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

The new reality of exascale computing faces many challenges in achieving optimal performance on large numbers of nodes. A key challenge is the efficient utilization of the message-passing interface (MPI), a critical component for process communication. This paper explores communication optimization strategies to harness the GPU-accelerated architectures of these supercomputers. We focus on MPI applications where processors form a two-dimensional process grid, a common arrangement in applications involving dense matrix operations. This configuration offers a unique opportunity to implement innovative strategies to improve performance and maintain effective load distribution. We study two applications— Dist-FW (Apsp:all-pair-shortest-path) and HPL-MxP (LU factorization with Mixed precision)—on two accelerated systems: Summit (IBM Power and NVIDIA V100) and Frontier (AMD EPYC and MI250X). These supercomputers are operated by the Oak Ridge Leadership Computing Facility (OLCF) and are currently ranked #1 and #5 on the Top500 list. 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 an unprecedented scale. Accurately predicting application performance becomes crucial for cost reduction as the computational 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 EuroMPI2023: the 30th European MPI Users' Group Meeting (EUROMPI '23), September 11–13, 2023, Bristol, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3615318.3615327

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE)..

EUROMPI '23, September 11–13, 2023, Bristol, United Kingdom

1 INTRODUCTION

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. One of the key aspects is the Message Passing Interface (MPI) which plays a crucial role in process communication. This paper focuses on optimizing applications where processors form a two-dimensional process grid, with communication optimization as the main target. We delve into two specific applications — the distributed memory-Floyd Warshall (Dist-FW) algorithm [9] for computing all-pair shortest paths (Apsp) and the High-performance Linpack in mixed precision (HPL-MxP) [18] benchmark.

The distributed-memory Floyd Warshall algorithm calculates the shortest paths between all pairs of vertices in a graph. By using semi-ring multiplication operations, it can efficiently compute the shortest paths, particularly for large-scale graphs. These operations resemble mathematical rings but without the need for additive inverses. Meanwhile, the High-Performance mixed-precision LINPACK (HPL-MxP) benchmark measures the performance of computer systems in solving dense systems of linear equations using LU decomposition. HPL-MxP is a variant of High-Performance LINPACK (HPL) that focuses its functionality mainly on mixed precision. This approximation allows it to emulate the lower precision typical of AI-like workloads.

For clarity, we refer to the distributed mixed-precision LU solve as HPL-MxP and the distributed Floyd Warshall algorithm as Dist-FW. The implementation of HPL-MxP is called OpenMxP [27], and the application of Dist-FW is called Dsnapshot [32]. Both of these algorithms, herein referred to as Dist-FW and HPL-MxP, respectively, encounter common communication optimization issues. They require row and column broadcasts at every iteration, typically incorporating a look-ahead strategy to overlap computation with communication. Overcoming the hurdle of optimizing this communication, alongside efficiently mapping computation within a two-dimensional grid, is a focal point of this study.

Furthermore, the use of graphics processing unit (GPU) accelerators in systems 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 overlap of communication with computation. The OLCF systems have two memory subsystems—one attached to the CPU and one to a GPU. Data locality tied with communications

plays an important role in communication performance, which directly impacts application performance.

Our paper directly addresses these challenges to increase the efficiency of communication operations for 2D partitioned applications. We achieve this by combining communication with computation and aligning the architecture. We thoroughly evaluate various broadcast algorithms and summarize their performance characteristics. Our proposed "chunked 2D broadcast" algorithm blends row and column broadcasts, enhancing inter-process communication performance and ensuring a balanced distribution of workload. As demonstrated in OpenMxP and Dsnapshot, this approach significantly improves performance. The latter nearly reaches one exaflop, while the former sets a new record, nearing 10 exaflops. Additionally, we recommend a hyperbolic model to meet the urgent need for accurate performance prediction and reduce costs at large computational scales. This model serves as an alternative to the traditional one-sided asymptotic model, simplifying the optimization process, reducing the need for large-scale runs, and providing a more efficient way to predict performance at exascale levels.

This study offers more than specific solutions. We pinpoint several problems and propose useful MPI features that could make optimization strategies more effective. We also evaluate and contrast various communication strategies at an unparalleled scale, providing insights beneficial for a broader range of applications. Therefore, this work is a crucial step in advancing performance capabilities at the exascale level, implying broader implications for the future of high-performance computing. Building on this work, our future research aims to further improve communication optimization and develop a resilient communication library to handle network failures or delays.

2 BACKGROUND

2.1 System Information

Our study focuses on two different architectures of supercomputers hosted at OLCF: Frontier and Summit. A brief overview of the key architectural specifications and software stacks can be found in Table 1c. The key difference we would like to point out is the location of the network interconnect (NICs). In the Frontier system, the NICs reside on the GPUs, which eliminates the traditional overhead of copying network data back to the CPU. On the other hand, the Summit system contains two sockets per node, with each socket having one NIC. The node architectures are shown in Figures 1a and 1b. Another significant difference is the ratio of graphics compute dies (GCDs) to GPUs. Summit has one GCD per GPU, while the Frontier system has two GCDs per GPU. Each GCD is utilized as a separate accelerator.

2.2 HPL-MxP

The HPL-MxP benchmark was designed to evaluate the mixed precision capability of the system 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 to the HPL benchmark, HPL-MxP allows the input matrix to have an appropriate condition number for omitting the pivoting step during the LU factorization process [17, Chapter 9]. More importantly, it permits 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 with three specific procedures. Initially, the matrix is transformed into an estimated triangular form using mixed precision block Gaussian elimination [35]. Once the estimated LU factorization of $A \cong \tilde{L}\tilde{U}$ has been obtained, the estimated solution $\tilde{x}$ to $Ax = b$ can be efficiently calculated by solving the two triangular systems of linear equations ($\tilde{L}\tilde{U}\tilde{x} = b$). Finally, the solution is further refined to attain FP64 precision accuracy ($b - A\tilde{x} < \epsilon$, where ϵ is FP64 machine epsilon) by applying iterative refinement (IR) [36].

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. [13, 33]. 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 2

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 2.

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

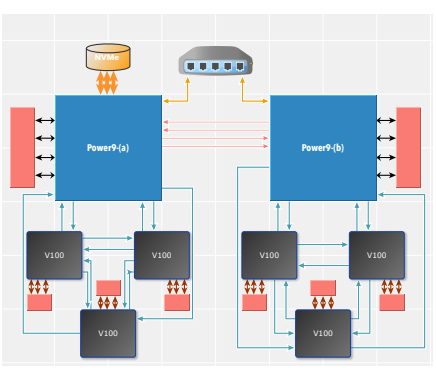
Further expanding, LU factorization can be viewed as solving for $L_{1,1}$, $L_{1,2}$, $U_{1,1}$ and $U_{2,1}$, then update the $A_{2,2}$, see part (c) of Figure 2. The high-level algorithmic steps for block-based LU factorization at step k could be represent as the following:

(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 $A^{(k+1)} = L_{2,2}U_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$.

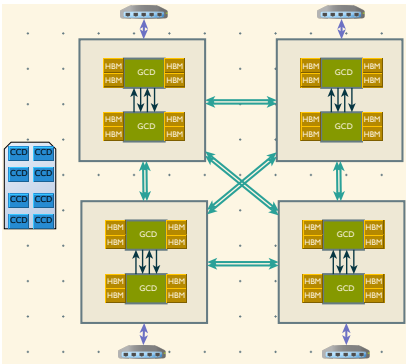
Iterative refinement Even when A is well-conditioned, computing its LU factorization suffers from precision limitations of floating point arithmetic, especially when working in mixed precision. As a result, the mixed precision LU factorization $A \rightarrow \tilde{L}\tilde{U}$ holds only an approximation 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 purposes



(a)

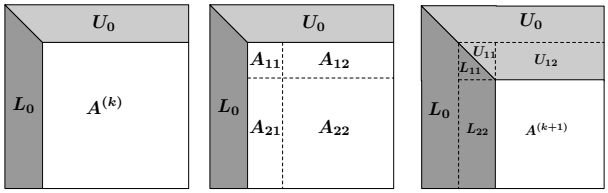


(b)

	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

(c)

Figure 1: (a) 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). (b) Frontier Node architecture: Bandwidth for connection: 50 GB/S GPU to GPU (green), 50 GB/S GPU to NICs (blue). (c) Key architectural and software stack specifications for Summit and Frontier.



(a) (b) (c)

Figure 2: (a) The kth step of block LU factorization. (b) Partitioning of the trailing matrix. (c) Outcome of the kth step.

the IR is stopped once the solution discrepancy $\tilde{d}$ gets below some ration of FP64 machine epsilon. The IR procedure does not require significant run time and will be omitted from this paper.

Related Work In 2006 Kurzak and Dongarra [24] 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 [25]. Dense linear algebra library ScaLAPACK [6] which developed in 1992 supported distributed computing. In 2009, MAGMA library [1],[3] enabled the GPU support for BLAS. In 2017, Haidar et al. added mixed precision variants to MAGMA [15, 16]. In 2019, a library called SLATE [14] expand the capability to multiple precision with distributed multi-GPU support. However, these libraries focused on the portability and usability and did not attempt to achieve the maximum performance of target system. On the CPU front, the Fugaku HPL-MxP code [22, 30] is the first to break the exascale barrier in 2020 with a CPU-only implementation.

The implementation being modified in our paper is from the OLCF [27] and is called OpenMxP, which is the first code that obtains almost 8 exaflops on the Frontier system. This work is heavily influenced by prior efforts and uses OpenMxP as a baseline but makes further communication and performance optimizations to achieve almost 10 exaflops.

2.3 Floyd-Warshall

Our second application focuses on the calculation of all pair-shortest paths (Apsp).

Classical Floyd-Warshall Algorithm: In simpler terms, Apsp calculation is the process of determining the shortest paths between all pairs of vertices in a weighted graph, where the graph is denoted as $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. We can have negative weights, but we avoid cycles where the total weight is negative. In cases where the graph is dense—meaning $m = O(n^2)$—we can employ the Floyd-Warshall algorithm. This algorithm involves three nested loops, each iterating over all vertices. As a result, its sequential complexity amounts to $O(n^3)$ operations.

The Floyd-Warshall algorithm works by updating a matrix, $Dist_{ij}$, which stores the shortest path length between any two vertices v_i and v_j. If there's no discovered path, this distance initializes to ∞. The algorithm ensures that $Dist_{ij}$ is minimum with at most k vertices acting as intermediaries at each iteration k. Thus, the Floyd-Warshall algorithm uncovers more paths between vertices, reducing the number of ∞ $Dist_{ij}$ entries.

Semi-ring Notation: Apsp calculation can also be seen from an algebraic lens. Apsp may be understood algebraically as computing the matrix closure of the weight matrix, W, defined over the tropical semiring [11]. 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

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

This interpretation of the Min-Plus product helps to understand the following blocked version of distirbuted Floyd-Warshall algorithm. (Algorithm 2).

Blocked Floyd-Warshall algorithm

The Blocked Floyd-Warshall algorithm, an optimized variant of the classic Floyd-Warshall algorithm, significantly improves cache efficiency and performance by subdividing the distance matrix into smaller blocks. The method takes as input a distance matrix A and a block size b.

The distance matrix A of dimension n, is divided into $n_b = \frac{n}{b}$ blocks, each of size $b \times b$. These blocks will be the fundamental

units of operation within the algorithm. The algorithm proceeds with three core steps, iteratively applied for each block along the diagonal of the matrix.

- **Diagonal Update:** traditional Floyd-Warshall algorithm is applied to the k^{th} diagonal block of the matrix. This results in an updated block, $A(k,k)$, which stores the shortest path lengths between its represented vertices.
- **Panel Update:** This step updates the k^{th} block row and column. For all blocks in the k^{th} block row, $A(k,j)$, we apply the in-place Min-Plus multiply with $A(k,k)$ from the left. Similarly, all blocks in the k^{th} block column, $A(i,k)$, undergo the same operation but from the right. This refreshes the distances within these blocks with the new distances from the k^{th} diagonal block.
- **Min-Plus Outer Product:** It involves the outer product of the k^{th} block row and block column and updates each block $A(i,j)$ through a Min-Plus operation. The updated value is the result of the operation between its present value and the Min-Plus product of block $A(i,k)$ and block $A(k,j)$.

Through these iterative steps, the Blocked Floyd-Warshall algorithm efficiently discovers the shortest paths in a blocked fashion, leveraging the locality of reference to improve cache efficiency and overall computational performance.

Related Work. In 1987, Jenq and Sahni [19] developed the first 2D distributed memory algorithm for the Apsp. In 1991, Kumar & Singh [23] analyzed the different Apsp algorithms that developed the compute-communication overlapping scheme without blocking. Besides overlapping, Solomonik et al. proposed a communication-avoiding parallel Apsp which uses 2.5D process grid [34]. The first distributed GPU Apsp showed good performance for smaller clusters [9], but the centralized communication scheme limits the scalability beyond 64 GPUs. In 2020, Sao et al. [31] first used optimized semi-ring matrix multiplication as a building block for computing the Floyd-Warshall algorithm. In 2021, Sao et al. [32] demonstrated the first scalable GPU Dist-FW. This algorithm was used in Dsnapshot [20] and Coast [21].

Besides Dist-FW, a different effort to define graph algorithms in terms of linear algebra was initiated by the GraphBlas Forum [12]. This development motivates approaches combining graph algorithms with the MPI runtime, such as CombBLAS [4] and SuiteSparse [7]. However, these libraries mainly focus operations on sparse data set and portability. There has not been an effort focused on performance on massive data sets.

2.4 2D Block-cyclic data distribution

The 2D block-cyclic data distribution scheme splits a matrix A into blocks, (i,j). Each block stands for a submatrix $A[ib:(i{+}1)b, jb:(j{+}1)b]$ with a block size b. In some applications such as, Sparse direct solvers, the block size can beuneven. Specific processes receive block assignments (i,j) based on row and column indices $P_r(i)$ and $P_c(j)$.

This scheme presents numerous benefits. It guarantees data ownership, meaning it is always clear which process is responsible for which submatrix. Consequently, the location of the data is always known. Additionally, it establishes the owner update policy, meaning that only the owner process updates a block, thereby eliminating coordination among multiple processes. The scheme achieves load balancing via fine-grained blocked partitioning, which prevents imbalances when updating parts of the matrix. Random submatrices of A usually distribute evenly among processes, supporting 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 mention alternative data distribution strategies and their development. These include 2D data distribution without block-cyclic characteristics, Partitioned Global Address Space [8], and tiled block-cyclic distribution [14]. 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) [10], Co-Array Fortran (CAF) [28], and Chapel [5]. PGAS is particularly useful when dynamic load balancing is required [26]. Each alternative differs in how the local submatrix is stored locally and frameworks such as Slate support various options. Many of the optimizations readily apply to when the local storage is in a different format.

2.5 Distributed Implementation

For the distributed version of HPL-MxP and Dist-FW algorithms, we partition the global matrix A into 2D Block-cyclic data distribution described in section 2.4 and implemented the numerical steps in section 2.2 and 2.3. For HPL-MxP, we used accelerator-specific BLAS (Basic Linear Subprograms) and additional native-implemented casting kernels for precision conversion. The kernels we used are SGETRF_nopivot (single precision LU factorization), STRSM (single precision triangular solve) and GEMM_ex (mixed precision general matrix matrix multiplication) from AMD rocBLAS [2] and Nivida cuBLAS [29], For Dist-FW, we implemented the required SEMIRING-GEMM kernel. Pseudo code is provided in Algorithm 1 and Algorithm 2. The two implementations are called OpenMxP and Dsnapshot.

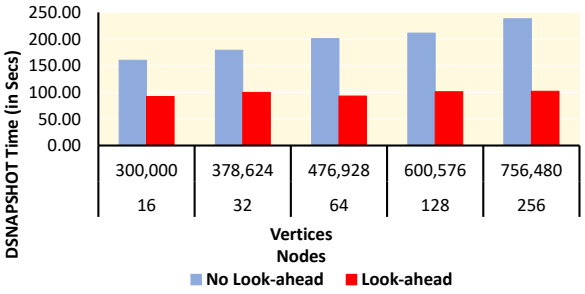


Figure 3: Dsnapshot weak scaling on Summit for overlapping compute and communication.

Algorithm 1 Distributed mixed Precision LU on 2D process grid

Input: N, B, P_r, P_c
Fill global matrix A with random numbers.
On each MPI process p_{id} do in parallel:
for $k = 1,\ 2,\ 3 \ldots\ n_b$ **do**
  $P_{ir}, P_{ic} \leftarrow processmapping(k)$ // $A_{k,k}$ owner process index
  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 $P(:, P_{ic})$
  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
  $A(k+1:n, k+1:n) \leftarrow$ ***GEMM***$(L, U, A(k+1:n, k+1:n))$
IR omitted

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

function ParallelFW($A, P = P_r \times P_c$) ▷ A is distributed in block-cyclic fashion ▷ my process Id is pid
  for all MPI process pid in parallel **do**
    for $k \in \{1, 2, \ldots, n_b\}$ **do**
      if $pid = p_{k,k}$ **then**
        $A(k,k) \leftarrow$ Floyd-Warshall$(A(k,k))$
        Broadcast$(A(k,k), P_r(k))$
        Broadcast$(A(k,k), P_c(k))$
      if $pid \in P_r(k)$ **then**
        Receive$(A(k,k), p_{k,k})$
        $A(k,:) \leftarrow A(k,:) \oplus A(k,k) \otimes A(k,:)$
        Broadcast$(A(k,:), P_c(pid))$
      else
        Receive$(A(k,:))$
      if $pid \in P_c(c)$ **then**
        Receive$(A(k,k), p_{k,k})$
        $A(:,k) \leftarrow A(:,k) \oplus A(:,k) \otimes A(k,k)$
        Broadcast$(A(:,k), P_r(pid))$
      else
        Receive$(A(k,:))$
      for $i \in \{1, 2, \ldots, n_b\}$ **do**
        for $j \in \{1, 2, \ldots, n_b\}$ **do**
          if pid owns $A(i,j)$ **then**
            $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$

3 OPTIMIZATION

3.1 Overlapping Communication with Computation

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, overlapping communication with computation via look-ahead optimization minimizes delays and maximizes resource use, leading to improved performance in distributed computing systems. In Fig 3, we show the effect of look-ahead using weak scaling of Dsnapshot on Summit as an example. We will consider the look-ahead is applied to all runs for the following paper.

3.2 Mapping 2D Partitioned Algorithm to Architecture

To enhance performance in distributed computing systems, we must effectively adapt the computation to the architecture. We achieve this optimization through two strategies: increasing bandwidth and reducing latency. Increasing bandwidth boosts the rate of data transfer between nodes over time. This strategy necessitates a solid understanding of constraints like network topology, communication protocols, and hardware limitations specific to the application. Similarly, reducing latency, or the time data takes to travel between nodes, requires the same depth of knowledge regarding network topology and hardware constraints.

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 NVME. 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 NICs 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. The default method of assigning rank can lead to poor performance as all the ranks on the node are consecutive in one direction, often resulting in an unbalanced grid structure when partitioning into a 2D grid. In Figure 4a, 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 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.

System Specific Features. Communication 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.

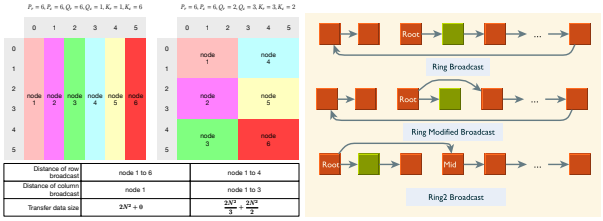
A Summit node contains two NICs that are each connected to a socket. One of the key features it provides is to bind the two NICs to serve one send/recv by stripping and relaying data through the CPU-CPU connection. This binding feature increase the effective bandwidth for most bandwidth bound problems. Given HPL-MxP and Dist-FW are both dense matrix operations with 2D Block-cyclic data distribution, which consist of bandwidth bounded communication, binding features would significantly improve the application performance.

Different from Summit, Frontier NICs are directly connecting to GPU instead of CPU. GPU-aware MPI is highly suggested to be used to remove data transfer overhead for distributed GPU applications. This prompted to select the residency of input matrix A on GPU and use GPU-aware MPI to achieve maximum performance.

We highlight the advantage of different combinations of Q_r and Q_c with architecture specific features in Figure 5. The performance was measured using OpenMxP and the percentage improvement of floating points operations per second (FLOPS) over the default setting. 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/grid arrangement, leading to substantial improvements in communication efficiency.

3.3 Optimizing Broadcast Operations

In the HPL-MxP 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.



(a) (b)

Figure 4: (a) Example of different node local grid, and the metric that impact the communication time. (b) Different Variant of Ring Broadcast used in our experimentation

Summit P=2916	None Binding	Binding		
	Qr=6 Qc=1	Qr=6 Qc=1	Qr=3 Qc=2	Qr=2 Qc=3
IBcast	25.67%	46.15%	49.92%	49.48%
Bcast	100.00%	135.79%	**155.00%**	141.79%
1r	81.68%	131.15%	135.69%	132.66%
2rM	81.37%	131.18%	135.93%	132.61%
Frontier P=1024	**No GPU-aware**	**GPU-aware MPI**		
	Qr=8 Qc=1	Qr=8 Qc=1	Qr=4 Qc=2	Qr=2 Qc=4
Bcast	100.00%	156.61%	163.81%	163.29%
1r	128.07%	182.28%	192.39%	193.59%
2rM	134.42%	189.58%	**194.61%**	194.45%

Figure 5: OpenMxP performance improvement over the baseline for different grid and platform specific features. Baseline are marked in green, best improvement are marked in red. 1r/2rM stand for 1-ring /2-rings-Modified broadcast.

Tree-based broadcast algorithms, such as binary trees and Fibonacci trees, are commonly used in distributed computing systems to achieve efficient data communication among nodes. However, for 2D partitioned matrix applications like HPL-MxP 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 different number of rings (see Fig. 4b). The number indicate the number of rings that execute concurrently. The selection of the appropriate broadcasting algorithm depends on the constraints of the network topology and performance requirements.

Chunked 2D broadcast. Our proposed algorithm, *Chunked Ring Broadcast*(Alg 3), adapts Ring broadcast to mitigate latency costs linked with large message transmissions. This algorithm divides the original message into approximately equal parts, or 'chunks' of a size termed *CSize*. It conducts two simultaneous broadcasts, one each along the process row and column.

If a process belongs to the root row or column, it initiates the relaying of message chunks via the 'relay' function. The 'relay' function receives a chunk and passes it to the succeeding process in the ring. This procedure continues, with the algorithm monitoring the relayed chunks for both row and column broadcasts, denoted by variables i_r and j_c.

Subsequently, the algorithm enters a loop to check for the complete relay of all chunks. If chunks are pending for relay, it confirms their receipt. Upon receiving a chunk, the 'relay' function triggers

Algorithm 3 Chunked Broadcast Algorithm

procedure CHUNKED2DBROADCAST
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
  $i_r = 0$, $j_c = 0$
  for each process (px, py) in parallel **do**
    if I am in the root row **then**
      for $i \leftarrow 0$ to $N_r - 1$ **do**
        relay(P_r, R_r, $Buf_r[i]$)
      $i_r \leftarrow N_r$
    if I am in the root column **then**
      for $j \leftarrow 0$ to $N_c - 1$ **do**
        relay(P_c, R_c, $Buf_c[j]$)
      $j_c \leftarrow N_c$
  while $i_r \neq N_r$ or $j_c \neq N_c$ **do**
    if $i_r \neq N_r$ **then**
      check receive(i_r)
      if received **then**
        relay(i_r)
        $i_r \leftarrow i_r + 1$
    if $j_c \neq N_c$ **then**
      check receive(j_c)
      if received **then**
        relay(j_c)
        $j_c \leftarrow j_c + 1$
end procedure

to advance it to the following process in the ring, and the corresponding counter (i_r or j_c) increments.

The chunking mechanism actively pipelines broadcasts for row and column communicators, reducing RING broadcast latency and tolerating network fluctuations. The granularity of chunked transmission lets chunks fail and recover independently, minimizing the influence of network issues on overall message transmission. Empirical evidence shows that this chunked broadcast implementation outperforms others (Fig. 7b), providing greater resilience against unexpected network behaviors. It smoothly manages network faults, but further research must identify the origins of these faults and the reasons for this algorithm's increased resilience.

Discussion of various experiments. The results and data presented bring into focus the varying efficiencies of different broadcast algorithms in the context of the HPL-MxP and DIST-FW algorithms. A broad range of communication patterns and their effects on workloads were studied on Frontier. We denote 1-RING as 1r, 2-RINGS as 2r, modified RING as rM and chunked version with C at the end.

We focus on only the first 1200 iterations for the DIST-FW workload because its communication workload stays constant throughout the run. In Figure 6a the 2-RINGS-MODIFIED-CHUNKED version of the RING broadcast surpasses most other communication methods for DIST-FW. It does not produce the lowest per iteration runtime but produce the highest overall performance. This suggests that using the chunked technique can effectively reduce the high latency costs commonly linked to broadcasts.

On the other hand, 1-RING-CHUNKED method shone promise in the HPL-MxP context, as Figure 6b shows. This variant seemed to align well with the decreasing workload of HPL-MxP over the iterations and performed better than other RING implementations.

Process 0 provided the perspective for the reported runtimes measurements. We noted a consistent rise and fall in both workloads in the runtime. Process 0's position at the ring's tail caused the increase, while its location at the source resulted in the fall. The figure illustrates that the rise halves from a 1-RING to a 2-RINGS due to the ring's shorter length. Additionally, the integration of OpenMP had a notable impact. We modified Algorithm 3 to utilize separate OpenMP threads for each communication direction (Row/Column). This modification substantially alleviated congestion during 2D grid communication, especially when two messages arrived simultaneously (Fig 6c). By effectively managing this common scenario, we reduced communication time considerably. Therefore, incorporating OpenMP proved beneficial by decreasing communication time and enhancing the broadcast algorithm's performance.

The effect of different chunk sizes on the communication was also investigated (Fig 7a). Surprisingly, a chunk size of 2 MB appeared to outperform other sizes on the Frontier fabric. This provides valuable insights for future broadcast implementations on this system, suggesting an optimal chunk size for minimizing latency and maximizing efficiency. We measured the bandwidth usage for a full HPL-MxP workload (Fig 7b). Notably, 1-RING-CHUNKED with OpenMP shows superior performance as the message size increases. An interesting trend appeared in the average bandwidth across varying data sizes during an HPL-MxP workload run (Fig 7c). Even as the number of nodes doubled, the average bandwidth consistently stayed between 95-98%. This suggests that the RING broadcast algorithms, particularly the chunked versions, efficiently reduce latency and preserve bandwidth.

In summary, our findings emphasize the promise of the RING broadcast algorithms, particularly their chunked version, for 2D partitioned matrix applications like HPL-MxP and DIST-FW. Utilizing pipelining and chunking techniques and multi-threading solutions like OpenMP are powerful methods for enhancing communication efficiency in distributed computing systems. Further exploration of these advanced broadcasts' resilient behavior, especially against unpredictable network conditions, is a valuable focus for future research.

4 APPLICATION PERFORMANCE

To evaluate the use of the optimization from section 3, we attempted the close-to-full scale runs on Frontier to evaluate the improvement.

The full scale **DIST-FW** adding chunked communication was run on Frontier. In Fig 8a, we observe only a slight improvement in application performance on chunked and the original 2-RINGS broadcast, as DIST-FW is primarily a computed bound problem at this scale. However, the new broadcast suffers significantly less from frequent network jitters than the original.

The comprehensive **OPENMXP** program runs on Frontier and includes LU factorization and iterative refinement (IR). However, because IR only accounts for 1-2% of the total execution time, we will mainly focus on the LU factorization results. As shown in Fig 8b, we compared the OPENMXP's runtime per iteration

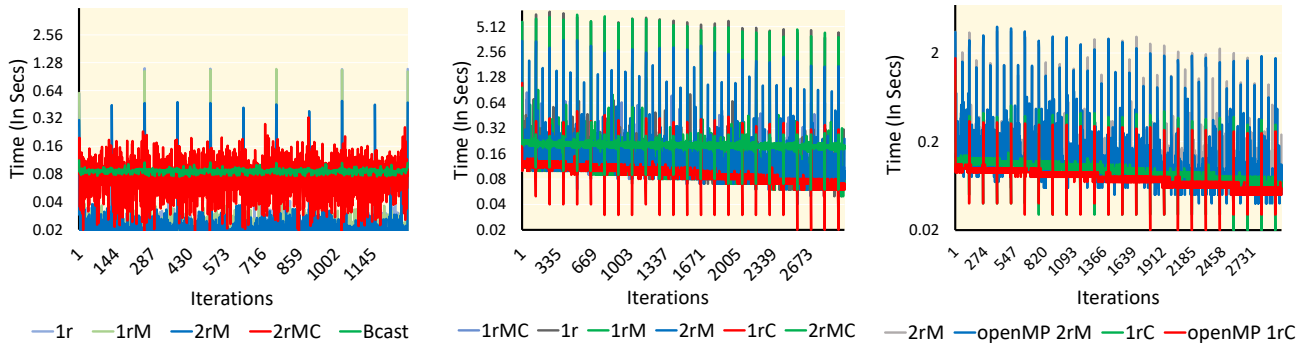


Figure 6: (a) Per iteration broadcast time of Dist-FW communication on 2,809 Frontier nodes ($P_r = P_c$ = 212) and N = 6,946,816 for different broadcasts. (b) Per iteration broadcast time of HPL-MxP communication on 2,048 Frontier nodes ($P_r = P_c$ = 128) and N = 16.05M for different Ring broadcast. (c) Per iteration broadcast time of (b) with openMP. We color the broadcast based on the rank of overall performance. (1st Red, 2nd Green, and 3rd Blue)

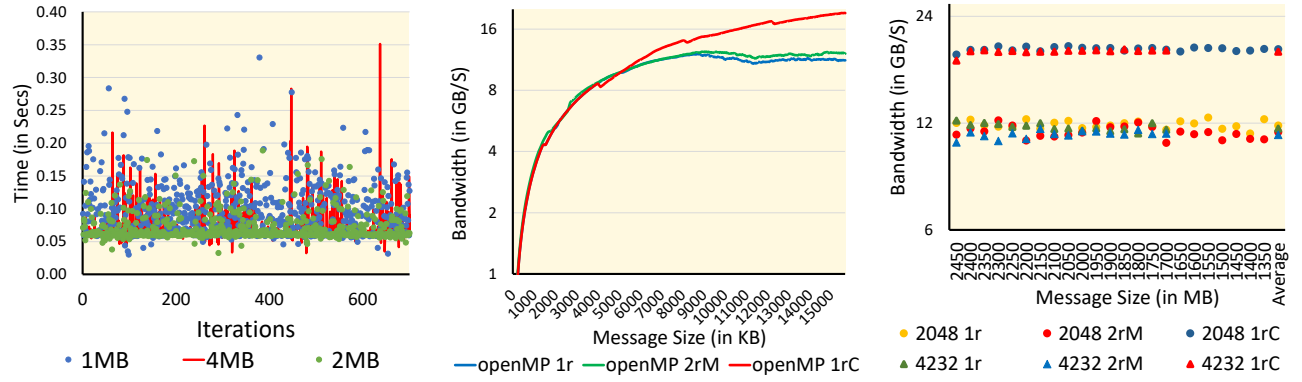


Figure 7: (a) Per iteration broadcast time of Dist-FW communication on 2,809 Frontier nodes ($P_r = P_c$ = 212) and N = 6,946,816 for different chunk size. (b) Average bandwidth of various HPL-MxP communication size on 2,048 Frontier nodes. (c) Average bandwidth of HPL-MxP communication on 2,048 and 4,232 Frontier nodes ($P_r = P_c$ = 128 and 184) for different Ring and size.

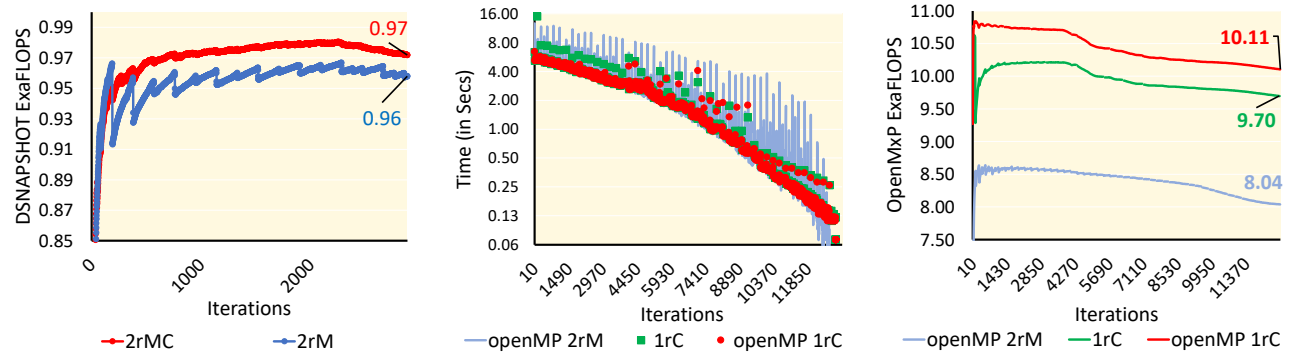


Figure 8: (a) Sustained Flops of Dsnapshot on 9,025 nodes ($P_r = P_c$ = 380) and N = 7.5M. (b) Per iteration runtime of OpenMxP on 9,248 nodes ($P_r = P_c$ = 272) and N = 34.24M for Ring broadcasts. (c) Sustained Flops of (b). Omitting IR.

with the previous Frontier HPL-MxP ranking. We performed these runs using ROCM/5.4.3 and replaced device_synchronization with hipEvent_synchronization in the new 1-ring-Chunked setup. This change resulted in the shortest per iteration runtime. Fig 8c reports the sustained exaFlops performance of all three runs. We found that the 2-rings-Modified model failed to maintain the ROCM improvements at scale due to its high communication overhead. However, with the use of 1-ring-Chunked communication, OpenMxP managed to achieve nearly **10 ExaFlops** in the mixed precision LU solution - marking *a first in this field*. Both Dist-FW and OpenMxP demonstrated that chunked variants experienced fewer disruptions from unexpected network timeouts than their non-chunked counterparts, a factor that was crucial for reaching these performance benchmarks.

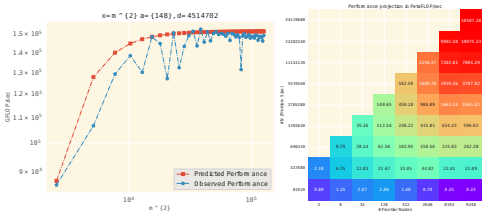
5 PERFORMANCE MODEL

5.1 Motivation

Optimizing high-performance computing (HPC) applications relies heavily on effective performance modeling. This approach allows us to set realistic performance goals, identify and resolve performance issues, and steer optimization strategies. Yet, performance modeling presents several obstacles:

- The performance of different implementations for various routines depends on operand sizes.
- Fluctuations in operand sizes, such as those in HPL, complicate performance prediction.
- As problem scale changes, so do operand sizes across scales, adding difficulty to performance prediction.
- Developing analytical models for opaque components, like closed-source libraries, is challenging.
- Analytical models, while useful, often display a discrepancy between theoretical and observed performance due to the abstraction of hardware architecture details.

Addressing these challenges, we aim to create composable performance models. These models will maintain accuracy across various operations and accommodate the limitations as mentioned earlier. Our proposed solution centers on the use of hyperbolic performance models. This approach will provide a more accurate, generalizable HPC application performance modeling method.



(a) (b)

Figure 9: (a) Predicted vs observed performance of GEMM_ex. (b) OpenMxP performance prediction with combined models

5.2 Hyper Model

Predicting performance accurately in parallel applications is a significant challenge. We need a versatile model that can handle a variety of situations, not just those with large problem parameters. "Asymptotic models", the traditional models, often fail when problem sizes per process decrease while the number of processes increases. Due to their single asymptote ($x \to \infty$), this limitation calls for a more flexible approach. We propose hyperbolic performance models as a solution. They provide two linear asymptotes ($x \to \infty$ and $x \to 0$), making them simple yet effective.

A hyperbolic performance model is a function $y = f(x)$ that outlines a hyperbola on an xy-plane. Here, y represents performance, a throughput-oriented metric like FLOP/sec to be predicted, and x stands for the problem parameter. We describe this function as:

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

In this function, η is a performance parameter, while a and d are constants to be determined. Roughly, 'a' corresponds to peak throughput, and 'd' connects to latency (not necessarily the MPI communication latency). As an empirical model, it doesn't show an obvious analytical relationship, so we need some measurements. The traditional 'least-squares fit' method may not be sufficient to estimate the values of a and d that best match our observed data, as it heavily favors larger operand sizes. Instead, we identify the values of a and d that minimize the difference between the observed performance, $g(\eta_i)$, and the model's prediction, $f(\eta_i)$, for a specific problem parameter η_i:

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

We utilize a simple method to get approximate values of a and d. Here, we take 'a' as the maximum observed performance value and choose 'd' as the value of η_i that makes $g(\eta_i)$ closest to $a/2$.

Hyperbolic performance models show remarkable flexibility in accommodating changes in operand sizes and problem scales. They effectively handle the theoretical and observed performance gap by using observed performance data to determine model constants. As a result, they serve as an efficient and versatile tool for the performance modeling of HPC applications. Their simplicity and rational function form allow for faster approximations, even in quick, rough calculations.

5.3 Application Performance Modeling

We follow a structured approach when modeling performance. First, we model each component without assumptions or approximations. At this phase, the performance of some operations might depend on multiple variables.

Our model enables us to calculate iteration times accurately, considering variations in operand size. This feature proves beneficial when handling diverse workloads and computational tasks. For instance, operand sizes in OpenMxP, including the trailing matrix and panel size, vary with each iteration.

Unlike traditional asymptotic models, our hyperbolic model captures these changes effectively, thus predicting performance more accurately. As shown in Figures 9a, our model accurately tracks most mixed precision GEMM behavior with various local matrix sizes. However, we notice some differences between predicted and observed performance. To our knowledge, the mixed precision GEMM performance in rocBLAS primarily relies on the underlying

auto-tuning library (Tensile). Therefore, reducing the performance fluctuation for various sizes remains a work in progress. We also use our model to predict scaling performance by integrating it with the model designed for broadcast. We applied this model to forecast the scaling performance of OpenMxP with 1-ring-Chunked broadcast, as depicted in Figure 9b. The prediction was within 5% of the actual run. Beyond the above use case, we can utilize this model to establish a more accurate analytical expression for application performance. The structure of our hyperbolic model often reflects in these analytical expressions of iteration times, indicating the model's accuracy. We aim to employ these models for dynamic algorithm selection during runtime.

6 DISCUSSION

In this section, we elaborate on several challenges we faced during the optimization and note possible future work.

6.1 Compute and Communication Overlapping

In the implementation of the look-ahead optimization, we designed three approaches to overlap computation and communication for the diagonal process in HPL-MxP, as illustrated in Figure 10a. Notably, the size of the GEMM update varies throughout the run.

Initially, we employed a straightforward method: utilizing a non-blocking broadcast. This allowed kernels with separate data to proceed without communication delays. However, we encountered a challenge on Summit, which is detailed in the subsequent subsection. The second approach involved a non-blocking GEMM kernel with GPU_Device_synchronized to manage dependencies. Yet, this method required restricting the overlap of two broadcasts to only one part of the computation synchronization chunk. To optimize this second approach, we synchronized the communication with the GEMM_Update. Ultimately, we adopted event-based synchronization to strike a balance between programming simplicity and maximizing the overlap of computation and communication, ensuring the most efficient control over dependencies.

This enhancement process was complex due to the absence of a stream parameter in MPI. Many current accelerators use the stream parameter to handle data dependencies. It might be beneficial for the MPI interface to incorporate the stream parameter to manage data dependency on accelerators, creating something like a stream triggered MPI implementation.

6.2 Underlying MPI Implementation

During the optimization of the cross platform Dsnapshot and OpenMxP, we encountered two different implementations of the MPI library, namely Spectrum-MPI (developed by IBM) and Craympich (developed by HPE). Our application's performance was significantly impacted by the divergent behavior of each of MPI library. When employing non-blocking communication on Summit (MPI_Icast, MPI_Isend, MPI_Irecv), we observed a considerable reduction in bandwidth, as shown in Figure 5.

Additionally, when activating GPU-awareness for Spectrum-MPI to grant NICs direct access to GPU memory, we noticed an unexpected behavior with Spetrum_MPI_Bcast and Spetrum_MPI_Ibcast. The library broadcast appears to synchronize the device prior to initiating the broadcast, causing our overlapping strategies that depend on non-blocking GPU kernels to fail. This reinforces our suggestion, the need for an accelerator synchronization mechanism within the MPI standard.

6.3 Variation in runs

While collecting data for the Frontier system network, we encountered random communication hangs or delays. These irregular delays or hangs could occur anywhere from 10 seconds up to 60 seconds, as shown in Figure 10b. At the user level, MPI would detect a timeout, but the root cause of these spikes remained elusive. We suspect this behavior could be linked to redistribution of a new dynamic routing table or dynamic routing itself. In reference to Algorithm 3, chunked communication seems to exhibit greater resilience to this issue. In the future, we plan to design a broadcast algorithm that is topology-aware for large networks and more resilience in the face of transient issues.

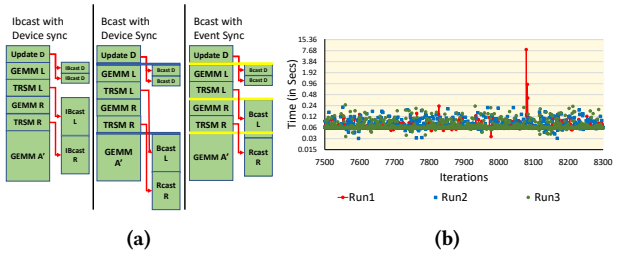


Figure 10: (a) Communication Computation Overlap Mechanism. (b) Network hang observed in DSNAPSHOT run.

7 CONCLUSION

In conclusion, this study highlights the complexity involved in improving communication between processes in 2D partitioned MPI applications. To improve inter-process communication performance, numerous factors, including algorithms, system design, and architecture, need to be considered. We design a new "chunked 2D broadcast" algorithm that cleverly combines row and column broadcasts. The implementation of this algorithm, alongside other optimized broadcasts for OpenMxP and Dsnapshot applications, led to significant performance improvements. Specifically, Dsnapshot reached nearly one exaflop, while OpenMxP set a new record, nearing **10 exaflops**.

We studied communication and computation overlaps across different scales and machines, gaining valuable insights. In addition, we presented a predictive model to simplify the optimization process and reduce the necessity for a large number of large-scale runs. As a practical contribution, we identified several issues and suggested useful MPI features to improve optimization strategies. This effort ultimately strives to push the boundaries of performance at the exascale level.

Future research will focus on expanding these techniques to a broader array of applications and further enhancing communication optimization using techniques such as communication-avoiding 3D algorithms. Moreover, designing a resilient communication library capable of mitigating transitive network failures or delays is another crucial area for future exploration.

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] AMD. [n.d.]. AMD ROCm Platform Portal. Accessed Oct. 21, 2021. https://rocmdocs.amd.com/en/latest/

[3] 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

[4] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[5] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[6] 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.

[7] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (dec 2019), 25 pages. https://doi.org/10.1145/3322125

[8] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned global address space languages. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 1–27.

[9] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. 2014. Efficient multi-GPU computation of all-pairs shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 360–369.

[10] Tarek El-Ghazawi and Lauren Smith. 2006. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 27–es.

[11] 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.

[12] 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.

[13] 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

[14] 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.

[15] 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

[16] 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

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

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

[19] Jing Fu Jenq and Sartaj Sahni. 1987. ALL PAIRS SHORTEST PATHS ON A HYPERCUBE MULTIPROCESSOR.. In *Proc Int Conf Parallel Process 1987*. Pennsylvania State Univ Press, 713–716.

[20] Ramakrishnan Kannan, Piyush Sao, Hao Lu, Drahomira Herrmannova, Vijay Thakkar, Robert Patton, Richard Vuduc, and Thomas Potok. 2020. Scalable Knowledge Graph Analytics at 136 Petaflop/s. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC41405.2020.00010

[21] Ramakrishnan Kannan, Piyush Sao, Hao Lu, Jakub Kurzak, Gundolf Schenk, Yongmei Shi, Seung-Hwan Lim, Sharat Israni, Vijay Thakkar, Guojing Cong, et al. 2022. Exaflops biomedical knowledge graph analytics. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[22] 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

[23] Vipin Kumar and Vineet Singh. 1991. Scalability of parallel algorithms for the all-pairs shortest-path problem. *J. Parallel and Distrib. Comput.* 13, 2 (1991), 124–138.

[24] 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).

[25] 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

[26] Xing Liu, Aftab Patel, and Edmond Chow. 2014. A new scalable parallel algorithm for Fock matrix construction. In *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 902–914.

[27] 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

[28] Robert W Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, Vol. 17. ACM New York, NY, USA, 1–31.

[29] NVIDIA. [n.d.]. NVIDIA CUDA Toolkit Documentation. Accessed Apr. 21, 2021. https://docs.nvidia.com/cuda/index.html

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

[31] Piyush Sao, Ramakrishnan Kannan, Prasun Gera, and Richard W. Vuduc. 2020. A Supernodal All-Pairs Shortest Path Algorithm. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*. ACM, 250–261. https://doi.org/10.1145/3332466.3374533

[32] Piyush Sao, Hao Lu, Ramakrishnan Kannan, Vijay Thakkar, Richard Vuduc, and Thomas Potok. 2021. Scalable All-pairs Shortest Paths for Huge Graphs on Multi-GPU Clusters. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 121–131.

[33] 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.

[34] Edgar Solomonik, Aydın Buluç, and James Demmel. 2013. Minimizing communication in all-pairs shortest paths. In *Proceedings of the 27th IPDPS*.

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

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