Optimizing Communication in 2D Grid-Based MPI Applications at Exascale

Hao Lu
luh1@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Piyush Sao
saopk@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Micheal Matheson
mathesonma@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Ramakrishnan Kannan
kannanr@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Feiyi Wang
fwang2@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Thomas Potok
potokte@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

ABSTRACT

Exascale computing, now a reality, still faces many challenges to achieve optimal performance with the load balanced on large numbers of nodes. A key challenge is the message passing interface (MPI) which is a critical component for process communication on exascale systems. This paper explores communication optimization strategies to harness the hybrid accelerated architectures of gpu accelerated supercomputers fully. We focus on MPI applications where processors form a two-dimensional process grid which is a common arrangement in applications involving dense matrix operations. This configuration offers a unique opportunity to implement innovative optimization strategies to improve performance and maintain effective load distribution. Building on this, we study two applications—APSP (all-pair-shortest-path) and HPL-MxP (LU factorization with Mixed precision)—on two accelerated architectures: Summit IBM Power with Nvidia V100 and Frontier AMD MI250X with AMD Epyc. We show how to scale up both applications to exascale levels and tackle the MPI challenges related to implementation, synchronization, and performance. We also compare the performance of several communication strategies at a unprecedented scale. Accurately predicting application performance becomes crucial for cost reduction as the computation scale grows. To address this, we suggest a hyperbolic model as a better alternative to the traditional one-sided asymptotic model for predicting future application performance at such large scales.

ACM Reference Format:
Hao Lu, Piyush Sao, Micheal Matheson, Ramakrishnan Kannan, Feiyi Wang, and Thomas Potok. 2023. Optimizing Communication in 2D Grid-Based MPI Applications at Exascale. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The advent of exascale computing has opened up new avenues of scientific discovery and innovation. However, it also presents a unique set of challenges in achieving optimal performance and efficient load balancing of the system resources. One of the key aspects of exascale computing is the Message Passing Interface (MPI), which plays a crucial role in process communication within these systems. This paper focuses on MPI applications where processors form a two-dimensional process grid which is a common arrangement for applications involving dense matrix operations. We delve into two specific applications — the distributed memory-Floyd Warshall algorithm for computing all-pair shortest paths (APSP) and the high-performance Linpack in mixed precision (HPL-MxP) [10] implementation.

The distributed memory-Floyd Warshall algorithm is employed for computing all-pair shortest paths between different vertices in a graph. Unlike traditional algorithms, this approach uses a semi-ring multiplication operation, a mathematical structure akin to a ring but devoid of the requirement of additive inverses. This unique feature allows for efficient computation and storage, particularly when managing large-scale graphs.

In contrast, HPL-MxP stands as a testament to the power of LINPACK (Linear Algebra Package), a software library specifically designed to perform numerical linear algebra computations on large-scale systems with a distributed memory architecture. The LINPACK benchmark measures the performance of computer systems in solving dense systems of linear equations using LU decomposition. The utilization of mixed precision in the HPL framework further extends its use for AI-like workloads on exascale systems.

Both the APSP and HPL-MxP cases share common challenges and strategies. At every iteration, row and column broadcasts are required, typically involving a look-ahead strategy to overlap computation with communication. One of the primary obstacles lies in optimizing this communication and efficiently mapping computation to the architecture within the context of a two-dimensional process grid.

The incorporation of Graphics Processing Unit (GPU) accelerators in supercomputers presents additional challenges. The programming model of GPUs differs from that of traditional Central Processing Units (CPUs), adding another layer of complexity when

trying to achieve an efficient communication overlap with computation.

This paper's primary focus is optimizing the broadcast operations for such applications, specifically overlapping communication with computation and efficiently mapping computation to the architecture. We also aim to provide a model for predicting application performance at the exascale level, vital for cost reduction as computation scales grow. Moreover, we will discuss the desired MPI features that can facilitate the expression of such optimization strategies, further boosting the performance and efficiency of exascale computing applications. Exscale, MPI, 2D grid

2 BACKGROUND

2.1 System Information

In this paper, our study focus on two different architectures of super computers that are hosted in OLCF, Frontier and Summit. A brief overview of the key architectural specifications and software stacks are in Table 1. The key difference we like to point out is the network interconnect location. The NICs reside on the GPUs in Frontier system, which removes the traditional overhead of copying network data back to CPU. On the other hand, Summit system contains two socket per node, and the two NICs reside on each of the socket. Hardware and software features for combining the two NICs for single off node MPI communication is provided. The node architecture is showed in Fig 1 and 2

Another main difference is the ration of GCDs to GPU. Summit system has one GCD per GPU and Frontier system contain two GCDs per GPU. Each GCD is used as a separate accelerator.

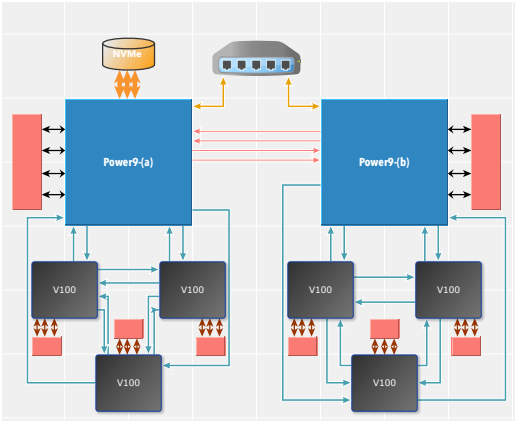


Figure 1: Summit Node architecture. Bandwidth for connection: 64 GB/S CPU to CPU (red), 16 GB/S CPU to NICs (orange), 50 GB/S GPU to CPU (blue)

2.2 HPL-MxP

The HPL-MxP benchmark was designed for evaluation of the system mixed precision capability by finding the unique solution to a dense system of linear equations $Ax = b$, where $A \in \mathbb{R}^{N\times N}$ is a full rank matrix and $x, b \in \mathbb{R}^N$ are the solution and right-hand side vectors, respectively. In contrast with the HPL benchmark, HPL-MxP allows

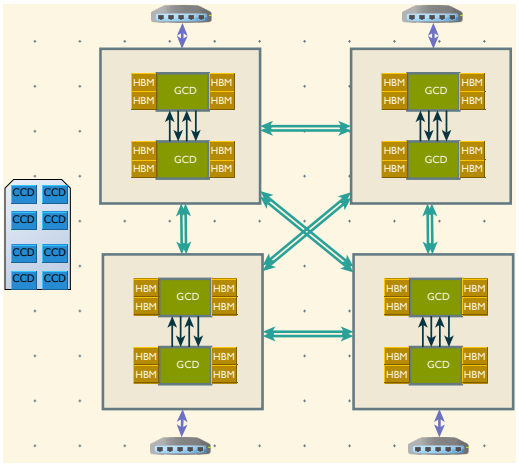


Figure 2: Frontier Node architecture. Bandwidth for connection: 50 GB/S GPU to GPU (green), 50 GB/S GPU to NICs (blue)

	Summit	Frontier
Number of Nodes	4608	9408
Processor	Power9	3rd Gen EPYC
CPU memory (Node)	512 GB	512 GB
GPU / # of GCDs (Node)	NVIDIA V100/6	AMD MI250X/8
memory per GPU	16 GB	128 GB
GPU Interconnect	NVLINK	Infinity Fabric
GPU Interconnect B/W	50+50 GB/s	50+50 GB/s
FP64 TFLOPS (GCD)	7.8	54.5
FP16 TFLOPS (GCD)	125	191
# of NICs	2x Mellanox EDR IB	4x Slingshot-11
NIC B/W (node)	12.5+12.5 GB/s	25+25 GB/s
MPI Library	spectrum-mpi/10.4.0.3	cray-mpich/8.1.25
GPU Library	cuda/11.4.2	rocm/5.4.3
Compilers	gcc/9.1.0	gcc/9.1.0

Table 1: Key architectural and software stack specifications for Summit and Frontier

for the input matrix to have an appropriate condition number to omit the pivoting step during the LU factorization [9, Chapter 9]. More importantly, it allows the use of a mixed precision solution to obtain lower precision $\tilde{L}$ and $\tilde{U}$ factors.

The benchmark requires the solution to be solved by three specific procedures. First, the matrix is transformed into an estimated triangular form using a mixed precision block Gaussian elimination [17]. Once the estimated LU factorization of $A \tilde{=} \tilde{L}\tilde{U}$ is obtained, the estimated solution $\tilde{x}$ to $Ax = b$ can be efficiently obtained by solving the two triangular systems of linear equations ($\tilde{L}\tilde{U}\tilde{x} = b$). Finally, the solution is further corrected back to FP64 precision accuracy ($b - A\tilde{x} < \epsilon$) by applying iterative refinement (IR) [18].

Block LU factorization. Block based Gaussian elimination partition a size n matrix A into $n_b \times n_b$ blocks, each with size $b \times b$ (i.e., $n_b = \frac{n}{b}$). The block size b is chosen to balance communication and computation. Consequently, transforming A into its LU factorization occurs in blocks, that is, each step of the Gaussian elimination computes b columns of L and b rows of U. [5, 16]. The total number of steps required for a complete LU factorization is n_b steps. Let $A^{(k)}$ denote the unfinished matrix at step k, and U_0 and L_0 denote the finalized part of matrix at step k, see part (a) of Figure 3

To factor the remaining $(N - kB + B) \times (N - kB + B)$ submatrix $A^{(k)}$ as $L^{(k)}U^{(k)}$, we represent as

$$A^{(k)} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix},$$

where $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, and $A_{2,2}$ are of sizes $B \times B$, $(N - kB) \times B$, $B \times (N - kB)$, and $(N - kB) \times (N - kB)$, see part (b) of Figure 3.

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & U_{2,2} \end{bmatrix}$$
$$= \begin{bmatrix} L_{1,1}U_{1,1} & L_{1,1}U_{1,2} \\ L_{2,1}U_{1,1} & L_{2,1}U_{1,2} + L_{2,2}U_{2,2} \end{bmatrix},$$

Further expanding, LU factorization can be viewed as solving for $L_{1,1}$, $L_{1,2}$, $U_{1,1}$ and $U_{2,1}$, then update the $A_{2,2}$, see part (c) of Figure 3. The high-level algorithmic steps for block-based LU factorization at step k could be represent as the following:

(1) Gaussian elimination for $A_{1,1}$ to find $L_{1,1}$ and $U_{1,1}$;
(2) Compute $L_{2,1} = A_{2,1}U_{1,1}^{-1}$;
(3) Compute $U_{1,2} = L_{1,1}^{-1}A_{1,2}$;
(4) Compute $A^{(k+1)} = L_{2,2}U_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$.

Iterative refinement. Even when A is well-conditioned, computing its LU factorization suffers from precision limitations of floating point arithmetic, especially when working in mixed precision. As a result, the mixed precision LU factorization $A \rightarrow \tilde{L}\tilde{U}$ holds only an approximation for the real factorization L and U. Performing IR by repeating the following steps until the required solution accuracy is reached:

(1) Compute the residual $r = b - A\tilde{x}$ in higher precision;
(2) Find an approximation $\tilde{d}$ of solution discrepancy $d = x - \tilde{x}$ by solving $\tilde{L}\tilde{U}\tilde{d} = r$
(3) Refine the approximate solution $\tilde{x}$ by assigning $\tilde{x} \leftarrow \tilde{x} + \tilde{d}$.

The estimate solution $\tilde{x}$ is refined closer to x every iteration, given the input was designed to converge. For benchmark purpose the IR is stopped once the solution discrepancy $\tilde{d}$ gets below some ration of FP64 machine epsilon. As this part of the procedures does not require significant run time we will omitted the detail in our implementation and discussion.

2.2.1 *Related Work.*

In 2006 Kurzak and Dongarra [12] were first to introduce the use of mixed precision to solve $Ax = b$. In 2010, Wang et al. offered a GPU-accelerated version of the algorithm for the first time [13]. Dense linear algebra library ScaLAPACK [3] which developed in 1992 supported distributed computing. In 2009, MAGMA library [1],[2] enabled the GPU support for BLAS. In 2017, Haidar et al. added mixed precision variants to MAGMA [7, 8]. In 2019, a library called SLATE [6] expand the capability to multiple precision with distributed multi-GPU support. However, these libraries mainly focused on the portability and usability, and does not tailored to target the maximum performance of target system. On the CPU front, the Fugaku HPL-MxP code [11, 15] is the first to break the exascale barrier in 2020 with a CPU-only implementation.

The implementation we used and updated in our paper is from the OLCF [14] and is called OpenMxP, which is the first code that obtains almost 8 exaflops on the Frontier system. This papers work is heavily influenced by these prior works, including using the OpenMxP code as a baseline, but takes it further with communication optimizations and performance modeling to achieve almost 10 exaflops.

2.3 Block Floyd-Wasrshall

FW uses a dynamic programming approach to computing all-pairs-shortest path. It initialized the distance of each pair Dist [i,j] to infinity. It initializes Dist with the input weights W. Then, in the k-th iteration, it checks for all pairs of vertices v_i and v_j if there is a shorter path between them via the intermediate vertex v_k. If so, FW updates Dist$[i, j]$. Therefore, Dist$[i, j]$ after k steps, which we denote by Dist$^k(i, j)$, may be defined recursively as

$$\text{Dist}^k[i, j] \leftarrow \min\left\{\text{Dist}^{k-1}[i, j], \text{Dist}^{k-1}[i, k] + \text{Dist}^{k-1}[k, j]\right\}.$$

Therefore, at the end of the n-th iteration Dist$^n[i, j]$ will be the length of the shortest path between v_i and v_j.

Apsp may be understood algebraically as computing the matrix closure of the weight matrix, W, defined over the tropical semiring [4]. In more basic terms, let $\oplus$ and $\otimes$ denote the two binary scalar operators

$$\begin{aligned} x \oplus y &:= \min(x, y) \\ x \otimes y &:= x + y, \end{aligned}$$

where x and y are real values or ∞. Next, consider two matrices $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$. The Min-Plus product C of A and B is

$$C_{ij} \leftarrow \bigoplus_k^{\oplus} A_{ik} \otimes B_{kj} = \min_k \left(A_{ik} + B_{kj}\right).$$

To see its connection to graph path analysis, consider an example of the complete tripartite graph in **??**.

This interpretation of the Min-Plus product helps to understand the following blocked version of FW (**??**).

2.3.1 *Blocked Flyod-Warshall algorithm.*

Suppose we divide Dist into $n_b \times n_b$ blocks, each of size $b \times b$ (i.e., $n_b = \frac{n}{b}$). Let A_{ij} denote the (i, j) block of A, where $1 \le i, j \le n_b$. Then a blocked version of FW, called BlockedFw in **??**, can carry out the same Apsp computation as FW in the following three steps, as illustrated in **??**:

- **Diagonal Update:** Perform the classic FW algorithm on a diagonal block, A_{kk}.
- **Panel Update:** Update the k-th block row and column. For any block $A(k, j)$, $j \neq k$ in the block row, the update is a Min-Plus multiply with A_{kk} from the left, and for block $A(i, k)$ on the k-th block column is Min-Plus multiply with A_{kk} from right,

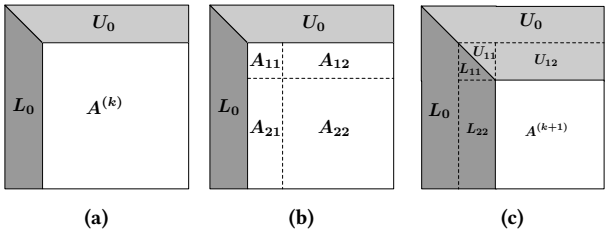


Figure 3: The kth step of block LU factorization. (a) The layout right before the kth step. (b) Partitioning of the trailing matrix. (c) The outcome of the kth step.

2.4 2D Block-cyclic data distribution

The 2D block-cyclic data distribution scheme divides a matrix, like a matrix A, into blocks denoted as (i, j). Each block represents a submatrix $A[ib : (i+1)b, jb : (j+1)b]$ with a block size b. This scheme is useful when the block size is uneven, as in sparse direct solvers. Blocks (i, j) are assigned to specific processes based on row and column indices $P_r(i)$ and $P_c(j)$.

This distribution scheme has several advantages. Firstly, it ensures data ownership, meaning the process responsible for a submatrix is always known. This guarantees precise knowledge of where the data resides. Secondly, the owner update policy ensures that the latest copy of a block is with the corresponding process. Therefore, only the owner process updates the block, eliminating the need for coordination among multiple processes.

Load balancing is achieved through fine-grained blocked partitioning, which mitigates imbalances when updating matrix parts. Random submatrices of A are usually equally distributed among processes, enabling arbitrary grid dimensions and efficient computations across the distributed system.

Communication within process rows and columns is vital in this scheme. Most communication occurs between processes in the same row or column, minimizing overhead and promoting efficient data exchanges.

While block-cyclic distribution offers significant benefits, it is important to consider alternative data distribution strategies. These include 2D data distribution without block-cyclic characteristics, global arrays (PGAS), and tiled block-cyclic distribution. In contrast to 2D data distribution, the PGAS (Partitioned Global Address Space) model provides a shared memory-like programming model while leveraging the distributed memory architecture. This approach is supported by languages such as Unified Parallel C (UPC), Co-Array Fortran (CAF), and Chapel. PGAS is particularly useful when dynamic load balancing is required. Each alternative differs in how the local submatrix is stored locally, introducing variations in communication patterns and load-balancing strategies. **add citations to models and Xing's paper**

2.5 Distributed Implementation

For distributed version of HPL-MxP and FW algorithm, we partition the global matrix A into 2D Block-cyclic data distribution described in section 2.4. Follow the numerical steps in section 2.2 for HPL-MxP, we used accelerator specific BLAS (Basic Linear Subprograms). The kernels we used are SGETRF_nopivot (FP32), STRSM (FP32) and GEMM_ex (FP16), with additional native implemented casting kernels for precision changes. Pseudo code is provided in Algorithm 1

Algorithm 1 Distributed GPU mixed Precision LU

Input: N, B, P_r, P_c
Fill global matrix A with random numbers.
(1) Block LU factorization
On each MPI process p_{id} do in parallel:
for $k = 1,\ 2,\ 3 \ldots\ n_b$ **do**
  Synchronize all processes
  $P_{ir}, P_{ic} \leftarrow processmapping(k)$ // $A_{k,k}$ owner process index
  (1a) Diagonal Update
  if $p_{id} == P(P_{ir}, P_{ic})$ **then**
    $A(k,k) \leftarrow$ ***GETRF***$(A(k,k))$
    Broadcast $A(k,k)$ to $P(P_{ir}, :)$ and $P(:, P_{ic})$
  (1b) Panel Update
  if $p_{id} \in P(P_{ir}, :)$ **then**
    Receive $A(k,k)$
    $A(k, k+1:n) \leftarrow$
      TRSM_L_LOW$(A(k,k), A(k, k+1:n))$
    $U \leftarrow$ ***TRANS_CAST***$(A(k, k+1:n))$
    Broadcast U to processes in $P(:, p_{ic})$
  else
    Receive U
  if $p_{id} \in P(:, P_{ic})$ **then**
    Receive $A(k,k)$
    $A(k+1:n, k) \leftarrow$
      TRSM_R_UP$(A(k,k), A(k+1:n, k))$
    $L \leftarrow$ ***CAST***$(A(k+1:n, k))$
    Broadcast L to processes in $P(p_{ir}, :)$
  else
    Receive L
  (1c) Update Trailing Matrix
  $A(k+1:n, k+1:n) \leftarrow$ ***GEMM***$(L, U, A(k+1:n, k+1:n))$

For distributed FW, we implemented the required SEMIRING-GEMM kenerl. Pseudo code is provided in Algorithm 2

3 IMPLEMENTATION AND OPTIMIZATION

3.1 Overlapping Communication with Computation

Add citations for look ahead scheme One crucial strategy to enhance performance in distributed computing systems is to overlap communication with computation. This is achieved by the HPL-MxP and Dist-FW algorithms using a look-ahead optimization. This method breaks down operations into smaller, manageable components. It allows the system to rank computation and communication tasks based on cost, facilitating simultaneous execution of computations even while waiting for data from other nodes.

Unlike the traditional bulk-synchronous structure, look-ahead optimization doesn't force processes to stall until each node finishes its current task. Instead, it allows the next computation or communication step to proceed. This technique, known as pipelining, enables overlapping communications between different stages of a calculation without requiring each process to wait for others.

This approach reduces idle time, where no work is done, and boosts efficiency by using resources more effectively. In summary,

Algorithm 2 Parallel Floyd-Warshall algorithm on 2D process grid

```
1: function ParallelFW(A, P = P_r × P_c)   ▷ A is distributed in block-cyclic fashion   ▷ my process Id is pid
2:   for all MPI process pid in parallel do
3:     for k ∈ {1, 2, ..., n_b} do
4:       if pid = p_{k,k} then
5:         A(k,k) ← Floyd-Warshall(A(k,k))
6:         Broadcast(A(k,k), P_r(k))
7:         Broadcast(A(k,k), P_c(k))
8:       if pid ∈ P_r(k) then
9:         Receive(A(k,k), p_{k,k})
10:        A(k,:) ← A(k,:) ⊕ A(k,k) ⊗ A(k,:)
11:        Broadcast(A(k,:), P_c(pid))
12:      else
13:        Receive(A(k,:))
14:      if pid ∈ P_c(c) then
15:        Receive(A(k,k), p_{k,k})
16:        A(:,k) ← A(:,k) ⊕ A(:,k) ⊗ A(k,k)
17:        Broadcast(A(:,k), P_r(pid))
18:      else
19:        Receive(A(k,:))
20:      for i ∈ {1, 2, ..., n_b} do
21:        for j ∈ {1, 2, ..., n_b} do
22:          if pid owns A(i,j) then
23:            A(i,j) ← A(i,j) ⊕ A(i,k) ⊗ A(k,j)
```

overlapping communication with computation via look-ahead optimization minimizes delays and maximizes resource use, leading to improved performance in distributed computing systems.

3.2 Mapping 2D Partitioned Algorithm to Architecture

We must effectively adapt the computation to the architecture to enhance performance in distributed computing systems. We can optimize this in two ways: by increasing bandwidth and reducing latency. The former strives to boost data transfer between nodes over time. This requires a clear grasp of network topology constraints, communication protocols, and hardware limitations specific to the application. On the other hand, reducing latency aims to cut data travel time between nodes, which also demands a thorough knowledge of network topology and hardware constraints.

Proposed models like fat-tree and torus networks can ease bandwidth usage during inter-node communication. These models provide multiple paths with varying connectivity degrees based on node proximity within the network. Overlapping computation optimization depends on several factors, including available resources such as network adapters, the employed communication protocols (TCP/IP or RDMA), and network topology constraints (fat-tree, torus, dragonfly, etc). These factors demand careful thought when designing an efficient distributed computing system.

While reducing bandwidth requirements or optimizing architectural mapping to increase bandwidth is generally possible, latency optimization is more challenging with fewer techniques available. Fortunately, latency is normally less critical in dense matrix operations. Although this paper doesn't primarily focus on latency optimization, we discuss situations where such optimizations might be beneficial and how the Message Passing Interface (MPI) can aid these efforts.

3.2.1 *Choosing Grid dimensions.*

Contrary to popular belief, a square process grid (i.e., $P_r = P_c$) doesn't necessarily minimize communication costs as it often overlooks the network architecture. For instance, while $P_r = P_c$ reduces total communication for each process, it fails to distinguish between data transferred to a different node, and data exchanged within the same node, neglecting the significant differences in bandwidth and latency.

To decrease communication time, we should focus on the slowest link used by the application, often the network interface card (NIC), when not using IO or NVM. By mapping computation to architecture, we can optimize the NIC usage and maximize bandwidth. This involves identifying the data sent via each NIC and optimizing its transmission.

Consider MPI processes arranged in a 2D grid of dimension $P_r \times P_c$. If a subset of processes Q share a single NIC, we can arrange them in a logical grid $Q_r \times Q_c$, resulting in a logical 2D grid arrangement of NIC with dimensions $K_r \times K_c$ where $K_r = P_r/Q_r$ and $K_c = P_c/Q_c$. We aim for $K_r \approx K_c$ to minimize data transfer through the NIC.

In practice, we usually assign ranks in a specific manner to achieve this configuration. For example, we use an explicit resource file on the Summit supercomputer to determine where each rank resides. In MPICH, we can set this by using:

```
MPICH_RANK_REORDER_METHOD=3
```

and creating a file:

```
MPICH_RANK_REORDER.grid
```

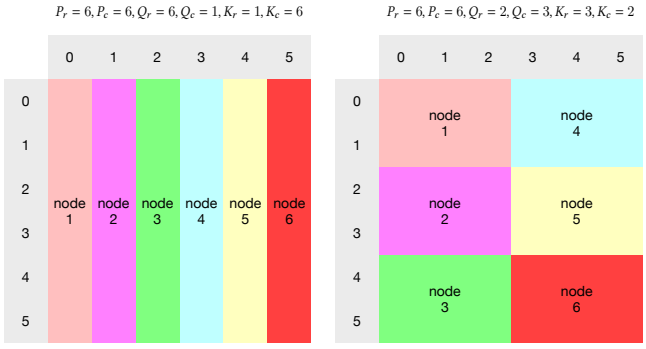
that contains a list of ranks in the i-node in its i-th line. The default method of assigning rank can lead to poor performance as all the ranks on the node are consecutive, often resulting in an unbalanced grid structure when partitioning into a 2D grid.

In Figure 4, we present a logical representation of $Q_r(P)$ and $Q_c(Q)$ for six nodes in both the default and optimized configurations. In the default scenario, each Network Interface Card (NIC) transfers data at a cost of $2n^2$, indicating high communication overhead. However, in the optimized scenario with more effective node arrangements, the communication cost decreases to $5n^2/3$. This demonstrates that adjusting the grid structure and rank distribution can significantly reduce communication overhead and improve NIC bandwidth utilization.

On the other hand, Figure 5 highlights the advantages of different combinations of Q_r and Q_c in terms of effective bandwidth. It provides insights into how adjusting these parameters affects the effective bandwidth and allows us to identify the optimal configuration for maximizing this metric. This figure emphasizes the importance of carefully mapping computation to architecture through rank distribution and grid arrangement, leading to substantial improvements in communication efficiency.

- **F**ig 4 Logically Describe the qr(P) qc(Q) that reduce the inter-node message
- **F**ig 5 show case the benefit of qr qc in terms of actual measurement of bandwidth

- Fig 6 show case the hpl-mxp improvement we can harvest for Summit specific, binding 2 port, using spectrum optimized bcast seem best
- Fig 7 show case the hpl-mxp improvement we can harvest for Frontier specific, max improvement from GPU-aware mpi, smaller increase for qr qc.



Distance of row broadcast	node 1 to 6	node 1 to 4
Distance of column broadcast	node 1	node 1 to 3
Transfer data size	$2N^2 + 0$	$\frac{2N^2}{3} + \frac{2N^2}{2}$

Figure 4: Example of different node local grid, and the metric that impact the communication time.

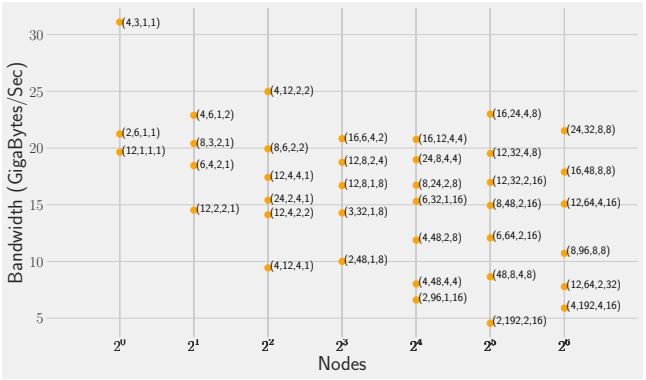


Figure 5: Summit Example of different node local grid, and the metric that impact the communication time.

3.3 Optimizing Broadcast Operations

In the HPL-AI and Dist-FW algorithms, we need to perform broadcasts across process rows and columns in each iteration. The broadcast source shifts one step per iteration to ensure all processes have updated data. For example, process $P_r(k)$ handles the broadcast in the k-th iteration, and in the subsequent iteration (k+1), process $P_r(k+1)$ takes over this task. The same pattern applies to column communication operations.

Tree-based broadcast algorithms, such as binary trees and Fibonacci trees, are commonly used in distributed computing systems

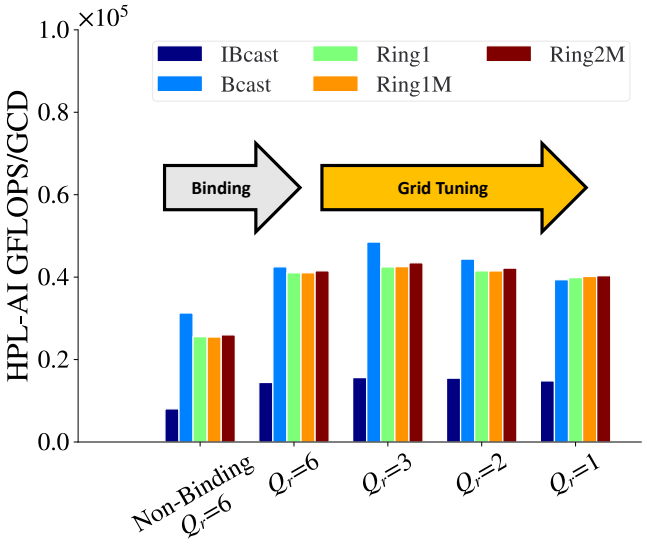


Figure 6: Summit HPL-MxP architecture features, 2916 GCD, LN=61440,b=768

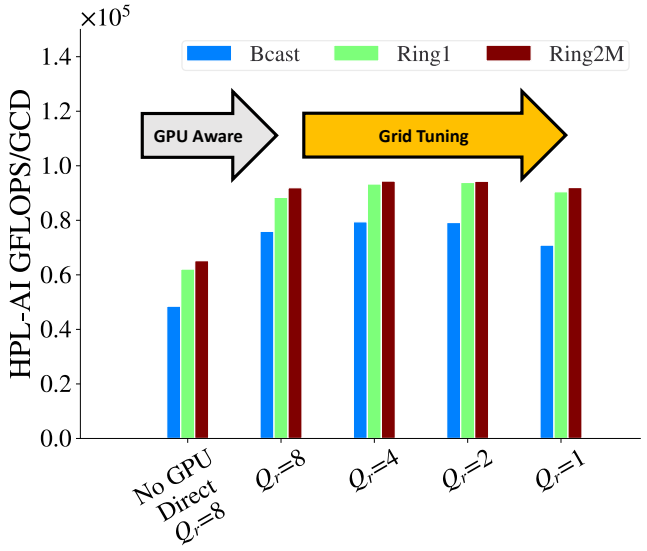


Figure 7: Frontier HPL-MxP architecture features, 1024 GCD, LN=119808,b=3072

to achieve efficient data communication among nodes. However, for 2D partitioned matrix applications like HPL-AI and Dist-FW algorithms, the ring family of broadcasts may outperform tree-based broadcasts as it aligns well with the communication pattern. The ring family of broadcast algorithms offers various variants, including modified and bidirectional rings (two-directional). The selection of the appropriate broadcasting algorithm depends on the constraints of the network topology and specific performance requirements, such as reducing latency or optimizing bandwidth utilization.

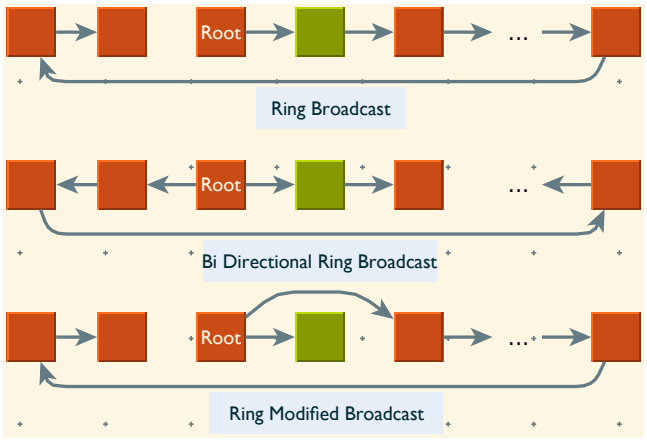


Figure 8: Different Variant of Ring Broadcast used in our experimentation

Algorithm 3 Chunked Broadcast Algorithm

```
1: procedure ChunkedBroadcast
Require: CSize, P_r, P_c, R_r, R_c, N_r, N_c, Buf_r, Buf_c,   ▷ Variables: Chunk Size, Process row/column communicator, Row/Column root, Number of chunks for row/column broadcast, Buffer for row/column broadcast
2:   i_r = 0, j_c = 0
3:   for each process (px, py) in parallel do
4:     if I am in the root row then
5:       for i ← 0 to N_r − 1 do
6:         relay(P_r, R_r, Buf_r[i])
7:       i_r ← N_r
8:     if I am in the root column then
9:       for j ← 0 to N_c − 1 do
10:        relay(P_c, R_c, Buf_c[j])
11:      j_c ← N_c
12:  while i_r ≠ N_r or j_c ≠ N_c do
13:    if i_r ≠ N_r then
14:      check receive(i_r)
15:      if received then
16:        relay(i_r)
17:        i_r ← i_r + 1
18:    if j_c ≠ N_c then
19:      check receive(j_c)
20:      if received then
21:        relay(j_c)
22:        j_c ← j_c + 1
23: end procedure
```

3.3.1 A modified 2D broadcast. Besides the tree broadcast and ring broadcast, there is another version of the broadcast which is essentially a modified version of the ring broadcast. It modifies it in two ways: combining the two broadcasts and breaking down the message into smaller parts. The idea behind this broadcast is to avoid high latency costs associated with string broadcasts. Breaking the message into smaller parts allows us to pipeline the broadcast and transmit each chunk individually. Additionally, this modified broadcast merges both the process row and column broadcasts.

To achieve this merge, we track which chunk is transmitted for each row and column at every step of the combined broadcast. We wait for messages to be received before incrementing pointers.

This broadcast implementation performs the best in all cases and has an added advantage. It is much more resilient to unexpected network behavior that we observed during testing. In some instances, certain network links would completely fail for specific messages.

We found that this modified version of the broadcast could withstand such deviations more gracefully than other versions. While we do not clearly understand why the faults occurs and why this broadcast algorithm shows resilient behavior, it is an area worth investigating in the future.

3.3.2 Discussion of various experiments. The results and data presented bring into focus the varying efficiencies of different broadcast algorithms in the context of HPL-AI and Dist-FW algorithms. A broad range of communication patterns and their effects on workloads were studied, focusing on the first 1200 iterations in the case of the APSP workload (Fig 9). It was observed that the 2RM-Pipelined version of the ring broadcast, wherein the message is broken down into chunks, outperformed most other communication methods. This suggests that the pipelining technique is an effective way to mitigate the high latency costs typically associated with broadcasts.

However, this is not the only communication method that showed promise. In the HPL-MXP workload, the 1ring chunked method was the standout, as shown in Fig 10. This variant seemed to work in harmony with the decreasing workload of HPL-MXP over the iterations, and it outperformed the other ring implementations.

A key finding from these experiments was the remarkable effect of incorporating OpenMP to alleviate congestion during 2D grid communication (Fig 11). The simultaneous arrival of two messages often leads to congestion, and effectively handling this scenario can result in significant communication time reduction. The use of OpenMP proved to be an effective solution, further reducing communication time and enhancing the performance of the broadcast algorithm.

The effect of different chunk sizes on the communication was also investigated (Fig 12). Surprisingly, a chunk size of 2 MB appeared to outperform other sizes. This provides valuable insights for future broadcast implementations, suggesting an optimal chunk size for minimizing latency and maximizing efficiency.

Finally, a noteworthy pattern emerged when evaluating the average bandwidth across different data sizes during an HPL-MXP run (Fig 13). As the number of nodes doubled, the average bandwidth remained 95-98%. This indicates that the ring broadcast algorithms, specifically the chunked versions, are efficient not just in terms of latency but also in terms of bandwidth utilization.

Overall, these results highlight the potential of the ring family of broadcast algorithms, specifically their modified versions, for 2D partitioned matrix applications like HPL-AI and Dist-FW. The use of pipelining and chunking techniques and the incorporation of multi-threading solutions like OpenMP seem to offer a robust approach to improving communication efficiency in such distributed

computing systems. Future studies should further investigate the resilient behavior of these modified broadcasts, especially in the face of unpredictable network behavior.

- **F**ig 9 Show case different communication effect on APSP workload for the first 1200 iterations, workload remain the same throughout the run. 2RM-Pipelined (2ringM chunked), recorded in the yellow line, was outperformancing most other communication. (we dont have 1rC data)
- **F**ig **??** ALL type of ring data for hpl-mxp, want to remove.
- **F**ig 10 show case the basic implementation of all the different ring for hpl-mxp workload for first 3000 iterations. Clearly see the 1ring chunked outperform the rest of ring, recorded in the read line. Note the workload of hpl-mxp decrease as the iteration pass through
- **F**ig 11 show case the hpl-mxp ring improvement using openmp to resolve the congestion happens during the 2D grid communication, where 2 message come in the same time. Openmp further reduce the communication time, recorded in the yellow line
- **F**ig 12 Compare the effect of different chunk size, 4MB seem to outperform, recorded in the blue line.
- **F**ig 13 Average bandwidth on different size of data during hpl-mxp run. The last label is the average bandwidth across all size we recorded. We observe a 95-98% reduction of bandwidth as we double the nodes.

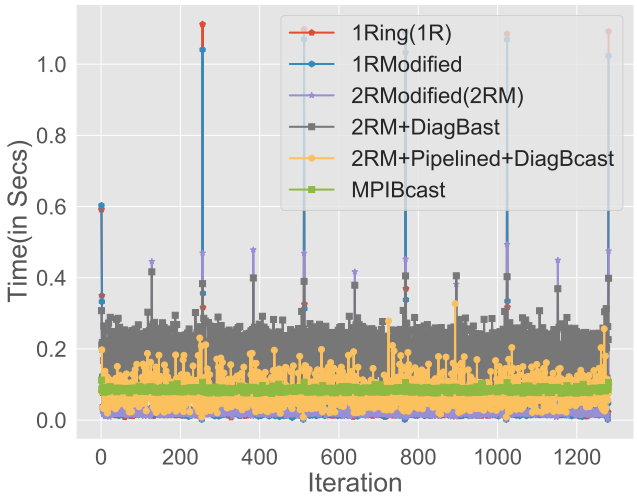


Figure 9: Frontier APSP communication for ring and broadcast compare. 2RingM chunked outperform, per GCD has 2 ranks of LN36864, PQ=128x128

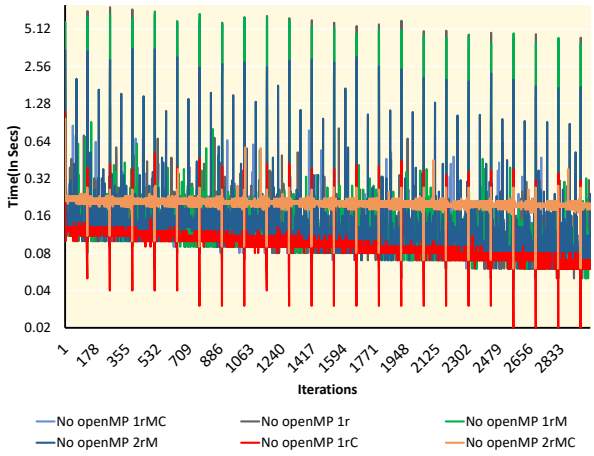


Figure 10: Runtime of Frontier Basic implementation of ring boradcast, PxQ=128x128, LN=125440, 3000iterations, spikes are due to longer latency of chain, 2ringM reduce it, chunking reduce even more

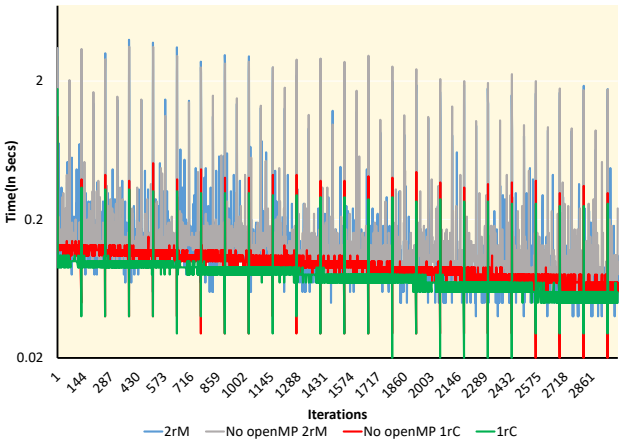


Figure 11: Runtime of Frontier openMP ring compare with no openMP, it further reduce peak for congestion happens on 2D grid,PxQ=128x128, LN=125440, 3000iterations

4 PERFORMANCE MODEL

4.1 Motivation

4.2 Hyper Model

4.3 Application Performance Modeling

When we work with parallel applications, we often face the challenge of making accurate performance predictions. We need a model that can handle all sorts of situations, not just when our problem parameters are large. Traditional models, called 'asymptotic models', struggle when problem sizes per process become small as the number of processes increases. So, we turn to a more flexible solution: hyperbolic performance models.

Let's simplify this complex term. A hyperbolic performance model is just a function, $y = f(x)$, which draws a hyperbola shape when plotted on a graph. Here, y is the performance we want to predict, and x is the problem parameter. We write this function as:

$$y := f(\eta) = \frac{a\eta}{\eta + d} \qquad a,\ ad \neq 0.$$

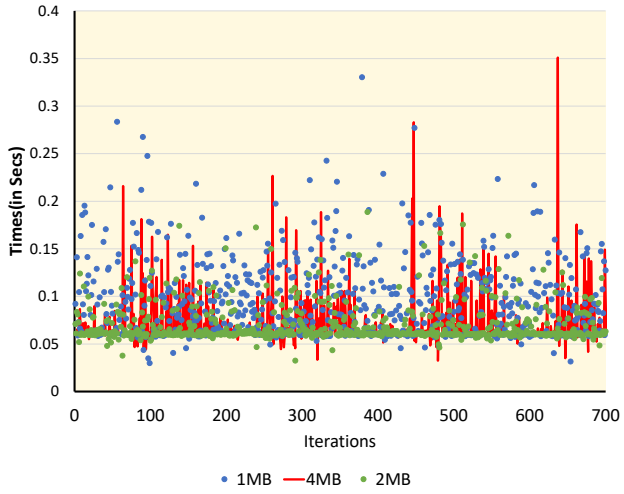


Figure 12: Comparison of different chunk size cf1024 is 4MB, which is out perform. 128x128

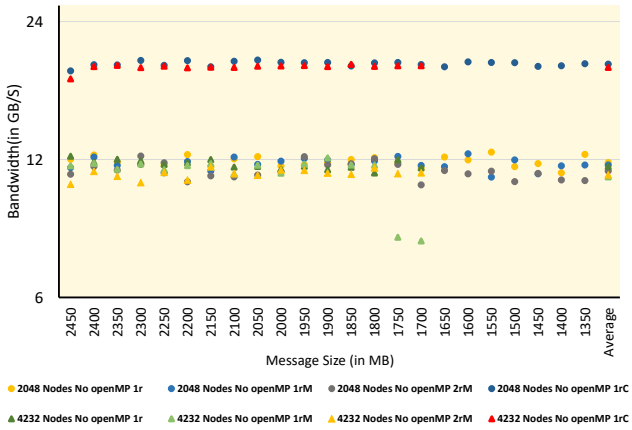


Figure 13: Average of Bandwidth of Frontier best ring scale comparison, going from 128x128 to 184x184, observe roughly 2-5% reduction on average bandwidth

where η is a performance parameter, and a and d are constants we need to determine.

Now, we need a way to estimate the values of a and d that best fit our observed data. To do this, we use a technique called 'least-squares fit'. This technique tries to minimize the difference between our model's predicted performance and the actual observed performance.

Let's say $g(\eta_i)$ is the observed performance for a given problem parameter η_i. We want our model, $f(\eta_i)$, to be as close as possible to $g(\eta_i)$. We aim to find the values of a and d that make this difference the smallest:

$$arg \min_{a,d} \sum \frac{1}{\eta_i} \cdot \left(\log \frac{f(\eta_i)}{g(\eta_i)}\right)^2.$$

Finding the exact values of a and d that minimize this difference can be difficult. So, we use a simpler method to get approximate values. We take a to be the maximum observed performance value, and d to be the value of η_i that makes $g(\eta_i)$ as close as possible to $a/2$:

$$a \approx \max_i g(\eta_i) \tag{1}$$

$$d \approx arg \min_{\eta_i} \left| g(\eta_i) - \frac{a}{2} \right|. \tag{2}$$

This approximation method usually gives a good fit when our observations aren't too noisy. But if they are, we can try different values of d close to the one we found, and choose the one that gives the best fit.

- Fig 14 Shows gemm performance on Frontier of various local matrix size. Serve as building block for performance model
- Fig 15 Shows gemm performance on Frontier of various local matrix size. Serve as building block for performance model
- Fig 16 and Fig 17 Shows broadcast performance on Frontier of various local matrix size in bandwidth and runtime. Serve as building block for performance model
- Fig 18 Performance prediction heatmap

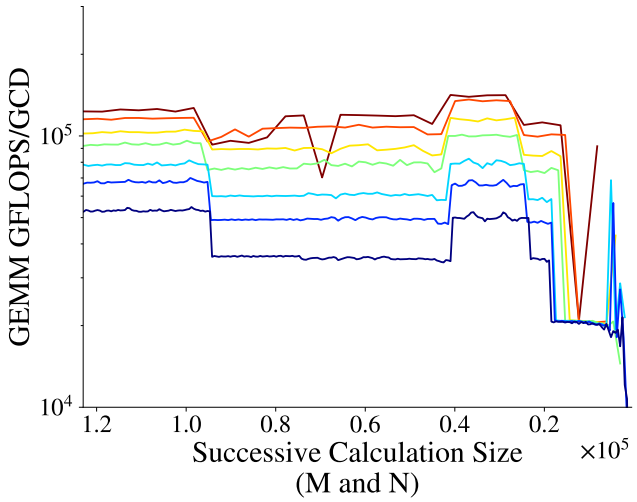


Figure 14: Performnace sweep for M on most significant kernel GEMM_ex on Frontier

5 APPLICATION PERFORMANCE

- Fig 19 Shows to total runtime after combine the compute and communication optimization vs the basic 1ring chunked on Frontier. We achieve 10.XX exaflops (I will update number) before IR, and 9.95 exaflops after the IR. This is the first application that harvest the 10exaflops on any system.
- Fig 21 and Fig 22 Similarly we report the strong and weak scaling of APSP on Summit after combine all the optimization

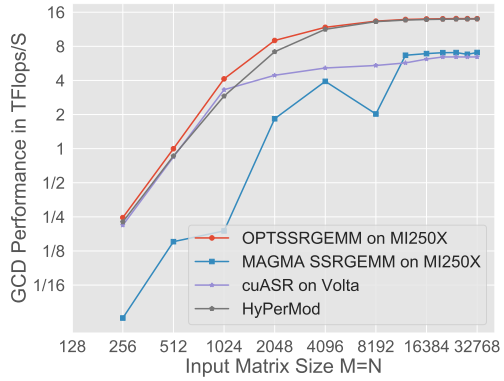


Figure 15: Performnace sweep for M on most significant semiring gemm on Frontier

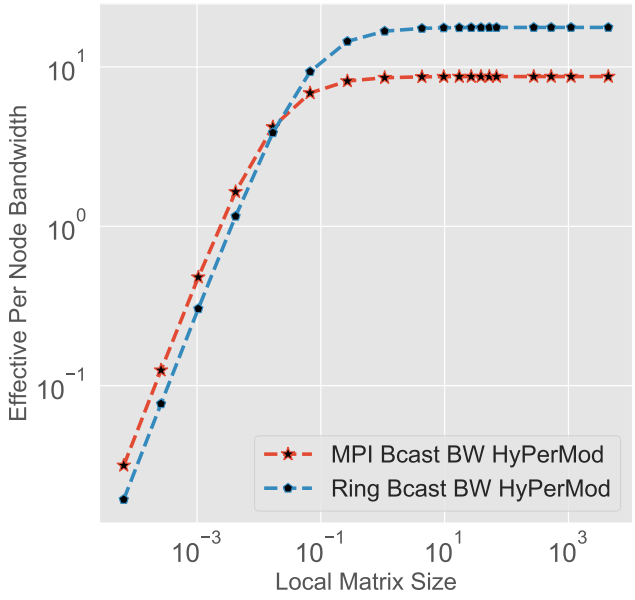


Figure 16: Performance of boradcast in terms of bandwidth on various local matrix size

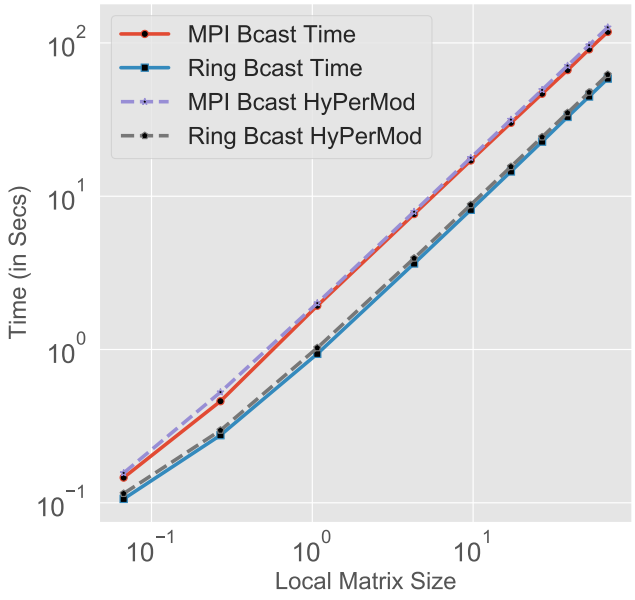


Figure 17: Performance of boradcast in terms of time spent on various local matrix size

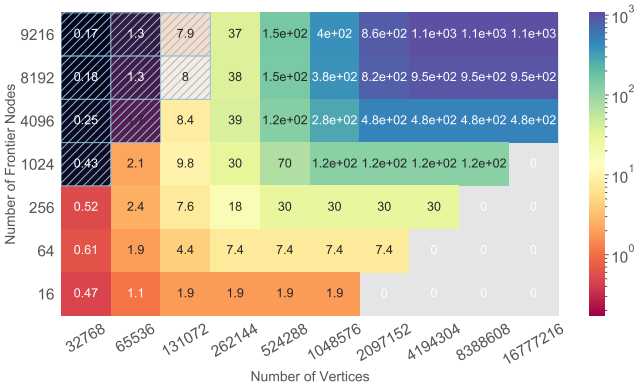


Figure 18: Cost performance prediction heatmap

6 DISCUSSION

In this section, we elaborate on several challenges we faced during the implementations.

6.1 Compute and Communication Overlapping

During the implementation of a look-ahead optimization, We designed three ways to overlap the computation and communication, see fig 23 for an HPL-MxP example for the diagonal process. Note that the last GEMM update does not remain of fixed size through out the run.

One simple way is to make use of a non-blocking broadcast to allow kernels with independent data to proceed with computation. However, this encounters several issues described in next subsection. The second way is to make use of a non-blocking GEMM kernel, using GPU_Device_synchronized to control the dependencies. However, by doing this, we are forcing the broadcast to only be overlapped with one chunk of data synchronization. With this strategy, the best performance we achieve was overlapping both communication with the GEMM_Update. Finally, for best programmability and maximum overlapping of compute and communication, we used event based synchronization to control the dependencies. Many current accelerators use the stream parameter to handle the data dependency, we suggest that maybe if the MPI interface included the stream parameterfor handling data dependency thatthe overlapping of processes will be easier.

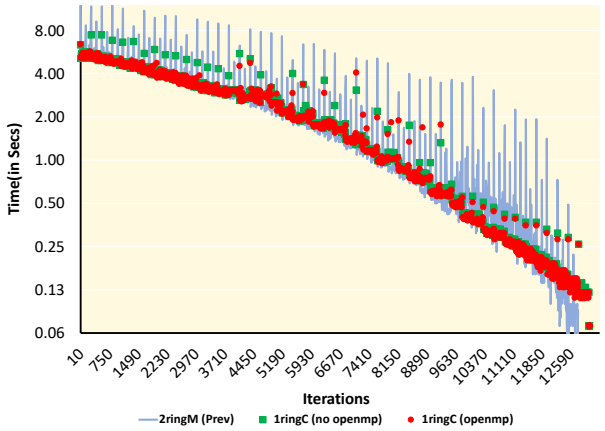


Figure 19: Total runtime breakout for 272x272 of HPL-MxP frontier.

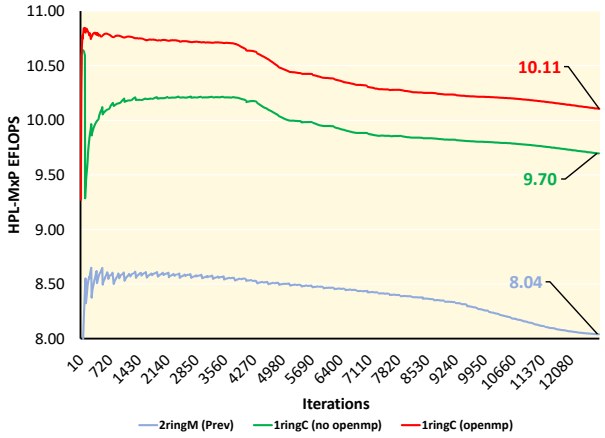


Figure 20: Floprate of full HPL-MxP frontier run. Omitting IR.

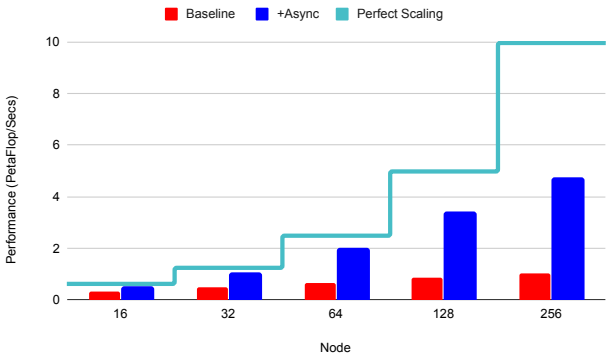


Figure 21: apsp strong scaling on summit

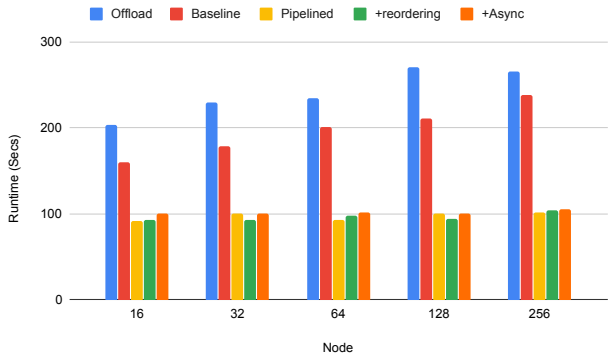


Figure 22: apsp weak scaling on summit modifiy this one to keep baseline and async

6.2 Underlying MPI Implementation

During the development of the cross platform APSP and HPL-MxP, we involved two different implementation of MPI library, namely Spectrum-MPI (developed by IBM) and Cray-mpich (developed by HPE). Our application performance was significantly impacted by the different behavior of each of the MPI. During the experiments when using non-blocking communication on Summit (MPI_Icast, MPI_Isend, MPI_Irecv), we observed a significant performance reduction on the bandwidth, see Fig 6. In addition, when we activate GPU-awareness for Spectrum-MPI, allowing NICs direct access to GPU memory, we observe a unexpected behavior for Spetrum_MPI_Bcast and Spetrum_MPI_Ibcast. The library broadcast appears to synchronize the device before invoking the broadcast, causing our overlapping strategies that depend on non-blocking GPU kernels to fail.

On the other hand, when we experiment the tuning of QR QC on Frontier, we did not observe significant improvement over the default setting. We are suspecting the implementation of Cray-mpich did not fully utilize the underlying GPU-links.

6.3 Variation in runs

During the process of gathering data for Frontier system network, we experienced random hangs or delays on communication, see fig **??**. At a user level, MPI will detect a timeout but we could not identify the cause of these spikes. We believe this could be due to slingshot routing table redistribution of new dynamic routes.

We would like to design a broadcast algorithm that is resilience for large network topologies experiencing transient issues.

7 CONCLUSION

REFERENCES

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. 180, 1 (2009), 012037.

[2] Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2020. Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs. In *2020 IEEE High Performance Extreme Computing Conference*. 1–7. https://doi.org/10.1109/HPEC43674.2020.9286214

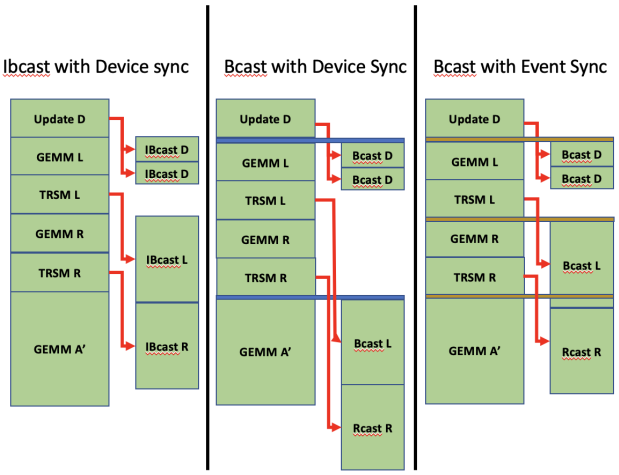


Figure 23: Discussion of better programability of MPI

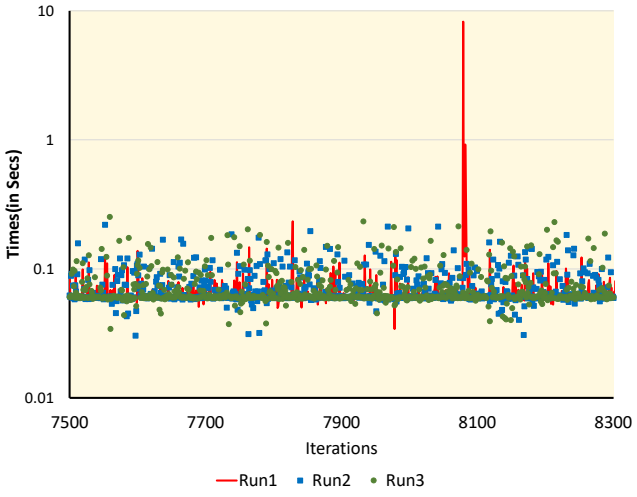


Figure 24: Discussion of network variance: shows the communication time per iteration recorded on the last rank using 2809 nodes for $P_r = P_c = 212$ and N =6946816. The time taken on the last 700 iterations with different chunk sizes and the variations among three different runs under same settings are shown in, respectively.

[3] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. 120–121.

[4] Jeremy T. Fineman and Eric Robinson. 2011. Fundamental graph algorithms. In *Graph Algorithms in the Language of Linear Algebra*, Jeremy Kepner and John Gilbert (Eds.). Society of Industrial and Applied Mathematics, Philadelphia, PA, USA, Chapter 5, 45–58.

[5] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. 1990. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Rev.* 32, 1 (Mar. 1990), 54–135. https://doi.org/10.1137/1032002

[6] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a modern distributed and accelerated linear algebra library. In *SC19:Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.

[7] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2020. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476, 2243 (2020), 20200110. https://doi.org/10.1098/rspa.2020.0110

[8] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2018. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 603–613. https://doi.org/10.1109/SC.2018.00050

[9] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics, USA.

[10] ICL. [n. d.]. HPL-AI Mixed-Precision Benchmark. Accessed Aug. 1, 2021. https://hpl-ai.org/

[11] Shuhei Kudo, Keigo Nitadori, Takuya Ina, and Toshiyuki Imamura. 2020. Implementation and Numerical Techniques for One EFlop/s HPL-AI Benchmark on Fugaku. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 69–76. https://doi.org/10.1109/ScalA51936.2020.00014

[12] Jakub Kurzak and Jack Dongarra. 2006. Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the CELL Processor. *University of Tennessee Computer Science Tech Report* UT-CS-06-580, LAPACK Working Note #177 (2006).

[13] Wang Lei, Zhang Yunquan, Zhang Xianyi, and Liu Fangfang. 2010. Accelerating Linpack Performance with Mixed Precision Algorithm on CPU+GPGPU Heterogeneous Cluster. In *2010 10th IEEE International Conference on Computer and Information Technology*. 1169–1174. https://doi.org/10.1109/CIT.2010.212

[14] Hao Lu, Michael Matheson, Vladyslav Oles, Austin Ellis, Wayne Joubert, and Feiyi Wang. 2022. Climbing the Summit and Pushing the Frontier of Mixed Precision Benchmarks at Extreme Scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41404.2022.00083

[15] RIKEN-RCCS. [n. d.]. HPL-AI implementation for Fugaku. Accessed Apr. 21, 2021. https://github.com/RIKEN-RCCS/hpl-ai

[16] Robert Schreiber. 1988. Block Algorithms for Parallel Machines. In *Numerical Algorithms for Modern Parallel Computer Architectures*, Martin Schultz (Ed.). Springer US, New York, NY, 197–207.

[17] Gilbert Strang. 2016. *Introduction to Linear Algebra* (5 ed.). Wellesley-Cambridge Press, Wellesley, MA.

[18] James H. Wilkinson. 1994. *Rounding Errors in Algebraic Processes*. Dover Publications, Inc., USA.