Performance and Productivity Evaluation of HPC Communication Libraries and Programming Models

by

Alex Johnson

B.S. in Electrical Engineering, University of Pittsburgh, 2019

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical and Computer Engineering

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Alex Johnson

It was defended on

April 1, 2021

and approved by

Ahmed Dallal, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Robert Kerestes, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Alan D. George, Ph.D., Professor, Department Chair, R&H Mickle Endowed Chair, Department of Electrical and Computer Engineering

Performance and Productivity Evaluation of HPC Communication Libraries and Programming Models

Alex Johnson, M.S.

University of Pittsburgh, 2021

To reach exascale performance, data centers must scale their systems, increasing the number of nodes and equipping them with high-performance network interconnects. Orchestration of the communication between nodes serves as one of the most performance-critical aspects of highly distributed app development. While the standard for HPC communication is two-sided communication as represented by Message Passing Interface (MPI), the use of two-sided communication may not effectively express certain communication patterns. It may also fail to take advantage of key performance-critical features supported by state-of-the-art interconnects such as remote direct memory access (RDMA). By contrast, one-sided communication libraries such as MPI's extensions for remote memory access (RMA) and OpenSHMEM can provide developers with the added flexibility of one-sided communication primitives and the capability to take advantage of RDMA. To investigate these approaches, this research provides comparative performance and productivity analysis of two-sided MPI, one-sided MPI and OpenSHMEM using kernels to simulate various communication and computation patterns representative of HPC apps. Performance is measured in terms of latency and achieved throughput using up to 320 nodes at the National Energy Research Scientific Computing Center (NERSC) Cori and Pittsburgh Supercomputing Center (PSC) Bridges-2 systems. Additionally, the productivity of the communication interfaces is analyzed quantitatively and qualitatively. RMA-based APIs are found to show lower latency and efficient scalability across the DAXPY, Cannon's Algorithm Matrix Multiply, SUMMA Matrix Multiply and Integer Sort kernels. Similarly, the RMA-based libraries achieve the best throughput, with OpenSHMEM achieving up to double the total concurrent data movement of MPI. Conversely, MPI's two-sided API produces the simplest programs in terms of lines of code and API calls, but it generally shows the highest latency across the evaluated kernels. The OpenSHMEM API achieves the highest performance for the four kernels and is simpler in terms of our productivity metrics than one-sided MPI for RMA-optimized codes. In contrast to these findings, two-sided MPI remains a strong library for HPC communication due to its robust set of API calls and optimized collective performance.

Table of Contents

List of Tables

List of Figures

Preface

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

This research also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges-2 system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

I would like to thank the University of Pittsburgh Center for Research Computing, National Energy Research Scientific Computing Center and Pittsburgh Supercomputing Center for their role in enabling this research and for their resources for debugging and configuring their systems.

I would also like to thank Jeff Hammond at NVIDIA for his guidance when I began working with SHMEM and one-sided MPI routines.

Finally, I would like to thank all at NSF SHREC who have helped support and review this work.

1.0 Introduction

As data centers continue to approach exascale performance, there is a growing emphasis on parallel-communication languages and libraries in an effort to harness high-performance computing (HPC) resources effectively. The first exascale system in the world, Frontier, scheduled to be completed in 2021, will deliver over 1.5 exaflops and feature tens of thousands of compute nodes [1]. Additionally, modern supercomputers are moving towards heterogeneous architectures, leveraging massively parallel hardware such as GPUs. Nonetheless, many apps require computational capability that is far beyond that of a single node, often employing over 1000 nodes [2] [3] [4]. The data center itself is a parallel architecture, with many nodes connected through complex network fabrics. Additionally, as these supercomputing centers scale upwards in number of nodes, their associated energy footprint matches that increase. For example, the Frontier system is projected to consume 30 megawatts [1]. These powerful systems provide new possibilities due to their computational potential, but also introduce novel challenges in terms of efficiency, reliability and programmability. Systems of this magnitude are difficult to efficiently program due to complex coordination between nodes, creating a significant challenge for app designers.

A major consideration for efficient execution on supercomputer resources is communication between nodes. For instance, large simulation apps fundamentally have memory and computational requirements that go beyond a single node, so having functionality to move data efficiently over a network has become necessary for many apps. Just as the computational capabilities of HPC nodes have improved, network interconnects such as Cray Slingshot and NDR InfiniBand are improving, supporting throughputs of over 200 Gb/s while also providing other novel features to enable more communication acceleration [5] [6]. One such feature is Remote Direct Memory Access (RDMA), which enables a node to access remote memory over the network without interrupting the remote CPU. This functionality allows for remote memory access (RMA) to occur without the need for separate runtime threads on the remote CPU to manage incoming communication actions, thereby freeing computational resources of the remote CPU.

High-level programming libraries enable rapid, simplified development of high-performance distributed HPC programs. Although low-level application program interfaces (APIs) like InfiniBand Verbs [7] and Distributed Memory Application (DMAPP) [8] expose highly extensible networking capabilities, most app developers use higher-level libraries such as MPI for communication on HPC systems. Higher-level libraries still allow for flexibility in communication patterns, but abstract away tedious tasks such as setting up socket connections, packing message payloads, and managing communication channels. These libraries also expose methods for performing collective routines such as *broadcast*() or *scatter*() which must be programmed manually in low-level APIs. A productive HPC library can provide high-level abstractions while also achieving high performance, potentially with the usage of modern features like RDMA and network-accelerated collective routines [9]. There is a trade-off between high- and low-level libraries, as abstraction can result in performance penalties, while the flexibility provided by lower-level APIs may increase the probability of hard-to-resolve bugs being introduced into code and hinder development.

While Message Passing Interface (MPI) has been the *de facto* communication library for HPC apps, other libraries and languages use new approaches, features, and communication models. MPI's primary model for communication is two-sided in nature, meaning a communication call of *send*() has a corresponding *recv*() call. Newer additions to the MPI standard add a secondary API for one-sided communication calls such as *put*() and *get*(). One-sided communication is synonymous to RMA; therefore it allows programmers to leverage RDMA on modern HPC interconnects. The RMA interface exposed by modern MPI will be referred to as MPI-RMA in this paper and can be thought of as its own API. SHMEM is another communication model implemented as a library and standardized as the OpenSHMEM specification. OpenSHMEM leverages a partitioned global address space (PGAS) memory model and also uses RMA as its primary communication method. It is crucial to investigate these newer APIs for potential productivity and performance benefits. Much of MPI's success has come from its relatively simple interface, so if a new communication paradigm is shown to provide better overall performance but at a significant cost in productivity, it may not be worth developers' time to learn. On the other hand, in scenarios when performance is critical, developers may choose the most performant API regardless of programming overhead.

To evaluate HPC communication libraries in terms of productivity and performance, this research investigates a set of distributed kernels containing a variety of communication patterns, a common practice as shown in [10] [11] [12]. Different kernels can stress specific aspects of a system, allowing for more granular performance insights to be understood. The communication patterns found in parallel implementations of various computational kernels vary widely, and can be representative of common patterns found in larger, supercomputer-scale apps. Using a controlled study with optimized implementations for all libraries and communication styles, this work studies a set of kernels for useful insights on an API's pattern-specific performance and overall programmability. The goals of this research are to provide an evaluation of MPI, MPI-RMA, and OpenSHMEM in terms of weak scaling performance as well as programmability. DAXPY, Cannon's Algorithm Matrix Multiplication, SUMMA and Integer Sort kernels are benchmarked using up to 320 nodes on two supercomputing centers, NERSC Cori and PSC Bridges-2, for comprehensive performance insights.

2.0 Background

The libraries explored in this research expose different communication primitives and potential for optimizations. This section first explains one- and two-sided communication, as understanding these core behaviors is crucial to the APIs and optimizations explored in this research. Subsequently, each API will be explained by detailing memory models and communication primitives.

2.1 One- and Two-Sided Communication Models

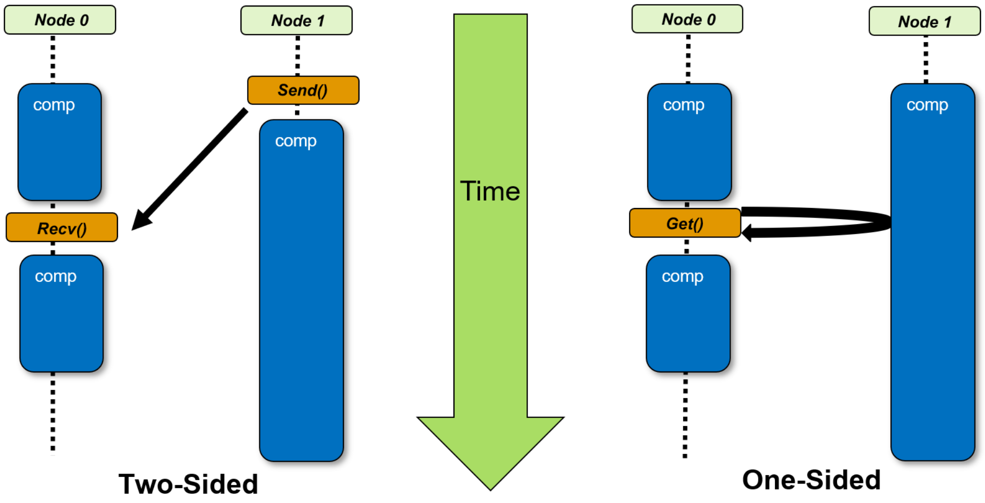

Figure 1: One- and two-sided communication

It is first important to distinguish between the two major communication paradigms explored in this research: one- and two-sided. Two-sided communication, also known as *message passing*, is a cooperative operation between both the receiving and sending processes. This paradigm is realized with the following communication primitives: *send*() and *recv*().

Fig. 1 shows a *send*()/*recv*() pair. Both nodes are involved in this communication as they must call the necessary primitives, both in the written code and at runtime. One-sided communication, commonly referred to as *RMA*, involves only one process, the caller of the communication routine. The primitives for this paradigm are *get*() and *put*(). Fig. 1 shows an example of a *get()* operation. On modern interconnects, this operation leverages RDMA, meaning Node 1's computation will be completely uninterrupted. While this explanation may present one-sided communication as being superior to two-sided routines due to decreased overhead, a common pitfall when using RMA is the need for added synchronization. In Fig. 1, if the data being accessed with the *get*() call is dependent on some condition or computation having occurred, nodes 0 and 1 must explicitly synchronize to ensure that the data being transferred is valid. On the other hand, the *send*()/*recv*() model expresses this elegantly as the pair of communication calls acts as a form of synchronization between processors at runtime.

2.2 MPI

MPI has been the *de facto* standard for HPC communication on distributed systems. As its name implies, the core communication mechanism of MPI is *message passing*. Regardless of if the underlying hardware is a Symmetric Multiprocessor (SMP), the programming model in MPI is that of a *distributed memory architecture* [13]. This programming model's prevalence in MPI stems from its origins in the early 1990s when multiprocessors were not as widely adopted. This model can be thought of as a trade-off; Abstracting away the locality of the data simplifies the programming model, but obfuscates optimizations that can be made for SMPs. Portability has been a driving goal of the specification since it was introduced, and this can be thought of as a key reason for its broad success. Along with point-to-point interfaces, MPI provides an extensive range of collective communication operations such as $MPI\_Bcast()$, $MPI\_Gather()$ and $MPI\_Alltoall()$.

2.3 MPI-RMA

One-sided communication functionality was added to MPI as part of MPI-2 in 1997. Operations that can be performed using two-sided communication can also be performed using one-sided communication. As such, this research treats the RMA extensions to MPI as a separate API, MPI-RMA. The core RMA operations for this API extension are *MPI_Get*() and *MPI_Put*().

Listing 1: Windowed memory management in MPI-RMA.

```
#include <mpi.h>
int main()
{
        // init code
        int* x; // pointer to memory
        MPI_Win winX; // window object

        MPI_Alloc_mem(sizeof(int), MPIINFO_NULL, &x);
        MPI_Win_create(x, sizeof(int), MPIINFO_NULL, &winX);

        // communication/computation code

        MPI_Win_free(&winX);
        MPI_Free_mem(x);

        // finalize code
}
```

Because the calling process must have knowledge of memory addresses on the target process, the API must provide some functionality for distributing addresses. MPI-RMA uses the concept of memory “windows” for distributing transfer parameters at runtime. Window creation is collective among all processors, distributing memory location information. Listing 1 shows sample code of MPI-RMA window set up. First, an MPI_win object must be created. The MPI specification recommends users allocate memory that is being exposed in a window with the *MPI_Alloc_mem*() function [13]. Specially allocated memory can be handled by the runtime library, which can attempt to establish memory at symmetric offsets, simplifying address translation. This allocated memory can then be exposed to all processors with the $MPI\_Win\_create()$ function, which takes the allocated memory and MPI_Win object as parameters. Because one-sided communication removes the implicit synchronization provided by a $send()/recv()$ pair, more explicit synchronization calls must be made in the program such as $MPI\_Win\_fence()$.

2.4 PGAS

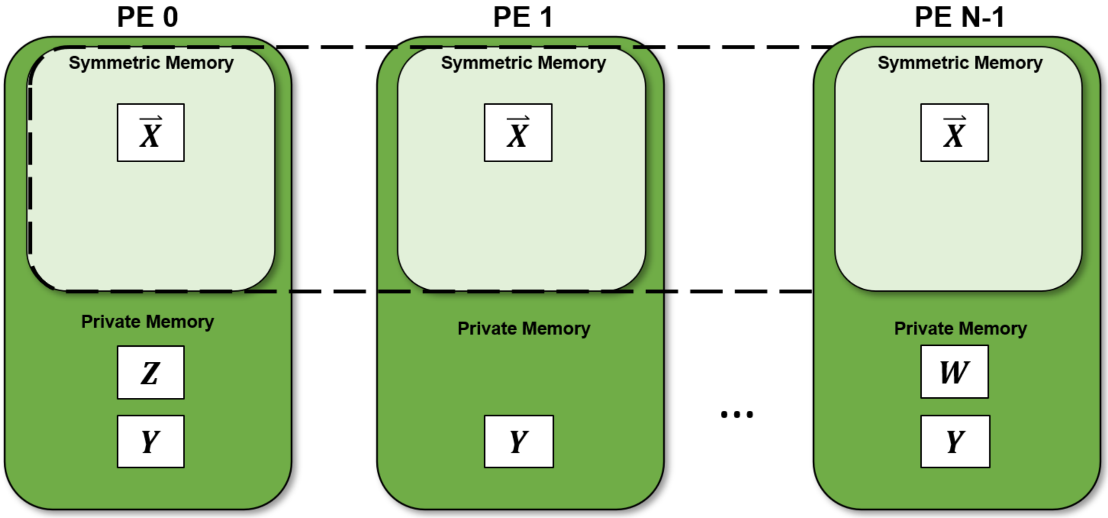

Figure 2: Illustration of PGAS memory model

In the PGAS programming model, a process is denoted as a Processing Element (PE). A PE can be a node in a distributed system or a logical core in an SMP system. This model aims to provide better programmability with the abstraction of logically partitioned data. In this system, data can be logically shared, extending the model of an SMP to a distributed system. Fig. 2 illustrates the PGAS memory model, showing symmetric and private memory regions. Private memory can be allocated in a PE-dependent manner, but symmetric memory must be allocated in a manner that creates symmetric size and offsets for all PEs. Data must still be explicitly communicated between PEs at the app level, therefore behavior is not actually the same as an SMP. As processors have knowledge of one another through the PGAS abstraction, one-sided communication can be exploited.

2.5 OpenSHMEM

PGAS languages such as Unified Parallel C (UPC) and Chapel exist, but OpenSHMEM—similar to MPI—is implemented as a library, currently with bindings for Fortran and C. SHMEM, or "shared memory" has existed since 1993 as a parallel programming model, first beginning as Cray-SHMEM [14] [15] [16]. A growing effort and body of literature around the OpenSHMEM standard shows its potential as an alternative to MPI for some apps due to superior performance on large clusters [17] [11]. While MPI's RMA routines allow for one-sided communication, developers do not benefit from an abstraction such as PGAS, making apps harder to develop and debug. The combination of one-sided routines and a PGAS memory model makes investigating OpenSHMEM worthwhile as there is potential for both productivity and performance gains.

Listing 2: Symmetric memory management in OpenSHMEM.

```
#include <shmem.h>
int main()
{
        // init code
        static int y; // symmetric memory
        int* x = shmem_malloc(sizeof(int));

        // communication/computation code

        shmem_free(x);

        // finalize code
}
```

The abstraction provided by OpenSHMEM is the "symmetric heap." Listing 2 illustrates the basic symmetric memory management seen in an OpenSHMEM program. Memory can be allocated from this region of memory using *shmem_malloc*()/*shmem_free*() similar to traditional heap memory management in C. Memory can also be assigned to the symmetric heap using the *static* keyword for a variable declaration. Communication can only occur to or from a symmetric region because RMA operations require the address of remote memory in the symmetric heap. OpenSHMEM implementations leverage the symmetric memory abstraction for remote address calculation and *shmem_malloc*() routines can be optimized by establishing memory at symmetric offsets, simplifying address translation [16]. The primary one-sided operations are *shmem_get*() and *shmem_put*(). While standard RMA operations in OpenSHMEM and MPI are blocking at the calling process, OpenSHMEM also provides non-blocking RMA operations in *shmem_get_nbi*() and *shmem_put_nbi*(). The API also exposes collective operations similar to those found in MPI. Examples of such operations include *shmem_broadcast*() and *shmem_collect*() which mirror *MPI_Bcast*() and *MPI_Reduce*(), respectively.

3.0 Related Research

Communication libraries are consistently evaluated for their performance on a variety of apps. Often performance evaluations are done within a single API on different interconnects or implementations. For example, Hjelm [18] evaluates the overall performance of MPI-RMA with the OpenMPI library, while Jithin *et al.* [11] characterize OpenSHMEM scalability on InfiniBand systems. These works often use artificial microbenchmarks [19] or computational kernels for performance characterization [10] [11].

There is not significant conclusive evidence indicating a specific communication API's superiority in terms of performance, as various popular architectures, library implementations, and apps would need to be analyzed. Existing research comparing the APIs explored in this research also show inconsistent results. In [10], OpenSHMEM, MPI, and MPI-RMA are tested by creating code implementations of the NAS Parallel Benchmarks. OpenSHMEM performs the worst, primarily due to unoptimized collective operations in the library implementation tested. Conversely, Baker *et al.* [17] evaluate MPI-RMA and OpenSHMEM on the Smith-Waterman algorithm for genome sequence alignment, showcasing that the OpenSHMEM-based app executes over 2× faster than MPI at 128 nodes. This research does not aim to provide conclusive evidence as to which API is the most performant. Rather, it attempts to augment the existing performance research while also providing novel productivity insights.

Productivity is a nebulous and subjective concept, making it challenging to effectively quantify. Chamberlain *et al.* [20] detail the Chapel parallel programming language and discuss it and other parallel tools' programmability, but do not aim to quantify any of these concepts. Other research has aimed to compare MPI and OpenMP, employing qualitative discussion and detailed counts of API-specific lines of code (LOC) [21]. This measurement of LOC is the most common metric for productivity of high-performance tools, middleware, languages and libraries.

Wang *et al.* [22] similarly use LOC as their core productivity metric along with measurements of development time. This research does not measure development time, as it varies significantly from developer to developer. For example, because many developers in the HPC community are familiar with MPI, they will likely be able to write and test MPI-based code more efficiently than OpenSHMEM code. This scenario may result in a large reduction in development times, but would not necessarily accurately reflect a difference in productivity, which aims to capture inherent "ease-of-development." This research aims to use an approach similar to [21], by characterizing APIs using kernel performance while extending their quantitative productivity analysis by breaking code down in a more granular fashion.

4.0 Evaluated Kernels

The distributed kernels used in this research, their communication-computation patterns and the specifics of their various implementations are discussed in this section. Each of the four kernels stress a different communication-computation pattern commonly found in large distributed apps. Table I details the API calls used for each library, breaking them down by their core types: synchronization, communication, allocation, rank-query.

4.1 DAXPY

The DAXPY kernel is a simple Basic Linear Algebra Subprogram (BLAS) kernel, typically found in libraries like Eigen or OpenBLAS [23] [24]. It consists of a *scatter()/gather()* pattern with intermediate local computation, which is a common idiom in distributed programs. The kernel is based on one-dimensional arrays representing vectors and is given by Eq. 4.1. The data originates on the MASTER process (i.e. PE of rank 0), and is distributed to all other processes using a *scatter()* operation. Local portions of the vector are computed using Eq. 4.1 after which the vector is reassembled on the MASTER process with a *gather()* operation.

$$\vec{y} = \alpha \cdot \vec{x} + \vec{y} \tag{4.1}$$

The implementations investigated are:

- MPI
- MPI-RMA
- OpenSHMEM synchronous (sync)
- OpenSHMEM asynchronous (async)

The MPI implementation uses $MPI\_Scatter()$ and $MPI\_Gather()$ to distribute and reassemble the vector. The MPI-RMA implementation uses $MPI\_Get()$ and $MPI\_Accumulate()$.

The OpenSHMEM synchronous implementation uses *shmem_get*() and *shmem_put*(). The OpenSHMEM asynchronous implementation is an additional optimized version of the kernel using the OpenSHMEM API, and uses *shmem_get_nbi*(), *shmem_put*() and the *shmem_quiet*() synchronization call. This implementation attempts to use non-blocking RMA operations to compute the local vectors in a pipelined manner. The specific optimization is unique to the DAXPY kernel and is not explored for the other kernels.

4.2 Cannon's Algorithm Matrix Multiplication

The Cannon's algorithm Matrix Multiplication (Cannon's MM) kernel is a dense matrix multiplication algorithm specifically for $\sqrt{N} \times \sqrt{N}$ meshes of nodes or processors. The original algorithm given in [25] assigns individual matrix elements to each process and therefore is severely limited in the size of matrices that can be computed. However, the algorithm can be expanded to assign submatrices to each process as shown by Algorithm 1.

The core pattern found in this kernel is structured peer-to-peer communication, which is extremely common in simulation workloads which use 2D decompositions on virtual grids [2] [3] [11]. The algorithm computes the resultant matrix using submatrices of size $M/\sqrt{N} \times M/\sqrt{N}$. As seen in Algorithm 1, the submatrices are first distributed among the PEs on the 2D mesh with an initial skew. Subsequently, the submatrices of the resultant matrix, C, are computed as the multiplicand submatrices, A and B, are shifted between the PEs along the grid.

Algorithm 1: Cannon's Matrix Multiplication for M x M matrices with N PEs

```
Data: Two input M x M Matrices, A and B
Result: M x M Resultant Matrix, C
Map each processor of rank [0, N-1] to a 2D virtual address tuple (x,y):
$x := rank \mod \sqrt{N}$;
$y := rank/\sqrt{N}$;
Initialization: Skew Matrices
for $x \leftarrow 0$ to N-1 do
    Left circular shift submatrix B(x,y) by x, so it is assigned submatrix A(x, (y+x) mod N);
end
for $y \leftarrow 0$ to N-1 do
    Upward circular shift submatrix B(x,y) by y, so it is assigned submatrix B((y+x) mod N, y);
end
Computation & Communication
for $k \leftarrow 0$ to N do
    for $i \leftarrow 0$ to M - 1 do
        for $j \leftarrow 0$ to M - 1 do
            C[i,j] = C[i,j] + A[i,j] * B[i,i];
        end
    end
    Left circular shift each row of A by 1, so submatrix A(x,y) is assigned submatrix A(x, (y+1) mod N);
    Upward circular shift each column of B by 1, so submatrix B(x,y) is assigned submatrix B((x+1) mod N, y);
end
```

The implementations investigated are:

- MPI
- MPI-RMA
- OpenSHMEM

The MPI implementation uses *MPI_send_recv_replace*() to shift the submatrices between PEs, and uses *MPI_Scatter*() and *MPI_Gather*() to distribute and collect the initial and final matrices, respectively. The MPI-RMA implementation uses *MPI_Put*() to shift the submatrices between PEs, and uses *MPI_Scatter*() and *MPI_Gather*() to distribute and collect the initial and final matrices, respectively. The OpenSHMEM implementation uses *shmem_put*() to shift the submatrices between PEs, and uses custom RMA-based *scatter*() and *collect*() operations to distribute and collect the initial and final matrices, respectively. The RMA-based implementations (MPI-RMA and OpenSHMEM) use alternating buffers to hold submatrices of A and B, allowing matrices to be passed to the next PE without overwriting the currently used one. The MPI implementation avoids this hazard altogether due to its usage of *MPI_send_recv_replace*(), which guarantees that a matrix will not be overwritten at the potential penalty of added idle time.

4.3 SUMMA

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) kernel is a dense matrix multiplication algorithm for arbitrarily sized 2D meshes of nodes or processors. Similar to Cannon's MM, SUMMA uses a shift-based algorithm on the 2D grid, but instead of peer-to-peer shifts, matrix rows and columns are partially broadcasted [26]. This kernel therefore simulates a partial broadcast and reduce communication pattern using a 2D grid processor decomposition. SUMMA and its communication pattern are chosen because they simulate a common pattern found in apps like ray tracing and molecular dynamics [27] [2]. The algorithm is also chosen because partial broadcasts are not natively supported in current OpenSHMEM libraries, potentially requiring significant effort to implement which may demonstrate the maturity of the MPI API. For more information regarding this algorithm, the reader is referred to [26].

The implementations investigated are:

- MPI
- MPI-RMA
- OpenSHMEM

The MPI-based versions of the code take advantage of MPI's native support for creating custom ”communicator” groups, whereby the programmer is able to create groups of processors to perform collective operations on. The default behavior for collective operations in MPI is that they operate on all processors using the default communicator, MPI_COMM_WORLD. The $MPI\_Comm\_split()$ API call is used to divide communicators for both MPI version of the kernel, while a custom group-based broadcast solution needed to be written for the OpenSHMEM version. The MPI and OpenSHMEM implementations use their respective $scatter()$ and $gather()$ to distribute and collect the initial and final matrices. For communication of the row and column submatrices, the MPI-based codes use the $MPI\_Bcast()$ call, while the OpenSHMEM version uses a custom $broadcast()$ solution using RMA $put()$ calls.

4.4 Integer Sort

The integer sort kernel is based off of the Integer Sort (IS) kernel found in the NAS Parallel Benchmarks [28]. The goal is to sort N keys in parallel, using a pseudorandom number generator to generate the keys within a specified range to ensure the benchmark is repeatable. This work creates implementations based on the existing work of [29] and [30]. The sorting method used is a distributed bucket sort. Each PE receives a segment of the unsorted integers and places its given elements into buckets. Using an *all-to-all* pattern, the buckets are distributed to the proper PE which will then perform a local sort. For further information on the kernel, the reader is referred to [28].

The implementations investigated are:

- MPI
- MPI-RMA
- OpenSHMEM

The MPI implementation uses $MPI\_All\_to\_All()$ and $MPI\_All\_to\_Allv()$ to exchange bucket sizes and buckets, respectively. Both RMA-based implementations use a loop structure with $put()$ operations to realize an all-to-all operation. The execution stages of all kernel implementations are separated by their respective $barrier()$ API calls. This structure ensures that the IS kernel primarily compares the library-based all-to-all API call of MPI to RMA-based all-to-all operations created using MPI-RMA and OpenSHMEM.

Table 1: API Calls Used in Kernel Implementations

	Synchronization	**Communication**	**Allocation**	**Rank-Query**
MPI	*MPI_Barrier()*	*MPI_Scatter()*, *MPI_Gather()*, *MPI_send_recv_replace()*, *MPI_All_to_all()*, *MPI_All_to_allv()*		*MPI_Comm_size()*, *MPI_Comm_rank()*, *MPI_Cart_create()*, *MPI_Cart_shift()*, *MPI_Comm_split()*
MPI-RMA	*MPI_Barrier()*, *MPI_Win_fence()*	*MPI_Put()*, *MPI_Get()*, *MPI_Accumulate()*	*MPI_Alloc_mem()*, *MPI_Win_create*, *MPI_Free_mem()*, *MPI_Win_free()*	*MPI_Comm_size()*, *MPI_Comm_rank()*, *MPI_Cart_create()*, *MPI_Cart_shift()*
OpenSHMEM	*shmem_barrier_all()*, *shmem_quiet()*	*shmem_put()*, *shmem_put_nbi()*, *shmem_get()*, *shmem_collect()*, *shmem_broadcast()*	*shmem_malloc()*, *shmem_free()*	*shmem_n_pes()*, *shmem_my_pe()*

5.0 Experimentation

In this section, the experimental setup is discussed. First, the two supercomputing centers used to evaluate kernel performance are detailed. Next, the specific metrics and methodologies for data collection are discussed. This work investigates performance on two different HPC systems that use differing compute hardware, interconnects and software libraries. The goal of using multiple systems is to ensure that measured performance insights can generalize to many platforms. Performance trends observed on a single system may only reflect a specific hardware configuration or library implementation like the Cray libraries of the OpenMPI suite.

Cray-MPI and OpenSHMEMX use different underlying low-level APIs for their RMA with Cray-MPI MPICH [31] using Generic Network Interface (uGNI) and OpenSHMEMX using DMAPP [8], which can potentially explain performance discrepancies between the two RMA APIs.

5.1 Testbeds

The first testbed is the National Energy Research Computing Center (NERSC) Cori supercomputer system. This system consists of 2,388 dual-socket nodes with 32-core, 2.3 GHz Intel Xeon E5-2698 v3 processors and 128 GB of DDR3 memory [32]. The nodes are connected by the Cray Aries interconnect with a dragonfly network topology with native support for RDMA [33]. The default toolchain for this system, which uses the Intel 19.0.3.199 compiler, is used for all compilation with -O3 flags. The MPI implementation used is Cray-MPICH 7.1.1.0, which is optimized for performance on the given hardware [31]. The OpenSHMEM implementation used for this research is Cray-OpenSHMEMX 9.1.0, which again is optimized for the platform and uses the DMAPP library for its underlying RMA operations [34].

The second testbed is the Pittsburgh Supercomputing Center (PSC) Bridges-2 supercomputer system. This system consists of 488 dual-socket nodes with 64-core, 3.40 GHz AMD EPYC 7742 processors and 256 GB of DDR4 memory. The nodes are connected using HDR-200 InfiniBand interconnects with a fat tree network topology with native support for RDMA and SHARP [35]. The MPI and OpenSHMEM implementation used on this system are part of the OpenMPI 4.0.5 library package. The compiler toolchain used on this system is GCC 10.0.5 with -O3 flags enabled. OpenMPI is specifically configured for InfiniBand systems and is able to use Mellanox SHARP to accelerate collective operations such as *reduce*() and *broadcast*(). SHARP is enabled for all tests on PSC.

A notable difference between the communication libraries on the NERSC tesbed and those on the PSC testbed is that the NERSC libraries are completely separate from one another in terms of their underlying transport mechanisms. On NERSC, Cray-MPICH uses Generic Network Interface and OpenSHMEMX uses DMAPP for their underlying communication. On the other hand, on PSC, both SHMEM and MPI functionality fall under OpenMPI and use Unified Communication X (UCX) for their underlying transport handling [31] [34] [36].

5.2 Approach

For each test, five "burn in" iterations were performed, then the latency was averaged over 50 iterations. This setup was used for all node and problem size configurations shown in the Results section. This research measures weak scalability for each kernel. When measuring weak scaling, the total problem size is scaled correspondingly with the number of processes. This metric is chosen because it shows how well an app can exploit the increasing resources at hand, making it valuable when inferring performance for large-scale apps found on exascale systems. Tests are performed with up to 320 nodes on both testbed systems.

Throughput of each kernel is calculated using Eq. 5.1. This metric encapsulates the kernel and its API-specific implementation's ability to concurrently process and communicate large amounts of data. Throughput can be thought of as another view of a kernel's scaling ability, as it also encapsulates latency. As with weak scaling latency, data for this metric is collected for up to 320 nodes on both systems.

$$Throughput \quad (Bytes/s) = \frac{Total \quad Data \quad Transferred \quad (Bytes)}{Latency \quad (s)} \tag{5.1}$$

When measuring LOC for productivity, comments, blank lines, and debug print statements are ignored. MPI and OpenSHMEM share similar *init()* and *finalize()* library calls for runtime startup and cleanup, respectively. These calls occur in identical locations within the code and have identical functionality. As the calls are identical between libraries and do not fit within the categories of *synchronization*, *communication*, or *allocation*, they are similarly ignored for API call counting. Library calls to query the number of active processors or current rank are also counted in the *rank-query* category, as other querying operations such as $MPI\_Cart\_shift()$ are only used in the matrix multiplication kernels.

6.0 Results

The main performance metrics for this research are weak scaling in terms of latency and throughput. Fig. 3, 8, 13 and 18 show latency weak scaling for the four kernels from 1 to 320 nodes on the NERSC Cori system. Fig. 5, 10, 15 and 20 show latency weak scaling for the four kernels from 1 to 320 nodes on the PSC Bridges-2 system. Fig. 4, 9, 14 and 19 show throughput weak scaling for the four kernels from 1 to 320 nodes on the NERSC Cori system. Fig. 6 11, 16 and 21 show throughput weak scaling for the four kernels from 1 to 320 nodes on the PSC Bridges-2 system.

Productivity is measured in terms of LOC and API calls broken down into four categories: synchronization, communication, allocation, and rank-query. Table II summarizes the productivity results with total LOC and total API calls for each kernel. Fig. 7, 12, 17 and 22 break down the API calls for the DAXPY, Cannon's Algorithm Matrix Multiplication, and Integer Sort kernels, respectively.

6.1 DAXPY

The DAXPY kernel is evaluated with a per-PE vector size of 1,000,000 double-precision values (8 MB). Two vectors, $\vec{x}$ and $\vec{y}$, must be distributed to each processor with a result vector being returned, making the total message size for each PE 24 MB. Both OpenSHMEM-based kernels show the most stable scaling up to 320 nodes as can be seen in Fig. 3 and 5 on NERSC and PSC respectively.

Between 1 and 64 nodes, the OpenSHMEM async implementation remains over 15% faster than MPI-RMA and over 30% faster than MPI in terms of runtime. At 320 nodes, the OpenSHMEM synchronous implementation has over 1%, 13% and 21% lower latency than the OpenSHMEM asynchronous, MPI-RMA, and MPI implementations, respectively. This region is also where the kernel achieves its highest throughput as can be seen in Fig 4 and Fig. 6 where the SHMEM-based code is able to achieve up to 46% and 105% higher throughput than the MPI-based solution on NERSC and PSC, respectively.

Fig. 7 breaks down the API calls used by type. The MPI version of the code is the simplest in terms of LOC and API calls, with all other implementations using over 2× more API calls to realize the kernel. The MPI version uses no *allocation* library calls, while the OpenSHMEM and MPI-RMA versions use 6 and 10, respectively.

Table 2: Summary of Productivity Metrics for Evaluated Kernels

Kernel	**Version**	**Lines of Code**	**Total API Calls**
DAXPY	MPI	259	7
	MPI-RMA	282	22
	OpenSHMEM sync	262	12
	OpenSHMEM async	269	14
Cannon's MatMult	MPI	405	12
	MPI-RMA	453	40
	OpenSHMEM	559	28
SUMMA	MPI	353	15
	MPI-RMA	392	44
	OpenSHMEM	602	28
Integer Sort	MPI	1045	8
	MPI-RMA	1215	16
	OpenSHMEM	1029	11

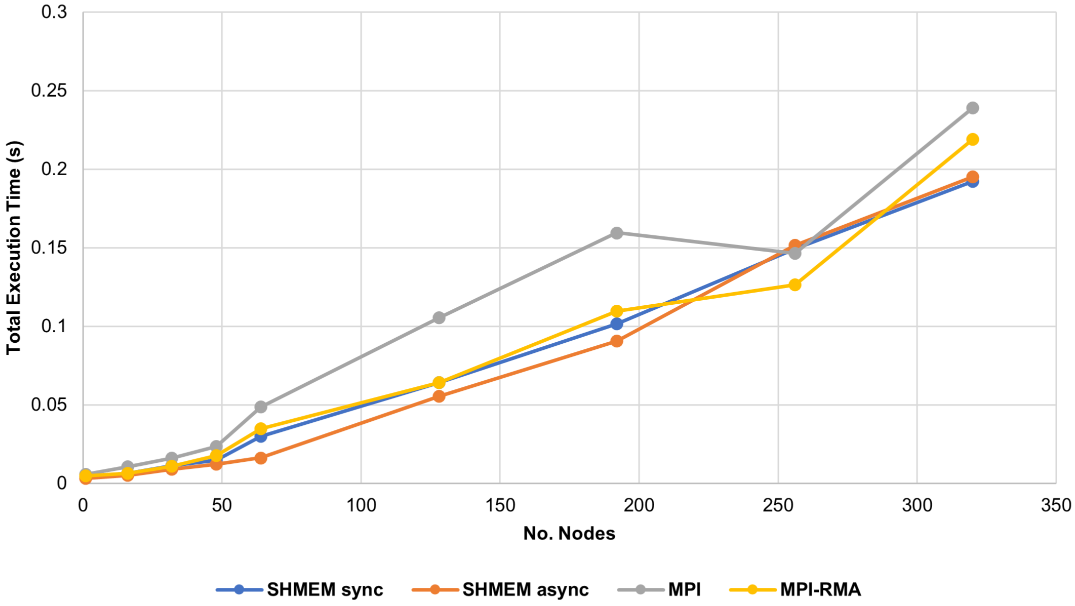

Figure 3: DAXPY kernel latency weak scaling from 1 to 320 nodes on NERSC.

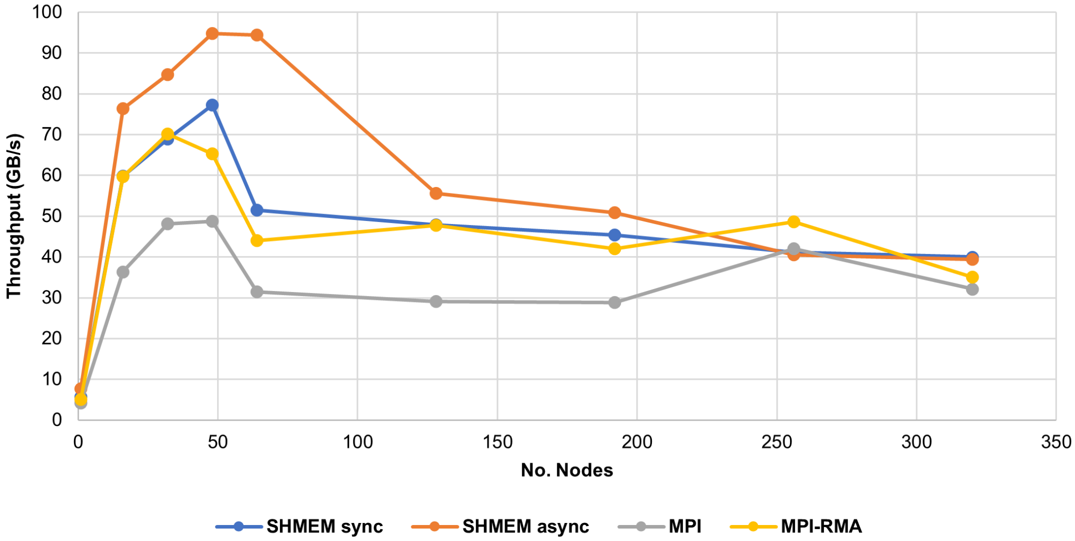

Figure 4: DAXPY throughtput weak scaling from 1 to 320 nodes on NERSC.

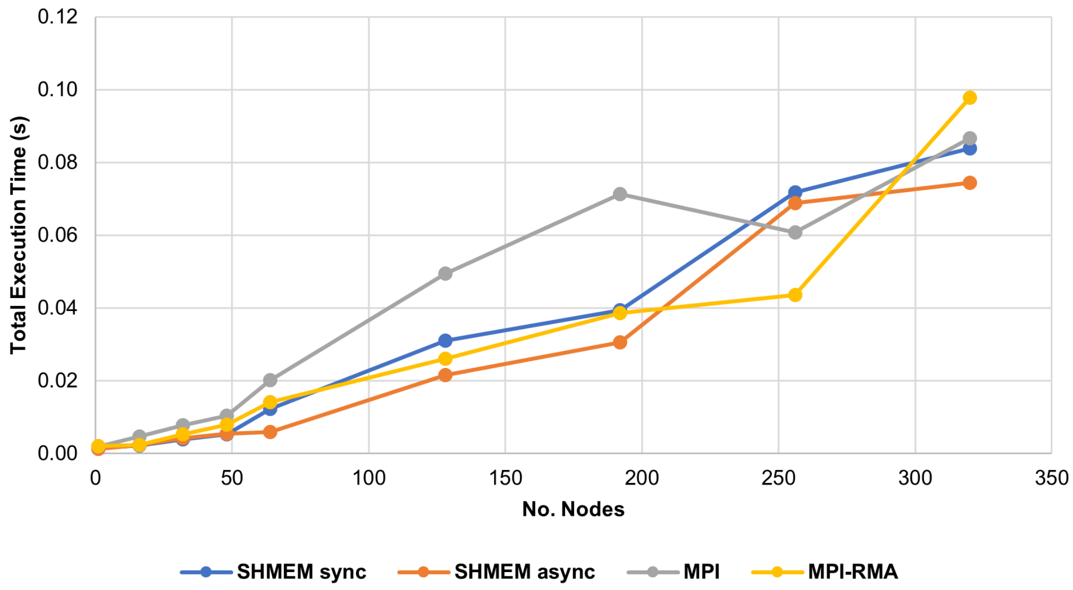

Figure 5: DAXPY kernel latency weak scaling from 1 to 320 nodes on PSC.

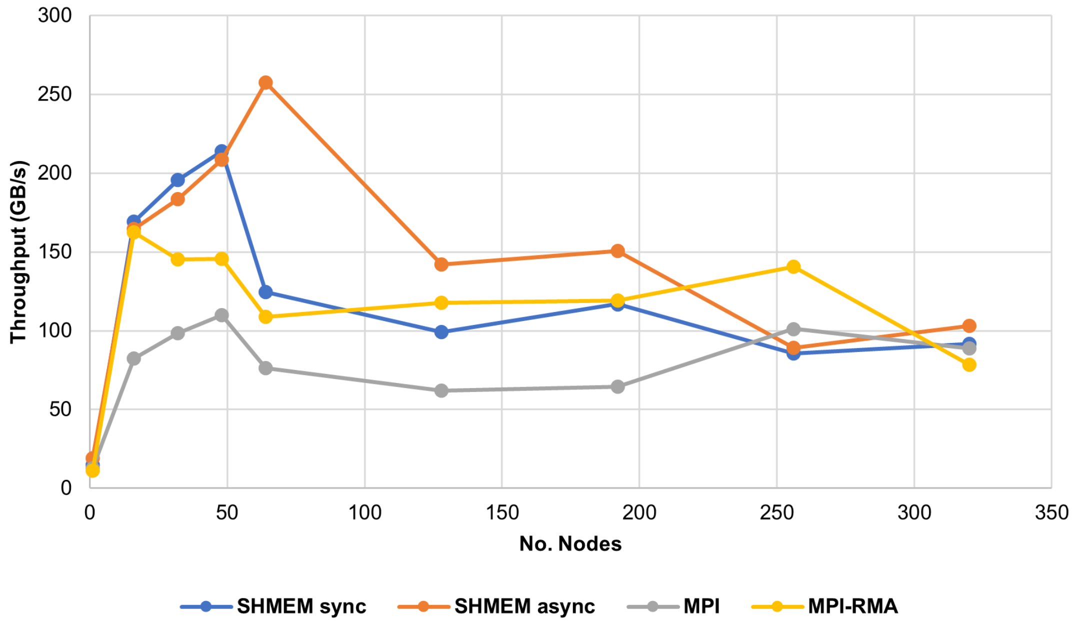

Figure 6: DAXPY throughtput weak scaling from 1 to 320 nodes on PSC.

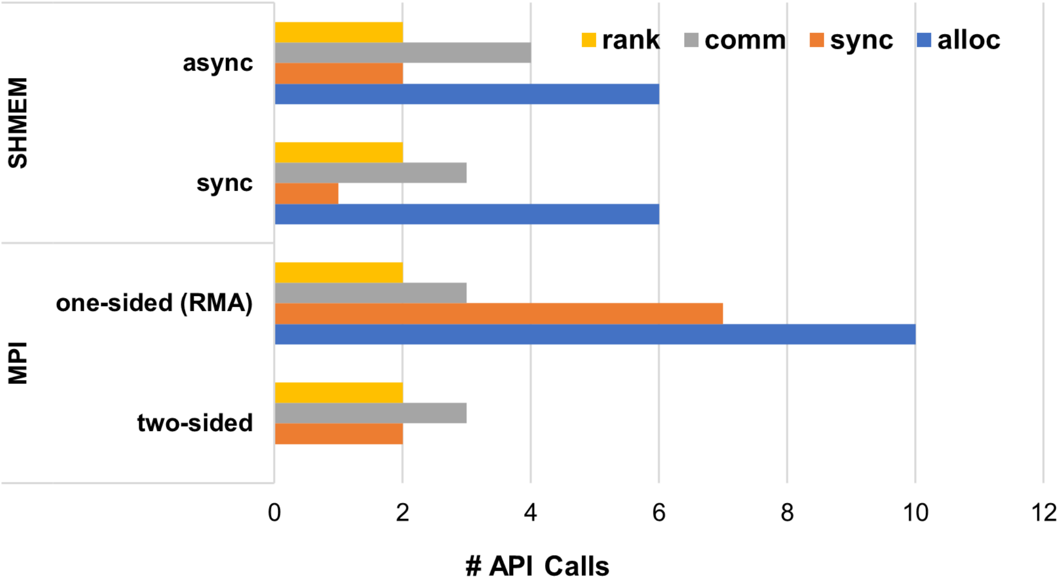

Figure 7: Breakdown of API calls for the various DAXPY kernel implementations explored

6.2 Cannon's Algorithm Matrix Multiplication

The Cannon's Algorithm Matrix Multiplication kernel is evaluated with a per-PE submatrix size of 160,000 double-precision values (1.28 MB). The OpenSHMEM-based kernel has the lowest latency for nearly all node configurations up to 289 PEs on both systems At 289 nodes on NERSC, the OpenSHMEM-based kernel has over 8% lower latency than the MPI-RMA-based kernel and over 13% lower latency than the MPI-based kernel. At 289 nodes on PSC, the MPI-RMA kernel is the fastest with 38% lower latency than the MPI-based kernel and 14% lower latency than the OpenSHMEM-based kernel. In terms of throughput, implementations on both systems remain within 25% of each other for all node configurations.

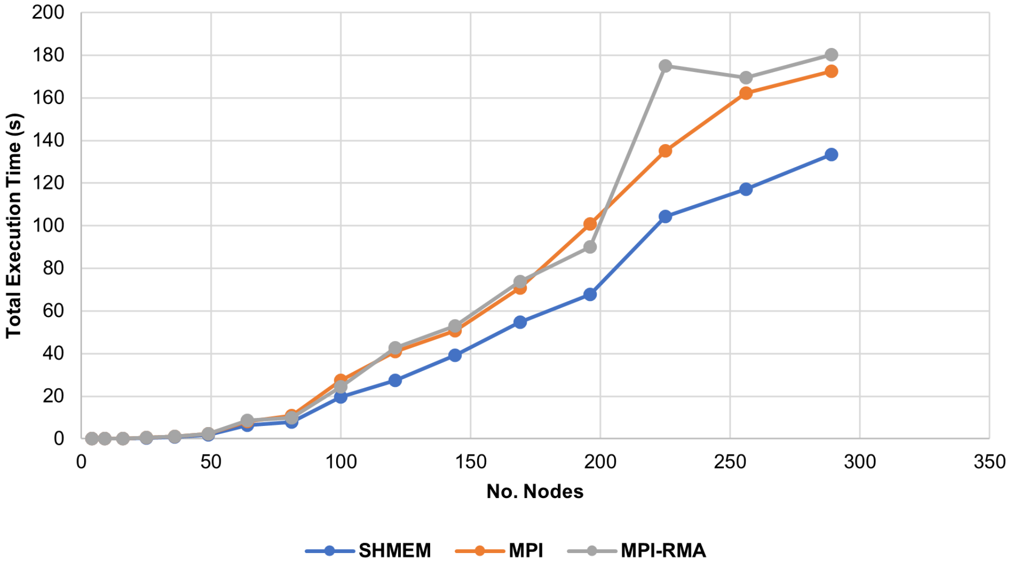

Figure 8: Cannon's MM latency weak scaling from 1 to 289 nodes on NERSC.

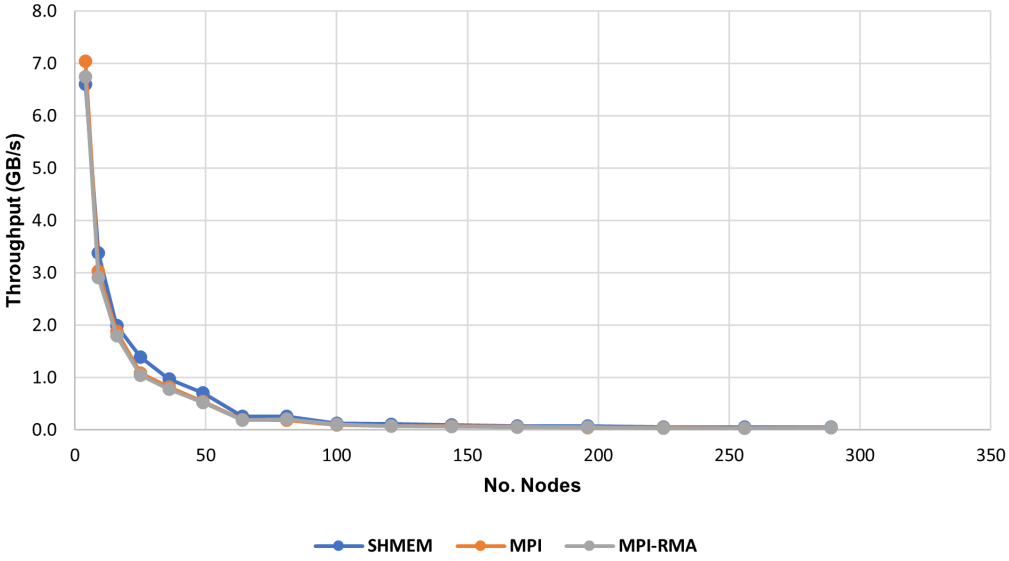

Figure 9: Cannon's MM throughtput weak scaling from 1 to 289 nodes on NERSC.

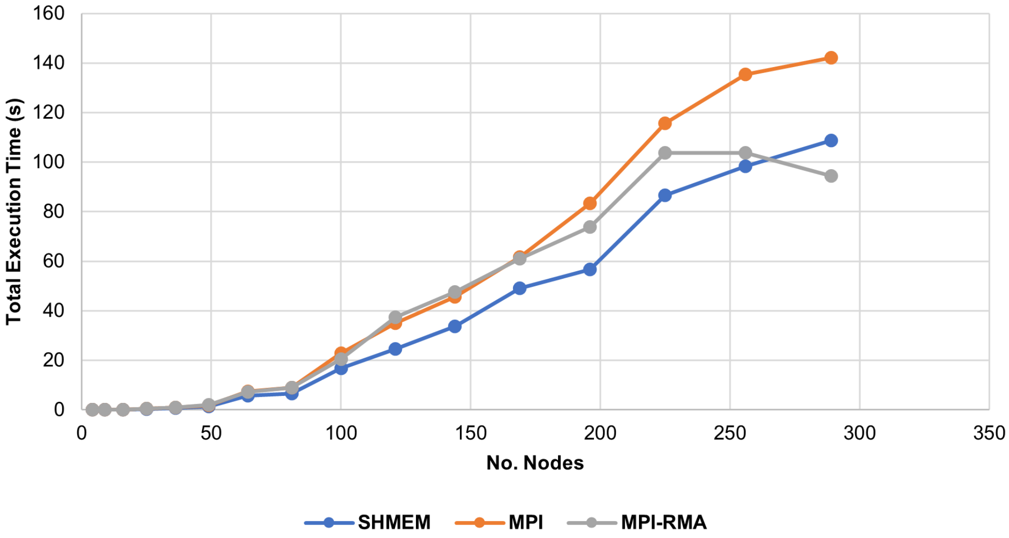

Figure 10: Cannon's MM latency weak scaling from 1 to 289 nodes on PSC.

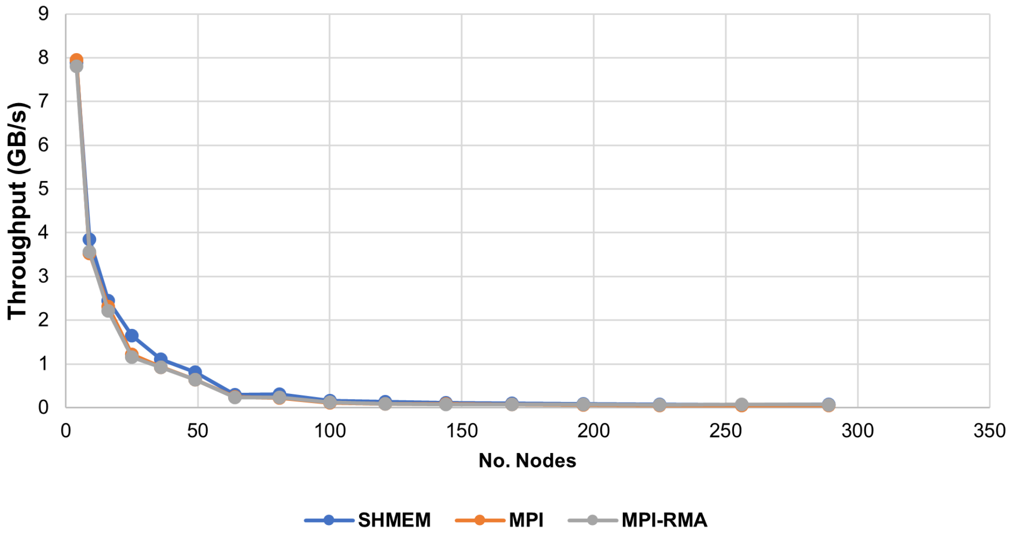

Figure 11: Cannon's MM throughtput weak scaling from 1 to 289 nodes on PSC.

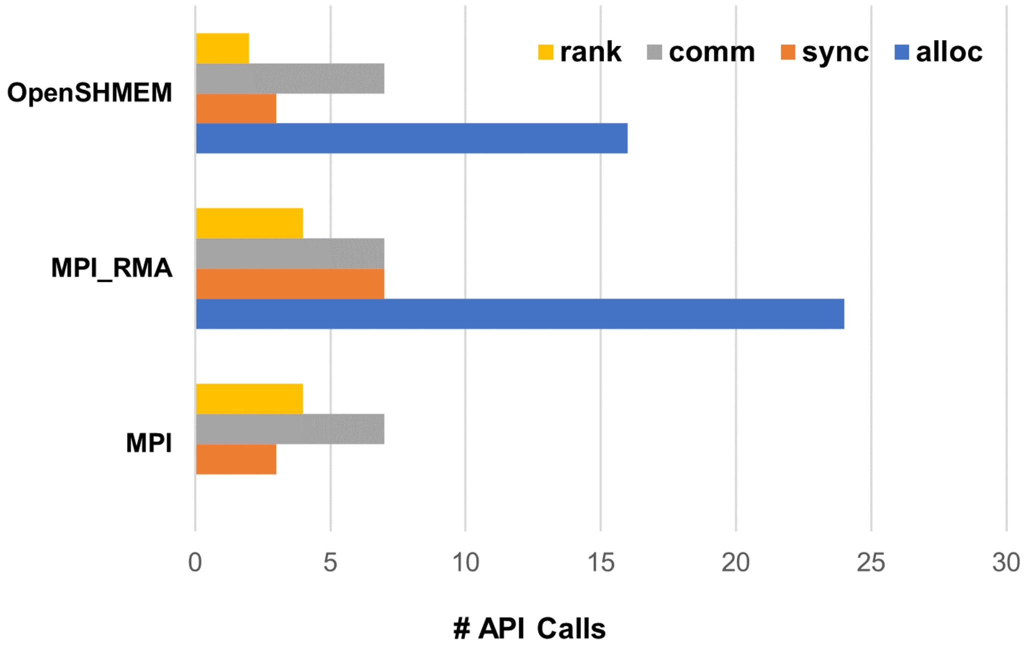

Figure 12: Breakdown of API calls for the various Cannon's MM kernel implementations explored

The API breakdown for the Cannon's MM kernel can be seen in Fig. 12. The OpenSHMEM-based kernel uses over 150 more lines of code than the MPI-based kernel, primarily due to additional utility code required to translate 1D PE addressing to a 2D virtual grid. This functionality is currently directly supported by utility functions in MPI, while it is not in OpenSHMEM 1.4. The two additional *rank*-type API calls found in the MPI- and MPI-RMA-based kernels account for these utility functions, effectively saving over 150 lines of code for this kernel.

6.3 SUMMA

The SUMMA kernel is evaluated with a per-PE submatrix size of 160,000 double-precision values (1.28 MB) to match the Cannon's MM kernel. The OpenSHMEM-based kernel has the highest latency for all node configurations on NERSC but the lowest latency for all node configurations on PSC. The OpenSHMEM-based kernel uses over 235 more lines of code than the MPI-based kernel from additional utility code required to translate 1D PE addressing to a 2D virtual grid and to execute group-based collective operations using RMA primitives.

Fig. 17 breaks down the API calls used by the SUMMA kernel. Similar to results found for the Cannon's MM kernel, 2D virtual PE addressing is currently directly supported by utility functions in MPI, while it is not in OpenSHMEM 1.4. The three additional *rank*-type API calls found in the MPI- and MPI-RMA-based kernels account for these utility functions. Additionally, this kernel's partial broadcast needed to be implemented by hand for the OpenSHMEM version of the code. These two shortcomings of the OpenSHMEM API contributed to a 70% increase in LOC over MPI, but led to superior performance on PSC.

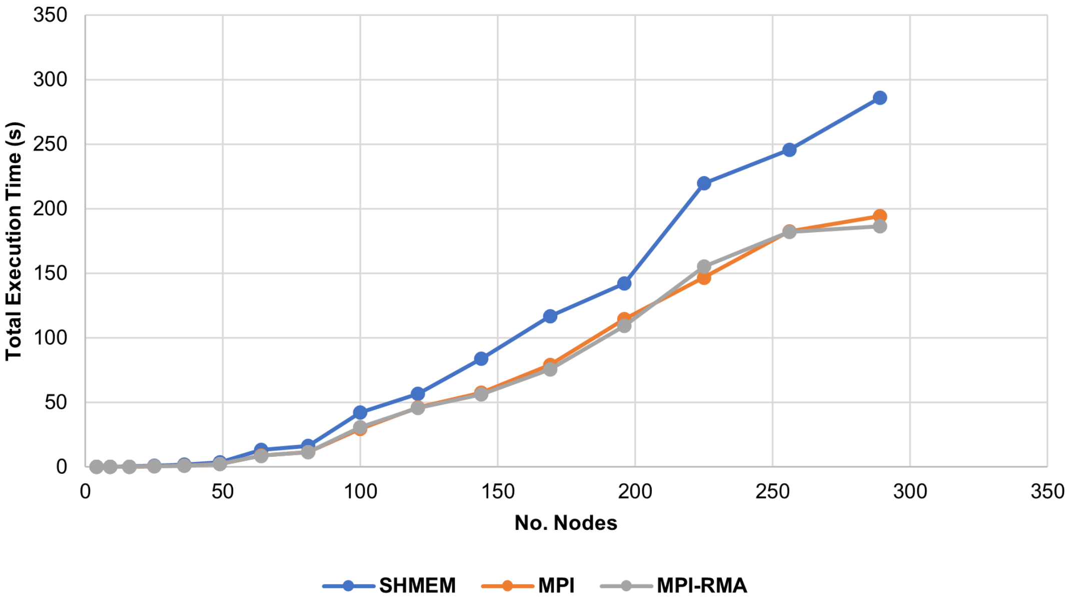

Figure 13: SUMMA latency weak scaling from 1 to 289 nodes on NERSC.

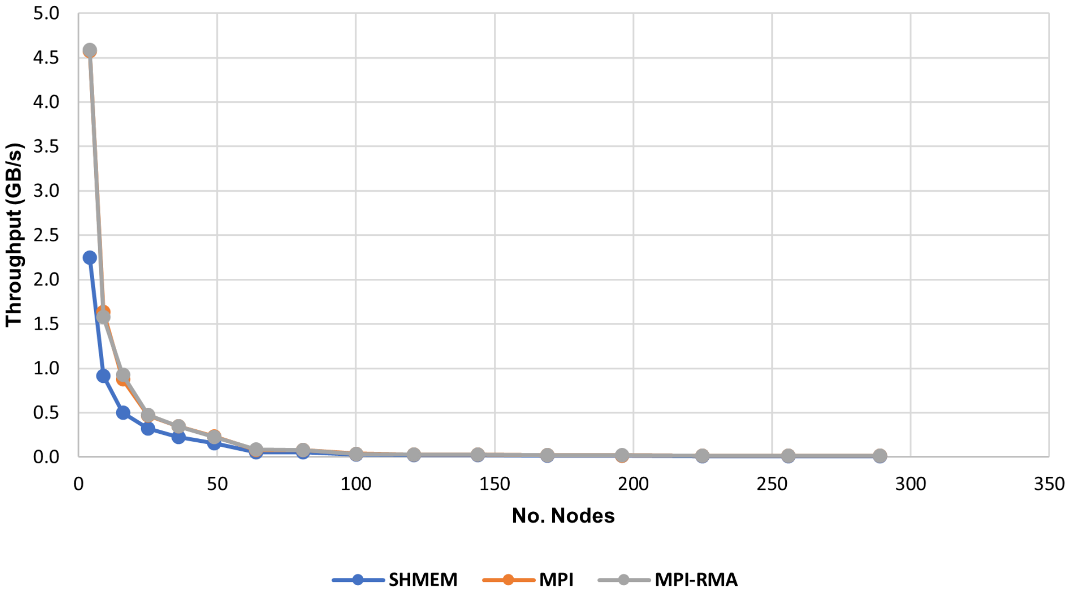

Figure 14: SUMMA throughtput weak scaling from 1 to 289 nodes on NERSC.

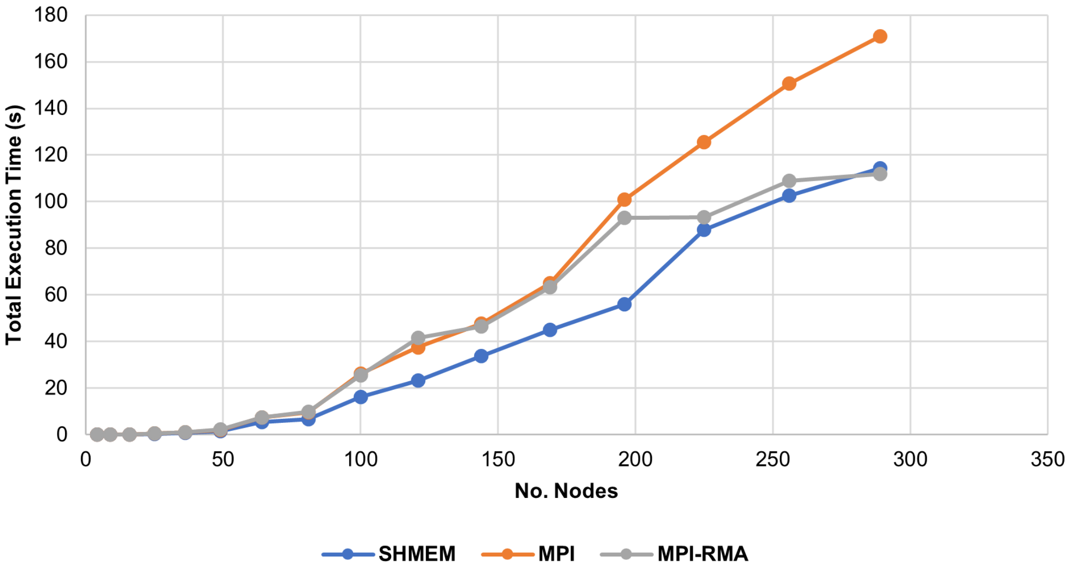

Figure 15: SUMMA latency weak scaling from 1 to 289 nodes on PSC.

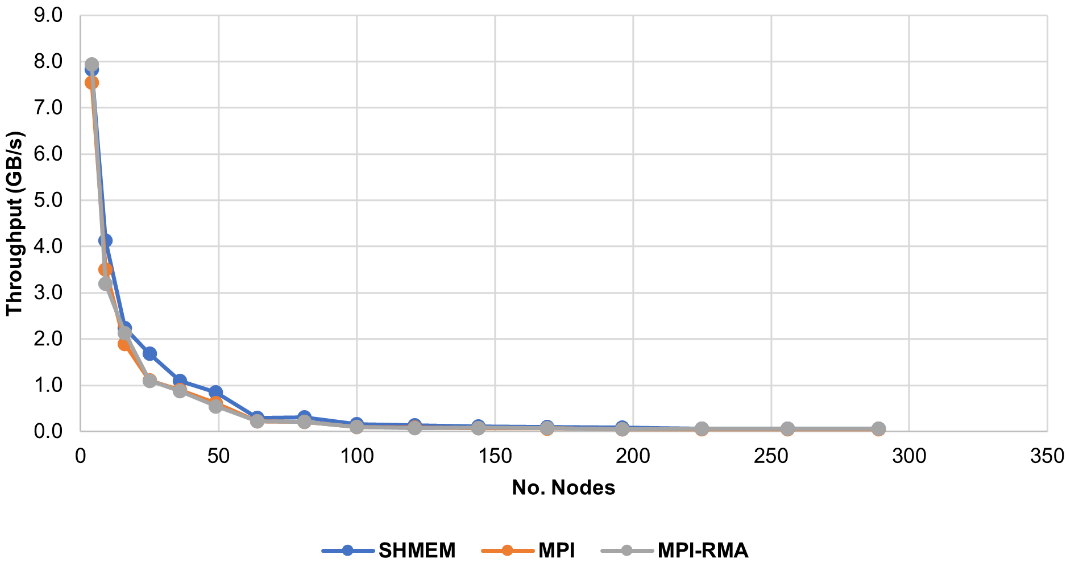

Figure 16: SUMMA throughtput weak scaling from 1 to 289 nodes on PSC.

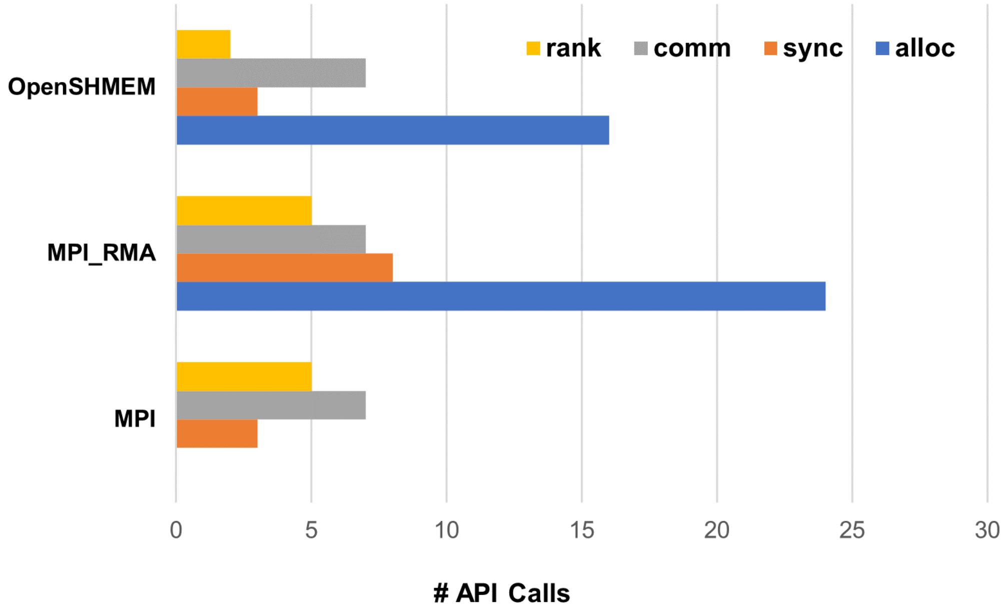

Figure 17: Breakdown of API calls for the various SUMMA kernel implementations explored

6.4 Integer Sort

The Integer Sort kernel is evaluated with a per-PE key count of 2^{28} 32-bit integer values. The keys are initialized on each PE and only bucket information is communicated between PEs with a variable message size based on the random distribution of values. This kernel shows the smallest performance differences between APIs, as they are all within 5% of each other in terms of execution time and throughput for both NERSC and PSC.

Table II summarizes the productivity metrics for the Integer Sort kernel while Fig. 22 breaks down the API calls used by type. The OpenSHMEM version uses the least LOC with 1029. The MPI version uses the least API calls with 8. The additional *allocation* calls used by the MPI-RMA implementation come from RMA *window* object creation and freeing, which does not need to occur in MPI or OpenSHMEM.

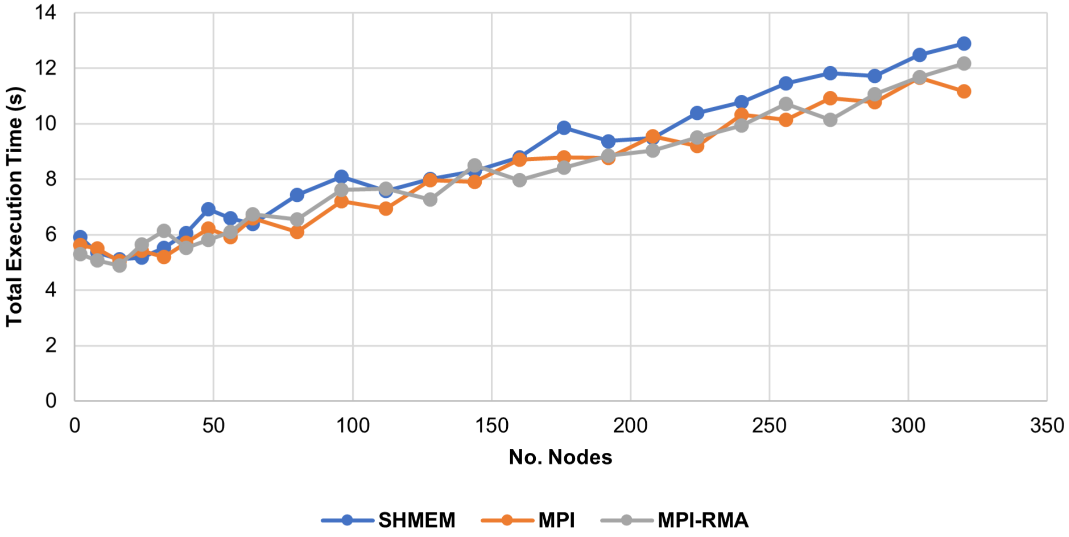

Figure 18: Integer Sort latency weak scaling from 1 to 320 nodes on NERSC.

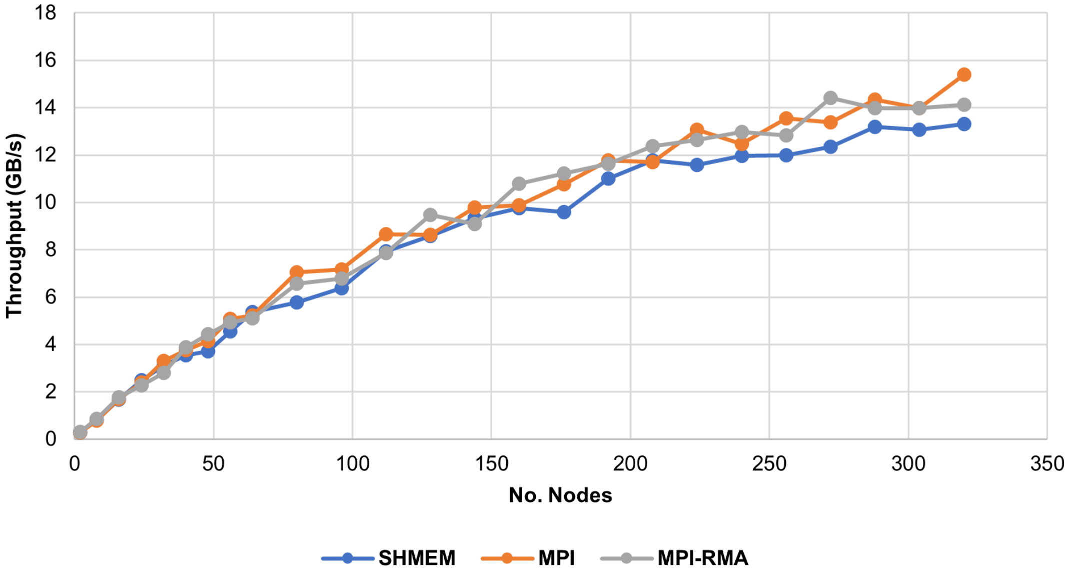

Figure 19: Integer Sort throughput weak scaling from 1 to 320 nodes on NERSC.

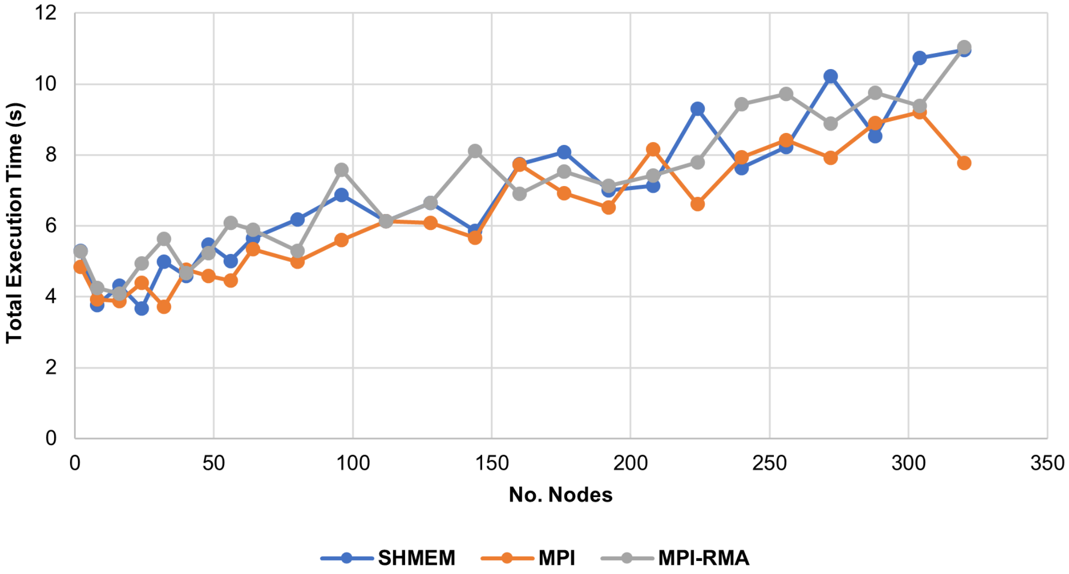

Figure 20: Integer Sort latency weak scaling from 1 to 320 nodes on PSC.

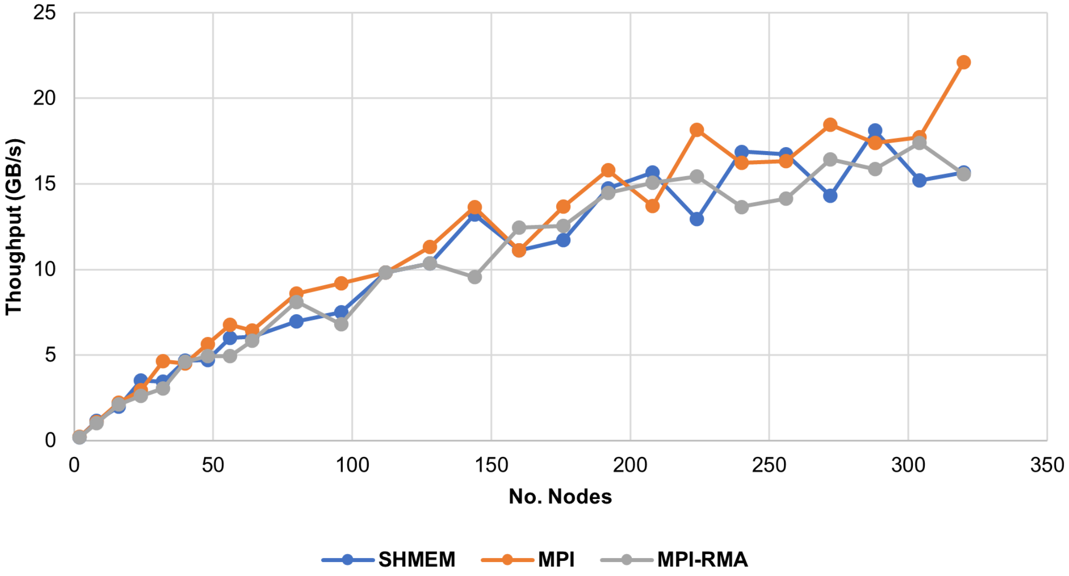

Figure 21: Integer Sort throughput weak scaling from 1 to 320 nodes on PSC.

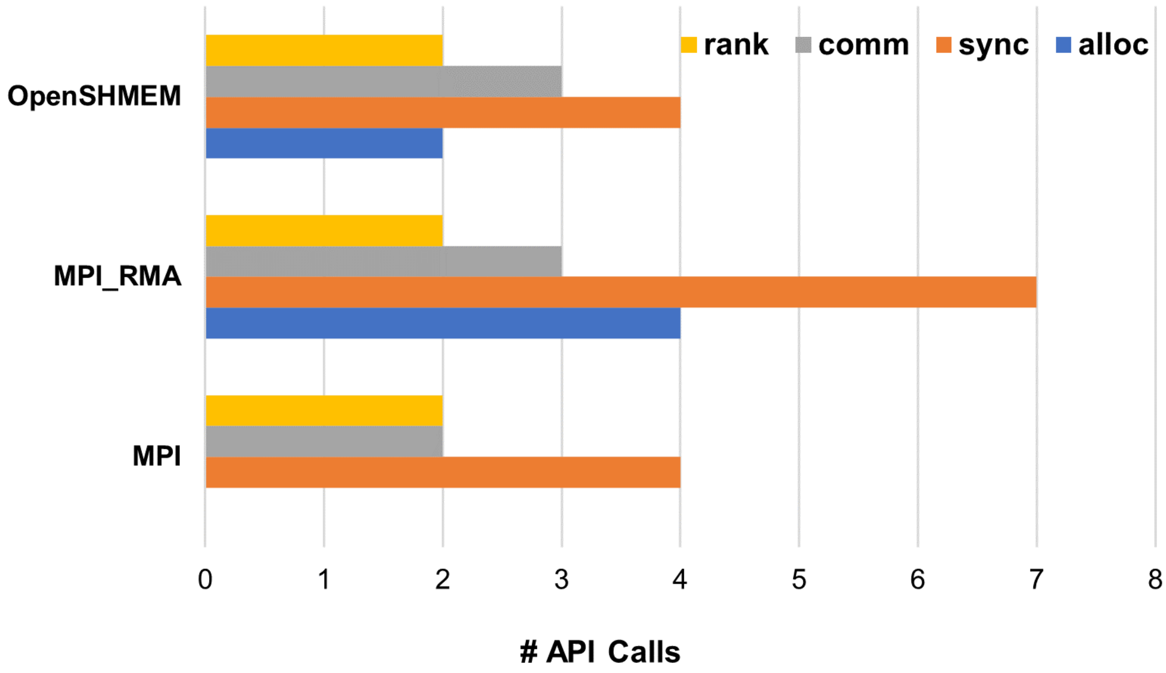

Figure 22: Breakdown of API calls for the various Integer Sort implementations explored

7.0 Discussion

The performance of the kernels depends on an implementation's level of optimization, its ability to take advantage of modern interconnect features such as RDMA, and its API's flexibility to allow for novel optimizations. This section analyzes why these factors are reflected in the results seen in this research across the set of kernels and their respective communication/computation patterns. Additionally, the differences between performance on the two testbed systems is analyzed. The productivity metrics investigated indirectly measure each API's "ease-of-use" and can provide the reader some indication of the work required in distributing an app on an HPC platform.

7.1 Performance

In terms of weak scaling, the RMA-based versions (MPI-RMA or OpenSHMEM) performed the best with OpenSHMEM-based kernels scaling the best for the kernels that primarily leverage point-to-point communication (DAXPY and Cannon's MM). This result is expected, as when properly utilized, one-sided protocols are able to leverage RDMA, enabling faster overall communication due to less overhead by the library runtime. Performance for RMA-based APIs on the SUMMA and Integer Sort kernels is less conclusive. For SUMMA, OpenSHMEM scales the worst on NERSC, while on PSC OpenSHMEM and MPI-RMA scale the best. For the Integer Sort kernel, all API versions behave very similarly in terms of throughput and latency.

The results for the DAXPY kernel show the largest scalability difference below 100 nodes, with the OpenSHMEM asynchronous version scaling the best. This scalability can most notably be seen in Fig. 4 and 6 where throughput of the OpenSHMEM version exceeds more than double that of the MPI version. The pipelined, computation-communication overlap pattern used for this implementation scales well under 100 nodes, but begins to converge with the OpenSHMEM synchronous version afterwards. This convergence may be due to the fact that this implementation splits the vector into multiple chunks per-PE, thereby introducing more total communication requests. The additional requests may overwhelm the network when more PEs are added, as they all must compete for data from the MASTER node. Still, this optimization shows a significant performance benefit at low node counts, indicating that RMA-based optimizations could provide substantial latency reduction as long as network traffic remains manageable.

The Cannon's MM kernel performs similarly to the DAXPY kernel with the OpenSHMEM-based kernel showing the lowest runtime for all node configurations. This app mainly shows the benefits of RMA-based peer-to-peer communication, as operations can be more easily overlapped. In the two-sided MPI-based kernel, both the *sender* and *receiver* must be finished with computation before sending and receiving data, while in the RMA-based implementations, A PE can *put()* its data to its receiver using RDMA, leaving computation uninterrupted.

The SUMMA kernel implements the same matrix multiplication as seen in the Cannon's MM kernel, but uses a partial broadcast scheme as opposed to peer-to-peer messaging. Here, the inherent benefit of peer-to-peer communication that RMA provides is lost, as on NERSC, the OpenSHMEM implementation is the worst performing of all implementations. The MPI- and MPI-RMA-based codes can be seen to be similar in terms of both throughput and latency, which can be attributed to their common usage of $MPI\_Bcast()$ to realize partial collectives. The OpenSHMEM version's custom RMA-based collective solution can be seen to perform the best on PSC, but is the worst on NERSC. It is suspected that the Cray-MPICH library contains more hardware-specific optimized collective calls than those found on the OpenMPI-based PSC system despite the PSC tests' usage of Mellanox SHARP.

The DAXPY, Cannon's MM and SUMMA kernels require the data to be initialized and distributed from the MASTER PE, whereas for the Integer Sort (IS) kernel, the data is initialized at each PE as specified by the benchmark [28]. This difference explains the overall trends in scaling. The size of the data originating at the MASTER PE grows as more PEs are added to the problem. Therefore, a communication bottleneck exists at the *scatter()* portion of the DAXPY, Cannon's MM and SUMMA programs, which becomes more limiting as nodes are added, explaining the upward trends in their runtimes. The IS kernel has the best scaling of all evaluated kernels, due to the fact that no such bottleneck exists. The performance difference between all APIs is the least notable for this kernel. This is likely due to the all-to-all communication pattern of this kernel, which utilizes the least RMA API calls of the kernels.

It is worth noting that marginal differences (below 10%) in performance seen in the results may in fact reflect larger runtime reductions in real apps. The apps that leverage supercomputers such as NERSC and PSC can consist of many kernels executing sequentially, often executing for hours or days [2] [3]. Marginal latency reductions across an entire app at this time-scale can mean hours of runtime saved, freeing more time for other jobs and improving the overall efficiency of a supercomputer.

7.2 NERSC and PSC Comparison

Overall, the performance levels seen between NERSC and PSC are expected primarily due to the generational difference between the systems. The Cori system on NERSC was delivered in 2017 whereas the Bridges-2 system at PSC was delivered in 2021. The general 50% to 80% performance improvement seen from running equivalent kernels on NERSC to PSC can be adequately explained by hardware improvements from newer CPU, memory and interconnects.

As noted, NERSC and PSC use different libraries to implement the OpenSHMEM and MPI specifications. PSC uses OpenMPI to realize both OpenSHMEM and MPI with both using the common backend of UCX [36]. We speculate that this common backend contributes to the similarity in performance between the various implementations of the collective-based kernels like Integer Sort and SUMMA on PSC.

The most notable difference between performance of a single kernel on the two platforms is the SUMMA kernel. On NERSC, the OpenSHMEM version which uses a custom RMA-based solution to implement its partial broadcasts is the worst performing. The MPI and MPI-RMA versions which primarily use the MPI library *broadcast*() operation have nearly identical performance. This trend is reversed on PSC where the OpenSHMEM variant has the lowest latency. This scaling behavior of the kernel may indicate a discrepancy in collective API call performance between the MPI libraries found on NERSC and PSC, as the behavior of point-to-point calls is consistent for all kernels between the systems.

7.3 Productivity

MPI's API simplicity stems from its two-sided nature. One-sided APIs such as OpenSHMEM and MPI-RMA require memory to be specially allocated using library calls, whereas two-sided MPI does not. In a two-sided call, the addresses of the *send()* and *recv()* buffers are communicated explicitly; in a one-sided call, the remote *target* buffer address of a *put()* or *get()* must be held by the communication library at runtime, requiring an allocation call to have occurred previously to distribute target address information between PEs. OpenSHMEM's PGAS abstraction simplifies the memory allocation process, as symmetrically allocated memory is inherently RMA-compatible. Additionally, the pattern of $shmem\_malloc()/shmem\_free()$ in an OpenSHMEM program is identical to the normal $malloc()/free()$ pattern found in equivalent C MPI programs, incurring no true additional productivity overhead. With this in mind, the API call counts between OpenSHMEM and MPI are much closer when revisiting Fig. 7, Fig. 12, Fig. 17 and Fig. 22 if allocation calls are ignored.

The RMA-based APIs also require additional *synchronization* calls to be placed in code, replacing the implicit synchronization points created by a $send()/recv()$ pair. While added synchronization calls incur productivity overhead, they also can directly enable optimizations. A synchronization point can be placed later in the program's execution, overlapping more communication and computation. These optimizations are used in the DAXPY and CMM kernels, pointing to the fact that the additional overhead from RMA-based communication can be directly offset by performance gains. Additionally, synchronization uses fewer API calls in OpenSHMEM than MPI-RMA, as can be seen in Fig. 7, Fig. 12, Fig. 17 and Fig. 22. This is because in MPI-RMA synchronization must be called on a *per-window* basis, whereas in OpenSHMEM, synchronization occurs on a PE's entire symmetric memory. The more granular synchronization approach found in MPI-RMA could in fact be exploited for better performance, as less objects are required to be synchronized in a single call, although a scenario where this could be exploited was not found in this research.

8.0 Conclusions

In this work, four distributed kernels were studied using the MPI, MPI-RMA and OpenSHMEM APIs to compare scalability and programmability. Cray-MPICH and Cray-OpenSHMEMX were the library implementations used to evaluate the APIs on NERSC while OpenMPI was used to evaluate the APIs on PSC. Each kernel stresses a unique communication-computation pattern: the DAXPY kernel simulates *scatter()/gather()* with intermediate computation, the Cannon's Algorithm Matrix Multiplication kernel highlights structured peer-to-peer communication with intermediate computation, the SUMMA kernel emphasizes group-based collectives and the Integer Sort kernel simulates an all-to-all pattern. The kernels were evaluated on up to 320 nodes on the NERSC and PSC HPC systems.

One-sided communication libraries such as OpenSHMEM and MPI-RMA perform better in terms of weak scaling for the evaluated kernels that required peer-to-peer or custom communication solutions, but incur more productivity overhead to create equivalent programs. Two-sided MPI allows for the simplest programs, both in terms of lines of code and number of API calls used, and even achieves the best performance for collective-based kernels like SUMMA and Integer Sort on some systems. However, two-sided MPI lacks the granularity and potential for novel optimizations found in one-sided APIs, illustrating a trade-off between performance and productivity.

As OpenSHMEM performs better than MPI-RMA with reduced programmer overhead, it was found to be the most viable one-sided communication API for distributed HPC, demonstrating the best overall performance with productivity that approaches the simplicity of MPI. This research illustrates the benefits of developing custom, performance-optimized communication solutions for specific execution patterns seen in a variety of HPC apps. The performance levels demonstrated by strategic OpenSHMEM programming on the kernel level have the potential to extend to hours or days of runtime and resource reduction when applied to real HPC apps, enabling significantly more efficient supercomputing center utilization.

9.0 Future Work

This work investigates kernels that focused on *scatter()/gather()*, structured peer-to-peer, partial-collective, and all-to-all communication patterns. One communication pattern that is commonly seen in large simulation work on HPC data centers is unstructured peer-to-peer communication on 2D or 3D virtual grids. This pattern could be thoroughly explored with a comparison study using a larger, dynamic-simulation app.

Along with the longstanding approach of lower-level communication libraries explored in this work, there is a new generation of languages and libraries to be explored for HPC. The Chapel language has been under development for over 10 years and promises performance similar to C-based libraries like MPI, but with far simpler codes due to data- and task-parallel constructs being standard language features. Libraries like Apache Spark and Apache Hadoop expose software frameworks in more modern languages like Python, promising scalability and simplicity at the cost of runtime overhead.

Communication libraries themselves are ever-evolving. New additions to the OpenSHMEM 1.5 specification address some shortcomings that may make it less attractive to users of MPI. One key example is PE groups called “teams” similar to the concept of “communicators” in MPI [37]. The newest specification also adds utility functions to decompose teams into 2D virtual grids, which would greatly simplify the OpenSHMEM version of the matrix multiply kernels. At the same time, the MPI specification is evolving. A large portion of future plans includes addressing one-sided operations, so productivity findings for both APIs will need to be continually evaluated [13]. Similarly, individual library implementations are regularly optimized with new features, algorithms, and hardware support, opening them to further evaluation as advancements are continually made.

Bibliography

[1] "Frontier spec sheet," May 2019.

[2] S. Mondal, R. Gupta, S. Park, B. Yoon, T. Bhattacharya, and H.-W. Lin, "Moments of nucleon isovector structure functions in 2+1+1 -flavor QCD," *Physical Review D*, vol. 102, Sep 2020.

[3] K. D. Fong, J. Self, B. D. McCloskey, and K. A. Persson, "Onsager transport coefficients and transference numbers in polyelectrolyte solutions and polymerized ionic liquids," *Macromolecules*, vol. 53, no. 21, pp. 9503–9512, 2020.

[4] R. C. Vincent *et al.*, "Li5VF4(SO4)2: A prototype high-voltage Li-ion cathode," *ACS Applied Materials & Interfaces*, vol. 12, no. 43, pp. 48662–48668, 2020. PMID: 33047963.

[5] "HPE slingshot: The interconnect for the exascale era." White Paper, 2020.

[6] "Nvidia Mellanox InfiniBand NDR 400G architecture." White Paper, 2020.

[7] D. Roland *et al.*, "Linux RDMA and InfiniBand."

[8] M. ten Bruggencate and D. Roweth, "DMAPP - an api for one-sided program models on baker systems." White Paper, 2010.

[9] A. Gainaru, R. L. Graham, A. Polyakov, and G. Shainer, "Using InfiniBand hardware gather-scatter capabilities to optimize MPI all-to-all," in *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, (New York, NY, USA), p. 167–179, Association for Computing Machinery, 2016.

[10] S. Pophale *et al.*, "OpenSHMEM performance and potential: A NPB experimental study," in *Proceedings of the 1st Conference on OpenSHMEM Workshop*, Oct. 2013.

[11] J. Jose, J. Zhang, A. Venkatesh, S. Potluri, and D. K. D. Panda, "A comprehensive performance evaluation of OpenSHMEM libraries on InfiniBand clusters," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools* (S. Poole, O. Hernandez, and P. Shamis, eds.), (Cham), pp. 14–28, Springer International Publishing, 2014.

[12] E. F. D'Azevedo and N. Imam, "Graph 500 in OpenSHMEM," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies* (M. Gorentla Venkata, P. Shamis, N. Imam, and M. G. Lopez, eds.), (Cham), pp. 154–163, Springer International Publishing, 2015.

[13] "MPI: A message passing interface standard 2019 draft specification," Nov. 2019.

[14] T. El-Ghazawi and L. Smith, "UPC: Unified Parallel C," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, (New York, NY, USA), p. 27–es, Association for Computing Machinery, 2006.

[15] "The Chapel parallel programming language."

[16] "OpenSHMEM application programming interface version 1.4," Dec. 2017.

[17] M. Baker, A. Welch, and M. Gorentla Venkata, "Parallelizing the Smith-Waterman algorithm using OpenSHMEM and MPI-3 one-sided interfaces," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies* (M. Gorentla Venkata, P. Shamis, N. Imam, and M. G. Lopez, eds.), (Cham), pp. 178–191, Springer International Publishing, 2015.

[18] N. T. Hjelm, "An evaluation of the one-sided performance in Open MPI," in *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, 2016.

[19] J. R. Hammond, S. Ghosh, and B. M. Chapman, "Implementing OpenSHMEM using MPI-3 one-sided communication," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools* (S. Poole, O. Hernandez, and P. Shamis, eds.), (Cham), pp. 44–58, Springer International Publishing, 2014.

[20] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[21] R. Brown and I. Sharapov, "Performance and programmability comparison between OpenMP and MPI implementations of a molecular modeling application," in *OpenMP Shared Memory Parallel Programming* (M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, eds.), (Berlin, Heidelberg), pp. 349–360, Springer Berlin Heidelberg, 2008.

[22] G. Wang, H. Lam, A. George, and G. Edwards, “Performance and productivity evaluation of hybrid-threading HLS versus HDLs,” in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2015.

[23] G. Guennebaud, , *et al.*, “Eigen.” http://eigen.tuxfamily.org, 2020.

[24] W. Saar, W. Qian, Z. Chothia, C. Shaohu, and L. Wen, “OpenBLAS.” https://www.openblas.net/, 2020.

[25] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, USA, 1969. AAI7010025.

[26] R. A. V. D. Geijn and J. Watts, “SUMMA: Scalable universal matrix multiplication algorithm,” tech. rep., Concurrency: Practice and Experience, 1995.

[27] B. Burley *et al.*, “The design and evolution of Disney’s Hyperion renderer,” *ACM Trans. Graph.*, vol. 37, July 2018.

[28] D. Bailey *et al.*, “The NAS Parallel Benchmarks,” tech. rep., National Aeronautics and Space Administration, 1994.

[29] J. Hemstad and U. R. Hanebutte, “ISx: A scalable integer sort mini-application,” in *Supercomputing 2015*, (Austin, Texas), Nov. 16-19 2015.

[30] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, and L. G. Fernandes, “Efficient NAS benchmark kernels with C++ parallel programming,” in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 733–740, 2018.

[31] P. Mendygral, “Cray MPI for KNL,” CMPSCI Tech. Rep., Argonne National Lab, 2018.

[32] K. Antypas, N. Wright, N. P. Cardo, A. Andrews, and M. Cordery, “Cori: A Cray-XC pre-exascale system for NERSC.” White Paper, 2014.

[33] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, “Cray XC series network,” tech. rep., Cray Inc., 1994.

[34] N. Ravichandrasekaran, B. Cernohous, D. Pou, and M. Pagel, *Introducing Cray OpenSHMEMX - A Modular Multi-communication Layer OpenSHMEM Implementation*, pp. 41–55. 01 2019.

[35] J. Urbanic, "Bridges-2 early user workshop."

[36] P. Shamis *et al.*, "UCX: An open source framework for HPC network APIs and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 40–43, 2015.

[37] "OpenSHMEM application programming interface version 1.5," Jun. 2020.