UNICONN: A Uniform High-Level Communication Library for Portable Multi-GPU Programming

Doğan Sağbili*,Sinan Ekmekçibaşı*, Khaled Z. Ibrahim†, Tan Nguyen†, Didem Unat*
*Department of Computer Science and Engineering
Koç University, Istanbul, Turkiye
{dsagbili17, sekmekcibasi23, dunat}@ku.edu.tr
†*Applied Mathematics and Computational Research Division*
Lawrence Berkeley National Laboratory, CA, USA
{kzibrahim, tannguyen}@lbl.gov

***Abstract*—Modern HPC and AI systems increasingly rely on multi-GPU clusters, where communication libraries such as MPI, NCCL/RCCL, and NVSHMEM enable data movement across GPUs. While these libraries are widely used in frameworks and solver packages, their distinct APIs, synchronization models, and integration mechanisms introduce programming complexity and limit portability. Performance also varies across workloads and system architectures, making it difficult to achieve consistent efficiency. These issues present a significant obstacle to writing portable, high-performance code for large-scale GPU systems.**

We present UNICONN, a unified, portable high-level C++ communication library that supports both point-to-point and collective operations across GPU clusters. UNICONN enables seamless switching between backends and APIs (host or device) with minimal or no changes to application code. We describe its design and core constructs, and evaluate its performance using network benchmarks, a Jacobi solver, and a Conjugate Gradient solver. Across three supercomputers, we compare UNICONN's overhead against CUDA/ROCm-aware MPI, NCCL/RCCL, and NVSHMEM on up to 64 GPUs. In most cases, UNICONN incurs negligible overhead, typically under 1% for the Jacobi solver and under 2% for the Conjugate Gradient solver.

***Index Terms*—GPU, Multi-GPUs, MPI, NCCL/RCCL, NVSHMEM, communication libraries**

I. INTRODUCTION

In modern HPC and AI supercomputers, GPUs have become the dominant accelerators, and applications increasingly rely on multi-GPU clusters to meet growing memory and performance demands. To enable data movement across GPUs and nodes, a variety of communication libraries are available [1], ranging from GPU-aware MPI and vendor-specific collectives like NCCL and RCCL, to one-sided communication runtimes such as NVSHMEM. These libraries, each with distinct programming models and runtime behaviors, are widely used in application frameworks (e.g., PyTorch [2], AMReX [3], PETSc [4]), linear algebra libraries and solvers (e.g., ScaLAPACK, MAGMA [5], SuperLU [6]), and programming models (e.g., Kokkos [7], Galois [8], Charm++ [9]).

Despite the availability of various GPU communication libraries, users still face challenges in programming complexity and portability across clusters. A key issue is the inconsistency in API semantics and GPU runtime integration. For example, while MPI and NCCL/RCCL use a synchronous, two-sided communication model, NVSHMEM employs a one-sided, asynchronous model, leading to differences in buffer address handling and synchronization. Additionally, MPI lacks native GPU stream support, requiring manual synchronization, while NVSHMEM introduces further complexity with its device-side APIs and specialized kernel launch functions. These disparities extend to initialization methods, requiring developers to be more than knowledgeable about each library. Moreover, the performance of the communication library depends on factors like message size, node architecture, and system setup. Thus, no single library universally delivers optimal performance across systems and workloads. As a result, adapting code to different libraries often requires significant changes, emphasizing the need for unified and portable solutions.

To alleviate these concerns, efforts to unify multiple communication libraries have emerged in specific AI and HPC contexts. Aluminum targets deep learning workloads by combining MPI and NCCL under a two-sided model, enabling communication-computation overlap and custom collectives [10]. MCR-DL dynamically selects the best-performing backend for each message size during training on NVIDIA GPUs [11]. Other efforts, such as PETSc [12], mix MPI and NVSHMEM to leverage two-sided and one-sided communication in parallel numerical solvers. However, these solutions are either domain-specific, tied to particular frameworks, or lack a general-purpose API that unifies host and device communication across vendors with minimal code changes.

This paper presents UNICONN, a high-level, portable, and unified communication library that supports point-to-point (P2P) and collective operations on both host and device APIs. Designed to simplify multi-GPU programming, UNICONN enables developers to switch between multiple communication backends at compile time with minimal or no changes to application code. It supports NVIDIA and AMD GPUs and offers a modular design that facilitates integrating new communication backends or vendor support.

UNICONN provides four key abstractions: `Environment`, `Communicator`, `Memory`, and `Coordinator`, which together simplify backend initialization, resource management, and kernel coordination. The `Coordinator` enables flex-

```
1  //Host-side API
2  jacobi_kernel<<<..., stream>>> (Anew, A, iy_start, iy_end, nx, A_buf);
3  cudaStreamSynchronize(stream);
4  MPI_Isend(Anew + offset1, nx, MPI_FLOAT, top, 0, MPI_COMM_WORLD, req);
5  MPI_Isend(Anew + offset2, nx, MPI_FLOAT, bottom, 0,
       MPI_COMM_WORLD,req+1);
6  MPI_Irecv(Anew_buf + nx, nx, MPI_FLOAT, bottom, 0, MPI_COMM_WORLD,
       req+2);
7  MPI_Irecv(Anew_buf, nx, MPI_FLOAT, top, 0, MPI_COMM_WORLD, req+3);
8  MPI_Waitall(4, req, stat);
```

Listing (1) GPU-aware MPI

```
1  //Host-side API
2  jacobi_kernel<<<..., stream>>> (Anew, A, iy_start, iy_end, nx, A_buf);
3  ncclGroupStart();
4  ncclSend(Anew + offset1, nx, ncclFloat, top, world_comm, stream);
5  ncclSend(Anew + offset2, nx, ncclFloat, bottom, world_comm, stream);
6  ncclRecv(Anew_buf + nx, nx, ncclFloat, bottom, world_comm, stream);
7  ncclRecv(Anew_buf, nx, ncclFloat, top, world_comm, stream);
8  ncclGroupEnd();
```

Listing (2) GPUCCL

```
1  //Device-side API
2  __global__ void jacobi_kernel(...) {
3     if (blockIdx.x == gridDim.x - 1) { //Top compute
4        nvshmemx_float_put_signal_nbi_block (Anew_buf+ nx,
               A_buf + nx, nx, sync_arr+1, iter+1,
               NVSHMEM_SIGNAL_SET, top);
5        if (threadIdx.x == 0)
6           nvshmem_signal_wait_until(sync_arr,
                  NVSHMEM_CMP_EQ, iter+1);
7     } else if (blockIdx.x == gridDim.x - 2) {//Bottom compute
8        nvshmemx_float_put_signal_nbi_block (Anew_buf, A_buf,
               nx, sync_arr, iter+1, NVSHMEM_SIGNAL_SET,
               bottom);
9        if (threadIdx.x == 0)
10          nvshmem_signal_wait_until(sync_arr+1,
                  NVSHMEM_CMP_EQ, iter+1);
11    } else { /*Inner domain compute*/}
12 }
13 //Host-side API
14 void *kernelArgs[] = {...};
15 nvshmemx_collective_launch((void *)jacobi_kernel,
       grid_dim_x * grid_dim_y + 2, dim_block, kernelArgs,
       0, stream);
```

Listing (3) GPUSHMEM Device

Fig. 1: Code examples of GPU-aware MPI, GPUCCL, and GPUSHMEM for a 2D Jacobi solver on NVIDIA GPUs. GPUCCL uses group operations; GPUSHMEM uses device-side communication APIs.

ible GPU kernel launch strategies via a templated launch mode parameter and supports both two-sided and one-sided communication models. These capabilities help developers write portable, backend-agnostic code while maintaining performance across multi-GPU systems. In summary, our contributions are as follows:

- We design UNICONN, a unified and portable high-level C++ API that supports both point-to-point and collective operations across host- and device-initiated APIs.
- We introduce a Coordinator construct that enables compile-time switching across communication libraries, one-sided and two-sided models, and host/device APIs with minimal effort.
- We implement three communication backends: GPU-aware MPI, GPUCCL (NCCL/RCCL), and GPUSHMEM. The first two conform to the host API specification on AMD/NVIDIA GPUs, while the latter supports both host and device APIs on NVIDIA GPUs.
- We evaluate UNICONN using standard network benchmarks (latency and bandwidth), as well as Jacobi and Conjugate Gradient (CG) solvers, and compare performance against native implementations.
- UNICONN introduces negligible or acceptable overheads. In network benchmarks, overheads range on average from 5% to 1% for small and large messages. In the Jacobi solver on 64 GPUs, performance was within 1% difference of the native implementations. For CG, overheads range on average under 2% with 2 nodes, 8 GPUs on Perlmutter and Lumi.

II. BACKGROUND AND MOTIVATION

This section gives background on the commonly available communication libraries in HPC clusters and discusses their programmability issues and our motivations to design UNICONN.

A. Communication Libraries

1) GPU-aware MPI: MPI is widely used in HPC to build applications that scale to thousands of nodes. *GPU-aware MPI* can distinguish between host and device buffers, allowing data transfers to bypass the host CPU. This capability enables developers to use device buffers directly in MPI calls, allowing the MPI implementation to leverage a direct GPU-to-GPU data path established by GPUDirect RDMA [13] or ROCnRDMA [14]. Without this optimization, device buffers must be copied to or from host memory before being sent or received via MPI, introducing additional host-device transfers to the data path. Currently, most MPI implementations such as MVAPICH2, OpenMPI, and HPE Cray MPICH support GPU awareness either natively or through UCX [15]–[20].

2) GPUCCL: The GPU Collective Communication Library, we call *GPUCCL* in short, is a collection of vendor-provided, topology-aware collective primitives and, more recently, point-to-point communication APIs for inter-device communication [21]. NVIDIA, AMD, and Intel provide their respective libraries as NCCL, RCCL, and oneCCL [22]–[24]. *GPUCCL* implements communication and computation for a collective within a single kernel, reducing kernel launch overhead and providing specialized solutions. Since *GPUCCL* natively supports each vendor's GPU programming model, the APIs include a stream argument, allowing separate streams to overlap communication and computation on a GPU when needed, and the GPU itself manages the ordering of the operations.

3) GPUSHMEM: GPU-centric OpenSHMEM libraries, that we call *GPUSHMEM* in short, are Partitioned Global Address Space (PGAS) libraries from GPU vendors that implement the OpenSHMEM specification for GPUs. These libraries provide efficient one-sided put/get APIs for processes to access remote data objects and support point-to-point and collective communication between GPUs within

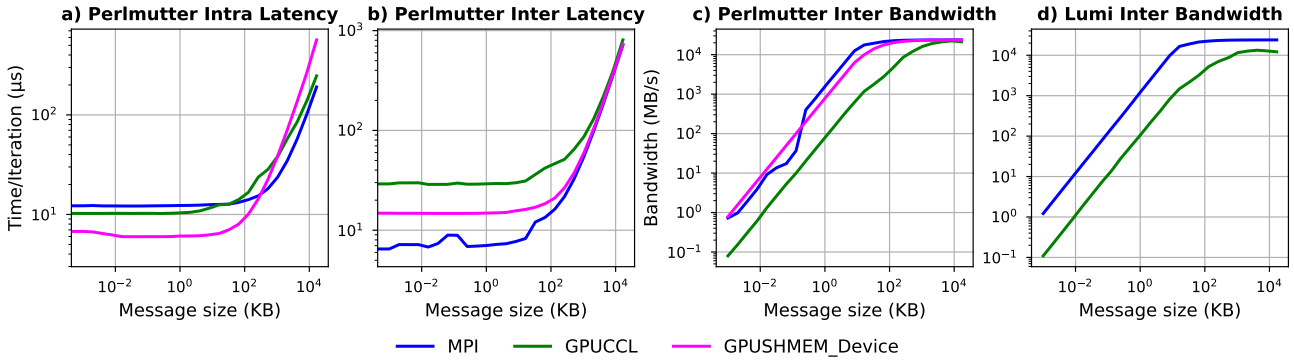

Fig. 2: Communication performance results within and across nodes in Perlmutter and Lumi supercomputers. Both the x and y axes are logarithmic. Blue and green lines represent solutions that use host-side APIs and magenta lines represent solutions that use device-side APIs.

and across nodes. GPUSHMEM libraries extend the OpenSHMEM API specification by providing stream-aware host-side and device-side APIs with additional execution granularity information, such as warp and thread-block. The thread-block and warp variants of the device-side APIs require all threads in the corresponding hierarchy to communicate cooperatively. NVIDIA for CUDA, AMD for HIP, and Intel for SYCL have their implementations as NVSHMEM, rocSHMEM, and Intel SHMEM, respectively [25]–[28].

B. Programmer's Productivity and Portability

Despite the availability of multiple library options provided by GPU vendors, users still face programming complexity and portability challenges across supercomputing platforms. One major issue is the variation in API semantics and GPU runtime integration, as illustrated in Fig. 1, which shows an example of a single iteration from a 2D Jacobi solver.

While all libraries use a buffer address and size pair, MPI and GPUCCL follow a two-sided communication model with `send/receive` operations that are synchronous between processes. In contrast, GPUSHMEM uses a one-sided model with `put/get` operations that are asynchronous with respect to the other participating GPU. This difference is evident in Listing 1 and Listing 2 versus Listing 3, where GPUSHMEM specifies the receiver's buffer address in the sender's API.

As seen in Fig. 1, GPU-aware MPI does not officially integrate a GPU stream concept into its API, whereas GPUCCL and GPUSHMEM support it natively. Thus, MPI needs *cudaStreamSynchronize* to synchronize CPU and GPU progress. Also, to reduce latency from kernel launch overheads to a GPU stream, GPUCCL features grouping calls to aggregate individual communication operations into a single kernel launch.

GPUSHMEM differs from others by enabling device-side APIs, adding another layer of complexity for developers. In addition to the execution granularity of the device APIs, which is described in Section II-A3, GPUSHMEM requires a specialized kernel launch function that enables synchronization and collective APIs on a GPU kernel. This is showcased on Listing 3, where the GPU kernel launches with *nvshmemx_collective_launch*, which also limits the number of thread blocks a kernel can launch due to a lack of preemptive scheduling on GPUs.

Moreover, each of the libraries mentioned uses a different method to initialize inter-process GPU communication (not shown in the listings): some depend on a CPU communication library, as GPUCCL does; others require tweaking environment variables; and some, like GPUSHMEM, require the use of specialized functions.

These differences result in inconsistent APIs, communication ordering, and initialization schemes, requiring deep familiarity with each library. Supporting multiple libraries often forces developers to rewrite communication interfaces, reducing productivity.

C. Performance

Communication libraries can exhibit significant performance variation due to differences in the hardware and software stacks of HPC systems. Hardware variation includes GPU types, NICs, interconnects, and GPU layouts within and across nodes. On the software side, differences in runtimes, frameworks, and technologies like GPUDirect/ROCnRDMA or GDRCopy [13], [14], [20], [29] also impact library performance across clusters.

To illustrate the performance differences of the libraries, we have conducted simple communication benchmarks derived from the OSU communication benchmarks [30]. The results are shown in Fig. 2. In these benchmarks, we measured the latency and one-way bandwidth of CUDA-aware MPI, NCCL, and device-side NVSHMEM with increasing message sizes on both Perlmutter and Lumi supercomputers.

Fig. 2 a) and b) illustrate the performance of different communication paths within and across two nodes in the same supercomputer. These results reaffirm that no single library optimally utilizes interconnects on intra-node and inter-node communication. Comparing Fig. 2 c) and b), we observe that libraries exhibit contrasting performance across two different

supercomputers, underscoring the importance of library integration with the system architecture. Fig. 2 shows performance differences across libraries for different message sizes, emphasizing the need to switch between backend libraries for the application to observe the best performance.

Overall, the results demonstrate that the optimal communication library for an application depends on multiple factors: message size, whether the communication occurs within or across nodes, and what interconnect and/or network are being used to communicate between GPUs within and/or across nodes respectively. As a result, applications should be capable of leveraging different communication libraries to achieve performance portability.

III. Design Goals

This section presents an overview of the goals and requirements for the UNICONN library.

- **Uniform Communication Interface:** A common interface should support the semantics of the widely used communication models such as MPI and OpenSHMEM.
- **Performance:** The library should meet application performance needs similar to native implementation by introducing negligible overhead.
- **Portability across GPU runtimes:** Since NVIDIA and AMD GPUs are prevalent in the current supercomputers, the library should support their runtimes.
- **Device-side API support:** The library should allow users to invoke device-side functions and host-side counterparts. In addition, the library should provide mechanisms to cooperate with kernels that have device-side functions with the existing compute kernels in the application.
- **Effortless transition:** The library should allow seamless switching between host- and device-side supports with minimal to zero code changes. This feature lets developers incrementally introduce device-side API into their multi-GPU applications while preserving existing host-side API implementations.
- **Asynchronous progress:** The library should enable asynchronous progression by allowing underlying libraries to advance multiple independent communication primitives alongside computation.

IV. Uniconn Library

To address the programmability and performance variation issues outlined in Section II and achieve the goals in Section III, we introduce UNICONN, a unified, portable C++ template communication library for multi-GPU platforms. This section provides a high-level overview of UNICONN, illustrating its abstractions and operations with example code, the UNICONN implementation of the Jacobi solver shown in Listing 4.

An application using UNICONN generally follows three main execution phases: *Setup*, *Progression*, and *Termination*. In *Setup*, required resources are created—this includes initializing the library, setting up communication and GPU kernel constructs, and allocating memory. In *Progression*, these resources are used to launch GPU kernels and perform communication operations. Finally, in *Termination*, allocated memory is released and communication resources are destroyed.

```
1  int main(int argc, char *argv[]) {
2  // Environment initialization
3     Environment<Backend> env(argc, argv);
4     int rank = env.WorldRank();
5     int size = env.WorldSize();
6     int local_rank = env.NodeRank();
7     env.SetDevice(local_rank);
8  // Communication initialization
9     Communicator<Backend> comm;
10    auto *comm_d = comm.toDevice();
11    int npes = comm.GlobalSize();
12    int mype = comm.GlobalRank();
13 // Memory management
14    A_buf = Memory<Backend>::Alloc<float>(2 * nx);
15    Anew_buf = Memory<Backend>::Alloc<float>(2 * nx);
16    sync_arr = Memory<Backend>::Alloc<uint64_t>(4);
17    cudaMalloc(&A, nx * chunk_size_high * sizeof(float));
18    cudaMalloc(&A_new, nx * chunk_size_high * sizeof(float));
19 // Coordinator construction
20    Coordinator<Backend, LaunchMode:X> jacobi_step(stream);
21    void *argsHost[] = {...};
22    void *argsPdev[] = {...};
23    void *argsFDev[] = {...};
24 // Kernel binding with launch modes
25    jacobi_step.BindKernel<LaunchMode::PureHost> (jacobi_kernel,
          dim_grid_host, dim_block_host, 0, argsHost);
26    jacobi_step.BindKernel<LaunchMode::PartialDevice>
          (jacobi_p_dev, dim_grid_p_dev, dim_block_p_dev, 0,
          argsPdev);
27    jacobi_step.BindKernel<LaunchMode::PureDevice> (jacobi_f_dev,
          dim_grid_dev, dim_block_dev, 0, argsDev);
28    comm.Barrier(stream);
29 // Jacobi time loop
30    for (iter = 0; iter < iter_max; ++iter) {
31       jacobi_step.LaunchKernel();
32       jacobi_step.CommStart();
33       jacobi_step.Post(A_buf , Anew_buf, nx, sync_arr + 1, iter
             + 1, top, &comm);
34       jacobi_step.Post(A_buf + nx, Anew_buf + nx, nx, sync_arr,
             iter + 1, bottom, &comm);
35       jacobi_step.Acknowledge(Anew_buf + nx, nx, sync_arr, iter
             + 1, top, &comm);
36       jacobi_step.Acknowledge(Anew_buf, nx, sync_arr + 1, iter +
             1, bottom, &comm);
37       jacobi_step.CommEnd();
38       std::swap(a_new, a);
39       std::swap(a_new_comm, A_buf);
40    }
41    comm.Barrier(stream);
42    cudaStreamSynchronize(stream);
43 // Memory free
44    cudaFree(A);
45    cudaFree(Anew);
46    Memory<Backend>::Free(A_buf);
47    Memory<Backend>::Free(Anew_buf);
48    Memory<Backend>::Free(sync_arr);
49 }
```

Listing 4: UNICONN host API code example for a 2D Jacobi solver on NVIDIA GPUs.

A. Library Selection

There are distinct types available in the API that represent each backend library supported in UNICONN. All interfaces in UNICONN are designed to be used with a type template argument to specify an underlying backend library to use in the application. This allows the UNICONN library to support multiple libraries under a single API and allow developers to switch, add, or extend to backend libraries without disrupting an application's existing design.

Currently, UNICONN supports three libraries: GPU-aware MPI, NCCL/RCCL, and NVSHMEM, which are denoted as types of `MPIBackend`,

```
1  __global__ void jacobi_dev(...) {
2    if (blockIdx.x == gridDim.x - 1) {
3      Compute_Row(...); // Compute top row
4      Coordinator<Backend>::Post<ThreadGroup::BLOCK> (A_buf,
           Anew_buf, nx, sync_arr + 1, iter + 1, top, comm);
5      Coordinator<Backend>::Acknowledge<ThreadGroup::BLOCK>
           (Anew_buf + nx, nx, sync_arr, iter + 1, top, comm);
6    } else if (blockIdx.x == gridDim.x - 2) {
7      Compute_Row(...); // Compute bottom row
8      Coordinator<Backend>::Post<ThreadGroup::BLOCK> (A_buf + nx,
           Anew_buf + nx, nx, sync_arr, iter + 1, bottom, comm);
9      Coordinator<Backend>::Acknowledge<ThreadGroup::BLOCK>
           (Anew_buf, nx, sync_arr + 1, iter + 1, bottom, comm);
10   } else {
11     Compute_Domain(...); // Inner domain compute
12   }
13 }
```

Listing 5: UNICONN device API example using the *PureDevice* launch mode

```
1  __global__ void jacobi_p_dev(...) {
2    Compute_Block(...); // Compute block
3    int block_iy = blockIdx.y * blockDim.y + offset1;
4    int block_ix = blockIdx.x * blockDim.x + 1;
5    if ((nx - 1) > block_ix){
6      if ((block_iy <= offset1) && (offset1 < block_iy +
           blockDim.y)) {
7        Coordinator<Backend>::Post<ThreadGroup::BLOCK> (A_buf +
             block_ix, Anew_buf + block_ix, min(blockDim.x, nx - 1
             - block_ix), nullptr, 0, top, comm);
8      }
9      if ((offset2 < block_iy + blockDim.y) && (block_iy <=
           offset2)) {
10       Coordinator<Backend>::Post<ThreadGroup::BLOCK> (A_buf + nx +
             block_ix, Anew_buf + nx + block_ix, min(blockDim.x, nx
             - 1 - block_ix), nullptr, 0, bottom, comm);
11     }}}
```

Listing 6: UNICONN device API example using the *PartialDevice* launch mode

`GpucclBackend`, and `GpushmemBackend`. rocSHMEM as a `GpushmemBackend` for AMD GPUs was not mature during the development, and we leave it as future work. An example of how to select a backend library can be seen in Listing 4, where the `Backend` is given as a template argument to the constructs of UNICONN and can be any type that is listed earlier.

B. Environment

Communication libraries often have different initialization schemes. For instance, GPU-aware MPI uses its initialization function (e.g., `MPI_Init`) along with GPU runtime device selection functions for setup and teardown. In contrast, libraries like GPUCCL and GPUSHMEM typically depend on another communication library, such as MPI, for their initialization and termination routines, making development error-prone and cumbersome.

To address this, UNICONN introduces the `Environment` abstraction, which handles the initialization and termination of the underlying communication library and sets the GPU device for execution. To set the correct GPU through `SetDevice`, `Environment` provides global and node-local process size and rank APIs. Before invoking any UNICONN operations, an application must create at least one `Environment` instance and specify a valid GPU. An example is shown in lines 3-7 in Listing 4.

C. Communicator

The `Communicator` encapsulates processes and resources for collective or point-to-point operations, analogous to an MPI Communicator or OpenSHMEM Team. UNICONN provides operations to create and destroy communicator instances, get both process size and rank within a communicator, and split an existing communicator into sub-communicators. To support device-side communication, the communicator includes a `toDevice()` function, which returns a valid GPU address for use within GPU kernels. An example of how to create and get the global size of the communicator and the global rank of the process within the communicator can be seen in lines 9 and 13 in Listing 4. Additionally, the communicator supports barrier operations on both the host and device sides, enabling synchronization across all GPUs within a communicator instance.

D. Memory Management

The UNICONN library requires all processes in an Environment to allocate and relinquish memory through the `Memory` construct for use in communication operations via `Alloc()` and `Free()` functions which is shown in between lines 14 and 16 and 46 and 48, respectively in Listing 4. This decision has been made since GPUSHMEM libraries enforce all processes to use a symmetric heap for communication. For GPU-aware MPI and GPUCCL, buffers allocated through the `Memory` construct may improve performance due to data locality by having a separate memory region for data transfer.

E. Coordinator

The `Coordinator` is the most critical abstraction in UNICONN for managing coordination between GPU computation and communication. This section will discuss the multiple aspects of `Coordinator` that allow easy transition between underlying communication libraries and host-side and device-side APIs. The constructor and destructor of `Coordinator` take a GPU stream as an argument.

1) ***Launch Mode****:* A *Launch Mode* template parameter controls a `Coordinator` instance's behavior. *Launch Mode* decides which GPU kernel to use to launch and which host or device APIs to enable for communication for the application. We currently support three types of *Launch Modes*: *PureHost*, *PureDevice*, and *PartialDevice*.

- **PureHost**: In this mode, the `Coordinator` instance uses only host-side APIs and launches GPU kernels exclusively for computation.
- **PureDevice**: This mode is the alternative to *PureHost*. The `Coordinator` instance relies solely on device-side APIs, and a GPU kernel that manages computation and communication is launched. This mode is available only when using the `GPUSHMEM` libraries. An example GPU kernel is shown in Listing 5.
- **PartialDevice**: This mode provides a middle ground between *PureHost* and *PureDevice* for p2p communications. The UNICONN library allows the developer to send

multiple messages from within a GPU kernel without immediate synchronization with the receiver; synchronization occurs later through host-side APIs. This enables developers to partition messages into smaller chunks aligned with the GPU kernel's computation pattern and send them asynchronously. An example GPU kernel is shown in Listing 6, and in Host API lines 33 until 36 in Listing 4 are handling the synchronization between neighboring GPUs. For collective operations, the behavior is identical to that of the *PureHost* mode. Like *PureDevice*, this mode is available only with `GPUSHMEM` libraries.

*2) **GPU Kernel Management**:* In Section III, we described our motivation for enabling the UNICONN library to switch between host-side and device-side APIs without requiring any code changes. Achieving this flexibility requires a mechanism for switching between different GPU kernels, either for computation alone or for computation combined with communication, within an application.

Incorporating device-side function calls into a kernel can increase register usage per thread, thereby reducing GPU occupancy compared to a computation-only kernel [31]. Furthermore, developing a new kernel with embedded communication primitives requires additional effort. This work primarily involves managing thread hierarchies and balancing computation and communication workloads within the GPU kernel. Thus, we designed the `Coordinator` construct to support launching different GPU kernels, controlled via the `LaunchMode` template parameter, to address these challenges. With this capability, the library can launch kernels using the appropriate GPU kernel launch functions and parameters the underlying backend library requires, through the `BindKernel` and `LaunchKernel` constructs. This design enables developers to incrementally extend their codebases to incorporate device-side APIs without disrupting existing workflows, while allowing each GPU kernel to be independently optimized.

The `BindKernel` construct is used to bind GPU kernels and the required parameters to a `Coordinator` instance. This binding is performed by comparing the `LaunchMode` template argument to both the function and the class, then storing the appropriate arguments. An example of `BindKernel` is shown at lines 20-27 in Listing 4, where a `Coordinator` instance is initialized with `LaunchMode::X`. Here, `X` can be any valid value of `LaunchMode`, and only the corresponding `BindKernel` invocation stores the parameters based on the `X` value. The `LaunchKernel` construct is used to initiate the GPU kernel launch. This construct does not take any parameters, as all the necessary data for the GPU kernel launch is already stored within the `Coordinator` instance, as illustrated at line 31 in Listing 4.

F. Communication Operations

The UNICONN library supports point-to-point and collective communication operations over a UNICONN Communicator, through both host and device-side APIs. These primitives are implemented within the `Coordinator` construct and are managed through the `LaunchMode` template parameter

```
1  class Coordinator<Backend,LaunchMode::X> {
2  void Post<T>(T* sendbuf, T* recvbuf, size_t size, uint64_t*
       sig_loc, uint64_t sig_val, int dest_id,
       Communicator<Backend>* comm);
3   void Acknowledge<T>(T* recvbuf, size_t size, uint64_t* sig_loc,
       uint64_t sig_val, int src_id, Communicator<Backend>* comm);
4   void AllGather<T>(T* sendbuf, T* recvbuf, size_t count,
       Communicator<Backend>* comm); // +In-Place +Vectorized
5   void AllReduce<ReductionOperator::OP,T>(T* sendbuf, T* recvbuf,
       size_t count, Communicator<Backend>* comm); // +In-Place
6   void Reduce<ReductionOperator::OP,T>(T* sendbuf, T* recvbuf,
       size_t count, int root, Communicator<Backend>* comm); //
       +In-Place
7   void AlltoAll(T* sendbuf, T* recvbuf, size_t count,
       Communicator<Backend>* comm); //+Vectorized
8   void Broadcast<T>(T* buf, size_t count, int root,
       Communicator<Backend>* comm);
9   void Gather(T* sendbuf, T* recvbuf, size_t count, int root,
       Communicator<Backend>* comm); //+In-Place +Vectorized
10  void Scatter(T* sendbuf, T* recvbuf, size_t count, int root,
       Communicator<Backend>* comm); /*+In-Place +Vectorized*/ }
```

Listing 7: Main UNICONN host communicaton interface

described in Section IV-E1. All host-side communication APIs provided by UNICONN are shown in Listing 7. In this section, we discuss the design choices made for the UNICONN communication APIs to conform to two-sided and one-sided communication models of existing libraries, while maintaining a uniform interface across host and device APIs.

*1) **Data Types**:* To ensure type safety for host and device APIs, UNICONN uses a type template parameter to determine the most suitable argument type based on the given data type for the communication functions.

*2) **P2P Primitives**:* The `Post` and `Acknowledge` operations are analogous to MPI's `send` and `recv` operations. However, these operations require additional arguments to support two-sided and one-sided communication models. In one-sided communication, the sender has direct access to the receiver's buffer address, and the receiver is notified via a signaling operation, which is an atomic update to a counter.

Consequently, the `Post` operation (Listing 7, Line 2) requires a send buffer, a receive buffer, and a signal buffer to support both semantics and leverage the asynchronous behavior of one-sided P2P communication. Similarly, the `Acknowledge` operation (Listing 7, Line 3) requires a receive buffer and a signal buffer to allow the receiver to be notified by the sender.

Importantly, the communication semantics of `Post` and `Acknowledge` preserve those of the underlying backend: with MPI or GPUCCL, communication completion is synchronized; with GPUSHMEM, communication completion remains asynchronous between GPUs.

*3) **Collectives**:* The collective operations in both the host and device APIs are similar to those provided by MPI and GPUSHMEM, which are shown in Listing 7. Additionally, UNICONN includes vectorized and in-place variants of some collective operations, indicated by `+Vectorized` and `+In-Place` comments. For reduction operations such as `AllReduce` and `Reduce`, a template parameter called `ReductionOperator` specifies the reduction operator to apply.

*4) **Device-side Thread Granularity Selection**:* To expose and efficiently utilize different levels of GPU execution granu-

larity, device-side APIs include an additional template parameter named `ThreadGroup`, which specifies the thread hierarchy to use for a primitive. The UNICONN library currently supports `THREAD`, `WARP`, and `BLOCK` granularities.

G. Operation Grouping

UNICONN includes two Host APIs on the `Coordinator`, namely `CommStart` and `CommEnd`, to allow for independent and non-blocking execution of communication operations listed between these functions. `CommStart` prepares the `Coordinator` instance for non-blocking execution and allows for registration of communication primitives. `CommEnd` ensures the completion of the registered communication operations before executing the next GPU operation on the GPU stream of the `Coordinator` instance.

H. Revisiting Jacobi Example

To provide a top-down view of the UNICONN constructs, we revisit the Jacobi example with the help of Listing 4. The program begins with the `Setup` phase by initializing the `Environment` object, which sets the rank, communication world, and local rank. It then creates a `Communicator` followed by memory allocation for solver buffers.

Afterwards, a `Coordinator` is instantiated with a GPU stream to manage kernel launches and communication bindings, configured with a chosen `LaunchMode` that we denote with `X`. All constructs are templated with a `Backend`, allowing the application to target different communication libraries. Depending on the selected launch mode, `BindKernel` is invoked to specify the kernel function. Although the example shows three possible bindings, only one is active at runtime based on the chosen `LaunchMode::X`. Automating the switching of these backends is a subject of future work.

Later, the Jacobi time loop follows, where each iteration launches the computation with `LaunchKernel`, prepares the `Coordinator` for non-blocking communication operations with `CommStart`, and registers the communication requests with calls such as `Post` and `Acknowledge` to synchronize halo exchanges. At the end of an iteration, `CommEnd` is called to ensure completion of communication operations on the GPU stream before moving on with the next iteration.

Finally, after `Barrier`, the program explicitly deallocates memory and uses the RAII technique, which ensures all previously allocated constructs are destroyed automatically at the end of their lifetime. This includes the `Environment` instance, which invokes the backend library's termination.

V. IMPLEMENTATION

The UNICONN library is implemented as a C++ template library, using templates to create an interface for every construct and function within the UNICONN API. This approach enabled us to implement constructs for each underlying communication library as template specializations of the respective construct template. Additionally, for each template parameter discussed in Sections IV-E1, IV-F3, IV-F4, and the data types in Section IV-F1, we utilize template specializations and instantiations for the corresponding template functions. This design allows us to separate implementations while maintaining a unified function signature and enables extending the UNICONN library to future programming models and communication libraries.

The UNICONN library interacts with GPU runtimes using vendor-agnostic function macros for portability across supercomputers and GPU vendors. These macros are expanded into specific GPU runtime functions based on the compile-time definitions for the target GPU runtime. Additionally, UNICONN provides default arguments to select the underlying communication library and *LaunchMode* for an application. These selections can be controlled through compile-time definitions that match the supported libraries and *LaunchModes*.

A. Semantic Coverage

Most communication operations in UNICONN can be directly mapped to corresponding backend functions. When a backend lacks a one-to-one match, UNICONN composes the operation using equivalent P2P primitives. For MPI, UNICONN supports both blocking and non-blocking P2P and collectives for native datatypes, mapping `Post` to `MPI_Send` or `MPI_Isend` depending on whether the operation belongs to a group. We are aware of GPU-aware MPI having mature one-sided API for P2P operations, and we will investigate this support as future work.

With NCCL/RCCL, UNICONN maps host-side APIs directly when possible, and constructs grouped P2P operations for unsupported collectives. For NVSHMEM, available host and device-side collectives are mapped directly to respective operations when possible; otherwise, UNICONN emulates them by using Put/Get operations with barriers to ensure ordering and completion.

Crucially, UNICONN preserves the completion semantics of each backend. For example, `Post` and `Acknowledge` follow synchronized communication completion for MPI and GPUCCL, aligning with their two-sided models. In contrast, NVSHMEM maintains asynchronous communication between GPUs using one-sided `PutWithSignal` and `WaitSignal`.

VI. EXPERIMENTS

To evaluate the portability and performance of UNICONN, we implemented and conducted three experiments across three supercomputers featuring both NVIDIA and AMD GPUs. These experiments include a network microbenchmark, a 2D Jacobi solver, and a Conjugate Gradient (CG) solver. Each experiment was executed using both the native API of the communication library, referred to as *libname: Native*, and the UNICONN API with the corresponding backend, denoted as *libname:* UNICONN (e.g., MPI-Native and MPI-Uniconn).

A. Experiment Setup

1) Supercomputers: Experiments were executed on three supercomputers: Perlmutter, Lumi, and MareNostrum5, whose hardware and software specifications are summarized in Table I. All reported network bandwidth values in the table are for unidirectional single GPU to GPU communication.

Supercomputer	CPU	GPU	Node Interconnect	Network Interconnect	GPU Runtime	MPI	GPU CCL	GPU SHMEM
Perlmutter GPU	1× AMD EPYC 7763	4× NVIDIA A100 (40 GB)	4× NVLink 3.0 (100 GB/s)	4× 200 Gb/s HPE Cray Slingshot 11	CUDA 12.4	Cray MPICH 8.1.30	NCCL 2.24.3	NVSHMEM 3.2.5
LUMI-G	1× AMD EPYC 7A53	4× AMD MI250X (128 GB)	1–4× Infinity Fabric (50 GB/s/link)	4× 200 Gb/s HPE Cray Slingshot 11	ROCm 6.0.3	Cray MPICH 8.1.29	RCCL 2.18.3	N/A
MareNostrum5 ACC	2× Intel Xeon 8460Y+	4× NVIDIA H100 (64 GB)	4× NVLink 4.0 (150 GB/s)	4× 200 Gb/s NDR InfiniBand	CUDA 12.6	OpenMPI 4.1	NCCL 2.18.5	NVSHMEM 3.1.7

TABLE I: Hardware and software characteristics of the supercomputers used in the experiments.

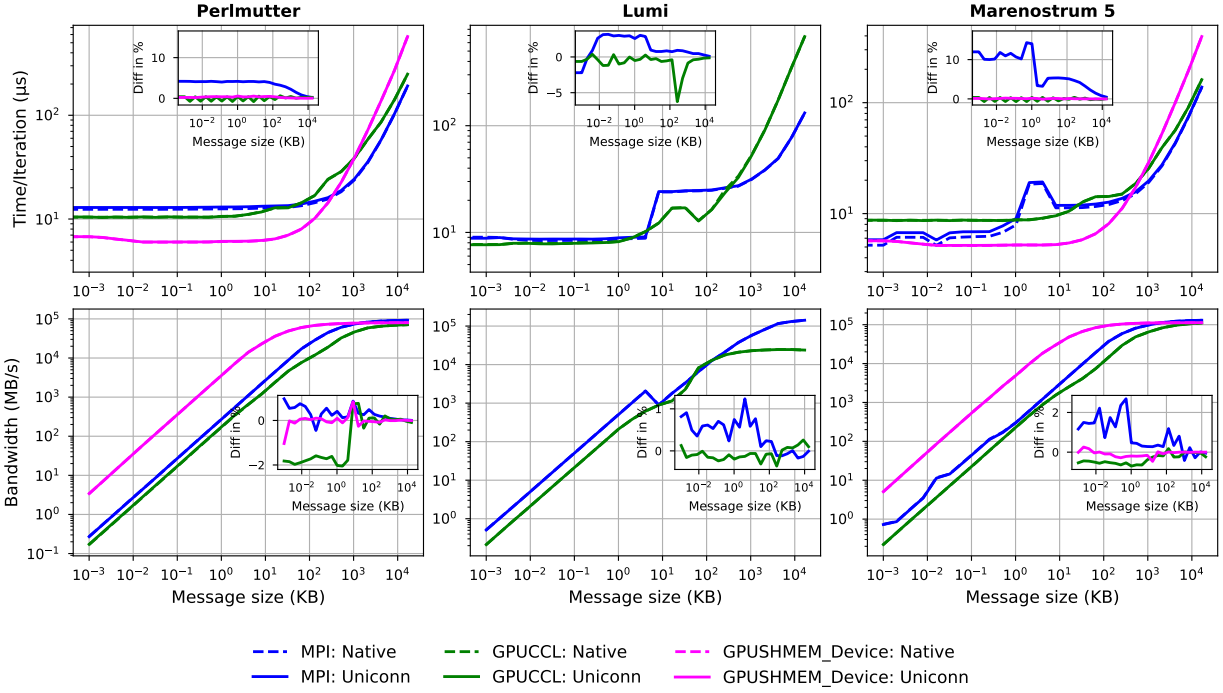

Fig. 3: Latency and bandwidth intra-node results

2) Measurements: We measure completion time using CUDA/HIP event-based timing, where start and end events are recorded on the main stream of the application. The host synchronizes with the device using the end event to determine execution time. Before timing begins, each experiment performs warm-up iterations, followed by a host and device synchronization. Each measurement is repeated ten times to reduce the impact of outliers and network variability. After omitting the lowest and the highest measurements, the average execution time is reported.

B. Network Benchmarks

To stress-test the limits of our API, we implemented both latency and bandwidth benchmarks, which are adapted from the OSU communication benchmarks [30], using the UNICONN Host and Device APIs. We then compared the performance of these UNICONN implementations against their native counterparts. For the latency benchmark, we ran 100K iterations with 10K warm-up iterations for message sizes below 8 KiB, and 10K iterations with 1K warm-up iterations for larger sizes. The bandwidth benchmark used 64 concurrent messages over 1000 iterations (100 warm-up) for messages smaller than 8KiB, and 200 iterations (20 warm-up) for larger messages.

The intra-node and inter-node benchmark results are presented in Fig. 3 and Fig. 4, respectively. All benchmarks were conducted using two GPUs connected through a node or network interconnect. Native implementations are shown with dashed lines, while UNICONN variants are depicted using solid lines. Each figure includes an embedded plot highlighting the performance difference between the native and UNICONN implementations as a percentage.

In intra-node and inter-node benchmark results, UNICONN host API performance differs by at most 7% for intra-node and at most 3% for inter-node cases, on average. This variation can be attributed to network interference, and interactions between the CPU and GPU streams, particularly with MPI, which produced irregular results across message sizes. When we investigated the variation and the overhead of Uniconn over MPI, the main reason is the decision logic for calling either blocking or non-blocking MPI operations to support operation grouping in communication API. Additionally, each blocking MPI call queries the GPU stream for pending op-

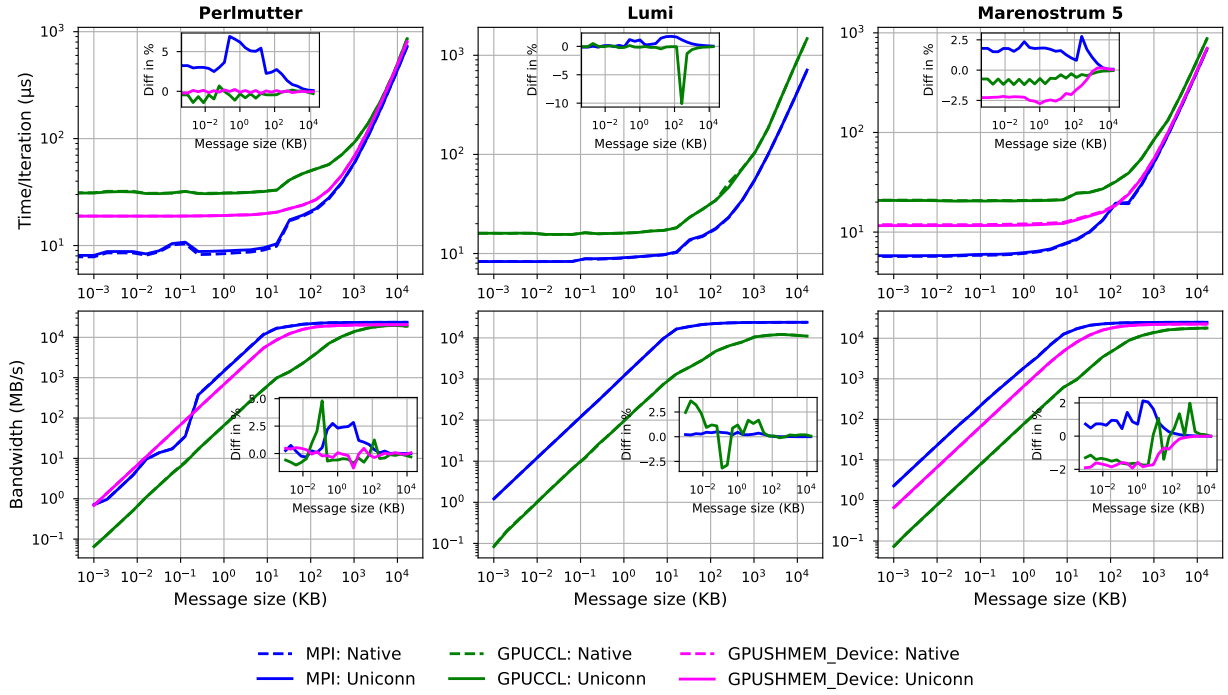

Fig. 4: Latency and bandwidth inter-node results

erations, introducing extra overhead. Due to MPI's multiple P2P modes, these stream queries interfere with communication progress, especially for small messages in the Acknowledge API, leading to variability in overhead across message sizes. For GPUCCL, the performance are within 1% with respect to native implementation for both intra-node and inter-node on average. These findings suggest that the Host API is efficient enough for developers to consider replacing the native communication library API.

In addition, UNICONN's device API performs similarly with respect to the native device APIs of GPUSHMEM, as shown in both Fig.3 and Fig.4. This trend is due to how the implementation of UNICONN's Device APIs are inlined inside the library header files and compiled together with the application code, eliminating per-warp function call overhead within GPU kernels and allowing better optimization from the compiler. This performance is evident in intra-node latency and bandwidth benchmarks, where the overhead is at most 0.08% on average for both.

While our engineering efforts aim to reduce overhead toward 0%, we argue that developers can still benefit if the overhead is significantly smaller than the performance gap between backend libraries. Our programming model enables the selection of the best-performing backend with minimal migration efforts. Automating migration from legacy models remains a topic for future work.

C. Jacobi 2D Solver

The iterative Jacobi solver is a stencil computation that updates each grid element based on the values of its immediate neighbors within a fixed-size window. Such operations are common in scientific simulations and image processing. Thanks to their regular communication pattern and predictable volume, stencil codes serve as a strong use case for evaluating the UNICONN P2P performance. In our evaluation, we use a 5-point 2D star stencil on a grid of size $2^{14} \times 2^{14}$ with single-precision floats. The grid is partitioned equally across GPUs along the y-axis, so each GPU handles a $2^{14} \times 2^{14}/N$ domain, where N is the number of GPUs. We run 100K iterations, with 10K as warm-up.

Fig. 5 compares UNICONN's performance against native libraries on up to 64 GPUs. This corresponds to 16 nodes on Perlmutter and MareNostrum5, and eight nodes on LUMI, where the HIP/ROCm stack treats each MI250X Graphics Compute Die (GCD) as a separate GPU.

The results in Fig. 5 show that UNICONN performs comparably to the native implementations, with less than 1% average difference across all GPU counts. However, the results from all backends, become less stable as the GPU count increases. This variation may stem from cascading delays between GPUs, which can occur in both native and UNICONN implementations. Such delays may arise when one process affects another process's progress, slowing the overall application due to the halo exchange communication pattern.

D. Conjugate gradient

Conjugate Gradient (CG) is used to solve large linear systems of the form $Ax = b$, where A is sparse, symmetric, and positive definite. In this experiment, each GPU holds a partition of matrix A and the corresponding portion of the

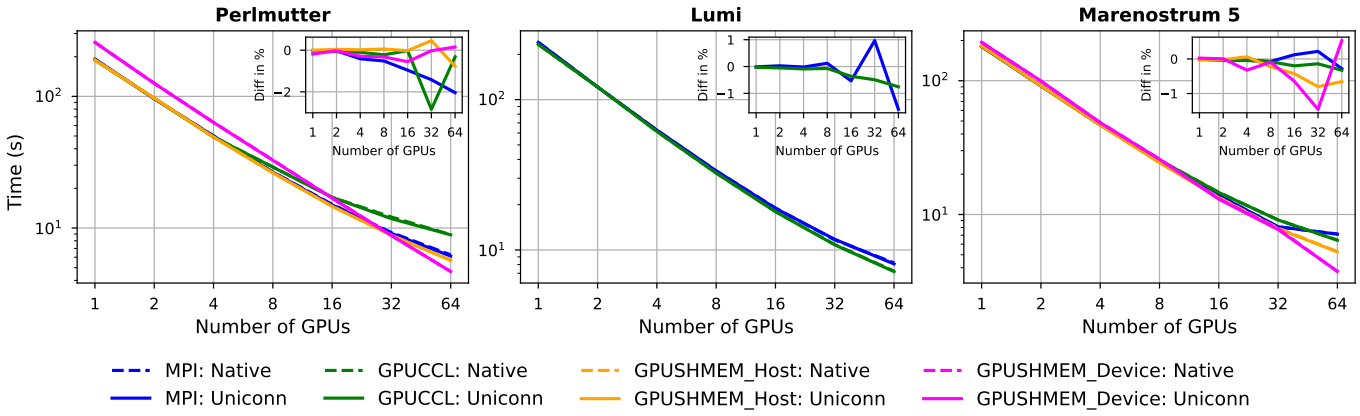

Fig. 5: Jacobi 2D results, Lower is better.

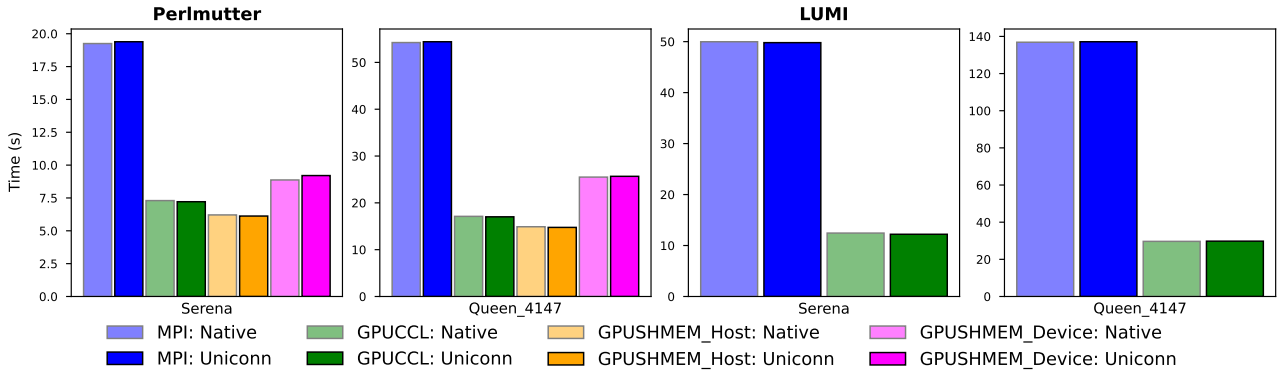

Fig. 6: CG results on 8 GPUs, Lower is better

input vector x. Each CG iteration involves a sparse matrix-vector multiplication (SpMV), which requires exchanging the relevant parts of x across GPUs to account for nonzero columns in their local matrix partitions. And two dot products require an *AllReduce* operation across all GPUs. We wanted to observe how UNICONN collective APIs are performing relative to their Native counterparts in a widely used linear solver.

We distribute the rows of A equally in length across GPUs, without accounting for the number of nonzeros in the input matrix. Inter-GPU communication during SpMV uses the *AllGatherv* collective primitive in both the Native and UNICONN APIs. Our experiments use two matrices from the SuiteSparse Matrix Collection: Serena and Queen_4147, with 1,391,349 and 4,147,110 rows, respectively [32]. We ran 10K iterations (no warm-up) on 8 GPUs using two nodes each on Perlmutter and LUMI.

As shown in Fig. 6, CG results are promising in which the UNICONN implementations are performing on par with the Native implementation such that the average difference between Native and UNICONN is less than 1% with a slight slowdown in Device API on the Serena matrix which is around 3%. This solidifies that UNICONN Host and Device APIs are suitable for collective operations. However, when we examine MPI results, the runtime is significantly higher than in other versions. A collective implementation may have caused this result. To check if a collective operation is creating a bottleneck, we tested all MPI and GPUCCL versions with Allgatherv disabled and MPI native and UNICONN versions' runtime were similar to the GPUCCL runtime.

E. Programmability

We include a table comparing SLOC across backends which is listed in Table II. While Uniconn's SLOC is slightly higher with respect to native implementations, this overhead reflects the inclusion of both host and device APIs in the same code-base via the Coordinator API. We also argue qualitatively that developing against a single communication library improves productivity and reduces maintenance burden, especially when compared to managing multiple code variants for different backends.

Library	Latency	Bandwidth	Jacobi2D	CG
MPI	112	122	162	773
GPUCCL	122	131	184	775
GPUSHMEM_Host	N/A	N/A	173	818
GPUSHMEM_Device	139	154	233	810
Uniconn	125	148	246	842

TABLE II: Source line of code (SLOC) for each experiment

VII. Related work

Combining multiple communication libraries under a unified interface is being explored for specific AI or HPC workloads. Aluminum is a communication library for AI workloads that unifies MPI and NCCL under a two-sided interface [10]. It supports communication-computation overlap via non-blocking operations with explicit wait operation, a dedicated progression engine and includes optimized collective algorithms to reduce latency. Our work differs between Aluminum by using a grouping model (similar to NCCL/RCCL) to support asynchronous operations, and it additionally supports Device APIs and GPU kernel management; enabling device-side communication and extensibility for custom backends and algorithms.

MCR-DL is a communication framework for distributed deep learning on NVIDIA GPUs that enables simultaneous use of multiple libraries [11]. It offers a thin Python wrapper over multiple communication libraries for PyTorch, requiring explicit backend selection and `torch.Tensor` buffers within their API. Each communication backend is implemented separately in C++, and the framework includes a tuning suite to select the optimal backend per message size. In contrast, UNICONN provides a backend-agnostic, templated C++ host and device API applicable to a broader range of GPU applications on both NVIDIA and AMD.

Automatic backend selection is an important but orthogonal concern to UNICONN, and depends on multiple factors which are briefly discussed in section II-C. A machine learning-based approach, such as the one used in MDLoader, can help guide this choice and could be adapted in UNICONN [33].

Other projects, such as [12], combine two-sided and one-sided models using MPI and NVSHMEM to improve performance in specific applications on NVIDIA GPUs. In contrast, UNICONN is designed as a general-purpose layer to support existing and emerging communication libraries with minimal code changes for new or existing applications.

With the growing availability of vendor-supported libraries, several studies have benchmarked GPU-aware MPI, NCCL/RCCL, and NVSHMEM [1], [34]. On systems like Leonardo and Alps, NCCL often outperforms MPI for collectives on NVIDIA GPUs. RCCL performs well for large messages on AMD but poorly on small-message collectives. NVSHMEM has shown competitive results for Jacobi solvers on Summit [35], [36], but its reliance on CPU progress threads limits device-initiated communication. While InfiniBand GPUDirect Async (IBGDA) addresses this, its deployment is still limited. Recent work on the Karolina Supercomputer demonstrates fully CPU-free Jacobi and CG solvers using NVSHMEM with better scaling than MPI on up to eight GPUs [37]. These trends underscore the need for a unified, portable communication abstraction like UNICONN to navigate the diverse capabilities and performance trade-offs across platforms.

VIII. Conclusion

On modern systems accelerated with GPUs, applications can employ various communication libraries like MPI, NCCL/RCCL, and NVSHMEM for data movement. However, the distinct APIs of these libraries, different semantics, and varying performance across systems create programming complexity and hinder performance portability. In this paper, we introduced UNICONN, a high-level, portable C++ communication library that supports both point-to-point and collective operations across GPU clusters. UNICONN enables seamless switching between communication backends and APIs (host or device) with minimal changes to the application code. We evaluated UNICONN with latency/bandwidth benchmarks and two numerical solvers. The results demonstrated that UNICONN introduces negligible overhead while ensuring performance portability across diverse systems. Future work will focus on expanding UNICONN's application support and backend compatibilities, automating the migration from legacy code, and performance-guided automated backend library selection.

Acknowledgment

Authors from Koç University have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number DE-AC02-05CH11231, and used resources of the National Energy Research Scientific Computing Center (NERSC). We acknowledge EuroHPC Joint Undertaking for awarding the project ID EHPC-DEV-2024D10-091 access to the MareNostrum5 supercomputer in Spain and the LUMI supercomputer in Finland.

References

[1] D. Unat, I. Turimbetov, M. K. T. Issa, D. Sağbili, F. Vella, D. D. Sensi, and I. Ismayilov, "The landscape of gpu-centric communication," 2024. [Online]. Available: https://arxiv.org/abs/2409.09874

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[3] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. P. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "Amrex: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, 2019. [Online]. Available: https://doi.org/10.21105/joss.01370

[4] J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. T. Mills, T. Munson, B. F. Smith, and S. Zampini, "The PetscSF scalable communication layer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 842–853, 2022.

[5] A. Abdelfattah, N. Beams, R. Carson, P. Ghysels, T. Kolev, T. Stitt, A. Vargas, S. Tomov, and J. Dongarra, "Magma: Enabling exascale performance with accelerated blas and lapack for diverse gpu architectures," *The International Journal of High Performance Computing Applications*, vol. 38, no. 5, pp. 468–490, 2024. [Online]. Available: https://doi.org/10.1177/10943420241261960

[6] X. S. Li, P. Lin, Y. Liu, and P. Sao, "Newly released capabilities in the distributed-memory superlu sparse direct solver," *ACM Trans. Math. Softw.*, vol. 49, no. 1, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3577197

[7] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[8] V. Jatala, R. Dathathri, G. Gill, L. Hoang, V. K. Nandivada, and K. Pingali, "A study of graph analytics for massive datasets on distributed multi-gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 84–94.

[9] J. Choi, Z. Fink, S. White, N. Bhat, D. F. Richards, and L. V. Kale, "Gpu-aware communication with ucx in parallel programming models: Charm++, mpi, and python," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 479–488.

[10] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An asynchronous, GPU-aware communication library optimized for large-scale training of deep neural networks on HPC systems," in *Proceedings of the Workshop on Machine Learning in HPC Environments (MLHPC)*, 2018.

[11] Q. Anthony, A. A. Awan, J. Rasley, Y. He, A. Shafi, M. Abduljabbar, H. Subramoni, and D. Panda, "Mcr-dl: Mix-and-match communication runtime for deep learning," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 996–1006.

[12] J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. T. Mills, T. Munson, B. F. Smith, and S. Zampini, "The petscsf scalable communication layer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 842–853, 2022.

[13] NVIDIA, "GPUDirect RDMA," https://docs.nvidia.com/cuda/gpudirect-rdma/, 2023.

[14] AMD, "ROCnRDMA," https://github.com/rocmarchive/ROCnRDMA, 2023.

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

[16] OpenMPI, "Open MPI v5.0.x Documentation: CUDA," https://docs.open-mpi.org/en/v5.0.x/tuning-apps/networking/cuda.html, 2023.

[17] HPE, "Cray MPICH Documentation," https://cpe.ext.hpe.com/docs/mpt/mpich/intro_mpi.html, 2021.

[18] K. Shafie Khorassani, J. Hashmi, C.-H. Chu, C.-C. Chen, H. Subramoni, and D. K. Panda, "Designing a rocm-aware mpi library for amd gpus: Early experiences," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 118–136.

[19] AMD, "ROCm Documentation: GPU-Enabled MPI," https://rocm.docs.amd.com/en/latest/how_to/gpu_aware_mpi.html, 2023.

[20] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "Ucx: an open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[21] A. Weingram, Y. Li, H. Qi, D. Ng, L. Dai, and X. Lu, "xccl: A survey of industry-led collective communication libraries for deep learning," *J. Comput. Sci. Technol.*, vol. 38, no. 1, p. 166–195, Mar. 2023. [Online]. Available: https://doi.org/10.1007/s11390-023-2894-6

[22] NVIDIA, "NCCL," https://developer.nvidia.com/nccl, 2025.

[23] AMD, "RCCL," https://rocm.docs.amd.com/projects/rccl/en/latest/, 2025.

[24] Intel, "OneCCL," https://uxlfoundation.github.io/oneCCL/index.html, 2025.

[25] NVIDIA, "Nvshmem," https://developer.nvidia.com/nvshmem, 2024.

[26] AMD, "ROC_SHMEM," https://github.com/ROCm-Developer-Tools/ROC_SHMEM, 2023.

[27] K. Hamidouche and M. LeBeane, "Gpu initiated openshmem: Correct and efficient intra-kernel networking for dgpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 336–347. [Online]. Available: https://doi.org/10.1145/3332466.3374544

[28] Intel, "Intel® SHMEM," https://github.com/oneapi-src/ishmem, 2023.

[29] NVIDIA, "Magnum IO GDRCopy," https://developer.nvidia.com/gdrcopy, 2023.

[30] O. S. U. Network-Based Computing Laboratory (NBCL), "Osu micro benchmarks," https://mvapich.cse.ohio-state.edu/benchmarks/.

[31] J. Baydamirli, T. Ben Nun, and D. Unat, "Autonomous execution for multi-gpu systems: Compiler support," in *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '24. IEEE Press, 2025, p. 1129–1140. [Online]. Available: https://doi.org/10.1109/SCW63240.2024.00155

[32] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[33] J. Bae, J. Y. Choi, M. L. Pasing, K. Merita, P. Zhang, and K. Z. Ibrahim, "Mdloader: A hybrid model-driven data loader for distributed graph neural network training," in *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '24. IEEE Press, 2025, p. 1046–1057. [Online]. Available: https://doi.org/10.1109/SCW63240.2024.00145

[34] D. De Sensi, L. Pichetti, F. Vella, T. De Matteis, Z. Ren, L. Fusco, M. Turisini, D. Cesarini, K. Lust, A. Trivedi, D. Roweth, F. Spiga, S. Di Girolamo, and T. Hoefler, "Exploring gpu-to-gpu communication: Insights into supercomputer interconnects," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00039

[35] C.-H. Hsu, N. Imam, A. Langer, S. Potluri, and C. J. Newburn, "An initial assessment of nvshmem for high performance computing," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 1–10.

[36] T. Groves, B. Brock, Y. Chen, K. Z. Ibrahim, L. Oliker, N. J. Wright, S. Williams, and K. Yelick, "Performance trade-offs in gpu communication: A study of host and device-initiated approaches," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 126–137.

[37] I. Ismayilov, J. Baydamirli, D. Sağbili, M. Wahib, and D. Unat, "Multi-gpu communication schemes for iterative solvers: When cpus are not in charge," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 192–202. [Online]. Available: https://doi.org/10.1145/3577193.3593713