Node-Aware Stencil Communication for Heterogeneous Supercomputers

***Abstract*—High-performance distributed computing systems increasingly feature nodes that have multiple CPU sockets and multiple GPUs. The communication bandwidth between these components is non-uniform. Furthermore, these systems can expose different communication capabilities between these components. For communication-heavy applications, optimally using these capabilities is challenging and essential for performance. This work presents approaches for automatic data placement and communication implementation for 3D stencil codes on multi-GPU nodes with non-homogeneous communication performance and capabilities. Benchmarking results in the Summit system show that choices in placement can result in a 20% improvement in single-node exchange, and communication specialization can yield a further 6x improvement in exchange time in a single node, and a 16% improvement at 1536 GPUs.**

***Index Terms*—stencil, CUDA, GPU, heterogeneous, MPI, communication, topology, node**

I. Introduction

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. Along with a discrete grid space, a "stencil" determines the neighboring grid points required to update any grid point in space. Stencils across applications vary in their type and order. Two types of order-1 (which means one immediate neighbor in each direction is used for calculating a grid point value) 3D stencils are shown in Fig. 1(a) and (b). Stencil in Fig. 1(a) requires one neighbour point along the axis in all directions, while in Fig. 1 (b), stencil also requires the neighbor points along the diagonals in different planes.

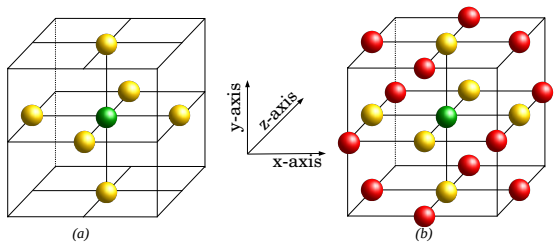


Fig. 1: An illustration of 3D stencils (a) Neighbor grid points required for updating the center (green) point are only along axis (b) Grid points required are along the axis and diagonals in each plane.

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 [1], [2], 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 [3]. Once the stencil data is initialized on the GPU, it remains there without further exchange with the host. The data-reuse between neighboring nodes 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 domain data may be much larger than a single GPU's memory. We survey various prior GPU and non-GPU stencil codes since 2-15, and see a range of 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 [3]–[6]. This motivates our selection of

Distributed stencil algorithms are usually implemented in terms of separate subdomains, which require halo exchange to update the grid points at the boundaries of the subdomains. Fig. 2 shows an example stencil 2D domain, being decomposed into four subdomains. Each subdomain has a halo of width r, wide enough to cover all neighbors at the boundary of the subdomain. Before the boundary grid points can be updated, current data from the neighbor needs to be transferred to each subdomain's halo. Halo exchange requires communication between GPUs within and/or across the compute nodes. The number of neighbors with which a GPU is required to communicate halos depends on the shape and type of the stencil, the pattern used to divide the compute domain among GPUs and nodes for parallel execution, and the type of boundary conditions. In this work, we consider periodic boundary conditions but the techniques are easily applicable to other types of boundary conditions.

For example, the 3D stencil in Fig. 1(a) would require halos to be exchanged along all the six faces of the compute subdomain, and hence six neighboing GPUs would be involved in the halo communication, one for each face. The stencil in Fig. 1 (b) has grid points not only along axis but also along the diagonals. It would require communication not only with six neighbors along the faces but also twelve neighbors along the edges of the compute subdomain.

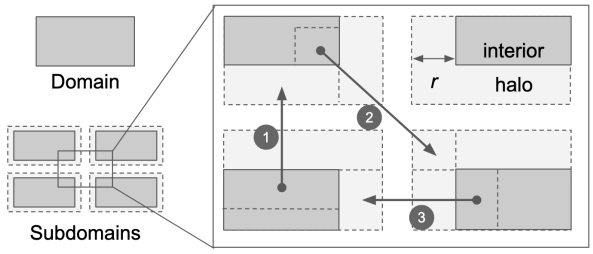


Fig. 2: An example of a 2D domain, decomposed into four equal subdomains, with three labeled halo transfers ❶ ❷ ❸ that show transfer of data from the interior of a region to the halo of a neighboring region. There are more transfers than the ones shown.

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 efficient data placement and communication strategies for large-scale stencil computations on such clusters is a challenging task. The choice of hardware features and library support are key to attain the best possible performance. For example, GPUDirect can be used to to directly exchange data between GPUs rather that staging through the CPU. Furthermore, CUDA-Aware MPI helps send messages in pipelined manner and transparently uses GPUDirect whenever is possible. Thus, significant development and tuning efforts are required for a knowledgeable developer to select the best data placement and communication strategies to achieve the best possible performance.

In this paper, we propose a set of techniques to handle these challenges for stencil applications, and implement those techniques in a CUDA/C++ library. This library automatically discovers system topology and the supported hardware and system capabilities. Based on that information, it chooses optimal data placement and communication strategy for 3D stencil halo exchange. Specifically, this work contributes

- a structured three-phase solution to optimize GPU-GPU stencil communication on heterogeneous clusters
- capability-based communication optimization based on subdomain communication requirements and GPU communication capabilities
- runtime node-aware data placement for stencil subdomains using node-level topology information
- high intra-node communication performance regardless of ranks per node

Additionally, we incorporate well-understood stencil communication techniques:

- Hierarchical inter-node and intra-node domain partitioning to minimize communication
- Support for overlapping stencil computation and communication

The library supports flexible performance across any combination of ranks and GPUs utilized in a single node, as well as across nodes, and it is generalizable for stencils of any type and radius.

This paper is organized as follows: Section II presents some background information on CUDA and MPI communication. Section III explains the methodology of our communication library. Section IV presents an experimental evaluation. Section V includes a discussion of related work. Section VII presents a discussion of future work, and concludes.

II. Background

In this section, we give a brief background on MPI and CUDA APIs for different modes of communication in a multi-node and multi-GPU environment. This background is useful for understanding how asynchronous MPI and CUDA calls are performed to overlap communication and execution.

A. CUDA

Two kinds of GPU communication are used in this work: GPU/CPU and GPU/GPU. For GPU/GPU communications, we use `cudaMemcpyPeerAsync`. Peer access refers to the ability of a GPU to directly access memory on another GPU without involving the CPU. When supported by the underlying paltform, this is the highest-performance way to send data between GPUs. Peer access must be explicitly enabled at the beginning of the program. For GPU/CPU communications, we use `cudaMemcpyAsync` in conjunction with pinned memory on the host (created with `cudaHostAlloc` or `cudaMallocHost`). This allows overlapping of multiple CPU-GPU communications as well as the highest bandwidth between the CPU and the GPU.

These asynchronous CUDA operations are coordinated through streams and events. A stream is a sequence of operation that are executed in issue-order on the GPUs. CUDA operations in different streams can run in parallel, which is useful to overlap data transfer from host while GPU is still busy in computation. CUDA events are used to synchronize operations between streams when necessary.

B. MPI

For inter-process communication, we use MPI non-blocking `MPI_Isend` and `MPI_Irecv` routines to exchange data between multiple MPI ranks. Non-blocking routines enable a rank to send and receive multiple unrelated messages simultaneously. The typical way to move data between two GPUs on different ranks is done in four steps: (a) copy data from the source GPU buffer to host CPU memory, (b) use MPI ISend to send the host data from source and destination CPUs, (c) use MPI IRecv to receive data into the destination CPU memory, and (c) finally, copy the received data from the destination CPU memory to the receiving GPU buffer.

C. CUDA and MPI

CUDA and MPI can be composed in a straightforward way: for example, using CUDA to copy data from a source GPU to a CPU, using MPI to copy that data to another node, and using CUDA to copy that data to the destination

GPU. However, CUDA-Aware MPI allows GPU buffers to be directly passed to MPI calls, instead of copying them to the CPU first. There are two potential benefits to use CUDA-Aware MPI. First, data may not need to be staged in the CPU, improving latency and throughput. Secondly, underlying acceleration technologies such as GPUDirect can be utilized seamlessly. In this work, we investigate the performance of using CUDA-Aware MPI for inter-node GPU data exchange.

Another option is to make use of the `cudaIpc*` family of functions. These functions allow events and device memory pointers to be shared between processes, something is usually disallowed by the virtual memory system. They work by turning an event or a device pointer into an opaque object that can be shared by some inter-process-communication mechanism (MPI in this work), and then converted back into a valid event or pointer in the destination address space. This means that two processes can directly copy data between device buffers, or two processes can use a shared event to synchronize streams. `CudaIPC*` is used in the COLOCATEDMEMCPY exchange strategy in Section III.

III. METHOD

Before the stencil application begins, information about the system and stencil domain is used to optimize communication. This work addresses the challenge of communication scheduling with a three-phase setup process:

- *Partitioning*: decomposing the stencil domain into subdomains while minimizing required data exchange.
- *Placement*: placing the stencil data according to the communication performance promised by the underlying platform.
- *Specialization*: choosing the communication strategy that best realizes the promised performance.

After setup is complete, halos can be exchanged on demand by re-using the decisions made during setup.

A. Setup: Partitioning

Decompose the domain into one subdomain per GPU that minimizes surface-to-volume ratio. The intuition is to produce subdomains with a small surface-to-volume ratio: this allows largest amount of stencil computation (the volume) for the minimal amount of data exchange (the surface). For example, Fig. 3 shows four potential partitions of the same 2D domain: 2×2, 4×1, 3×3, and 9×1, and summarizes the total communication volume for each subdomain (V_s) and the total (V_d). The total communication volume is minimized when the subdomains have a minimal surface-to-volume ratio for a given number of partitions (4 and 9 in this example).

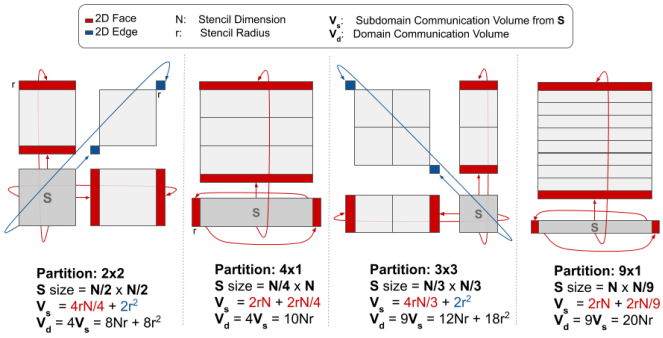


Fig. 3: Four example partitions of the same 2D domain: 2×2, 4×1, 3×3, and 9×1. The subdomain volume is shown as V_s, and the total data volume as V_d. The stencil radius is r. Communication volume is minimized when subdomain surface-to-volume ratio is minimized.

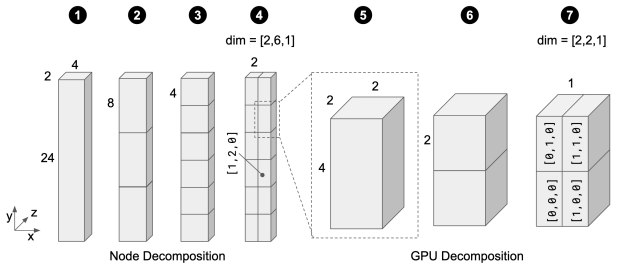


Fig. 4: Example of decomposing a 4×2×24 domain among 12 nodes with 4 GPUs. The domain is repeatedly partitioned among the longest dimension by the prime factors of the number of nodes. Then, the same process is applied for the number of GPUs per rank. The final result is a subdomain with a 3D index in the rank space and a 3D index in the GPU space. ❶ shows in the initial domain. ❷ shows the partition of y by 3, the first prime factor of 12. ❸ shows the partition of y by 2, the second prime factor of 12. ❹ shows the partition of x by 2, the final prime factor of 12. ❺, ❻, and ❼ repeat the process for 4 GPUs.

Since off-node exchange is more expensive than on-node (Section IV), we use a two-level strategy, where the partition is first done to minimize communication between nodes, and then again within nodes to minimize communication between GPUs. This may not ultimately minimize inter-GPU communication, but it does minimize the slower inter-node communication. To provide each node (or GPU) with a (roughly) equal subset of the domain, the domain must be split by integer divisors. The domain is repeatedly divided along its largest dimension by consecutive sorted prime factors of the number of nodes (or GPUs), which are all integers by definition. This provides the largest number of opportunities to divide, ensuring the best opportunity to make the resulting regions as cubical as possible.

Fig. 4 shows an example decomposition of a 4×24×2 domain among 12 nodes of 4 GPUs. The large aspect ratio is chosen to highlight the qualities of the decomposition approach. The prime factors of 12 are 3, 2, and 2. The first splits (❷) is along the the long y dimension by 3, then again along the long y dimension by 2 (❸), and finally along the x dimension by 2 (❹), yielding a final index space of `[2,6,1]`. Each resulting subdomain is assigned an index in the resulting 3D space. Index `[1,2,0]` annotated, which is in x position `1`, y position `2`, and z position `0`.

Each of those resulting subdomains is further decomposed with the same approach according to the number of GPUs on each node. ❺ highlights a single node-level subdomain, but the same decomposition process is applied to all the other node-level subdonains. In this example, ❻ splits the long y

dimension by two, then the x dimension by two (❼) to generate the subdomains for the four GPUs, which results in one approximately-cubical subdomain per GPU. Each subdomain has a system-level 3D index in the node space (which node-level partition its in), and a 3D index in the GPU space (which GPU-level partition it represents); the combination of node and GPU index is unique. Subdomains exchange halos with all neighbors in this combined index space.

B. Setup: Data Placement

The second phase is to assign each subdomain to a GPU to maximize use of the available communication bandwidth. The shape and adjacency of subdomains controls the amount of data exchanged between them, so not all subdomains on a node exchange the same amount of data. Fig. 5 shows an example four subdomains, each of size $M \times N \times P$. Subdomain `[0,0,0]` transfer the $M \times N$-sized face with `[0,1,0]`, but an $M \times P$-sized face with `[1,0,0]`.

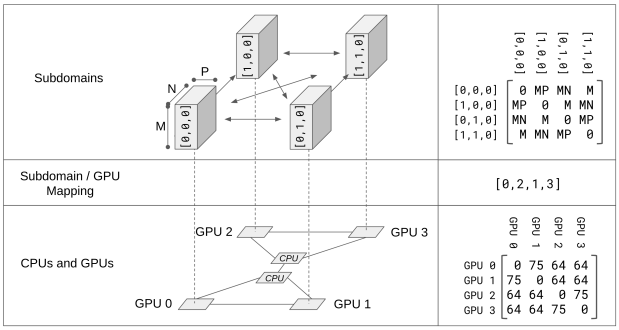


Fig. 5: Example of placing subdomains onto GPUs in a node. The communication volume between subdomains is determined by their index and shape. Those pairwise volumes are represented in a flow matrix w. Likewise, each GPU has a theoretical communication bandwidth to each other GPU. The reciprocal bandwidth is used as a distance matrix d. w and d are used in a quadratic assignment problem to place subdomains optimally on GPUs.

Subdomain placement is modeled as a quadratic assignment problem (QAP) The quadratic assignment problem is concerned with assigning a set of n facilities to n 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 subdomains with high exchange volume on GPUs that have high communication bandwidth. The assignment is a bijection f between facilities and locations. With real-valued square matrix w representing the flow between facility i, j d the distance between location i, j, and QAP minimizes the cost function

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

the sum of the flow-distance products under f.

We model the flow matrix as the required exchange volume between subdomains, and the distance matrix as the element-wise reciprocal of a matrix which captures the bandwidth of GPUs i and j in $d_{i,j}$. Fig. 5 The CUDA driver provides the Nvidia Management Library `libnvidia-ml`, which can be used to infer the connection and bandwidth between GPUs on 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.

C. Setup: Capability Specialization

In Section III-B, subdomains were assigned to GPUs to best match the theoretical bandwidth. The achievable GPU-GPU transfer bandwidth depends on the communication method [7], not just the node topology. In this section, we describe the final phase, where GPU-GPU exchange is implemented in terms of these communication methods.

In general, the exchange operation consists of taking the (possibly) non-contiguous halo region from the interior of the source subdomain, packing it into a contiguous buffer, sending that buffer to the destination GPU, and unpacking that buffer into the appropriate exterior of the destination subdomain. Fig. 6 shows an example of a pack operation on a 3D region. In this example, we consider an XYZ storage order, yeilding a non-continguous storage for the 3D region shown. The result of the pack operation is to copy that data into a contiguous buffer.

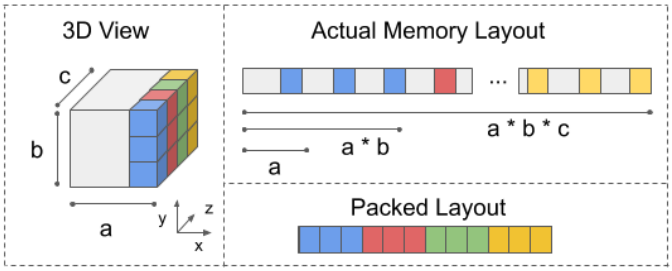
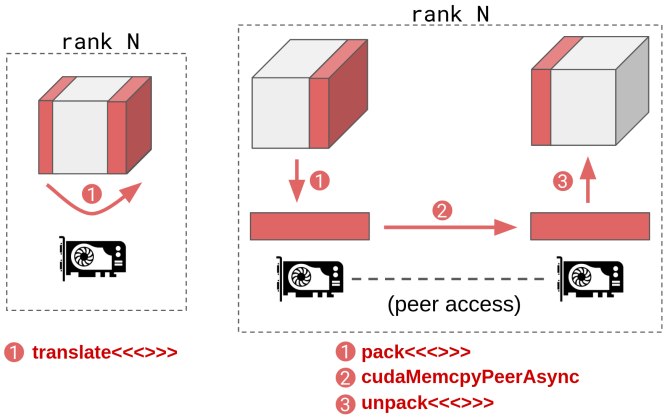


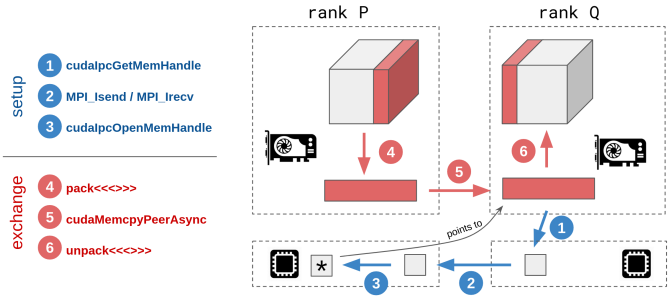
Fig. 6: Example of packing for a 3D region. In general, the linear storage order of the subdomain in memory causes the elemeents of the 3D region to be strided. The pack operation places only those elements in a dense buffer with some predetermined order.

In order to support high-performance exchanges in a variety of scenarios, we consider five GPU-GPU transfer methods. All methods are asynchronous, allowing them to be freely overlapped, even within a single process.

KERNEL (Fig. 7a): This method applies when a subdomain has a self-exchange. This occurs when the entire decomposition dimension has extent 1 in any direction, and there are periodic boundary conditions: the decomposition is only 1 subdomain wide, so the subdomain is on both the positive and negative boundary of that dimension. This method uses a CUDA kernel launched on the GPU to do an exchange within GPU memory. One kernel is launched per direction vector that needs to be exchanged. Since there is no packing or unpacking, this method is the lowest-overhead exchange. Each GPU uses its own stream to allow operations to overlap with other types of exchanges.

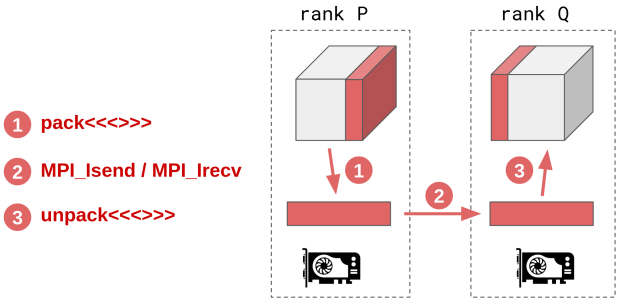


(a) The KERNEL (left) and PEERMEMCPY (right) exchange method on a subdomain. For KERNEL: ❶ represents a kernel transferring a halo from the interior to the exterior. For PEERMEMCPY: ❶ pack the non-contiguous 3D region into a buffer on the source GPU. ❷ `cudaMemcpyPeerAsync` to copy packed data to a buffer on the destination GPU. ❸ unpack the buffer into the halo of the destination subdomain.

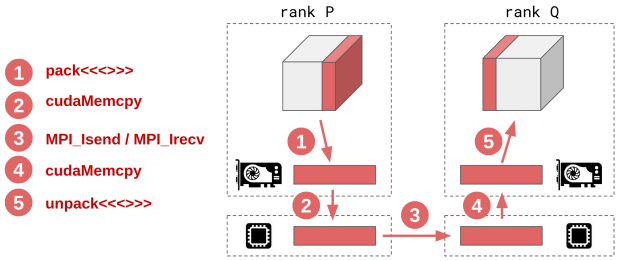


(b) During application initalization, the source and destination subdomains use the `cudaIpc*` interface to bypass MPI during following exchanges. ❶, ❷, ❸ show a `cudaIpcMemHandle_t` being sent through MPI from the destination to the source subdomain, to provide the source subdomain with a pointer that can be the target of a cudaMemcpy during future exchanges, without invoking MPI to send data between ranks. Then, in each exchange, the halo region is packed into a buffer on the source GPU (❹). The source domain uses `cudaMemcpyPeerAsync` ❺ to send data directly between the source and destination GPU, without MPI. ❻ marks unpacking the buffer into the halo region on the destination device.

Fig. 7: Summary of Kernel, PeerMemcpy, and ColocatedMemcpy exchange methods.



(a) First, the halo region is packed into a buffer on the source device (❶). Then, `MPI_Isend` is used with CUDA-aware MPI to transfer the buffer to the destination GPU (❷). Finally, the buffer is unpacked into the 3D region on the destination (❸).



(b) The halo region is packed into a buffer on the source device (❶). That buffer is copied to pinned memory on the host (❷). MPI is used to transfer the buffer between ranks (❸). The buffer is transferred (❹) to the destination device and unpacked (❺).

Fig. 8: Summary of CUDAAwareMPI and Staged exchange methods.

PEERMEMCPY (Fig. 7a): This method applies when two subdomains are in the same process, and the corresponding GPUs have peer access (Section II). A CUDA kernel on the source GPU (❶) packs the non-contiguous 3D region into a buffer on the device. ❷ uses `cudaMemcpyPeerAsync` to copy packed data to a buffer on the destination device. ❸ uses a kernel on the destination GPU to unpack the buffer into the exterior of the destination subdomain. Each GPU pair uses its own stream.

COLOCATEDMEMCPY (Fig. 7b): When multiple MPI ranks are co-located on a node, the virtual memory barrier between processes prevents straightforward data transfer between ranks. In general, when passing data between subdomains on different ranks, we would default to either the CUDAAwareMPI or Staged methods. However, when two ranks are on the same node, MPI can be bypassed entirely during each exchange through the `cudaIpc*` set of runtime APIs. Once, during the setup phase, the destination uses `cudaIpcGetMemHandle` to create an opaque handle (❶), which it passes through MPI to the source domain (❷). The source domain can use `cudaIpcOpemMemHandle` to convert this handle into a device pointer that is valid in its address space (❸). Then, during the exchange, the usual process of packing (❹), cudaMemcpyAsync (❺), and unpacking (❻) is used to send data between ranks, without using MPI at all.

CUDAAWAREMPI (Fig. 8a): CUDA-Aware MPI allows CUDA device pointers to be passed to the `MPI_Send/Recv` family of functions (instead of only pointers to system memory). Therefore, as long as the MPI system is "CUDA-Aware", this regime can be used to pass data between any two GPUs. Data is packed with a kernel (❶) into a flat buffer, transferred (❷) to the destination device with `MPI_Send/Recv`, where it is unpacked (❸).

STAGED (Fig. 8b): Any system with both CUDA and MPI will support this method. First, the region is packed (❶) into a flat buffer on the source GPU. Then, that buffer is copied (❷) to a pinned buffer on the host with `cudaMemcpyAsync`. `MPI_Send/Recv` is used to transfer the buffer to the destination (❸), where it is copied back ot the device (❹), and finally, unpacked into the subdomain with a kernel (❺).

Every subdomain in a rank queries the placement information (Section III-B) to determine where all of its neighboring subdomains are, in terms of node index, GPU index, as well as owning rank and CUDA device ID. With that information, peer

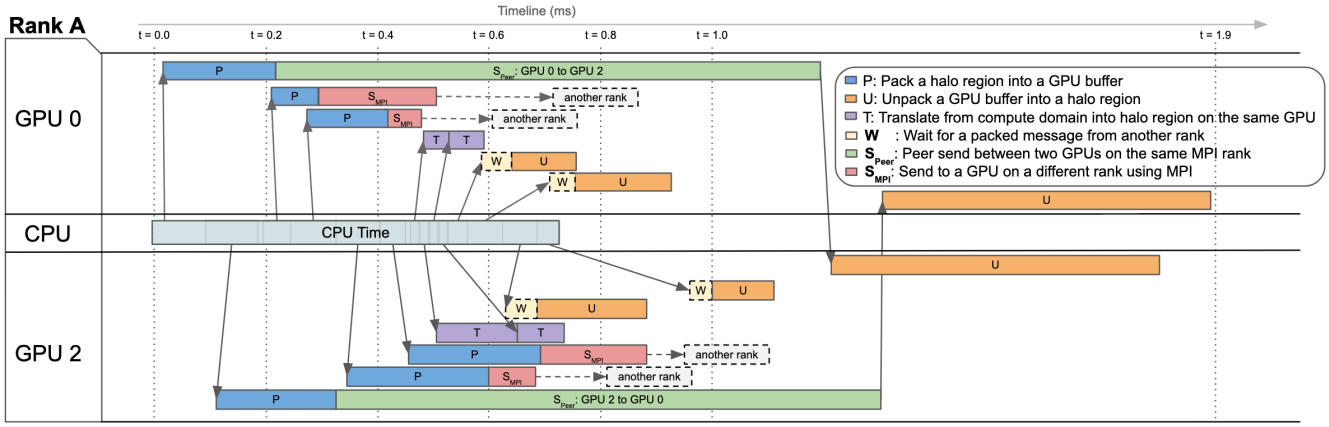


Fig. 9: An example timeline of overlapped exchange operations for a 512^3 subdomain with four SP quantities on a single rank controlling GPU 0 and GPU 2. Some data exchanges are contained within the rank, and some go to other ranks.

access between needed devices is enabled, if available. Then, for each subdomain exchange, the first applicable method from this section is selected. On our test platform, CUDA-Aware MPI is always slower than staged, so it is never selected.

D. Async and Overlap

To achieve good communication performance, it is crucial to overlap as many unrelated communication operations as possible. As described in Section II, both CUDA and MPI provide asynchronous transfer operations: the `cudaMemcpy*Async` and `MPI_Isend/Irecv` families respectively. When a subdomain exchange is only a sequence of CUDA operations (as in the kernel, PeerMemcpy, and ColocatedMemcpy), the asynchronous operations can be added to a stream, and all executed whenever resources are available. When the subdomain exchange involves both CUDA and MPI operations, we prevent serialization by implementing Sender and Reciever objects as state machines.

After starting the pure-CUDA asynchronous exchanges, we loop over all the state machines and check if each sequential phase of their operation is done, and they are ready to proceed to the next step. If so, they are moved to the next state. For example, the staged Sender would have two states, one where it is packing into the device buffer and copying to the host (both operations in a CUDA stream), and then a following state where it is using `MPI_Isend` to transfer the buffer to the receiver. Once all stateful senders and receivers are complete, we block on the truly asynchronous operations, and then the exchange is considered complete.

Fig. 9 shows how operations are effectively overlapped. It was recorded during a one-node exchange of 512^3 subdomain per GPU with four single-precision (SP) quantities between two MPI ranks, each of which controlled two GPUs.

IV. Results

A. Evaluation Platforms

Evaluation was carried out on the Summit [8] system at Oak Ridge National Laboratory and Fig. 10 and Table I summarizes the node.

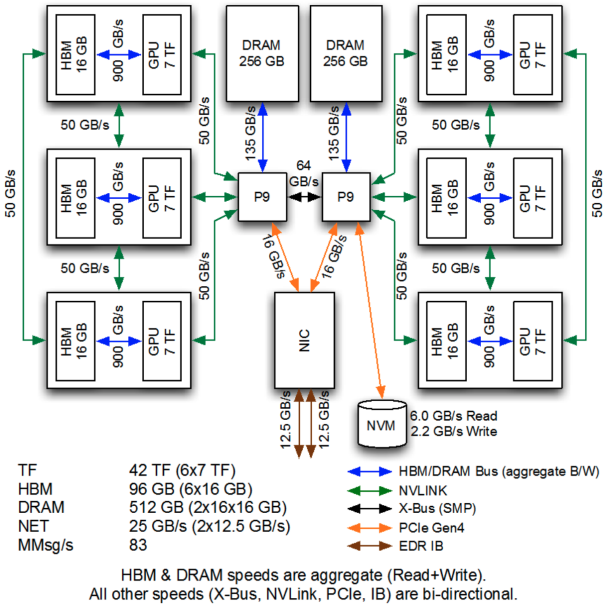


Fig. 10: Summit node architecture and bandwidths between CPUs, GPUs, and network interface card [8]. GPUs in the same triad have more bandwidth between them, affecting optimal data placement.

B. Data Placement Performance

Thanks to the domain partitioning, each GPU's subdomain shape usually has an aspect ratio close to one. When subdomains have small aspect ratios, all exchanges between domains are similar, and data placement has no performance effect. However, for a small number of subdomains or very high-aspect-ratio domains, the resulting subdomains can also have a high aspect ratio.

Fig. 11 shows an example of a compute domain of $1440\times1452\times700$, which produces 6 subdomains of $720\times484\times700$. On a six subdomain node, the largest possible ratio of dimensions for subdomains under most conditions is 3/2. This example is chosen to closely match this worst-case scenario. Under the node-aware placement, high-volume halo exchanges between subdomains occur on high-bandwidth

CPU	OS	Kernel	GPUs	CUDA Driver	MPI	nvcc	cc
22-core POWER9	RHEL 7.6	4.14.0-115.8.1.el7a.ppc64le	V100-SXM2-16GB	418.67	Spectrum 10.3.0.1	10.1.168	g++ 4.8.5

TABLE I: Summit hardware summary

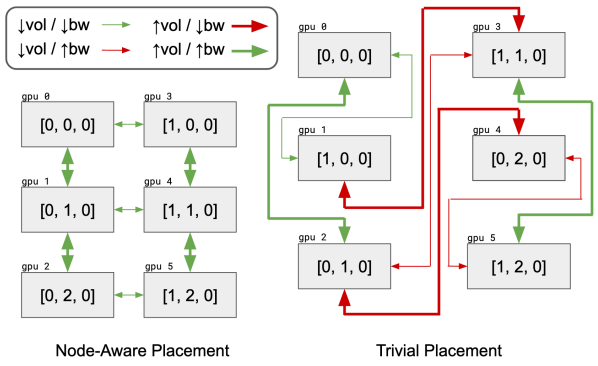


Fig. 11: Example of data placement exchanges for 6 subdomains of 720×484×700. For brevity, not all exchanges are shown, and z-axis is omitted. Required exchanges and corresponding sizes are determined by subdomain position. On the left, subdomain exchange volume is well-matched to GPU interconnect bandwidth. On the right, the same subdomains are poorly placed, resulting in high-volume exchanges on low-bandwidth links. Consult Fig. 10 for GPU bandwidth.

links, and low-volume occurs on low-bandwidth links (see Fig. 10 for bandwidth between GPUs). In a trivial placement, where the subdomain id is linearized and assigned to GPUs, some of the resulting high-volume exchanges occur on the low-bandwidth SMP link across sockets. In this scenario, node-aware data placement results in a 20% speedup thanks to better utilization of hardware links.

C. Communication Specialization Performance

Fig. 12a shows the performance effect of communication specialization on a single node, with a fixed amount of data per GPU, and four SP quantities. Experimental configurations are described with a string like "Xn/Xr/Xg/NNNN/ca", where Xn refers to X nodes, Xr refers to X ranks per node, Xg refers to X GPUs per node, NNNN refers to the extent of a single dimension of the domain, and ca refers to CUDA-aware, if used. "+remote" means only the STAGED or CUDAAWAREMPI exchange method is enabled. "+colo" means remote and the COLOCATEDMEMCPY exchange are enabled. "+peer" means the previous, plus PEERMEMCPY exchange is enabled "+kernel" means the previous, plus KERNEL exchange is enabled. Exchange times are the average of 30 exchanges measured by `MPI_Wtime`. For multi-node cases, the node with the longest exchange is used for the presented value.

Within each group of columns, additional communication capabilities are enabled. When only the STAGED capability is enabled, performance is at its worst, as all halo exchanges are implemented through `MPI_Isend`. As we move from one rank controlling all the GPUs to one rank controlling each GPU, the performance improves as more processes are recruited to participate in simultaneous memcopies underlying the on-node `MPI_Isend`.

When the COLOCATEDMEMCPY exchange is enabled, performance improves when more than one rank is on the node, as exchanges between GPUs owned by those ranks no longer invoke MPI. Even when compared to CUDA-Aware MPI, the colocated exchange is faster. This is because it only does the `cudaIpc*` exchange of buffers once during the setup phase (Fig. 7b), while the CUDA-aware MPI has to do it every time.

When peer exchange is enabled, all exchanges within ranks also no longer require MPI, and performance further improves. Finally, enabling the kernel exchange seems to have no effect on performance. This replaces self-exchanges through peer copy with a single kernel call, but the on-node time is still dominated by exchanges between GPUs. Ultimately, for a single node, communication specialization has a large impact on performance, yielding a 6x speedup over STAGED and a 2x speedup over CUDAAWAREMPI.

Fig. 12b and Fig. 12c show the performance effect of communication specialization on multiple nodes. Once communication off-node occurs, the on-node only provides small improvements, probably from replacing MPI calls with CUDA calls, and allowing the MPI system to only communicate between nodes. At 256 nodes, communication specialization provides a 1.16× speedup.

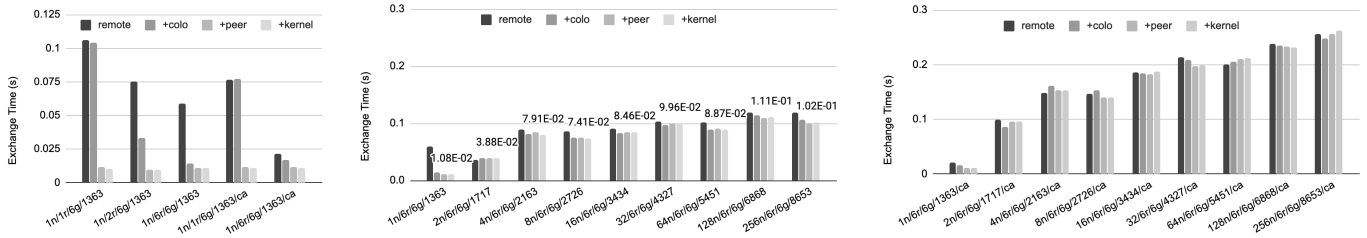
D. Weak Scaling

Fig. 12b and Fig. 12c show exchange time scaled out to multiple nodes, each with 6 ranks and 6 GPUs. Since the automatic partition and placement attempts to provide good performance for all domain shapes, we fix the domain shape to a cube for these experiments. The total grid volume closely matches 750^3 points per GPU, while maintaining an overall cube shape: it is computed as $\mathrm{round}(750 \times \mathrm{nGPUs}^{\frac{1}{3}})^3$, where nGPUs is the number of participating GPUs. Without CUDA-aware MPI, the exchange time flattens out after 32 nodes, when most nodes communicate with 26 distinct neighbors. As the off-node communication dominates the performance, enabling various on-node optimizations only provide a small benefit by removing some on-node MPI interactions. With CUDA-aware MPI, the performance degrades dramatically as more nodes are included, and intra-node optimizations cease to have the expected effect. This suggests that CUDA-aware MPI somehow prevents overlapping of on-node communications, and does not effectively parallelize independent transfers. Further investigation of these effects is taken as future work.

E. Strong Scaling

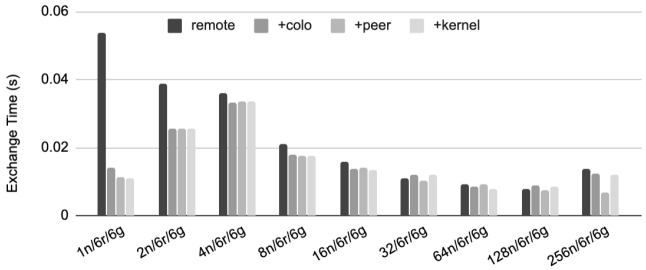
Fig. 13 shows exchange performance as a 1363^3 domain is distributed across 256 nodes, each with 6 MPI ranks and 6 GPUs. The configurations are annotated the same way as the weak scaling (Section IV-D).

As the stencil is distributed from 1-128 nodes, we see a drop in total exchange time. This is because each node only



(a) **(b)** **(c)**

Fig. 12: Exchange time vs. total domain size, scaled with the number of GPUs. Each configuration is labeled with a string of the form "Xn/Xr/Xg/NNNN/ca". *Xn* refers to *X* nodes. *Xr* refers to *X* ranks per node. *Xg* refers to *X* gpus per node. *NNNN* refers to the extent of each dimension of the domain. *ca* refers to CUDA-Aware, if enabled. (a) Exchange time on a single node with 1, 2, or 6 ranks. At 6 ranks, specialization provides a 6x speedup over STAGED only, and a 2× speedup over CUDAAWAREMPI. (b) Exchange times without CUDA-aware MPI, scaled to 256 nodes and 1536 GPUs. The "+kernel" time is annotated above each group. At 256 nodes, specialization provides a 1.16× speedup over STAGED. Enabling CUDA-aware MPI degrades performance severely and prevents specialization optimizations from improving it. (c) Exchange times with CUDA-aware MPI, scaled to 256 nodes and 1536 GPUs.



(a) Halo exchange time for a domain of 1363^3.

Fig. 13: Exchange time vs number of GPUs, for a fixed total domain size. Each configuration is labeled with a string of the form "Xn/Xr/Xg". *Xn* refers to *X* nodes. *Xr* refers to *X* ranks per node. *Xg* refers to *X* gpus per node.

exchanges locally with its neighbors, and the communication volume decreases as more nodes are included. For a small number of nodes, the impact of the on-node capability optimizations are more substantial, as the exchange time is not yet dominated totally by MPI. Once we reach 32 nodes, capability specialization stops improving performance. The amount of data transferred becomes small enough that the additional bandwidth offered by optimizations does not impact the overall exchange time. At 256 nodes, we cease to see strong scaling as the subdomains become very small. We do not evaluate a similar scenario with CUDA-aware MPI, as it did not provide performance improvement in weak scaling.

F. CUDA-Aware MPI

When CUDA-Aware MPI is enabled, the STAGED exchange is replaced with the CUDAAWAREMPI exchange. Fig. 12a shows that on a single node, using CUDA-aware MPI provides some benefit, even in the case of a single rank. For a single rank, the STAGED method copies the data from GPU to CPU, then `MPI_Isend`s the CPU data to the same rank, and then copies it back to the GPU. CUDA-Aware MPI omits the GPU-to-CPU transfer, but is still only about 25% faster. In practice, it is probably rare to invoke MPI between a rank and itself, so this is probably a case that CUDA-aware MPI is not optimized for. In the case of six ranks per node, it is three times faster. This is probably because the `MPI_Isend` can be implemented as a cudaMemcpy. Using our intra-node optimizations on top of CUDA-Aware MPI still provides an additional 2x speedup, since our COLOCATED method only does the inter-process communication during the stencil setup.

V. RELATED

To our knowledge, none of the prior work addresses all the related challenges for an efficient communication across GPUs in large clusters. It involves automatic discovery of the cluster topology and hardware capabilities on a node, to partition the domain, placing the partitions optimally across GPUs and the use of right combination of CUDA and MPI libraries to efficiently exploit the hardware. In this work, we address all those challenges and further abstracts away the implementation details for the ease of programmability. We focus our discussion of related work on multi-GPU stencil codes, node-aware GPU communication techniques, work that provides some degree of automatic stencil code generation for distributed memory (since it must therefore *also* handle communication automatically).

A. Stencil Communication

[9] suggest a 1D grid decomposition to avoid communication in dimensions where data in halo regions is not stored in contiguous locations. However, this limits scaling for large domains as overall communication volume is not minimized. Moreover, stencils with only axis-aligned grid points are mentioned which further limits the application range.

[10] presents an efficient overlap between communication and execution as a fundamental key to efficient execution. However, this neglects how to achieve efficient communication in the first place. They use the equivalent of our STAGED communication method.

[3] presents a two-node multi-GPU 1D decomposition of a stencil grid. They use the PEERMEMCPY exchange method for on-node exchanges, and a collective multi-GPU pack for

off-node. Their implementation is hard-coded for the target system and a single MPI rank per node.

[11] uses a similar prime-factor based 3D partitioning strategy in their stencil code.

B. Node-Aware GPU Communication

[12]–[14] is a group of related works focusing on node-aware topology for GPU-GPU communication. Like this work, they compare a GPU communication matrix with a topology matrix, for deciding which MPI ranks should be placed on which GPUs. Our work differs in five main ways. First, they restrict their consideration to one GPU per MPI process, so do not consider optimizing the communication case within a process or a GPU. Second, they do not consider the effect of their work on multi-node executions. Third, we characterize the placement as a quadratic assignment problem instead of a graph embedding, though they are interchangable in many cases. Fourth, we recognize that GPU-GPU communication defined at the algorithm level, not the MPI level, and so our placement algorithm operates on stencil subdomains instead of MPI ranks. Fifth, due to our domain-specific approach, we are able to do subdomain placement without requiring an initial profiling pass. [12] uses a 2D stencil microbenchmark to evaluate the effect of node-aware placement on message latency and bandwidth, and observe no effect. Our work shows an effect for real stencil exchanges in practice.

Several works [15], [16] provide multi-GPU collective communication patterns. Thanks to the semantics of the collectives, these works are able to implement patterns that take advantage of typical node topologies without adaptions to specific platform characteristics.

C. DSLs, Frameworks, and Code Generation

[17] is a DSL framework which decomposes the grid in 3D and supports stencils with grid points also along diagonals. To hide the communications latency the framework supports overlap with the execution. This framework also implicitly copies halos with non-unit stride through `cudaAllocHost` to avoid multiple calls to `cudaMemcpy`. This is the same as the packing we describe, except without a following explicit transfer step to the host. [18] presents another DSL framework which supports 3D decomposition of the compute domain. They implement the equivalent of our STAGED exchange method. In [19] once the domain is partitioned, sub-domains are mapped to GPUs using blocking and circular configuration. Blocking configurations first assign partitions to dual GPUs on the same graphic card, while circular configuration assigns partitions in an alternating fashion between the graphics cards. This strategy touches on node-ware configurations, but does not consider multi-GPU nodes or many node topologies. [20] presents a framework for executing stencil codes on GPU clusters with subdomains that are larger than GPU memory. Their temporal blocking mechanism causes inter-node communication to be implemented like our STAGED method. [21] present a stencil application in high-level OpenCL programming framework. It only considers a single node, and data is exchagned between GPUs by first being staged through the CPU. [22], [23] introduce a software framework for large-scale stencils on GPUs with an emphasis on overlapping communication with execution. [24] is a compiler which generates stencil code for for CPU-GPU clusters. It seems to leverage the same communication techniques used in [10].

VI. Future Work

Currently, the communication methods require some amount of GPU kernel execution to pack and unpack data. Especially on single-node exchanges, these operations can keep the GPU occupied for a substantial part of the exchange process 9. This would hinder the ability to overlap exchange with stencil execution, and this performance effect should be considered in future work.

There are also approaches for avoiding packing and unpacking. Instead of packing, we could use the CUDA `cudaMemcpy3D*` and `cudaMalloc3D*` routines to transfer and allocate 3D regions. Another option would be to store halos separately from the compute domain, and provide a "smart pointer" to GPU kernels, that redirects each dereference to the right memory allocation. This would create a performance penalty in the kernel, but improve exchange time. Alternatively, devices with peer access can implicitly access data remote inside GPU kernels. We could use a single GPU kernel to transfer data between such devices, or avoid copying data between same-rank GPUs in the first place.

We note signs that performance could be improved. In the strong scaling (Section IV-E), exchange time stabilizes around 10ms at large scale. Figure 9 shows that the CPU time initiating transfers can be substantial, especially if there are more GPUs. Consolidating operations could improve performance. Second, the current implementation of the data placement algorithm is naive. [12] suggests that a similar algorithm should have a negligible impact on execution time when properly implemented. The data-placement strategy could also be extended to the entire system, to ensure nodes with neighboring data are placed physically close to one another. A higher-performance solution to the QAP would be required, as complete enumeration of the bijection function would be prohibitively expensive for more than a single node.

[12] uses an empirical measurement of latency, bandwidth, and distance between GPUs to inform the MPI rank placement. We could investigate if empirical measurements provide better results. Furthermore, we could extend this to include the achieved bandwidth between GPUs for all specializations on-node.

[3] packs all GPU halos on a single node into a single buffer before exchanging with other nodes, to reduce the number of messages and increase the message size. Fewer, larger MPI messages tend to achieve better performance, but our messages may already be few enough and large enough. [17] implements GPU-to-CPU packing through zero-copy memory. This may be faster than our implementation in some circumstances.

At an application level, [21] allows the user to trade off halo exchange size with iterations between exchanges. Fewer,

larger exchanges cause fewer synchronization points, but also grow super-linearly in required data size.

VII. Conclusion

We present and evaluate a set of techniques for optimizing 3D stencil communication on heterogeneous supercomputers. First, a hierarchical partitioning technique is used to minimize required data exchange between nodes, and then between GPUs. Second, node topology is used to inform data placement regardless of MPI ranks per node and GPUs per rank, with up to a 20% reduction in exchange time. Third, node GPU transfer capabilities are used to optimize data exchange between GPUs on a node, yeilding a further 6x speedup of single-node exchange over a naive transfer staged through the CPU, or a 1.16x improvement in a 1535 GPU exchange on 256 nodes. Prior works have tended to focus on single-GPU nodes, or neglected the intra-node communication optimization. Important open questions derived from this work include CUDA-aware MPI performance, and system-level data placement among nodes.

Acknowledgments

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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*, 2019, pp. 209–218.

[8] Summit documentation. [Online]. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[9] 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.

[10] 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*. IEEE, 2015, pp. 17–26.

[11] HPCG benchmark. [Online]. Available: https://www.hpcg-benchmark.org/

[12] I. Faraji, S. H. Mirsadeghi, and A. Afsahi, "Exploiting heterogeneity of communication channels for efficient gpu selection on multi-GPU nodes," *Parallel Computing*, vol. 68, pp. 3–16, 2017.

[13] S. H. Mirsadeghi, "Improving communication performance through topology and congestion awareness in HPC systems," Ph.D. dissertation, PhD thesis, Queen's University, Ontario, 2017.

[14] I. Faraji, "Improving communication performance in GPU-accelerated HPC clusters," Ph.D. dissertation, 2018.

[15] Nvidia collective communications library. [Online]. Available: https://github.com/NVIDIA/nccl

[16] gloo. [Online]. Available: hhttps://github.com/facebookincubator/gloo

[17] 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*, 2011, pp. 1–12.

[18] 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*, 2012, pp. 155–164.

[19] T. Lutz, C. Fensch, and M. Cole, "Partans: An autotuning framework for stencil computation on multi-GPU systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–24, 2013.

[20] T. Endo and G. Jin, "Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2014, pp. 132–139.

[21] 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.

[22] 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.

[23] T. Shimokawabe, T. Endo, N. Onodera, and T. Aoki, "A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 525–529.

[24] 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.