would allow the MPI implementation to plan and batch various accelerator operations.

TEMPI was designed with this future work in mind, and should allow future research capabilities to be added to any existing MPI platform.

7.2.4 High-Level Programming Systems

A high-level programming system typically adopts a more task-oriented model. Each task comes with input and output data, and the system is responsible for deciding when and where those tasks run (subject to dependency constraints). In the most holistic case, a high-level description can decouple the desired functionality from its implementation and provide portability across architectures and machines. On the surface, a high-level system is ripe for the kinds of automatic performance-dependent decisions described in this work. The key challenge would be to sufficiently minimize any runtime cost so that better utilization of the underlying system becomes the bottleneck.

Any high-level system that deals with non-contiguous data will be able to make use of the datatype handling strategy described in this work. In this work, it was fully integrated with MPI, but in general, the process of going from a description of non-contiguous data to a fast handling strategy is a valuable one. The most straightforward approach would be to extract that code out into its own library. That library could mirror the design of MPI's derived datatype system, or it could adopt a similar interface. Another approach would be to integrate it directly into a compiler or runtime system. This would allow the higher-level application code to entirely ignore the non-contiguous nature of the data.

Appendix

Artifacts

The source code for Comm|Scope is available under the Apache License 2.0. It is currently hosted at `https://github.com/c3sr/comm_scope`. An archive of the state of the code at the time of writing was uploaded to Zenodo [106], with the DOI 10.5281/zenodo.4586913 [107].

The source code for the stencil library is available under the Boost Software License 1.0. It is currently hosted at `https://github.com/cwpearson/stencil`. An archive of the state of the code at the time of writing was uploaded to Zenodo, with the DOI 10.5281/zenodo.4635277 [108].

The source code for TEMPI is available under the Boost Software License 1.0. It is currently hosted at `https://github.com/cwpearson/tempi`. An archive of the state of the code at the time of writing was also uploaded to Zenodo, with the DOI 10.5281/zenodo.4584107 [109].

References

[1] Intel, "Intel AVX," 2017. [Online]. Available: https://software.intel.com/en-us/isa-extensions/intel-avx

[2] *System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.4*, Advanced Micro Devices Std. 0.99.4, 2010.

[3] ARM, "NEON," 2017. [Online]. Available: https://developer.arm.com/technologies/neon

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[5] Huawei, "Kirin 970," 2017. [Online]. Available: http://consumer.huawei.com/en/press/news/2017/ifa2017-kirin970/

[6] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura et al., "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[7] Intel, "Intel Nervana hardware," 2017. [Online]. Available: https://www.intelnervana.com/intel-nervana-hardware/

[8] A. Kandangath and G. Badie, "What's new in Core Motion," 2011, Apple. [Online]. Available: https://developer.apple.com/devcenter/download.action?path=/wwdc_2011/adc_on_itunes__wwdc11_sessions__pdf/423_whats_new_in_core_motion.pdf

[9] "Virtex UltraSCALE+ product table," 2018, Xilinx. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable

[10] "Intel Stratix 10 FPGAs," 2018, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html

[11] "About Agilio SmartNICs," 2018, Netronome. [Online]. Available: https://www.netronome.com/products/smartnic/overview/

[12] L. Codrescu, "Qualcomm Hexagon DSP," 2013, Qualcomm. [Online]. Available: https://developer.qualcomm.com/qfile/27696/qualcomm-hexagon-architecture.pdf

[13] "Digital signal processors," 2018, NXP. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/digital-signal-processors:Digital-Signal-Processors

[14] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[15] "Vision processing unit," 2018, Movidius. [Online]. Available: https://www.movidius.com/solutions/vision-processing-unit

[16] "The evolution of EyeQ," 2018, Mobileye. [Online]. Available: https://www.mobileye.com/our-technology/evolution-eyeq-chip/

[17] N. Baker, "Mixed reality," Youtube. [Online]. Available: https://www.youtube.com/watch?v=u0eBd2m_wEs#t==27m16s

[18] PCI-SIG, *PCI Local Bus Specification*, Std. 3.0, 2001. [Online]. Available: https://pcisig.com/specifications

[19] Intel, *Accelerated Graphics Port Interface Specification*, Std. 1.0, 1996. [Online]. Available: http://www.playtool.com/pages/agpcompat/agp10.pdf

[20] Intel, *Accelerated Graphics Port Interface Specification Revision 2.0*, Std. 2.0, 1998. [Online]. Available: http://www.motherboards.org/files/techspecs/agp20.pdf

[21] Intel, *AGP Interface Specification*, Std. 3.0, 2002. [Online]. Available: http://download.intel.com/support/motherboards/desktop/sb/agp30.pdf

[22] PCI-SIG, *PCI-X Addendum to the PCI Local Bus Specification*, Std. 1.0a, 2000.

[23] PCI-SIG, *PCI-X Addendum to the PCI Local Bus Specification*, Std. 2.0, 2002.

[24] PCI-SIG, "PCI Express 3.0 frequently asked questions," 2014. [Online]. Available: http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2

[25] PCI-SIG, "PCI Express 4.0 frequently asked questions," 2014. [Online]. Available: http://www.pcisig.com/news_room/faqs/FAQ_PCI_Express_4.0/#EQ3

[26] Nvidia, "Nvidia Tesla P100," Tech. Rep., 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[27] Nvidia, "Nvidia Tesla V100 GPU architecture," Tech. Rep., 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[28] Nvidia, "Nvidia A100 tensor core GPU architecture," Tech. Rep., 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[29] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs," *Proc. VLDB Endow.*, vol. 14, no. 2, p. 114–127, Oct. 2020. [Online]. Available: https://doi.org/10.14778/3425879.3425883

[30] M. standards committee, *MPI: A Message-Passing Interface Standard Version 3.1*, Std. 3.1, 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[31] IBM, "IBM Spectrum MPI version 10 release 1 user's guide," Tech. Rep., 2016. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSZTET_EOS/eos/guide_101.pdf

[32] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A flexible high performance MPI," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 228–239.

[33] W. Gropp, "MPICH2: A new start for MPI implementations," in *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Berlin, Heidelberg: Springer-Verlag, 2002, p. 7.

[34] The MVAPICH Team, "Mvpiach2-gdr 2.3.5," Tech. Rep., 2020.

[35] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu, "Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297663.3310299 p. 209–218.

[36] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1735688.1735702 p. 63–74.

[37] R. J. Wysocki, *intel_pstate CPU Performance Scaling Driver*, 2017. [Online]. Available: https://web.archive.org/web/20201112004549/https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html

[38] *POWER ISA*, 2.07B ed., IBM, Jan. 2018.

[39] *AMD64 Architecture Programmer's Manual*, 3.26 ed., Advanced Micro Devices, May 2018.

[40] *num - NUMA policy library*, man7.org, 2020. [Online]. Available: https://man7.org/linux/man-pages/man3/numa.3.html

[41] "Summit user guide," 2020. [Online]. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[42] C. B. Stunkel, R. L. Graham, G. Shainer, M. Kagan, S. S. Sharkawi, B. Rosenburg, and G. A. Chochia, "The high-speed networks of the Summit and Sierra supercomputers," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 3:1–3:10, 2020.

[43] A. B. Caldiera, "IBM power system AC922 introduction and technical overview," Tech. Rep., 2018. [Online]. Available: https://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf

[44] C. Pearson, M. Hidayetoğlu, M. Almasri, O. Anjum, I. Chung, J. Xiong, and W. W. Hwu, "Node-aware stencil communication for heterogeneous supercomputers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 796–805.

[45] H. Hotta, M. Rempel, and T. Yokoyama, "High-resolution calculations of the solar global convection with the reduced speed of sound technique. I. the structure of the convection and the magnetic field without the rotation," *The Astrophysical Journal*, vol. 786, no. 1, p. 24, 2014.

[46] A. Beresnyak, "Spectra of strong magnetohydrodynamic turbulence from high-resolution simulations," *The Astrophysical Journal Letters*, vol. 784, no. 2, p. L20, 2014.

[47] O. Anjum, G. de Gonzalo Simon, M. Hidayetoglu, and W.-M. Hwu, "An efficient GPU implementation technique for higher-order 3D stencils," in *2019 IEEE 21st International Conference on High Performance Computing and Communications(HPCC)*. IEEE, 2019, pp. 552–561.

[48] J. Skála, F. Baruffa, J. Büchner, and M. Rampp, "The 3D MHD code GOEMHD3 for astrophysical plasmas with large Reynolds numbers-code description, verification, and computational performance," *Astronomy & Astrophysics*, vol. 580, p. A48, 2015.

[49] J. Pekkilä, M. S. Väisälä, M. J. Käpylä, P. J. Käpylä, and O. Anjum, "Methods for compressible fluid simulation on gpus using high-order finite differences," *Computer Physics Communications*, vol. 217, pp. 11–22, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001046551730098X

[50] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for GPU memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–81.

[51] "Nvidia GPUDirect," 2021. [Online]. Available: https://developer.nvidia.com/gpudirect

[52] M. A. Heroux, J. Dongarra, and P. Luszczek, "HPCG benchmark technical specification," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2013.

[53] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management service for parallel dynamic applications," *Computing in Science & Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

[54] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[55] Nvidia, "NVIDIA Nsight Systems," 2021. [Online]. Available: https://developer.nvidia.com/nsight-systems

[56] C. Chu, J. M. Hashmi, K. S. Khorassani, H. Subramoni, and D. K. Panda, “High-performance adaptive MPI derived datatype communication for modern multi-GPU systems,” in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 267–276.

[57] C. H. Chu, K. S. Khorassani, Q. Zhou, H. Subramoni, and D. K. Panda, “Dynamic kernel fusion for bulk non-contiguous data transfer on GPU clusters,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 130–141.

[58] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, “The MVAPICH project: Transforming research into high-performance MPI library for HPC community,” *Journal of Computational Science*, p. 101208, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750320305093

[59] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, “Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2,” in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 308–316.

[60] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. K. Panda, “HAND: A hybrid approach to accelerate non-contiguous data movement using MPI datatypes on GPU clusters,” in *2014 43rd International Conference on Parallel Processing*, 2014, pp. 221–230.

[61] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and J. Dongarra, “GPU-aware non-contiguous data movement in open MPI,” in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi-org.proxy2.library.illinois.edu/10.1145/2907294.2907317 p. 231–242.

[62] J. M. Hashmi, C.-H. Chu, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, “FALCON-X: Zero-copy MPI derived datatype processing on modern CPU and GPU architectures,” *Journal of Parallel and Distributed Computing*, 2020.

[63] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur, “Processing MPI derived datatypes on noncontiguous GPU-resident data,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2627–2637, 2014.

[64] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp. 13–22.

[65] A. Bienz, L. N. Olson, W. D. Gropp, and S. Lockhart, "Modeling data movement performance on heterogeneous architectures." arXiv:2010.10378v2, 2020.

[66] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.

[67] C. Schulz and J. L. Träff, "Better Process Mapping and Sparse Quadratic Assignment," in *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*, ser. LIPIcs, vol. 75. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 4:1–4:15.

[68] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, and M. Boyle, "Recommendation for the entropy sources used for random bit generation," NIST, Tech. Rep. 800-90B, 2018.

[69] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating on-node GPU interconnects for deep learning workloads," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems.* Springer, 2017, pp. 3–21.

[70] T. Ben-Nun, "MGBench," https://github.com/tbennun/mgbench, 2017.

[71] A. Li, S. L. Song, J. Cheng, X. Liu, N. Tallent, and K. Barker, "Tartan: Evaluating modern GPU interconnect via a multi-GPU benchmark suite," in *International Symposium on Workload Characterization, IEEE*, 2017.

[72] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE.* IEEE, 2014, pp. 1–6.

[73] "CUDA 9.2 Toolkit Downloads," https://developer.nvidia.com/cuda-92-download-archive, NVIDIA, 2018.

[74] D. Ernst, "cuda-benches," https://github.com/te42kyfo/cuda-benches, 2019.

[75] A. Li, "Tartan," https://github.com/uuudown/Tartan, 2018.

[76] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *ACM SIGPLAN Notices*, vol. 52, no. 8. ACM, 2017, pp. 235–248.

[77] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying NUMA and contention effects in multi-GPU systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 11.

[78] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli, "A comprehensive performance analysis of HSA and OpenCL 2.0," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 183–193.

[79] S. Roberts, P. Ramanna, and J. Walthour, "Ac922 data movement for coral," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–5.

[80] Y. Gu, W. Wu, Y. Li, and L. Chen, "UVMBench: A comprehensive benchmark suite for researching unified virtual memory in GPUs," *arXiv preprint arXiv:2007.09822*, 2020.

[81] S. Chien, I. Peng, and S. Markidis, "Performance evaluation of advanced features in CUDA unified memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019, pp. 50–57.

[82] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[83] S. Pai, "Benchmarking unified memory in CUDA 6.0," https://www.cs.rochester.edu/u/sree/automem/, 2018.

[84] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2370816.2370824 p. 33–42.

[85] J. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition*, 2009, p. 758.

[86] D. Jacobsen, J. Thibault, and I. Senocak, “An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters,” in *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2010, p. 522.

[87] Y. Zhang and F. Mueller, “Auto-generation and auto-tuning of 3D stencil codes on GPU clusters,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2259016.2259037 p. 155–164.

[88] M. Steuwer, M. Haidl, S. Breuer, and S. Gorlatch, “High-level programming of stencil computations on multi-GPU systems using the SkelCL library,” *Parallel Processing Letters*, vol. 24, no. 03, p. 1441005, 2014. [Online]. Available: https://doi.org/10.1142/S0129626414410059

[89] T. Shimokawabe, T. Aoki, and N. Onodera, “High-productivity framework for large-scale GPU/CPU stencil applications,” *Procedia Computer Science*, vol. 80, pp. 1646–1657, 2016.

[90] M. Sourouri, S. B. Baden, and X. Cai, “Panda: A compiler framework for concurrent CPU-GPU execution of 3D stencil computations on GPU-accelerated supercomputers,” *International Journal of Parallel Programming*, vol. 45, no. 3, pp. 711–729, 2017.

[91] M. Sourouri, J. Langguth, F. Spiga, S. B. Baden, and X. Cai, “CPU+GPU programming of stencil computations for resource-efficient use of GPU clusters,” in *2015 IEEE 18th International Conference on Computational Science and Engineering*, 2015, pp. 17–26.

[92] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, “Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2063384.2063398

[93] T. Lutz, C. Fensch, and M. Cole, “Partans: An autotuning framework for stencil computation on multi-GPU systems,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400718

[94] M. Sourouri, T. Gillberg, S. B. Baden, and X. Cai, "Effective multi-GPU communication using multiple CUDA streams and threads," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 981–986.

[95] I. Faraji, S. H. Mirsadeghi, and A. Afsahi, "Topology-aware GPU selection on multi-GPU nodes," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 712–720.

[96] R. Ross, R. Latham, W. Gropp, E. Lusk, and R. Thakur, "Processing MPI datatypes outside MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 42–53.

[97] The MVAPICH Team, "MVAPICH2 changelog," 2020. [Online]. Available: http://mvapich.cse.ohio-state.edu/static/media/mvapich/MV2_CHANGELOG-2.3.4.txt

[98] S. L. Johnsson, "CMSSL: a scalable scientific software library," in *Proceedings of Scalable Parallel Libraries Conference*, 1993, pp. 57–66.

[99] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," https://www.mcs.anl.gov/petsc, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[100] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[101] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.14, 2020. [Online]. Available: https://www.mcs.anl.gov/petsc

[102] FleCSI Team, *FleCSI*, 2021. [Online]. Available: https://laristra.github.io/flecsi/index.html

[103] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

[104] E. Chow, A. J. Cleary, and R. D. Falgout, "Design of the hypre preconditioner library," *Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pp. 106–116, 1999.

[105] The Trilinos Project Team, *The Trilinos Project Website*, 2020 (acccessed May 22, 2020). [Online]. Available: https://trilinos.github.io

[106] "Zenodo," 2021. [Online]. Available: https://zenodo.org/

[107] C. Pearson, "c3sr/comm_scope," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4586913

[108] C. Pearson, "cwpearson/stencil," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4635277

[109] C. Pearson, "cwpearson/tempi," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4584107