Multi-Node Matrix Multiplication using MPI
Final Term Project, CSC 746, Fall 2023

Shubh Pachchigar*
SFSU

ABSTRACT

This study aims to investigate the problem of matrix multiplication on distributed memory systems. Our research focuses on the core task of matrix multiplication, scaling it up across progressively larger problem sizes. The primary objective is to understand and exploit the memory utilization capabilities inherent in the underlying hardware architecture. In this investigation, we evaluate a distributed memory matrix multiplication (MMUL) operation implemented using MPI on CPU nodes. Throughout this study, we carefully examine two distinct implementations of matrix multiplication. These implementations differ initially in terms of their mesh decomposition and how matrix multiplication is carried out. The first approach utilizes all three types of mesh decomposition, including row, column, and tile decomposition, along with blocking communication procedure. In contrast, the second approach exclusively employs tile decomposition with non-blocking communication between CPU nodes. We assess the behavior of these implementations as we scale up the problem sizes, harnessing multiple cores across various nodes. We also determine which implementation method outperforms the others and explore the potential reasons for any performance differences. Our performance evaluation encompasses elapsed time, data movement, the volume of exchanged messages, and the communication-to-computation tables. Our findings conclusively demonstrate that the second approach leads to performance enhancements for arbitrary ranks. We conclude with insights into further opportunities for improvement.

1 INTRODUCTION

In the field of computer algorithms, many opportunities exist to improve both speed and efficiency. These improvements are not limited solely to the refinement of algorithms themselves but extend to strategic data organization in memory and the utilization of multicore processing. This study delves deep into this phenomenon, shedding light on the 'how' and 'why' behind these advancements. To accomplish this, we center our investigation around a commonly encountered computational task: matrix multiplication, implemented in two distinctive ways. Our main goal is to understand the behavior of multicore processors and gain insights into how harnessing their capabilities can yield substantial reductions in program runtime. It's worth noting that matrix multiplication operations find extensive applications in fields such as image processing and computer vision tasks. Their inherent parallelizability causes them exceptionally well-suited for high-performance computing on multi-core CPUs.

In this study, we apply the distributed matrix multiplication (MMUL) operation to random matrices and evaluate performance variations as we execute this operation on a CPU with increasing levels of problem sizes. We employ the Message Passing Interface (MPI) to execute the MMUL operation in a distributed manner. The Message Passing Interface (MPI) is a standardized communication protocol and programming model widely utilized in high-performance and distributed computing. It plays a crucial role in enabling efficient data exchange and synchronization among processes running in parallel or on separate nodes. This makes MPI integral to scientific simulations, numerical computations, and data-intensive applications. While the code for distribution and gathering remains consistent, we make modifications to the domain decomposition. We employ three distinct approaches to partition the problem: row-slabs, column-slabs, and tile decomposition, all of which will be thoroughly explained in the following sections. For additional details regarding these implementations, please refer to Section 3.

In each of the two implementations, there are three primary steps involved: scatter, matrix multiplication (mmul), and gather on four ranks. However, there are noteworthy distinctions between these approaches. In the first approach, the process initiates with the scattering of tiles, followed by the execution of matrix multiplication, and culminates with the gathering of the results. This approach employs different decompositions for all the matrices and distributes them in such a manner that each tile decomposition of the final matrix has all the necessary elements required for computation. Consequently, there is no need for further communication beyond the initial scatter operation. However, it's important to note that this approach entails the transfer of large rectangular matrices. Once each node computes its respective tile subsection, the results are subsequently gathered. In the second approach, a different methodology is adopted. Instead of employing three separate decompositions, each matrix is decomposed using the same approach. As a result, each node possesses only a portion of the necessary blocks for matrix multiplication of that specific subsection at any given time. Nodes request the remaining blocks while simultaneously performing the computation. This strategy accelerates the overall processing time. In Section 3, we will delve deeper into the details of how this is achieved and how the code is organized.

In the subsequent section, we will conduct a series of experiments in Section 4 using each of these approaches, aiming to compare their respective performance. The metrics employed to get these differences include elapsed time, data movement, and the communication-to-computation ratio. Despite the fact that the second approach entails slightly more communication, the data clearly shows that it surpasses the first approach in certain metrics. This illustrates the viability of the second approach and opens the possibility of its applicability in various other domains as well. Furthermore, this work delves into additional optimizations that can be implemented on top of this approach to achieve even greater performance enhancements.

2 RELATED WORK

This section outlines the prior research that will be utilized in our study. We examine two papers in this area. The first paper discusses an implementation using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) with blocking communications. The second paper tackles the challenge of overlapping communication. In their approach, they employ UPC (Unified Parallel C), a parallel variant of C, which enables overlapping through asynchronous functions. Notably, they allocate one core of the NUMA (Non-Uniform Memory Access) region exclusively for communication, while the remaining five cores are dedicated to computation, leveraging multithreaded BLAS routines.

*email:spachchigar@sfsu.edu

This paper [1] presents an MPI+OpenMP implementation for matrix multiplication using rowwise and columnwise block-striped decomposition in a multi-core cluster system. The implementation significantly reduces running time, demonstrating efficient use of multi-core CPUs. However, challenges include system overhead from OpenMP compiler directives and load imbalance across CPU cores. The hybrid MPI/OpenMP model outperforms pure MPI, optimizing parallel granularity. Despite its improvements, certain implementation aspects, like loop correlation and data race issues, need refinement. The paper concludes that the MPI+OpenMP model maximizes multi-core CPU resources, offering better performance than MPI alone.

Another paper [2] explores the integration of communication-avoiding and overlapping techniques in numerical linear algebra algorithms. It demonstrates how these methods reduce communication cost, a crucial factor for scaling linear algebra problems on exascale systems. The study focuses on matrix multiplication, triangular solve, and Cholesky factorization, employing communication-avoiding 2.5D algorithms to reduce data transfer volume, albeit at the cost of increased memory usage. Additionally, communication overlap is employed to minimize latency by pipelining messages and integrating computation work. Despite the benefits, challenges arise in balancing these techniques and understanding their complex interactions. The research offers novel implementations and a performance model, showcasing significant improvements in scalability, especially in high-core-count scenarios. However, there are trade-offs, such as extra memory usage and potential increases in latency, that need consideration.

In this study, our intention is to merge both approaches, aiming to harness the strengths of each. Specifically, we intend to employ the method of distributing work using MPI for coarse-grained parallelism, while also incorporating per-core fine-grained parallelism. Additionally, we plan to integrate the concept of non-blocking communication from the second paper. A more detailed explanation of this combination will be provided in the following section.

3 Implementation

This section introduces the distributed matrix multiplication operation, organized into four subsections. In the initial subsection, we provide a concise overview of the code structure, offering a general understanding of the code and also delving into the description of the three types of domain decomposition. The subsequent subsection outlines the fundamental functions within the code. Moving to the second subsection, we delve into the first strategy, which involves extracting a rectangular subarray based on a combination of domain decompositions and subsequently employing MPI functions to send it to other ranks in a blocking fashion. In the third subsection, we delve into the second strategy, providing an in-depth discussion of how the block subarray must be crafted carefully to be received in a non-blocking manner and placed within the appropriate buffer for further processing.

3.1 Overview of the MPI code harness

This implementation provides an overview of the code structure and offers insights into the functionality of each component and how they interconnect. This approach encompasses several key functions.

Initially, the code as shown in Listing 1 initializes a 2D vector called 'tileArray,' which serves as a container for crucial information and gets populated during the domain decomposition process, and because we are working with three matrices we will use three tileArray vectors. Subsequently, it initializes MPI using 'MPI Init()' and retrieves the rank and size of the current MPI communicator, typically 'MPI COMM WORLD,' to determine the process's rank 'myrank' and the total number of participating processes 'nranks'. The code also acquires the hostname of the current node and outputs a 'Hello world' message, providing details on the rank, total ranks, and the hostname where each process is executing. To ensure synchronization among processes, it employs 'MPI Barrier()'.

In cases where debugging is enabled, and the rank is 0, the code prints the 'AppState' and 'tileArray' to aid in understanding the computation's current state. It proceeds by invoking the 'computeMeshDecomposition()' function to determine how the problem domain is partitioned among different processes, thereby populating the 'tileArray.' Again due to three matrices involved we invoke 'computeMeshDecomposition()' function thrice. Three techniques for mesh decomposition are employed: row-slab, column-slab, and tiled decomposition. The first divides the entire matrix into small row-wise partitions, the second into column partitions, and the third employs a block/tile-like decomposition approach.

Throughout the code, it records time measurements for various computation phases using the C++ 'chrono' library. During the scatter and gather phases, data is exchanged between ranks through functions such as 'scatterAllTiles()' and 'gatherAllTiles()', facilitating data distribution and collection among the ranks. If the current rank is 0, the code generates timing results, encompassing scatter, MMUL, and gather times, as well as the count of messages sent and the total message size in megabytes.

Upon completing all computations and communications, the code calls 'MPI Finalize()' to perform cleanup of MPI resources before exiting. The code follows a structured workflow: it initiates domain decomposition to obtain tile metadata, then proceeds with data scattering, MMUL operations on individual tiles, and finally gathers all results back to the source rank.

Furthermore, each of the approaches now includes additional components within the header file. These components introduce new variables within the application state and Tile arrays, serving various purposes, including implementation-specific functions and metric collection. A description of these supplementary elements will be presented in subsequent sections.

```
class AppState
{
   INITIALIZE buffer1, buffer2, Adecomp, Bdecomp,
    Cdecomp;
   std::chrono::time_point<std::chrono::
    high_resolution_clock> start_time, end_time;
   std::chrono::duration<double> elapsed_mmul_time
    ;
};

class Tile2D
{
   INITIALIZE inputBuffer, outputBuffer, A, B and
    C
}

int main(){
    computeMeshDecomposition(C);
    computeMeshDecomposition(A);
    computeMeshDecomposition(B);

    scatterAllTiles(C);
    scatterAllTiles(A);
    scatterAllTiles(B);

    mmulAllTiles(A, B, C);

    gatherAllTiles(C);
}
```

Listing 1: Overview of the code harness: load three matrices and scatter, perform mmul operation and then gather.

3.2 Blocking Matrix Multiplication Implementation

The code in Listing 2 performs a blocking matrix multiplication, which involves several steps. Firstly, it computes mesh decomposition for all three matrices: A, B, and C. Subsequently, it scatters these matrices, performs matrix multiplication (mmul) on each of them, and finally gathers the results.

Here is a detailed description of the process: Initially, since three matrices are involved, we invoke the 'computeMeshDecomposition()' function three times. Each invocation yields three tileArray vectors: AtileArray, BTileArray, and CtileArray. These tileArray vectors are constructed individually by each rank to ensure that every rank is aware of the entire grid layout. Moreover, within the tile metadata, each of them contains vectors A, B, and C. These vectors will be populated during the scatter operation, and each tileArray will have its own copy of A, B, and C. At this point, it might appear inefficient to duplicate A, B, and C across tileArrays, and indeed, it is. We plan to address this inefficiency in the next approach. Once each tileArray is computed, only the source rank, which is zero, utilizes the tileRank to transmit the subarray to their respective tiles. The primary goal of this approach is to send all the required blocks of A and B in a manner that enables the computation of C's tile decomposition in a single iteration, without the need for further messages. Consequently, during the scatter phase, we send large messages, but we also duplicate messages to different destinations. This necessitates storing all the essential information in the tileArray vector.

Matrix A is decomposed using column decomposition, assigning a rank to each tile. Similarly, matrix B is decomposed into column matrices since we use four ranks. However, matrix C follows a tile decomposition approach, with each rank responsible for populating its corresponding C subarray with the correct values. For matrix A, the first tile must be sent to both rank 0 and rank 2, so we duplicate the tile metadata and append it with updated tileRank. This duplication is also done for the second tile in matrix A, which should be sent to rank 1 and rank 3. As for matrix B, we employ a different approach: row decomposition. The first tile is required by rank 0 and 1, while the second tile is needed by rank 2 and rank 3. Careful metadata crafting ensures that each rank receives the required tile, and any extra space is occupied in the metadata within the respective tileArray. Finally, matrix C is divided into four parts, with each part assigned to one rank.

With three tile arrays involved, we perform three scatter operations to each rank. In this process, we utilize a new variable called 'type,' which is an integer value indicating the specific matrix it represents. For instance, if it represents matrix A, 'type' is set to 0; for matrix B, it's set to 1, and for matrix C, it's set to 2. When a specific scatter operation is called for matrix A, we store the appropriate chunk of A subarray in the tile metadata of that specific rank. Similarly, for matrix B, we store the BTileArray, and for matrix C, we store it in vector C within the CtileArray. These values are then sent using the 'sendStridedBuffer' function. This function takes in an integer variable msgTag to 0. This variable will be used to tag the message being sent, which can help identify or categorize messages in a communication system. The purpose of this code is to perform the sending of data using the MPI Send() function from one MPI rank (specified as 'fromRank') to another MPI rank (specified as 'toRank'). The data to be sent is located in the source buffer (srcBuf), which has dimensions described by srcWidth and srcHeight. Then several variables are defined to describe the subregion of data that will be sent. $globalSize$ represents the dimensions of the entire source buffer ($srcHeight$ and $srcWidth$). $startOffset$ indicates the starting position of the subregion within the source buffer, specified as $srcOffsetRow$ and $srcOffsetColumn$. Finally, $subRegionSize$ represents the dimensions of the subregion that will be sent, given by $sendHeight$ and $sendWidth$. The code creates a custom MPI data type $subRegionType$ using MPI Type create subarray(). This data type is tailored to represent the subregion of data within the source buffer. It defines the subarray structure within the larger array, specifying dimensions and offsets. In this case, it defines a 2D subarray within a 2D array. After defining the custom data type, the code commits it using MPI Type commit(). This step is necessary to make the data type ready for use in communication operations. The actual data transfer occurs in this paragraph. It employs MPI Send() to transmit the subregion of data from $srcBuf$ to the destination rank specified as $toRank$. The message is tagged with $msgTag$, which can be useful for distinguishing different types of messages. The communication is performed within the MPI communicator MPI COMM WORLD. Lastly, the code frees the custom data type $subRegionType$ using MPI Type free(). This step is important for cleaning up any MPI-specific resources associated with the custom data type once it's no longer needed.

Another important code is responsible for receiving a specific subregion of data from one MPI rank to another using the MPI Receive function. It involves the creation of a custom data type to represent the subregion within the destination buffer and then receiving this subregion from the source rank. The code for recvStridedBuffer() initializes an integer variable msgTag to 0. This variable will be used to tag the received message, helping to identify or categorize messages in a communication system. The purpose of this code is to perform the receiving of data using the MPI Recv() function from another MPI rank (specified as 'fromRank') to the current MPI rank (specified as 'toRank'). The data to be received will be placed into the destination buffer (dstBuf), which has dimensions described by dstWidth and dstHeight. Then, several variables are defined to describe the subregion of data that will be received. $GlobalSize$ represents the dimensions of the entire dst buffer (dstHeight and dstWidth). dstOffset indicates the starting position of the subregion within the dst buffer, specified as dstOffsetRow and dstOffsetColumn. Finally, subRegionSize represents the dimensions of the subregion that will be received, given by expectedHeight and expectedWidth. The code creates a custom MPI data type subRegionType using MPI Type create subarray(). This data type is customized to represent the subregion of data within the source buffer, defining the subarray structure within the larger array, specifying dimensions and offsets. In this case, it defines a 2D subarray within a 2D array. From the receiving rank's perspective, we only receive the tile sent by the source therefore subRegionSize is same as the globalSize. After defining the custom data type, the code commits it using MPI Type commit(). This step is necessary to make the data type ready for use in communication operations. The actual data reception occurs in this paragraph. It employs MPI Receive() to receive the subregion of data from the source rank specified as fromRank. The received message is tagged with msgTag, which can be useful for distinguishing different types of messages. The communication is performed within the MPI communicator MPI COMM WORLD. Lastly, the code frees the custom data type subRegionType using MPI Type free(). This step is important for cleaning up any MPI-specific resources associated with the custom data type once it's no longer needed.

Upon reception on each rank's side, the matrix is initially stored in the tile inputBuffer. Subsequently, it is copied to the corresponding buffer, either A, B, or C. This buffer is then passed into the 'mmulAllTiles()' function, which operates on all tileArrays and executes CBLAS on each of the rectangular arrays. The results are stored back in the CtileArray, which is then gathered to obtain the final result.

3.3 Non-Blocking Matrix Multiplication Implementation

The code follows a similar initialization process as outlined in Section 3.2. Initially, we invoke 'computeMeshDecomposition' three times, but this time, all matrices are divided using tile decomposition. Consequently, there is no need to concern ourselves with duplicating

```
int main(){

  as.decomp = COLUMN_DECOMP;
  computeMeshDecomposition(&as, &AtileArray);
  as.decomp = ROW_DECOMP;
  computeMeshDecomposition(&as, &BtileArray);
  as.decomp = as.Cdecomp;
  computeMeshDecomposition(&as, &CtileArray);

  scatterAllTiles(as.myrank, CtileArray, as.C.
   data(), as.global_mesh_size[0], as.
   global_mesh_size[1], 0);
    scatterAllTiles(as.myrank, AtileArray, as.A.
   data(), as.global_mesh_size[0], as.
   global_mesh_size[1], 1);
    scatterAllTiles(as.myrank, BtileArray, as.B.
   data(), as.global_mesh_size[0], as.
   global_mesh_size[1], 2);

   mmulAllTiles(as.myrank, AtileArray, BtileArray
   , CtileArray);

   gatherAllTiles(as.myrank, CtileArray, as.
   output_data_floats.data(), as.global_mesh_size
   [0], as.global_mesh_size[1]);


}
```

Listing 2: Implementation of Blocking Matrix Multiplication function: distributes A in column, B in row and C in tile mesh and computes square dgemm

tileArray metadata, as each tile will have equal dimensions. Thanks to tile decomposition, each rank receives one tile of matrices A, B, and C.

Next, we perform scatter operations for these matrices, involving three scatter operations. As a result, rank 0 receives C0, A0, and B0; rank 1 receives C1, A1, and B1; rank 2 receives C2, A2, and B2; and finally, rank 3 receives C3, A3, and B3. During these scatter operations, we continue to use the 'sendStridedBuffer()' function, as discussed in Section 3.2, and employ the 'recvStridedBuffer()' function, as previously described. However, in this scatter, we utilize the 'type' integer variable differently. Instead of storing subarrays for A, B, and C in separate vectors, we consolidate them into two buffers called 'buffer1' and 'buffer2.' These buffers are present in the application state and are sized to match the input matrix. The reason for this is that instead of storing each tile of A, B, and C in separate vectors, we place everything into a single vector and calculate pointers to the start of the respective arrays. The arrangement of this vector is such that C precedes A and B, to ensure the initial scatter order is performed we use blocking sends. However, all subsequent calls are non-blocking. Blocking communication is used again during the gather phase.

We don't need to store three pointers because if we know the dimensions of one tile's width and height, all other tiles will have the same dimensions. Therefore, there's no need to store multiple pointers. We only calculate the three pointers when we perform the actual calculation by passing them to the 'square dgemm' routine. During the initial scatter, we include a 'jump' integer, which determines how the receiving ranks fill 'buffer1' in a specific order. Initially, 'C' values are stored, followed by 'A' and then 'B.' The 'jump' parameter helps indicate where to start storing values within the same array. Once everything is scattered in a blocking fashion, our actual operations commence. Each rank has 'buffer1' filled with tile values for A, B, and C, while 'buffer2' remains empty. We observe that computations for the main diagonal tiles can begin immediately, such as tile [0,0] and tile [1,1]. In contrast, tiles like [0,1] and [1,0] require data from other tiles before they can begin computation. To ensure an overlap of communication with computation, we arrange non-blocking send and receive operations in a carefully orchestrated manner.

The code shown in Listing 3 does the following: For rank 0, since it possesses a diagonal tile, it can directly perform computations. Consequently, we initiate the sending of two vectors, A0 and B0, from the input buffer. Each rank is responsible for sending two tiles and receiving two tiles to correctly compute C0. Initially, rank 0 starts the send operation for A using the MPI Isend routine, followed by a similar operation for B. Concurrently, it also initiates the receiving of A2 from rank 2 and B1 from rank 1. While rank 0 has started all the necessary receive operations, it can now commence the computation of the data in 'buffer1.' It's important to note that A2 and B1, which we were waiting for, are now filled in 'buffer2.' This arrangement is deliberate; it ensures that the communication buffer does not interfere with the computation buffer. Since we prioritize receiving before sending, we include a 'wait' command to ensure that 'buffer2' is filled. We then utilize these values to compute C once again after the first computation. It's worth noting that in the square dgemm operation, we pass C from the first buffer and A and B from the second buffer. We also include a 'MPI wait' at the end to ensure that the send operation is successful, as it is of lower priority.

For rank 1, where a diagonal tile is absent, our strategy is to initiate receiving first. This approach pairs send and receive operations across ranks, ensuring that each send operation doesn't need to wait excessively for acknowledgments, thereby speeding up data transfer. In cases involving off-diagonal matrices, such as tile [0,1], our initial step is to copy B1 to 'buffer2,' allowing us to receive B0 into 'buffer1' and A3 from rank 3 into 'buffer2' at the appropriate locations. We employ 'memcpy' initially and subsequently begin non-blocking receive operations into 'buffer1.' Following this, we commence two send operations: sending A1 to rank 3 and B1 to rank 0. At this stage, we must wait for the receive operation to complete because we don't yet have the necessary data for computation. Once the initial receive operation is finished, we proceed with square dgemm computation using CBLAS. Before this computation, we initiate another receive into 'buffer2,' perform the first computation, and then await the completion of the second receive. Finally, we wait for both send operations to conclude.

For rank 2, where a diagonal tile is again absent, we again prioritize receiving first. This approach maintains the pairing of send and receive operations across ranks, ensuring that each send operation doesn't experience prolonged waits, resulting in faster data transfer. Therefore here in tile [1,0], our initial step is to copy B2 to 'buffer2.' This enables us to receive B3 into 'buffer1' and A0 from rank 0 into 'buffer2' at the appropriate locations. We commence this process with 'memcpy,' followed by initiating non-blocking receive operations into 'buffer1.' Subsequently, we initiate two send operations: sending A2 to rank 0 and B2 to rank 3. At this point, we must wait for the receive operation to complete because we don't yet have the necessary data for computation. Once the first receive operation is completed, we proceed with square dgemm computation using CBLAS. Before initiating this computation, we begin another receive operation into 'buffer2,' perform the initial computation, wait for the second receive to conclude, and then execute the square dgemm operation again. Finally, we await the completion of both send operations.

Finally for rank 3, since it possesses a diagonal tile, it can directly perform computations. Consequently, we initiate the receiving of two vectors, A1 from rank 1 and B2 from rank 2. While doing this, it starts computation in the buffer 1. Then this rank waits for both receive to complete, rank 3 starts the send operation for A3 and

B3 using the MPI Isend routine. While rank 0 has started all the necessary send operations, it can now commence the computation of the data in 'buffer2.' It's important to note that A1 and B2, which we were waiting for, are now filled in 'buffer2.' We then utilize these values to compute C once again after the first computation. We also include a 'MPI wait' at the end to ensure that the send operation is successful, as it is of lower priority.

```
if(as.myrank == DIAGNOL){
        MPI_Isend(A);
        MPI_Isend(B);
        MPI_Irecv(A);
        MPI_Irecv(B);
        square_dgemm(buffer1);
        MPI_Waitall(2, recv_requests.data(),
    MPI_STATUSES_IGNORE);
        square_dgemm(buffer2);
        MPI_Waitall(2, send_requests.data(),
    MPI_STATUSES_IGNORE);
      }
      if(as.myrank== NOT DIAGONAL){
        memcpy(B);
        MPI_Irecv(B);
        MPI_Isend(A);
        MPI_Isend(B);
        MPI_Wait(&recv_requests[0],
    MPI_STATUS_IGNORE);
        MPI_Irecv(A;
        square_dgemm(buffer1);
        MPI_Wait(&recv_requests[1],
    MPI_STATUS_IGNORE);
        square_dgemm(buffer2);
        MPI_Waitall(2, send_requests.data(),
    MPI_STATUSES_IGNORE);
      }
```

Listing 3: Implementation of Non-blocking Matrix Multiplication function: arrange send and receive in some order and squeeze sqaure dgemm whil waiting.

4 EVALUATION

This section delves deeper into how the implementations presented in Section 3 are evaluated across a range of increasing problem sizes. The initial part outlines the required environment for replicating the experiments, while the subsequent sections provide detailed descriptions of the experiments conducted. All experiments are conducted on a fixed concurrency levels of 4.

The first experiment compares the blocking and non-blocking Matrix Multiplication approaches on a CPU described in Section 3.2 and Section 3.3 at different types of domain decompositions and the results of running our code at varying problem sizes, and describe the observations about the elapsed time we observe as we increase problem sizes. The second experiment describes the data movement performance study. This experiment presents information about the number of messages sent and the total data moved between all ranks. The third experiment computes the computation to computation ratio for both the techniques.

4.1 Computational platform and Software Environment

The system in use is Perlmutter at NERSC, equipped with hardware specifications including a 32KB MiB L1 cache, 4MB of L2 cache, and up to 32MB of base L3 cache. It runs on the SUSE Linux Enterprise Server 15 SP4 operating system, version 5.14.21-150400.24.69 12.0.74-cray shasta c, and is powered by an AMD EPYC 7763 Milan CPU capable of clock rates ranging upto 2.45 GHz. The system has 256GB of DRAM, utilizes the GCC 11.2.0 compiler from Cray Inc. with default optimization levels set to O3, and employs CMake version 3.20.4 alongside GNU Make version 4.2.1, built for x86 architecture.

4.2 Methodology

In this section, we will subject the implementations discussed in Section 3 to a series of tests and all of them are distributed across 4 nodes on Perlmutter. Each of these implementations will be executed on a varying problem size, [1024, 2048, 4096, 7000]. Each problem size generates three square matrices with random values. All experiments uses concurrency levels of 4. Following the completion of these experiments, we will evaluate and compare their performance using the elapsed time table, and data movement tables and communication to computation ratio. Four metrics in use here: elapsed time, total messages sent between ranks and corresponding data moved and communication to computation ratio. This metrics allows us to assess how effectively both the hardware and software leverage their capabilities to achieve optimal performance. To measure the elapsed time of the matrix multiplication, we will employ the chrono timer library.

Due the efficient subarray implementation discussed in Section 3.2 and Section 3.3, the number of messages sent per rank during scattering is one, and the same holds true for gathering. Consequently, we have incorporated code within the scatterAllTiles() function, as detailed in Section 3.2, to track all invocations of the $sendStridedBuffer$ function. Similarly, the gatherAllTiles() function also invokes the sendStridedBuffer function, which we take into account for our analysis. To determine the amount of data transmitted, we calculate the size of the inputBuffer, which is present in each tileArray, and multiply it by the size of a double value on the respective machine.

Calculating the computation-to-communication ratio posed a challenge in the second approach outlined in Section 3.3. This complexity arises from the presence of additional communication costs, in addition to the square dgemm routine. In contrast, the first approach in Section 3.2 involves no communication during the mmul operation, as scatter handles this aspect. In the second approach, we introduced additional variables within the application state, as described in Section 3.1. These variables are used to track elapsed time, start time, and end time. During the MMUL phase, all ranks perform two square dgemms. To isolate the square dgemm code for measurement, we encapsulate it within these variables and aggregate the results. Additionally, we incorporate a chrono timer wrapper around the entire MMUL operation, encompassing both communication and computation times. By employing this more refined chrono timer wrapper, we can extract the computation time and determine communication costs by subtracting it from the total time.

4.3 Experiment 1: Runtime Performance Study

The objective of the experiment conducted in this section is to compare the performance of the approaches discussed in Section 3.2 and Section 3.3. The experiments were conducted across varying problem sizes, and the elapsed times for blocking, non-blocking, and CBLAS implementations were compared. All reported times were measured using the chrono timer library, and the results were evaluated for comparison. From the results presented in Table 1, we observe that the problem size of 1024 yields the shortest execution time, aligning with our expectations. Surprisingly, the blocking mmul approach outperformed non-blocking and CBLAS, primarily because non-blocking communication incurs additional communication costs, contributing to longer overall MMUL phase times. In contrast, the blocking approach achieved the lowest total time, even though it involves transmitting more data than non-blocking due to the smaller problem size. A more detailed breakdown of communication and computation costs in non-blocking, as discussed in

Problem Size	Blocking MMUL	Non-Blocking MMUL	CBLAS
1024	0.01	0.05	0.04
2048	0.09	0.13	0.33
4096	0.69	1.04	2.70
7000	3.25	4.99	12.65

Table 1: Comparison of the elapsed time (in seconds) for the different matrix multiplication implementations.

Section 4.5, will provide insights into the specific MMUL times for each rank.

For a problem size of 2048, we again find that the blocking approach is four times faster than CBLAS, and due to the extra communication costs, it remains three times faster than CBLAS. In the case of a problem size of 4096, the advantages of MPI, whether blocking or non-blocking, over CBLAS become even more evident. Here, the speedup is again four times faster than CBLAS, with non-blocking achieving twice the speed of CBLAS. The gap between blocking and non-blocking continues to widen as communication overhead increases with larger tiles being moved around. With a problem size of 7000, we once more see a four-fold improvement for the blocking code and a 2.5-fold improvement for non-blocking over CBLAS. While these experiments do not provide conclusive evidence regarding the superiority of one approach over the other, the significant speedup compared to CBLAS clearly indicates that the distributed multi-node matrix multiplication approach is far more efficient than CBLAS on a single node. Further experiments will help pinpoint the exact differences between both approaches.

4.4 Experiment 2: Data Movement Performance Study

In this experiment, we conducted an in-depth analysis of the data movement occurring across all ranks. We compared the number of messages sent across each rank and the total number of megabytes of data moved during these exchanges. Since we are working with two different approaches, we present two columns of data for each approach in Table 2. Additionally, we maintained a constant concurrency level of 4, resulting in a consistent number of messages exchanged between ranks. However, as the problem sizes increased, we observed a corresponding increase in the amount of data moved.

Starting with the smallest size of 1024, each approach had its own set of characteristics. For the blocking approach, the number of messages sent was 12, as rank 0 scattered three times to each rank, and each rank scattered a single message back to rank 0 during the gather phase, totaling 12 messages. This resulted in a data movement of 95MB. On the other hand, the non-blocking approach involved more messages, specifically 20, because during the MMUL phase, each rank sent two tiles in addition to the tiles sent during the scatter phase. However, since all tiles were of the same size, the data movement value was affected, resulting in a lower value of 41MB, which is almost half of the blocking movement. Moving on to the problem size of 2048, the number of messages remained the same due to the constant concurrency level. However, due to increased tile sizes, we observed a 1.5-fold increase in data moved in the blocking MMUL approach, while the non-blocking approach caused a 4-fold increase. Both approaches had similar values, with non-blocking being slightly higher than blocking. Doubling the size again to 4096 led to a significant increase in both approaches, where blocking caused a 4x jump, and non-blocking resulted in a 5x jump. This trend continued for larger sizes, such as 7000, where blocking saw a 3x increase, and non-blocking experienced a 3x increase as well.

Based on this data, it appears that the first approach is better if our goal is to reduce memory transfers. This is evident because the first approach doesn't involve any data transfer during the MMUL phase, while the second approach does require it. However, this experiment focused on communication reduction. To better understand the impact of communication costs, we will proceed to Experiment 4.5

Problem Size	Block numMsg	Block DataMov	Non-block numMsg	Non-block DataMov
1024	12	95.12	20	41.94
2048	12	150.99	20	167.77
4096	12	603.98	20	671.09
7000	12	1764.00	20	1960

Table 2: Comparison of the number of messages and data movements for the blocking version and non-blocking version of matrix multiplication.

Problem Size	Block Comm	Block MMUL	Non-Block COMM	Non-Block MMUL
1024	0	0.01	0.01	0.04
2048	0	0.09	0.04	0.08
4096	0	0.69	0.35	0.69
7000	0	3.25	1.74	3.25

Table 3: Comparison of the number of communication and computation for the blocking version and non-blocking version of matrix multiplication.

to examine the extent of these costs and whether we should retain or eliminate them.

4.5 Experiment 3: Communication to Computation ratio

In this experiment, we aim to determine which of the two approaches mentioned in Section 3.2 and Section 3.3 is more efficient. Specifically, we compare the communication costs and computation costs between these two approaches. It's important to note that this ratio is primarily useful for the second approach, as the first implementation never involves communication. In this experiment, we are solely concerned with the communication overhead of the second approach, with the goal of comparing the actual computation costs between both approaches. For a problem size of 1024, the first approach incurs no communication costs, while some costs exist for the non-blocking approach. We observe that the cost for non-blocking is four times higher than that for blocking, resulting in a ratio of 4. Moving to a problem size of 2048, the ratio remains at 2. Surprisingly, computation times are the same between both approaches, yet communication costs increase fourfold with a doubling of the problem size. Once again, the ratio between communication and computation remains at 2. For a problem size of 4096, the ratio remains at 2, with computation times equal for both approaches. However, we observe an eightfold increase in communication costs with a twofold increase in problem size. Finally, for a problem size of 7000, compute costs remain similar between both approaches, with a fivefold increase in communication costs. Once more, the ratio remains at 2. In summary, we can conclude that the compute costs between both approaches are approximately the same for larger sizes, indicating that both approaches perform similarly in terms of computation.

4.6 Conclusion

In conclusion, based on our experiments, both approaches perform well for small problem sizes, as indicated by their similar compute costs from the last experiment. However, due to the communication costs, we find that blocking approach has an advantage. Nevertheless, it's important to note that this approach is highly optimized for fixed number of ranks, and there isn't an optimized algorithm available for arbitrary ranks. On the other hand, non-blocking approach, while incurring communication costs, possesses the advantage of being more versatile. The data in the metadata enables it to be easily extended for arbitrary ranks, although this extension was not explored in this study. Future work may involve extending this approach to accommodate arbitrary ranks. Therefore, if we are working with a fixed number of ranks, then blocking implementation is preferable. However, if we anticipate working with arbitrary numbers of ranks, then non-blocking approach offers more flexibility and adaptability.

REFERENCES

[1] Lidong He et al. “MPI+OpenMP Implementation and Results Analysis of Matrix Multiplication Based on Rowwise and Columnwise Block-Striped Decomposition of the Matrices”. In: *2010 Third International Joint Conference on Computational Science and Optimization*. Vol. 2. 2010, pp. 304–307. DOI: 10.1109/CSO.2010.123.

[2] Evangelos Georganas et al. “Communication avoiding and overlapping for numerical linear algebra”. In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.32.