Optimizing Large Irregular Boundary Exchanges with Increased Asynchrony

Gerald Collom
Amanda Bienz
University of New Mexico
Albuquerque, New Mexico, USA

Abstract

The performance of many parallel applications is limited by irregular communication. Implementations of irregular communication often involve separate phases of computation and communication that prevent applications from achieving peak system performance. This work presents multiple irregular boundary exchange optimizations that overlap computation with computation to improve system utilization. The optimizations are based on introducing asynchrony with both thread-level early work and early communication, as well as through alternate data structures that better support asynchronous data access. We implement these optimizations within a benchmark that performs a commonly used sparse matrix operation and present an analysis of their performance and scaling behavior on real-world sparse matrices. Compared to a typical approach, we measured speedups of up to 28 percent for a thinner block vector and 190 for a more square dense matrix.

ACM Reference Format:
Gerald Collom and Amanda Bienz. 2018. Optimizing Large Irregular Boundary Exchanges with Increased Asynchrony. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

One important parallel operation found in several HPC applications such as large language models, enlarged Krylov methods and graph algorithms [17, 19] is the multiplication of a sparse matrix and a tall skinny dense matrix. In this work, we use the abbreviation SpMBV [17] for this operation, where the tall skinny dense matrix is referred to as a block vector, shortened to BV. The resulting linear system of $AX = B$, is typically partitioned row-wise in one dimension, equivalent to standard SpMVs, except X and B each have multiple columns, as exemplified in Figure 1.

We depict in Figure 1 how local rows of A are often separated into the following on-process and off-process components [1, 20, 22]. An on-process matrix holds entries of A used in local computation, highlighted as solid colored blocks while an off-process matrix contains entries that are multiplied by nonlocal data, indicated by

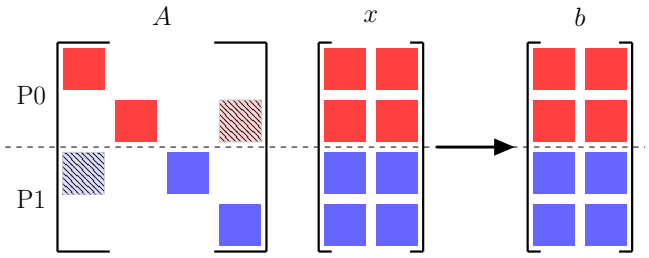

Figure 1: Example partitioning of a linear system multiplying a sparse matrix by a tall skinny dense matrix. Data dependencies emerge similar to those in SpMV but in this case, the additional dimension of the dense matrix results in larger messages consisting of multiple full rows.

pattern blocks. Typically, both of these matrices are stored using the compressed sparse row (CSR) format. Processes can perform local computations directly but must receive nonlocal dense matrix rows to complete off-process multiplications. Using Figure 1 as an example, the solid red blocks of A that belong to $P0$ are multiplied by the solid red blocks of X. Since these values of both A and X are stored locally on $P0$, these computations can be performed at any point during the overall operation without exchanging nonlocal data. Other computations, however, will not rely solely on local data. For example, the pattern block held by $P0$ must be multiplied by the fourth row of X stored on $P1$. In order to complete this multiplication, the nonlocal row of X must be sent from $P1$ to $P0$. For the partition of A belonging to $P0$ then, the on-process component consists of the solid red blocks, while the off-process portion will contain the red pattern block. Looking instead at $P1$, the on-process and off-process matrices are indicated by solid blue blocks, and pattern blocks respectively. In general, each participating process will separate a distinct set of columns of its partition of the sparse matrix into an on-process and an off-process matrix, depending on which rows of X the process holds and therefore which columns can be multiplied using local data, and which columns require nonlocal data. In this row-wise partitioning scheme of A and X, the on-process matrices generally align with the diagonal of A, while the off-process matrix contains off-diagonal entries of A.

Using the previously described partitioning of the linear system, one common approach to performing a distributed SpMBV consists of the following steps. First fully local computation is performed, completing all multiplications that only rely on data contained by the process, and computing a partial result. In the example in Figure 1, this would mean multiplying all solid colored blocks in A with solid colored blocks of X computing part of the result B. Then,

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

rows of the block vector held by the current process are sent to any processes that require it, those that hold nonzero entries of the sparse matrix in columns corresponding to the rows of the block vector held by the sending process. For example, $P0$ must send the first row of X to $P1$ and $P1$ must send the fourth row of X to $P0$. Following this step, processes await for all exchanged data to be received. The reason that all received block vector rows are waited on, rather than computing results for each row as they are received is based on how data in the sparse matrix is accessed when using the CSR format. Received block vector rows are multiplied by entries in corresponding columns of the local partition of the sparse matrix but with CSR, matrix entries are accessed row-wise. Locating the necessary columns of the sparse matrix using CSR requires stepping through every matrix row and for each one, traversing the row to check the column index of each value. Larger matrices warrant avoiding repeatedly iterating through the matrix as each block vector row arrives. Instead, after all rows are received, the sparse matrix is traversed once, adding products to the partial result to complete it. In the ongoing example, this would mean multiplying the pattern blocks on each process by the rows of X they received and adding the products to the previous partial result, completing the computation of B and the overall operation.

Additionally, several optimizations are often applied to SpMBV operations to improve performance. One major optimization is use of local threading on each process to parallelize local work consisting of both multiplications and packing data for communication. Packing data is required to transfer noncontiguous data to a contiguous message buffer so the data can be communicated. For example, if a few scattered rows of the block vector must be communicated, they must be copied from their discontinuous locations into one compact message buffer. With consideration to avoid excessive overheads from thread management, threading can parallelize this packing and required multiplications for a performance benefit. In the case of iterative SpMBVs, an MPI optimization that can be applied is persistent communication. Persistent communication improves iterative communication by amortizing setup costs. Another further optimization that builds on persistent communication is message partitioning. Partitioning messages allows for the parallelization of communication across threads. With partitioning, portions of messages can be sent independently without synchronizing on the communication of the entire message. This enhances performance by pipelining communication and computation through early work or early communication, resulting in improved network utilization.

This work makes the following contributions:

- Early communication with partitioned sends, reducing the amount of data sent to the NIC at one time and increasing network utilization.
- Asynchrony in off-process multiplication with CSC matrices, allowing for multiplication of off-process columns with portions of the dense matrix, increasing overlap of communication and computation.
- Early computation through partitioned receives and `MPI_Pready`, further increasing communication-computation overlap.

The remainder of this paper is structured as follows. Background is provided in Section 2 on the operation of multiplying a sparse matrix by a tall skinny dense matrix, on compressed matrix storage formats, and on persistent and partitioned communication optimizations. Related work is discussed in Section **??**. The implementations of three optimized communication strategies are detailed in Section 3. The performance of these approaches are analyzed in Section 4 through direct comparisons, strong scaling studies and studies on the effects of parameters such as the size of messages and the distribution of processes across compute nodes. Finally, concluding remarks and future directions are presented in Section 5.

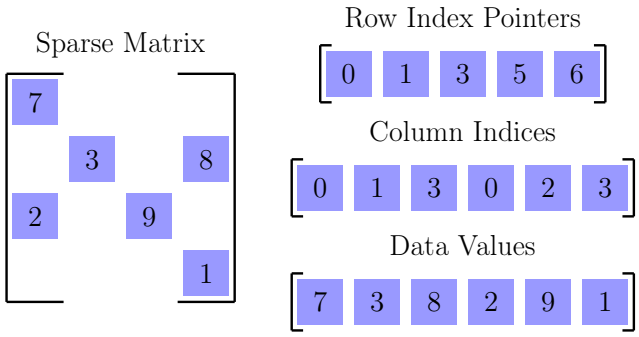

Figure 2: CSR matrix storage format. The value at each index in Row Index Pointers specifies the starting bound in the Column Indices and Data arrays of the row with the corresponding index. In the Column Indices array, the column index of each data value is listed at the same index as the data value.

2 Background

2.1 Sparse Matrix Block-Vector Multiply

The distributed multiplication of a sparse matrix times a block vector (tall skinny dense matrix) is a specific case of sparse matrix-matrix multiplication (SpMM). In parallel, the operation is typically decomposed equivalently to standard SpMVs using a row-wise one-dimensional partition, in which each process holds a portion of the rows of the matrix and equivalent rows of the input and output vectors. This partitioning is visualized in Figure 1. Consequently, data and communication dependencies emerge from the sparsity pattern of the matrix, just as in SpMV operations. Before nonzero matrix entries can be multiplied by nonlocal vector values, the data must be communicated from the process holding the partition of the vector containing those values. In SpMBV operations, the amount of nonlocal vector values requiring communication increases by the width dimension of the block vector, resulting in larger messages than in standard SpMVs. As the block vector width grows to approach square dimensions, the problem becomes equivalent to the more general SpMM operation, multiplication between a sparse matrix and a dense matrix.

2.2 Compressed Matrix Storage Formats

Sparse matrices can be stored more compactly than a full two dimensional array by taking advantage of their sparsity as zero-valued entries do not need to be stored. This is typically achieved with the use of the compressed sparse row (CSR) storage format. A matrix stored in this format consists of three arrays as shown in Figure 2: a row pointer array, a column indices array and a data array. The row pointer array describes the index within the column and data arrays at which each row in the matrix begins. Since the next entry in the row pointer array lists the starting index for the following row, that next entry defines the ending bound (exclusive) for data in the column indices and data values arrays belonging to the current row. For example in Figure 2, the first row of the matrix has data and column index values ranging from array index zero to just before array index one, so a single value. The data array contains every nonzero value from the matrix, and the column index of each value is stored in the column indices array at the same position of the corresponding data value. This condensed data format fully describes the an n-dimensional sparse matrix with nnz non-zeros using only $(n+1)+2(nnz)$ entries.

The CSC format requires equivalent arrays for pointers, indices, and values. Instead of a row pointer array, a column pointer array describes ranges of data for each column in the matrix. And instead of a column indices array, a row indices array stores the row index of each data value. This CSC format offers the same compressed storage as CSR with the key difference that the matrix is stored and accessed column-wise.

2.3 Partitioned MPI

Partitioned communication partitions each message across multiple threads and avoids synchronizing on the entirety of the message, allowing for portions of a message to be sent and received independently. Message partitioning interfaces are persistent, meaning there is an initialization phase, followed by repeated exchanges and cleanup. This optimization parallelizes communication across threads, and allows communicating parts of a large message immediately after the message buffer is prepared, without waiting on other parts of the message, termed early communication. This approach reduces network contention resulting from communicating all message data at once and overlaps computation with communication. On the receiving end of communication, partitioning also allows parts of a message to be unpacked and used in computation before the entire message is received, an optimization known as early work. This approach provides even further computation-communication overlap and reduces processor idle time.

While message partitioning can optimize the communication of large messages, it results in some overheads that can outweigh performance benefits if not properly considered. First, using threads with MPI requires the flag `MPI_THREAD_MULTIPLE` which, for some versions of MPI, can result in slowdowns [23]. The current message partitioning interface included in MPI also requires that all partitions be equal in size and that the number of sending and receiving partitions are equal. If message data cannot be evenly divided into the number of partitions used, the message data must be padded with unused data to make the division even. The complexity and overhead of this padding can result in reduced performance.

The partitioned communication interface consists of the following APIs. An initialization call, `MPI_Psend_init`, specifies standard communication arguments along with the number and size of partitions for use in subsequent communication exchanges. Each iteration begins with `MPI_Pstart`, stating a communication phase is beginning. After a partition's data is updated from computation and packed into the message buffer, the partition is marked as ready for communication with `MPI_Pready`. To wait for all parts of a message to arrive, `MPI_Wait` or a related method is used. Partitioned interface methods such as `MPI_Parrived` can also be used to test for the arrival of individual partitions. The latter routine can be used to perform early computation on one partition of a message before the entire message arrives.

2.4 Related Work

Sparse matrix-dense matrix multiplication is a heavily researched topic. The bottleneck of synchronization throughout sparse matrix operations has been previously explored, with one-sided communication yielding increased asynchrony [3]. Other popular optimizations include tiling [15, 18], row reordering [14], hierarchical reordering [13], and just-in-time (JIT) compilcation [9]. Data movement overheads have been optimized through locality-aware aggregation [2]

While this paper concentrates on the 1D partitioned sparse matrices, many sparse matrix operations can be improved through 2D, 2.5D, and 3D partitions [4, 16, 21]. In the case of multi-dimensional partitions, advanced multiplication algorithms such as SUMMA are standard practice [5].

Partitioned communication has been used in many contexts. Finepoints introduced the use of partitioned communciation to improve the performance of threaded applications [11]. Since being added to the MPI standard, partitioned communication has been implemented and evaluated within multiple MPI libraries [8, 10]. Partitioned communication has been proposed within collective communication [12] and has been used to optimize stencil communication [6]

3 Methodology

The approaches presented in this work attempt to optimize the typical block synchronous approach to iterative communication. In this common implementation, processes pack and exchange data, wait for all messages to be received and then complete computations dependent on the exchanged off-process data. There is some possible communication-computation overlap as applications can perform local computations during communication, but most MPI implementations lack strong progress and can not progress outside of MPI calls, reducing actual overlap.

We present optimizations that increase overlap by decreasing synchronization through the use of message partitioning and the CSC matrix storage format. Message partitioning with partitioned MPI interfaces increases communication-computation overlap by allowing threads to independently advance between communication and computation early rather than waiting to synchronize with other threads. The choice of matrix format also impacts this

overlap. Using the popular CSR format, the entire off-process block vector must be received before any computation can occur. If a partial receive were to be multiplied as soon as it arrived, the entire CSR matrix would need to be accessed to find the corresponding column indices. The CSC format, however, allows for any column of the matrix to be accessed directly, enabling early computation and increasing communication-computation overlap.

In order to analyze these optimizations, we implemented three methods for performing iterative SpMBVs and compared them in benchmark tests. The first method explores the use of the CSC format for early computation without consideration of threading or message partitioning. The second approach introduces thread-level message partitioning and has threads performs early communication immediately after packing message data. The final method expands upon the second, additionally removing synchronization between threads as message data is received to achieve early computation.

3.1 Data Access in CSR and CSC

Within a SpMBV, there are no dependencies between vector rows to complete the matrix multiplication. When receiving data it is possible to immediately perform computations that depend on the arriving data without synchronizing on the arrival of other communicated data. This asynchrony allows for early computation, and increases overlap with communication, improving network utilization. When using the CSR format and a vector row is received, however, completing all computations that depend on that vector row would require stepping through every row of the matrix and checking if any values in that row have the column index corresponding to the received vector row. Consider for example the matrix in Figure 1. When process 0 receives the fourth vector row, accessing the corresponding matrix value in the fourth column using CSR would require checking the column index of every value in the first and second rows, until the one was found with column index three. The CSC matrix format, however, allows for direct access to the column of matrix entries that must be multiplied by any given vector row received. Referring again to Figure 1, when process 0 receives the fourth vector row, the column of values to multiply it with is directly accessible using CSC, by jumping to the column with the same index as the received row.

When using multiple threads per process, however, early computation using the CSC format results in a race condition where multiple threads update the same values of the result vector. The race condition occurs because each thread processing a received vector row will compute a partial product for a column of the output vector and other threads will contribute partial products to the same entries. To resolve this race condition, the threaded CSC method presented in this work uses an atomic OpenMP directive to allow multiple threads to safely update the result vector.

3.2 Early Computation with CSC and `MPI_Test`

The first method analyzes the use of the CSC matrix storage format for early computation within iterative SpMBVs. We compare this approach against two other methods, beginning with a baseline typical implementation of iterative SpMBVs using the CSR format and nonblocking sends and receives. We also compare against an additional baseline that uses the CSC format instead of CSR but includes no other changes, to evaluate any overhead to the use of CSC. In the early computation method, however, MPI processes poll on incoming messages and immediately perform all related computations as they arrive, accessing data with the CSC format.

A characteristic code portion of the early computation approach is presented in Algorithm 1. The matrix entries that must be multiplied by off-process vector data are stored in A_off_csc using the CSC format. The resulting product is stored in $b.local$. The algorithm begins by creating an array to track which requests are yet to be received, initialized to include all receive requests. Then, the algorithm iterates over unreceived requests and for each one, tests its arrival with `MPI_Test`. If a message has arrived, the received data is immediately multiplied with *mult_append_msg*. Here, the CSC format enables direct data access to all matrix data needed for multiplication with the received data. The resulting partial product is added to the result vector. If the tested request has not arrived, it is added back into the list of unreceived requests. This method begins computation as soon as any message arrives and avoids synchronization on all messages being received before beginning any computation.

Algorithm 1: Early computation with CSC and `MPI_Test`

```
n_req ← num_recvs
next_n_req ← 0
Initialize req_idx with values 0 to num_recvs − 1
while n_req > 0
    next_n_req ← 0
    for i ← 0 to n_req − 1 do
        idx ← req_idx[i]
        MPI_Test(&(recv_requests[idx]), &flag, &status)
        if flag
            start ← recv_displs[idx]
            end ← recv_displs[idx + 1]
            mult_append_msg(A_off_csc, recv_data, b.local, start, end)
        else
            req_idx[next_n_req++] ← idx
    n_req ← next_n_req
wait_sends(A)
```

3.3 Threaded SpMBV Methods

Algorithm 2: Threaded SpMBV

```
parallel region
    for i ← 0 to A.n_rows − 1 do
        for j ← A.rowptr[i] to A.rowptr[i + 1] − 1 do
            for vec ← 0 to n_vec − 1        {n_vec refers to block vector width}
            do
                b[i × n_vec + vec] ← (A.data[j] × x[A.col_idx[j] ×
                  n_vec + vec]) + (β × b[i × n_vec + vec])
```

The second and third presented methods explore the same SpMBV operation performed iteratively but additionally introduces the use of threads and thread-level message partitioning with OpenMP and partitioned MPI interfaces. In the algorithms presented in this section, *n_vec* refers to the number of columns in the block vector. The baseline approach for the threaded case uses the typical CSR format and nonblocking sends and receives. Threads are used to parallelize

computation but communication is performed by a single thread. Threads perform local computations in parallel using a method shown in Algorithm 2. This algorithm iterates over matrix rows and divides them among threads to perform all necessary multiplications, storing the products in the product block vector b. The β term is used to differentiate computations on local and nonlocal data, using a β of zero for local computations to overwrite b, and using a β of one for nonlocal computations to instead add values to b. After local computations, a single master thread performs communication and all threads synchronize on the completion of communication before executing computations dependent on the received data, using Algorithm 2 with β value one. This baseline method is representative of a typical block synchronous approach.

Partitioned MPI interfaces, however, allow parallelization to extend into communication. The second optimization presented in this paper utilizes message partitioning for early communication. This and the following optimization use the simplifying assumption that the width of the block vector is evenly divisible by the number of partitions. If the above is true, partitions will always be equal in size, a requirement of the partitioned MPI interface. This method begins similar to the baseline with parallel local computations using Algorithm 2 but the parallel region extends into the start of communication. As presented in Algorithm 3, the approach iterates over all messages that must be sent and for each one, packs the data required by other processes. As threads pack data into a message buffer in parallel, the OpenMP `nowait` directive allows a thread to immediately advance to call Pready for the portion of data it has finished packing. This allows the data to be sent early rather than awaiting all threads to complete packing separate portions of the message. Next, a single master thread awaits for all nonlocal data to be received before completing nonlocal computations in parallel using Algorithm 2 as in the baseline method.

Algorithm 3: Parallel Packing and Pready

```
parallel region
    for each i ← 0 to num_sends − 1
        for j ← send_displs[i] × n_vec to send_displs[i + 1] × n_vec − 1
          {n_vec is block vector width}
        do
            idx ← idxs[⌊j/n_vec⌋]
            send_buffer[j] ← x[(idx × n_vec) + (j mod n_vec)]
          {nowait directive allows threads to asynchronously advance to Pready call}
        MPI_Pready(omp_get_thread_num(), &send_requests[i])
```

The final optimization evaluated in this work expands on the early communication in the previous method by also including early computation after receiving data with the use of the CSC matrix format. The algorithm for this early computation is presented in Algorithm 4. In this algorithm, messages are assumed to initialized with a number of partitions equal to the number of threads and for simplicity, each thread manages the partition with the same index. The method follows a similar pattern to the early SpMV computation in Algorithm 1 but polls for arriving data at the granularity of thread partitions rather than entire messages. Rather than having each MPI process use `MPI_Test` to check the arrival of messages, each thread uses `MPI_Parrived` to check the arrival of its partition of each message. When a partition has arrived, however, determining the range of values received is more complex than in Algorithm 1. While messages contain one or more whole rows of the block vector, partitions of messages can contain fractions of block vector rows which must be handled. The algorithm uses a subroutine to perform all multiplications for a single row or a single portion of one row of received block vector entries, `SpMV_off_proc_CSC`. The details of this subroutine are presented in Algorithm 5. This algorithm uses the CSC format to execute all multiplications between one or part of one vector row and one matrix column. The result of each multiplication computed for the received data is added to the corresponding entry in the output vector b.

Algorithm 4: Early computation with Parrived and CSC format

```
parallel region
    n_req ← num_recvs
    next_n_req ← 0
    part ← omp_get_thread_num()
    num_parts ← omp_get_num_threads()
    Initialize req_idx with values 0 to num_msgs − 1
    while n_req > 0
        next_n_req ← 0
        for i ← 0 to n_req − 1 do
            req_idx ← req_idxs[i]
            MPI_Parrived(&recv_requests[req_idx], part, &flag)
            if flag
                start ← recv_displs[req_idx]
                end ← recv_displs[req_idx + 1]
                part_size ← (end − start) × n_vec ÷ num_parts
                start_idx ← (start × n_vec) + (part_size × part)
                start_row ← ⌊start_idx ÷ n_vec⌋
                vec_row_start ← start_idx mod n_vec
                end_idx ← (start × n_vec) + (part_size × (part + 1))
                end_row ← ⌊end_idx ÷ n_vec⌋
                vec_row_end ← end_idx mod n_vec
                if vec_row_end == 0
                    vec_row_end ← n_vec
                else
                    end_row ← end_row + 1
                for j ← start_row to end_row − 1 do
                    end_pos ← n_vec
                    if j == end_row − 1
                        end_pos ← vec_row_end
                    SpMV_off_proc_CSC(n_vec, vec_row_start, end_pos, j,
                      A_csc, x_off_proc, b)
                    if j == start_row
                        vec_row_start ← 0
            else
                req_idxs[next_n_req++] ← req_idx
        n_req ← next_n_req
```

Algorithm 5: Sparse Matrix-Vector Multiplication (CSC Format)

```
for i ← A.colptr[col] to A.colptr[col + 1] − 1 do
    for vec ← vec_row_start to vec_row_end − 1 do
        b[A.col_idx[i] × n_vec + vec] += A.data[i] × x[col × n_vec + vec]
      {atomic update}
```

4 Results

In order to evaluate the methods presented in Section 3, we conducted a range of experiments on the Dane supercomputer at Lawrence Livermore National Lab [1]. Throughout this section we compare the speedup over a baseline for the following three communication optimizations:

(1) Early computation with the CSC format and `MPI_Test`
(2) Early communication with thread-level message partitioning and `MPI_Pready`
(3) A combination approach achieving early communication with `MPI_Pready` and early computation using the CSC format and `MPI_Parrived`

For the first optimization, we compare the speedup over a baseline for the CSC method performing early computation on a range of matrices. We also analyze the strong scaling behavior of the speedup. For reference in these two studies, the performance of an additional baseline method is also plotted in which the only change is the use of the CSC format over CSR without optimization.

We then compare the speedup of the second and third optimizations based on thread-level partitioning. For a range of matrices, we analyze the performance and strong scaling behavior of these thread-level optimizations. We additionally present analysis of the performance effects of varying two parameters. First we consider message size deriving from the number of columns of the block vector. Second, we evaluate the choice of how many MPI processes are assigned per node, or equivalently the number of cores and threads assigned to each MPI process.

4.1 Experimental Setup

We ran all experiments on the Dane supercomputer at Lawrence Livermore National Lab consisting of 112 core Intel Sapphire Rapids CPUs and Cornelis Networks interconnect. Experiments test real-world sparse matrices from SuiteSparse. [2] Our results are based on running approximately one second of iterations and taking the average iteration time using MPI timing functions. In order to avoid effects of neighboring jobs on the system, we ran each test five times, taking the average of averages. We used the default system MPI installation of `mvapich2` version 2.3.7. For on-node parallelism, we used OpenMP and for the partitioned communication library we used MPIPCL. [3]

4.2 Early Compute with CSC

This section presents analysis of the speedup and strong scaling of optimizations for a SpMBV using a single thread with a vector width of 25.

Figure 3 shows the performance implications of using CSC format across a variety of Suitesparse matrices. For the majority of tested matrices, CSC format results in slight slowdowns in comparison to the standard CSR format. However, when combining the CSC format with early work, multiplying each message as it arrives, a number of the multiplies were improved. For two of the matrices, this method performed about 15 to 20 percent slower and the method took roughly the same time as the baseline for several other

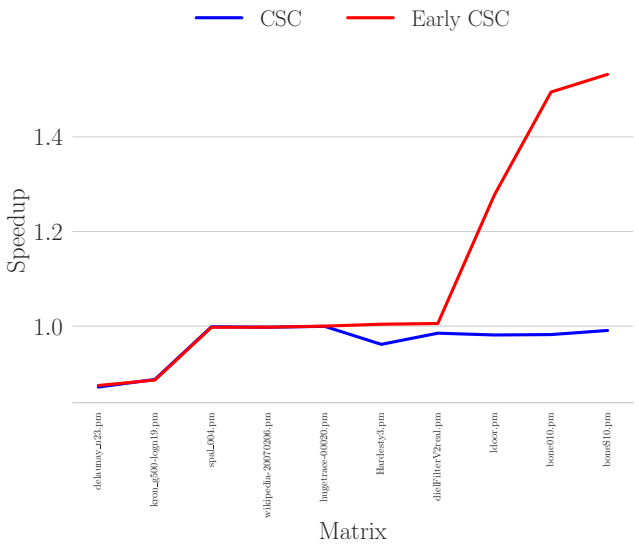

Figure 3: Performance comparison against a baseline for two methods: using the CSC method without further changes (blue) and early computation using the CSC format (red).

matrices. However, the remaining matrices achieved speedups of 20 to 60 percent, demonstrating a significant advantage.

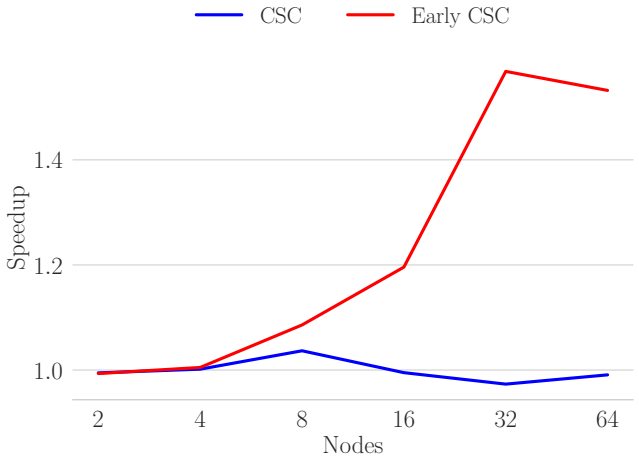

Figure 4: Strong scaling comparison of speedups for a single matrix using the CSC format (blue) and an early computation optimization (red).

Figure 4 shows the strong scaling behavior of early computation for the SuiteSparse matrix `boneS10`. While there were minimal differences between standard CSR and CSC approaches, early computation resulted in significant strong scaling improvements. At the scale of 32 nodes, performing early computation resulted in a speedup of nearly 60 percent.

[1] https://hpc.llnl.gov/hardware/compute-platforms/dane
[2] https://sparse.tamu.edu/
[3] https://github.com/mpi-advance/MPIPCL

4.3 Threaded Optimizations with Partitioning and CSC

In this section, we analyze the performance of two optimizations based on thread-level asynchrony within a SpMBV operation. The first approach achieves early communication with message partitioning and the second extends the first to include early computation using the CSC matrix format. The performance of these optimizations depends on the communication pattern resulting from the sparsity pattern of each matrix and parameters such as message size and the number of MPI processes assigned to each node. The results in this section include an initial performance comparison, a strong scaling study and analysis of the effects of two parameters: message size and the number of MPI processes assigned to each node.

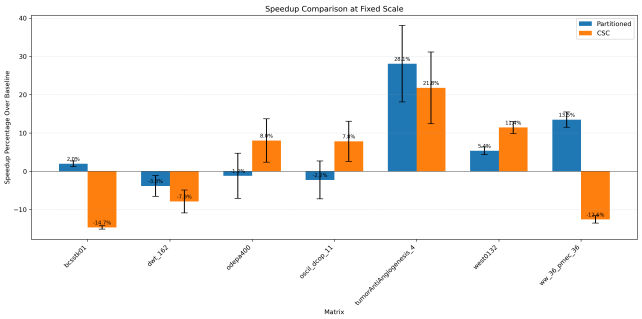

Figure 5: A comparison of speedups for two optimizations within a threaded SpMBV: 1) early computation (blue) and 2) additional early communication using CSC (orange).

Figure 5 shows the performance of a variety of Suitesparse matrices when multiplied by a block vector of width 32. The matrices were partitioned across 256 node, with one MPI process and two threads assigned to each of 112 cores per node. The performance of the optimizations varied per matrix. Using the early communication optimization, speedups ranged from 28 percent, the largest measured speedup, to a slowdown of 4 percent. For the combined optimization including early compute, we recorded a range of speedups from 22 percent in the best case to a slowdown of 15 percent in the worst case.

In Figure 6 we provide another comparison of speedups based on different parameters. We conducted this study on 64 nodes and assigned two MPI processes to each node, each utilizing 32 cores and 64 threads. Additionally, we tested a significantly larger block vector width of 131072. This case of a larger input vector and consequently larger messages resulted in more significant speedups from the optimizations. As opposed to the previous study, we did not measure slowdowns for any matrix tested. The early communication optimization resulted in a large speedup of 180 percent in the best case and a 3 percent speedup in the worst case. The early computation optimization resulted in a similar range of speedups but on different matrices, from the overall highest recorded speedup of 190 percent to a speedup of 2 percent in the poorest performing case.

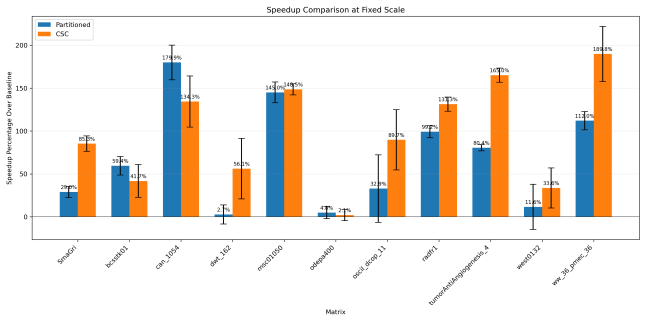

Figure 6: A comparison of speedups for two optimizations within a threaded SpMBV: 1) early computation (blue) and 2) additional early communication using CSC (orange).

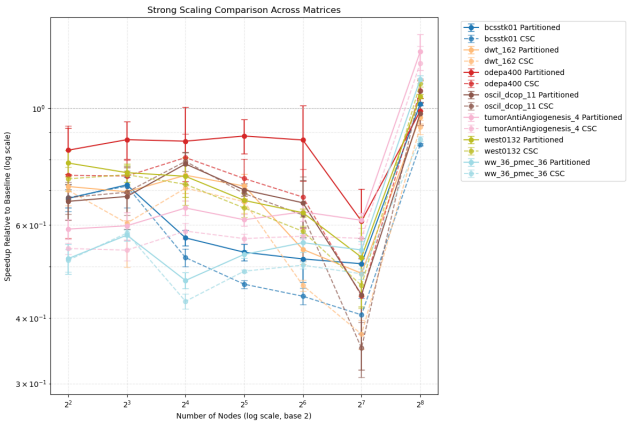

Figure 7: A strong scalign study of speedups for two optimizations within a threaded SpMBV over a range of matrices: 1) early computation (solid lines) and 2) additional early communication using CSC (dashed lines).

We present a strong scaling comparison of the two threaded SpMBV optimizations on several matrices in Figure 7. In this experiment we used a block vector width of 32 and one MPI process with two threads per core, totalling 112 MPI processes per node. Each matrix is plotted in a different color. Solid lines represent the speedup from early communication while dashed lines indicate speedups from the additional early computation optimization. We measured varying strong scaling behavior based on the sparsity and communication patterns of each matrix but generally the two optimizations resulted in either steady or increasing slowdowns up to 128 nodes. At the scale of 256 nodes, however, we found that performance sharply increased for all tested matrices resulting in speedups for some and reduced slowdowns for others. The maximum speedups we recorded for both optimizations occured using 256 nodes: 28 percent for early communication and 22 percent with the addition of early computation.

Since message partitioning introduces additional overheads for partition management, the payoff of optimizations based on partitioning depends on message size. For smaller messages, overheads

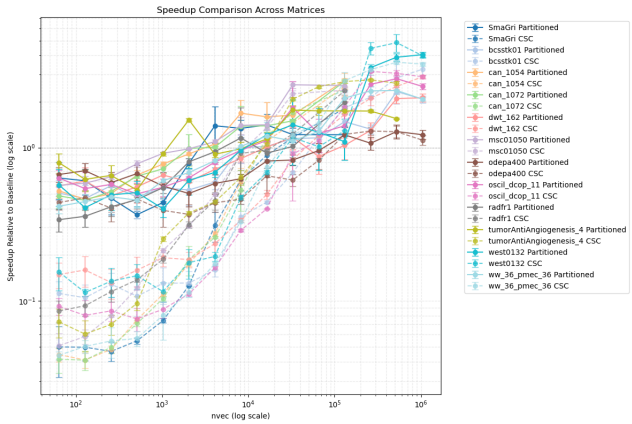

Figure 8: Measured speedups for threaded SpMBV optimizations over a range of values for the width (nvec) of the input block vector for several matrices.

can outweigh the speedup from optimizations and for larger messages, these overheads can be amortized resulting in an overall speedup. We present analysis of the effect of block vector width, and equivalently message size, on the speedups from the two threaded SpMBV optimizations in Figure 8. For this study we tested on 64 nodes and assigned 112 MPI processes per node. The performance we measured varied per matrix but followed the expected behavior described: slowdowns for smaller numbers of block vector columns and speedups for larger numbers of columns. The additional early compute optimization resulted in the worst recorded performance for smaller messages but as message size increased, performance began to overtake other approaches.

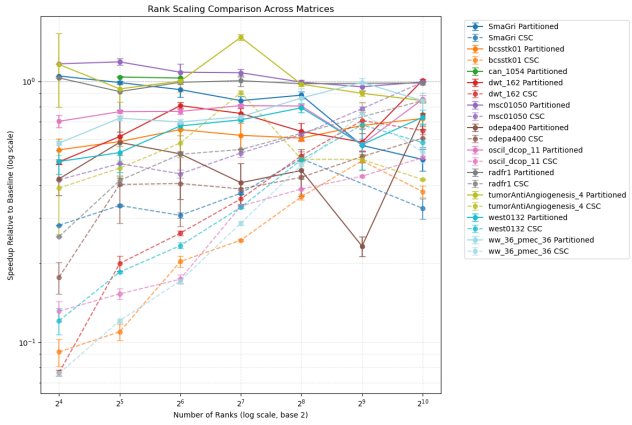

Figure 9: Measured performance for varying distributions of MPI processes (ranks) to available cores. Using 16 nodes, far left data was collected with one rank per node, far right data was collected with one rank per core.

Finally, we present a study of the choice of distribution of MPI processes accross available cores on a node in Figure 9. We ran this experiment on 16 nodes and tested a more square block vector with 131072 columns, essentially a square dense matrix. The available cores on a node can all be assigned to one MPI process, or multiple MPI processes can be assigned to a node, dividing available cores and threads among them, to a minimum of one core for each MPI process. We collected data for a range of process distributions. At one extreme we assigned one MPI process to each node, utilizing 64 cores and 128 threads. We tested increasing numbers of processes assigned to each node up to 64 per node, with each process using one core and two threads. The behavior we recorded suggests that for most matrices, early communication with partitioning is somewhat agnostic to the distribution of MPI processes, with some matrices performing better using fewer MPI processes per node, and others performing better with more. The addition of early computation with the CSC method, however, resulted in behavior more sensitive to process distribution. The performance we measured using this additional optimization was worse when assigning fewer MPI processes per node and better with more.

5 Conclusion and Future Directions

We present the implementation and performance analysis of three communication optimizations for SpMBV operations within parallel applications. These optimizations increase asynchrony and better overlap communication with computation through early communication using thread-level message partitioning and through early computation using the CSC matrix format. We analyze the performance of these optimizations under a variety of conditions and report speedups of up to 28 percent for a thinner block vector and 190 percent for a wider block vector closer to a square dense matrix. This work demonstrates how increased asynchrony between MPI processes and between threads can improve network utilization and overall communication performance in irregular boundary exchanges such as those in sparse matrix operations.

The optimized methods presented in this paper begin and end parallel OpenMP regions each iteration, launching threads repeatedly. These thread launches could be refactored to occur only once at initialization with one parallel region containing all iterations. Additionally, block updates to the result may result in improved performance over sporadic atomic memory updates as threads complete each multiplication. Task based models may provide further optimization by increasing the granularity of partitions and allowing for work stealing between threads while still achieving early communication. Furthermore, providing these optimizations within a neighborhood collective implementation could provide portable performance improvements. It may also be possible to combine the optimizations presented in this work with aggregation techniques introduced in previous work [7]. Small messages could be aggregated to avoid latency costs and result in larger messages more suitable to partitioning. Finally, a thorough analysis of the impact of the optimizations presented in this paper for various sparsity and communication patterns should be conducted to inform when each technique should be applied. This could consist of qualitative analysis for several example sparse matrices and their resulting message sizes and data dependency graphs, or if necessary training a model to predict optimization performance from sparsity patterns.

References

[1] HYPRE: High performance preconditioners. http://www.llnl.gov/CASC/hypre/.

[2] Bienz, A., Gropp, W. D., and Olson, L. N. Reducing communication in algebraic multigrid with multi-step node aware communication. *The International Journal of High Performance Computing Applications 34*, 5 (2020), 547–561.
[3] Brock, B., Buluç, A., and Yelick, K. Rdma-based algorithms for sparse matrix multiplication on gpus. In *Proceedings of the 38th ACM International Conference on Supercomputing* (New York, NY, USA, 2024), ICS '24, Association for Computing Machinery, p. 225–235.
[4] Buluc, A., and Gilbert, J. R. Challenges and advances in parallel sparse matrix-matrix multiplication. In *2008 37th International Conference on Parallel Processing* (2008), pp. 503–510.
[5] Buluç, A., and Gilbert, J. R. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing 34*, 4 (2012), C170–C191.
[6] Collom, G., Burmark, J., Pearce, O., and Bienz, A. Persistent and partitioned mpi for stencil communication. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)* (2024), pp. 1–7.
[7] Collom, G., Li, R. P., and Bienz, A. Optimizing irregular communication with neighborhood collectives and locality-aware parallelism. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (New York, NY, USA, 2023), SC-W '23, Association for Computing Machinery, p. 427–437.
[8] Dosanjh, M. G., Worley, A., Schafer, D., Soundararajan, P., Ghafoor, S., Skjellum, A., Bangalore, P. V., and Grant, R. E. Implementation and evaluation of mpi 4.0 partitioned communication libraries. *Parallel Computing 108* (2021), 102827.
[9] Fu, Q., Rolinger, T. B., and Huang, H. H. Jitspmm: Just-in-time instruction generation for accelerated sparse matrix-matrix multiplication. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2024), pp. 448–459.
[10] Gillis, T., Raffenetti, K., Zhou, H., Guo, Y., and Thakur, R. Quantifying the performance benefits of partitioned communication in mpi. In *Proceedings of the 52nd International Conference on Parallel Processing* (New York, NY, USA, 2023), ICPP '23, Association for Computing Machinery, p. 285–294.
[11] Grant, R. E., Dosanjh, M. G. F., Levenhagen, M. J., Brightwell, R., and Skjellum, A. Finepoints: Partitioned multithreaded mpi communication. In *High Performance Computing* (Cham, 2019), M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., Springer International Publishing, pp. 330–350.
[12] Holmes, D. J., Skjellum, A., Jaeger, J., Grant, R. E., Bangalore, P. V., Dosanjh, M. G., Bienz, A., and Schafer, D. Partitioned collective communication. In *2021 Workshop on Exascale MPI (ExaMPI)* (2021), IEEE, pp. 9–17.
[13] Islam, A. A. R., Xu, H., Dai, D., and Buluç, A. Improving spgemm performance through matrix reordering and cluster-wise computation, 2025.
[14] Jiang, P., Hong, C., and Agrawal, G. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2020), PPoPP '20, Association for Computing Machinery, p. 376–388.
[15] Labini, P. S., Bernaschi, M., Nutt, W., Silvestri, F., and Vella, F. Blocking sparse matrices to leverage dense-specific multiplication. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)* (2022), pp. 19–24.
[16] Lazzaro, A., VandeVondele, J., Hutter, J., and Schütt, O. Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5d algorithm and one-sided mpi. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (New York, NY, USA, 2017), PASC '17, Association for Computing Machinery.
[17] Lockhart, S., Bienz, A., Gropp, W., and Olson, L. Performance analysis and optimal node-aware communication for enlarged conjugate gradient methods. *ACM Trans. Parallel Comput. 10*, 1 (mar 2023).
[18] Niu, Y., Lu, Z., Ji, H., Song, S., Jin, Z., and Liu, W. Tilespgemm: a tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2022), PPoPP '22, Association for Computing Machinery, p. 90–106.
[19] Selvitopi, O., Brock, B., Nisa, I., Tripathy, A., Yelick, K., and Buluç, A. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *Proceedings of the 35th ACM International Conference on Supercomputing* (New York, NY, USA, 2021), ICS '21, Association for Computing Machinery, p. 431–442.
[20] Smith, B. *PETSc (Portable, Extensible Toolkit for Scientific Computation).* Springer US, Boston, MA, 2011, pp. 1530–1539.
[21] Solomonik, E., and Demmel, J. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In *Euro-Par 2011 Parallel Processing* (Berlin, Heidelberg, 2011), E. Jeannot, R. Namyst, and J. Roman, Eds., Springer Berlin Heidelberg, pp. 90–109.
[22] Trilinos Project Team, T. *The Trilinos Project Website*, 2024.
[23] Zambre, R., and Chandramowlishwaran, A. Lessons learned on mpi+ threads communication. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (2022), IEEE, pp. 1–16.