A More Scalable Sparse Dynamic Data Exchange

1st Andrew Geyko
Max Planck Institute
Universität des Saarlandes
Saarbrücken, Germany
ageyko@mpi-sws.org

2nd Gerald Collom
Department of Computer Science
University of New Mexico
Albuquerque, USA
geraldc@unm.edu

3rd Derek Schafer
Department of Computer Science
University of New Mexico
Albuquerque, USA
dschafer1@unm.edu

4th Patrick Bridges
Department of Computer Science
University of New Mexico
Albuquerque, USA
patrickb@unm.edu

5th Amanda Bienz
Department of Computer Science
University of New Mexico
Albuquerque, USA
bienz@unm.edu

***Abstract*—Parallel architectures are continually increasing in performance and scale while underlying algorithmic infrastructure often fails to take full advantage of available compute power. Within the context of MPI, irregular communication patterns create bottlenecks in parallel applications. One common bottleneck is the sparse dynamic data exchange, often required when forming communication patterns within applications. There is a large variety of approaches for these dynamic exchanges, with optimizations implemented directly in parallel applications. This paper proposes a novel API within an MPI eXtension library, allowing applications to utilize the variety of provided optimizations for sparse dynamic data exchange methods. Further, the paper presents novel locality-aware sparse dynamic data exchange algorithms. Finally, performance results show locality-aware approaches achieve up to 128x over existing approaches when exchanging only pattern of communication, and up to 54x when exchanging data to be communicated as well.**

***Index Terms*—sparse dynamic data exchange, MPI, HPC, parallel performance, locality-aware, extension library**

I. Introduction

Parallel architectures are continuously improving in scale, computational power, and efficiency, allowing for increasingly large and efficient high-performance computing (HPC) applications. Despite improvements in hardware capabilities, parallel applications fail to fully utilize emerging hardware due to the high costs associated with data movement and inter-process communication. Standard communication algorithms treat all communication as equal. However, the costs of various paths of communication on current and emerging computers vary greatly with relative locations of the sending and receiving processes, with notable differences between intra-socket, inter-socket, and inter-node communication.

A large number of parallel applications, including sparse solvers and simulations, are bottlenecked by the performance of irregular communication. This type of communication consists of each process sending messages to a subset of other processes, with the pattern of communication dependent on the specific problem being solved, such as the sparsity pattern of a given matrix. One bottleneck in irregular communication is forming a communication pattern, determining the set of processes to which each process must send as well as the set from which it receives.

The resulting communication pattern is application dependent. While structured problems require communication of a set number of neighboring processes, often up to 26, irregular communication can result in a range of patterns in which processes communicate with anywhere between none of and all of the other processes. Iterative methods and sparse solvers take a sparse matrix as input, such that each process holds a portion of the rows of the matrix. The communication pattern is dependent on the sparsity pattern of the given matrix, with each process receiving values corresponding to each non-zero column within its local rows. The number of processes with which each communicates can range from a handful to thousands, depending on the particular system to be solved and the number of processes across which it is partitioned.

Often, such as in the case of row-wise partitioned sparse matrix operations, each process can use the local sparsity pattern to determine what data should be received from other processes, but there is insufficient information regarding processes to which data should be sent. As a result, a crucial first step in every communication phase is determining this pattern of communication. The creation of this communication pattern requires a *sparse dynamic data exchange* phase (*SDDE*). In the context of the widely-used sparse matrix-vector multiply, the cost of creating the communication pattern through the SDDE often outweighs the cost of each subsequent step of communication.

Algebraic multigrid (AMG) is one example of a solver commonly bottlenecked by sparse matrix vector operations. AMG consists of a hierarchy of sparse matrices, throughout which solutions are iteratively refined through relaxation before restricting the problem to a coarser level. The finest level of the hierarchy is the system to be solved, with sparsity pattern dependent on input. On each coarse level, the matrix decreases in dimension, but increases in density, resulting in increasingly large communication requirements near the middle of the AMG hierarchy [1], [2]. Before SpMVs are performed, the communication pattern must be discovered for each increasingly dense matrix in the hierarchy through sparse dynamic data exchanges.

There are two main use cases for sparse dynamic data exchanges: exchanges with constant data sizes, such as determining only the communication pattern and per-message byte count, and variable-sized data exchanges, in which each process must dynamically send a variable amount of data to a subset of other processes. Cell adaptive mesh refinement applications, such as CELLAR [3] require an SDDE with constant size messages each time the mesh is altered. Each process can locally determine the subset of processes to which it must send and what data should be sent to each. However, an SDDE is needed to determine from which processes each receives, along with the size of each receive. This SDDE requires constant message sizes as only message sizes must be exchanged. A large majority of SDDEs, however, require variable-sized exchanges. Linear solvers typically consist of distributing sparse matrices across processes row-wise, allowing each process to locally determine the receive portion of the communication pattern. However, in this case, an SDDE is needed for each process to find the subset of processes to which it must send along with which data to send to each. As a result, indices corresponding to these data values must be sent, requiring each message to be equal to the size of subsequent iterations of communication, and therefore variable.

There are many existing approaches for the sparse dynamic data exchange, each hand implemented within applications that rely on the SDDE [4]–[6]. There are two typical implementations for the sparse dynamic data exchange, which can apply to both variable and constant data sizes. The personalized method consists of all processes performing an `MPI_Allreduce` to determine data exchange sizes before indices are dynamically communicated. Alternatively, the non-blocking implementation proposed by Torsten et al. [7] dynamically exchanges indices without the need for an initial reduction. The methods each outperform the other in a subset of sparse dynamic data exchange problems, with performance dependent on communication patterns, data exchange sizes, and process count. The personalized method outperforms the non-blocking approach in the case of SDDEs with very large message counts, as the required `MPI_Allreduce` has minimal overhead. However, SDDEs among a large set of processes in which there are few messages perform better when implemented through the non-blocking algorithm, avoiding the significant overhead of the reduction. Applications of constant-sized SDDEs, such as CELLAR, have further optimized the algorithm with one-sided MPI remote-memory access (RMA) communication. As all messages are of size `count`, each process p can use `MPI_Put` to directly put its contribution into the destination process's receive buffer at position $p \times$ `count`. This implementation, however, cannot be naturally extended to the variable-sized SDDE operations.

This paper presents optimizations to existing sparse dynamic data exchanges through message aggregation. Locality-aware optimizations greatly improve the performance and scalability of irregular data exchanges within the context of iterative methods. Communication within a region, such as intra-socket, greatly outperforms inter-region messages, such as inter-socket or inter-node. Locality-aware aggregation techniques gather data within a region and minimize the number of messages communicated between regions.

The contributions of this paper are the following.

- Presents a novel approach for locality-aware aggregation within sparse and dynamic communication;
- Presents locality-aware versions for each of the standard SDDE approaches;
- Presents new MPI eXtension APIs which allow for wrapping the presented algorithms behind a single API, increasing portability of the method, for dynamic exchanges of constant- or variable-sized data;
- Presents a performance study of the new approaches for both constant- and variable-sized dynamic exchanges across the communication patterns of 35 separate Suitesparse [8] matrices, and analyzes the speedup of the best locality-aware approach over the best standard approach;
- For constant-sized exchanges, speedup is achieved for all 35 communication patterns, with average and maximum speedups of 18x and 128x, respectively; and
- For variable-sized exchanges, speedup is achieved for 23 of the 35 matrices, with average and maximum speedups of 14x and 54x, respectively.

Further, all SDDE algorithms presented in this paper have been implemented behind the proposed API within the open-source MPI eXtension library MPI Advance [9], allowing for application programmers to replace current hand-implemented SDDE methods within widely used solvers.

The remainder of the paper is outlined as follows. An overview of the sparse dynamic data exchange problem and related works are presented in Section II. A common MPI eXtension (`MPIX`) API for SDDE algorithms is detailed in Section III. Existing algorithms for the SDDE are detailed in Section IV, while novel locality-aware extensions are described in Section IV-D. Scaling studies are presented in Section V, along with an analysis of when each algorithm performs best. Finally, concluding remarks and future work are discussed in Section VI.

II. Background

The sparse dynamic data exchange problem consists of a set of processes, each of which contains local data and is looking to operate on the global collection of data. Each process must receive a subset of the global data before it can complete its local portion of the operation. Initially, every process knows which global data it must receive from other processes. For instance, consider a sparse matrix operation, in which the sparse matrix and corresponding vectors are partitioned row-wise across all available processes. Each process must receive data corresponding to each non-zero column within its subset of rows. However, before data can be exchanged, each process must discover the subset of processes to which it must send all or part of its local data. The problem is formalized as follows.

Definition 1 (Sparse Dynamic Data Exchange Problem)
Let $\Sigma := \{\sigma_1, \ldots, \sigma_n\}$ be a collection of processes. For each $\sigma_i \in \Sigma$, let there be an associated local data segment D_i and a subset A_i of Σ of processes from which σ_i needs to receive data. The solution to the problem is a communication pattern such that after execution, each $\sigma_i \in \Sigma$ holds a copy of the set $\{j \mid \sigma_i \in A_j\}$.

Often, parallel applications rely on the SDDE problem when forming a communication pattern consisting of all processes to which each rank sends and receives data, the sizes of these messages, and the indices of the data to be communicated. There are multiple existing approaches to forming these communication patterns, including the personalized, non-blocking, and RMA approaches, detailed in Section IV. The performance of existing SDDE approaches consists of overhead costs, such as window synchronization or reduction costs, as well as the cost of point-to-point or one-sided dynamic communication of data. The trade-offs between existing implementations depend on whether the bottleneck of an SDDE is due to overhead or transfer of data. Factors that determine this overhead include the number of messages each process communicates, the size of these messages, and the total number of processes. Furthermore, the performance of the required point-to-point or one-sided communication is dependent on the locality of the messages, MPI implementation, and system architecture.

A. *Parallel Matrix Partitioning*

Applications and their underlying numerical methods, including Krylov subspace solvers, algebraic multigrid, ILU, and iterative smoothers, rely on sparse-matrix vector multiplication (SpMV). Sparse matrices are typically partitioned row-wise,

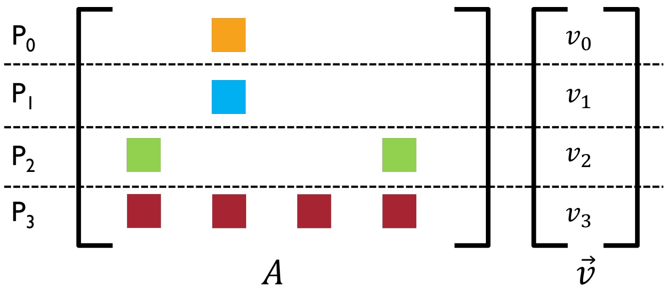


Fig. 1: An example of a 4×4 matrix and a vector distributed across 4 processes.

so that each process holds a contiguous subset of rows of the matrix along with corresponding vector values, as shown in Figure 1. Consider a system with n rows split row-wise across p processes. Each process holds $\frac{n}{p}$ consecutive rows of the matrix assuming the number of processes divides the number of rows; otherwise, some processes will hold one additional row. The vector values are partitioned similarly. While processes hold entire sparse rows, each column is considered either local or non-local, with local columns corresponding to local vector values and non-local columns corresponding to vector values local to other processes. There are other partitioning strategies, such as column-wise or 2.5d partitions [10], [11]. All SDDE strategies discussed in this paper are applied to row-wise partitions of sparse matrices and extend naturally to column-wise partitioning. While multi-dimensional partitioning strategies are outside the scope of this paper, the required SDDE problems are similar regardless of the partitioning strategy.

B. *Data Dependencies in Matrix-Vector Multiplications*

Consider the matrix displayed in Figure 1. When performing a matrix-vector product $A\vec{v}$ in parallel, each process must retrieve vector entries corresponding to off-process non-zeroes to perform the operation. For example, P_2 needs vector entries v_0 and v_3 in order to performs its local portion of the matrix-vector product. A complete data dependency chart is given by the following figure:

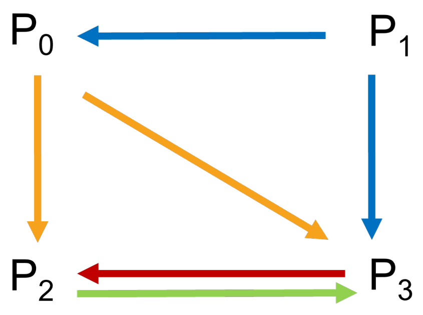


Fig. 2: The communication pattern associated with the example matrix in Figure 1

As shown in this example, a process does not know in advance which other processes need portions of its data as this communication pattern is dependent on the associated sparsity pattern. Therefore, before data can be exchanged, SDDE is required to determine the subset of processes with which each must communicate, along with which data must be communicated. Typically, this SDDE is performed once to create a communication package, which is then used during subsequent steps of communication. The SDDE requires communicating all non-zero column indices, similar in size to communicating the corresponding data within the vector during future steps of communication. However, the SDDE consists of dynamically receiving data with `MPI_Probe` operations, resulting in overheads from unexpected messages [12], [13]. As a result, the SDDE is often more expensive than subsequent exchanges unless a number of iterations are performed.

Communication patterns for irregular data exchanges are typically sparse, with each process communicating with a small subset of the total number of processes, typically $\mathcal{O}(\log(n))$ neighbors. For more dense communication patterns,

Send Variable	Input/Output	Definition	Receive Variable	Input/Output	Definition
`send_nnz`	Input	Number of dynamic sends	`recv_nnz`	Input/Output	Number of dynamic receives
`send_size`	Input	Size of dynamic sends	`recv_size`	Input/Output	Size of dynamic receives
`dest`	Input	Destinations of messages	`src`	Output	Sources of messages
`sendcounts`	Input	Send per-message count	`recvcounts`	Output	Receive per-message count
`sdispls`	Input	Send displacements	`rdispls`	Input	Receive displacements
`sendtype`	Input	Datatype of sends	`recvtype`	Output	Datatype of receives
`sendvals`	Input	Data to send	`recvvals`	Output	Data to receive

TABLE I: Variables required in MPIX sparse dynamic data exchange APIs. Red text indicates variables only required in the variable SDDE, while black variables are required in both APIs.

all-to-all exchanges are more efficient, removing the need for the SDDE phase and allowing for optimizations such as pairwise exchange.

C. *Related Works*

Irregular communication bottlenecks many parallel applications, including sparse solvers and simulations. As a result, there has been a significant amount of research into irregular communication optimizations. However, the majority of research into irregular communication optimizes messages after the communication pattern has been formed. Iterative point-to-point communication has been optimized within MPI through persistent communication [14], [15], which performs initialization once before reusing the communicator, amortizing setup costs over all iterations. MPI 4.0 provides partition communication [16], [17], an optimization for multithreaded applications that allows each thread to send a portion of a message when ready.

Many architecture-aware optimizations for irregular and collective communication utilize locality to reduce communication costs. Locality-aware aggregation minimizes the number and size of non-local messages, such as inter-node, through local, such as intra-node, aggregation of data [2], [18], [19]. These node-aware aggregation techniques assume an iterative nature to the communication, introducing overheads that are offset during per-iteration communication reductions. They would not yield performance gains for a dynamic exchange such as SDDE due to these initial overheads. Node-aware aggregation strategies, such as YGM, have also been applied to graph operations [20]. YGM communicates data less frequently to naturally aggregate messages, increasing the latency but decreasing the message count.

Aggregation has been used in regular, global communication, including collectives and I/O. Hierarchical collectives reduce inter-node communication costs by utilizing only one or a small number of MPI processes per node during inter-node communication [21], [22]. Further, multilane algorithms have been applied to collective operations, minimizing the size of inter-node messages, having each process per node send an equal and minimal amount of data [23]. Locality-aware aggregation has been applied to global all-to-all collectives and similar global collective I/O operations as well [24]–[27].

Communication constraints can be minimized through optimal partitions. Rather than a standard row-wise partition, graph or hypergraph partitioners minimize edge cuts between partitions [28]. Each partition is then assigned to a single process or node, reducing the amount of data needed to be exchanged throughout applications. While graph partitioners can be used in combination with sparse dynamic data exchanges, the use of these methods is outside the scope of this paper due to their large overheads.

III. MPI Extension APIs

A large variety of sparse dynamic data exchange implementations are used throughout existing parallel applications, with each application required to determine the optimal method for their program. This section presents the API that the authors have added to an open-source MPI eXtension library, MPI Advance. Existing methods and novel optimizations are added within the provided APIs, allowing an existing application to access any of the available optimizations rather than hand optimize. Further, the API allows for future research in which the optimal algorithm can be selected dynamically. Note that sparse data exchanges exist within MPI through neighbor collectives, but these methods do not allow for sparse *dynamic* data exchanges.

Sparse dynamic data exchange algorithms can be represented with two separate APIs: `MPIX_Alltoall_crs` and `MPIX_Alltoallv_crs`. All variables required for the APIs are defined in Table I.

The `MPIX_Alltoall_crs` method allows for dynamically exchanging constant data sizes with a sparse set of neighbors. One use case of this method occurs when processes know all data that must be sent to other processes, but have no knowledge of the receive portion of the communication pattern. This type of SDDE algorithm occurs within cell-based adaptive mesh refinement solvers, such as CELLAR [3].

The `MPIX_Alltoall_crs` API is defined in Figure 3. This method takes the send portion of the SDDE as input and returns received data. The variable `recv_nnz` is both input and output, allowing the user to input this information if it is already known rather than requiring the method to redetermine number of receives. Note the API is following current MPI standard requirements, and as a result `recvvals` must be allocated, potentially to some upper-bound, before the method is instantiated.

The `MPIX_Alltoallv_crs` method provides an interface for dynamically exchanging variable-sized data with a sparse set of neighbors. This method is required for any distributed sparse matrix operation, in which processes know

```
int MPIX_Alltoall_crs(int send_nnz,
        int* dest, int sendcount,
        MPI_Datatype sendtype,
        void* sendvals, int* recv_nnz,
        int* src, int recvcount,
        MPI_Datatype recvtype,
        void* recvvals, MPIX_Info* xinfo,
        MPIX_Comm* xcomm)
```

Fig. 3: `MPIX_Alltoall_crs` API

all data that must be received, but have no knowledge of processes to which they must send or what data must be sent to each. As a result, a variable-sized message must be exchanged containing this boundary information. Typically, this variable SDDE is used to initially form a communication package to then be used during each future sparse matrix operation, during which communication is typically implemented either through point-to-point messages or neighborhood collectives. Variable SDDEs exist within many widely used solvers, including Hypre BoomerAMG.

The `MPIX_Alltoallv_crs` API is defined in Figure 4. Similar to the non-variable API, this method takes the send

```
int MPIX_Alltoallv_crs(int send_nnz,
        int send_size, int* dest,
        int* sendcounts, int* sdispls,
        MPI_Datatype sendtype,
        void* sendvals, int* recv_nnz,
        int* recv_size, int* src,
        int* recvcounts, int* rdispls,
        MPI_Datatype recvtype,
        void* recvvals, MPIX_Info* xinfo,
        MPIX_Comm* comm)
```

Fig. 4: `MPIX_Alltoallv_crs` API

portion of the variable SDDE as input and returns received data. The parameters `recv_nnz` and `recv_size` are both input and output, allowing the user to input both receive count and/or receive size if either is known. Due to MPI standard requirements, this method also requires all pointers to be returned by the method to be allocated before they are passed to the method. This includes `recvcounts` and `rdispls`, which each need to be allocated to hold `recv_nnz` integers at minimum, as well as `recvvals`, which needs to be allocated to hold at least `recv_size` variables of size `recvtype`.

IV. Implementations

There are several existing methods for determining the communication pattern of a given irregular data exchange. The personalized and non-blocking methods, described in detail below, are widely used throughout existing parallel applications. Furthermore, an RMA-based optimization, as implemented within CELLAR, allows for `MPIX_Alltoall_crs` communication to be performed with one-sided communication. **All algorithms presented in this section, except for the RMA method, can be applied to both the `MPIX_Alltoall_crs` and `MPIX_Alltoallv_crs` methods, with minor changes required for the variable-sized SDDE, as highlighted in red.**

A. *Personalized Method*

The standard personalized method, utilized in a number of irregular codebases and described in [7], is detailed in Algorithm 1. The personalized protocol completes a sparse

Algorithm 1: Personalized

```
Input: rank                                  {Process ID}
args                        {MPIX_Alltoallv_crs Arguments}

ctr ← 0
for $i$ ← 0 to $send\_nnz$ do
    proc = dest[$i$]
    size = sendcounts[$i$]
    sizes[proc] = size
    MPI_Isend size at sendvals[ctr] to proc
    ctr ← ctr + size
MPI_Allreduce(sizes,MPI_SUM)
ctr ← 0
while $ctr < sizes[rank]$
    Probe for message and receive dynamically
    ctr ← ctr+ size of message
Wait for all sends to complete
```

dynamic data exchange by performing an `MPI_Allreduce` to gather the total number and size of messages that every process will receive during the dynamic communication phase. Data to be exchanged is then communicated with non-blocking sends. Finally, each process dynamically receives all messages sent to it using `MPI_Probe`.

When process counts are large, the overhead of the `MPI_Allreduce` increases. However, as message count increases, the reduction overhead is quickly outweighed by the dynamic exchange of data. The use of probes requires received data to be added to the unexpected message queue, further reducing performance for large messages. However, the `MPI_Allreduce` can yield large benefits as it allows all communication structures to be allocated before the dynamic communication step.

B. *Non-blocking Method*

The non-blocking method, presented in Torsten et al. [7] and detailed in Algorithm 2, is a modification of the standard algorithm which avoids the overhead and collective synchronization required by the `MPI_Allreduce`.

Algorithm 2: Non-blocking

```
Input: rank                                         {Process ID}
args                             {MPIX_Alltoallv_crs Arguments}

ctr ← 0
for i ← 0 to send_nnz do
    proc = dest[i]
    size = sendcounts[i]
    MPI_Isend size at sendvals[ctr] to proc
    ctr ← ctr + size
while All sends have not completed
    if Non-blocking probe finds a message
        Dynamically receive message
Non-blocking barrier
while Non-blocking barrier has not completed
    if Non-blocking probe finds a message
        Dynamically receive message
```

In this algorithm, each process p first sends non-blocking synchronous sends to every process to which it must communicate. A synchronous send is only considered complete when the destination process posts the associated receive. Each process dynamically receives messages until the synchronous sends have been completed on all processes. This is accomplished through MPI_Iprobe, which checks if a message is available, and only then is data received. While checking for messages, each process also tests its synchronous sends for completion Once all messages sent from a process have been received, that process calls a non-blocking barrier to indicate that it is finished sending. This process continues probing for messages until all processes have reached the barrier, at which point all processes' sends are received and dynamic communication is completed. While the non-blocking algorithm improves over the personalized method for large process counts by reducing synchronization and cost associated with the MPI_Allreduce, communication pattern structures must now be dynamically allocated. Furthermore, dynamic communication and unexpected messages remain a dominant cost of the method.

C. RMA Constant-Size SDDE

One-sided optimizations exist for sparse dynamic data exchanges in which all data is of a constant size, as implemented in CELLAR and detailed in Algorithm 3. Each process first allocates a shared memory window with enough space for all processes to add sendcount values. Then, each process can independently use MPI_Put to move the required data to position rank · sendcount on each corresponding process. Finally, each process can move received data from the window into the recvvals array.

The one-sided approach requires window creation, but this can be amortized over the cost of the application, and can potentially be reused within any subsequent one-sided communi-

Algorithm 3: RMA (MPIX_Alltoall_crs only)

```
Input: rank                                         {Process ID}
args                              {MPIX_Alltoall_crs Arguments}

Create window
Synchronize on window
proc_vals[num_procs × sendcount]
ctr ← 0
for i ← 0 to send_nnz do
    proc ← dest[i]
    orig_address ← sendvals[ctr]
    dest_address ←
     proc_vals[rank × sendcount]
    MPI_Put sendcount values from
     orig_address to dest_address
    ctr ← ctr + sendcount
Synchronize on window
for i ← 0 to num_procs do
    val ← proc_vals[i × sendcount]
    if val
        recvvals[recv_nnz++] ← val
```

cation. There is also a synchronization required to use a shared memory window. The authors currently use MPI_Fence for this, but more optimized forms of synchronization, such as locks, could be explored. RMA sparse dynamic data exchanges allow for all data to be exchanged without any dynamic MPI communication. However, the method does not naturally extend to variable-sized dynamic data exchanges. It can be used to determine communication sizes, followed by a standard data exchange to avoid dynamic messages, but the authors were unable to find a case where this would outperform other methods due to large matching costs. It is possible, however, that locality-aware data exchanges could improve the performance of the subsequent data exchange, but this optimization is outside the scope of this paper.

D. Locality-Aware Method

At large scales, the cost of both the personalized and non-blocking methods are bottlenecked by the point-to-point communication requirements. Irregular communication has been extensively modeled and analyzed [2], [18], [19], showing large costs associated with high inter-socket message counts due to lower bandwidth than intra-socket messages, injection bandwidth limits, queue search (or matching) costs, and network contention. This irregular communication can be optimized through locality-awareness, aggregating messages within a region, such as a socket, to minimize the number of messages communicated between regions. This greatly reduces the number of times inter-region latency is incurred along with associated queue search costs. While other locality-aware techniques remove duplicate values to reduce inter-region message sizes, this paper only concatenates messages as the overhead of determining duplicate values would outweigh the

Algorithm 4: Locality-Aware Personalized

```
Input: rank                                   {Process ID}
args                     {MPIX_Alltoallv_crs Arguments}
local_rank          {Local rank of process within region}
region_size              {Number of processes per region}

ctr ← 0
for i ← 0 to send_nnz do
    proc = dest[i]
    region = get_region(proc)
    size = sendcounts[i]
    Copy proc, size, and
      sendvals[ctr:ctr+size] into buf[region]
    ctr ← ctr + size
for each region to which rank sends
    proc = region·region_size+local_rank
    Non-blocking send of buf[region] to proc
MPI_Allreduce(sizes)
ctr ← 0
while ctr < sizes[rank]
    Probe for message and receive dynamically
    ctr ← ctr+ size of message
for proc ← 0 to region_size do
    sizes[proc] = size of buf[proc]
    Non-blocking send of buf[proc] to proc
MPI_Allreduce(sizes)
ctr ← 0
while ctr < sizes[rank]
    Probe for message and receive dynamically
    ctr ← ctr+ size of message
Wait for all sends to complete
```

benefits of reduced message size within a single iteration of communication, such as required during the SDDE problem.

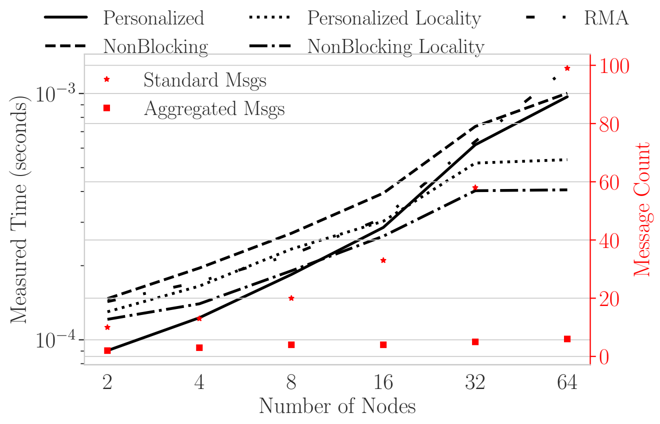
This paper presents locality-aware aggregation for both the personalized and non-blocking sparse dynamic data exchanges, as shown in Algorithms 4 and 5, respectively. Both locality-aware optimizations consist of concatenating all messages that a process p must send to any process within a single region. For example, the node-aware version of this method would aggregate all messages that process p sends to any process on node n, and send all data as a single message to a corresponding process. The corresponding process is determined to be the process in the region of destination with the same local rank as the sending process p, where local rank is determined as the rank of p within its region. For instance, if there are PPN processes per region and ranks are laid out sequentially across the regions, each process p has local rank $p \mod \texttt{PPN}$. A process with local rank r communicates only with other processes of local rank r during the inter-region step. Additional metadata, including the process of destination and size of each message, is sent with the concatenated message.

Algorithm 5: Locality-Aware Nonblocking

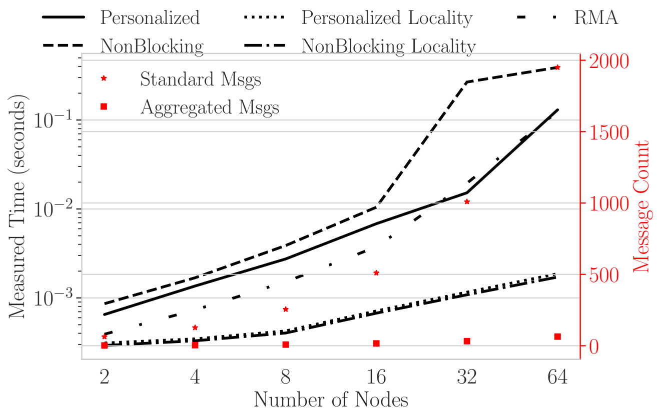
```
Input: rank                                   {Process ID}
args                     {MPIX_Alltoallv_crs Arguments}
local_rank          {Local rank of process within region}
region_size              {Number of processes per region}

for i ← 0 to send_nnz do
    proc = dest[i]
    region = get_region(proc)
    size = sendcounts[i]
    Copy proc, size, and
      sendvals[ctr:ctr+size] into buf[region]
    ctr ← ctr + size
for each region to which rank sends
    proc = region·region_size+local_rank
    Non-blocking send of buf[region] to proc
while All sends have not completed
    if Non-blocking probe finds a message
        Dynamically receive buf from origin_proc
        for each proc, size, indices in buf
            Copy origin_proc, size, indices
              into local_buf[proc]
Non-blocking barrier
while Non-blocking barrier has not completed
    if Non-blocking probe finds a message
        Dynamically receive buf from origin_proc
        for each proc, size, indices in buf
            Copy origin_proc, size, indices
              into local_buf[proc]
for proc ← 0 to region_size do
    sizes[proc] = size of buf[proc]
    Non-blocking send of buf[proc] to proc
MPI_Allreduce(sizes)
ctr ← 0
while ctr < sizes[rank]
    Probe for message and receive dynamically
    ctr ← ctr+ size of message
Wait for all sends to complete
```

After the inter-region communication is completed with either the personalized or non-blocking approach, all data is redistributed within the region. This is currently implemented as a personalized algorithm due to the fact that there are often a small number of processes within each socket or node, and they typically will redistribute data to a large percentage of other processes within the region. In recent years, the number of cores per node has largely increased. On emerging systems, such as sapphire rapids, it is possible that some sparsity patterns would be better optimized with the non-

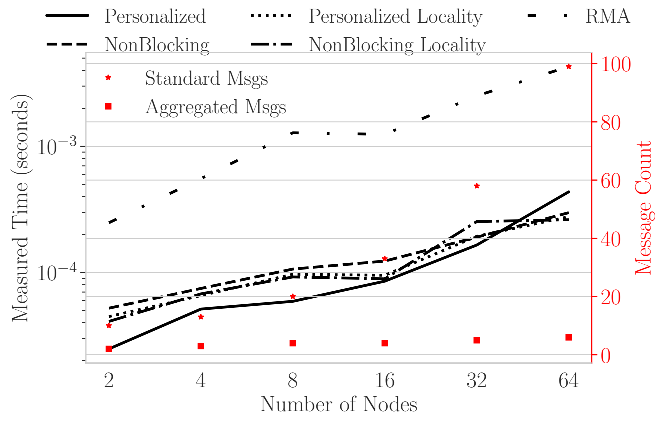


(a) dielFilterV2clx

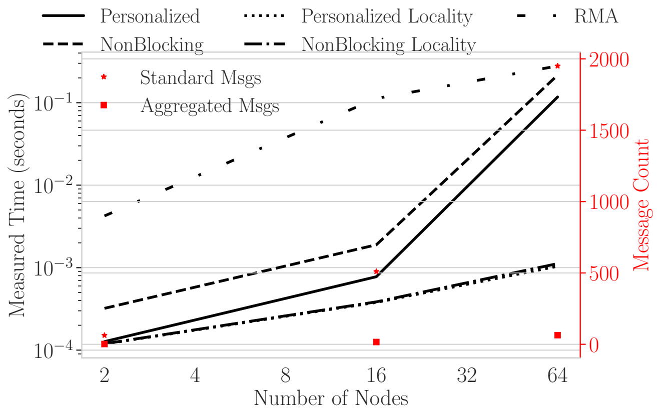


(b) NLR

Fig. 5: The cost of the various `MPIX_`**`alltoall`**`_crs` methods using **Mvapich2**.

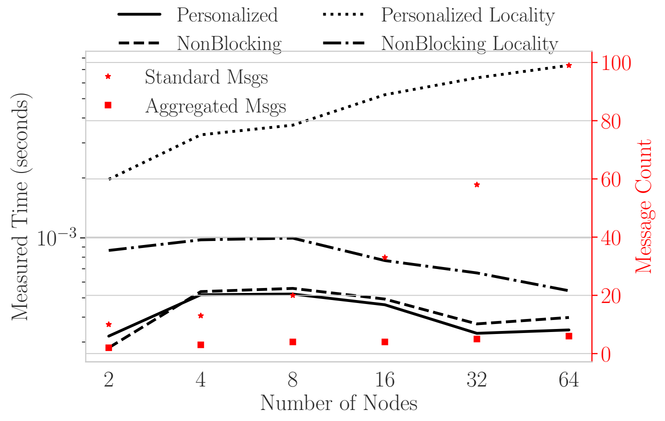


(a) dielFilterV2clx

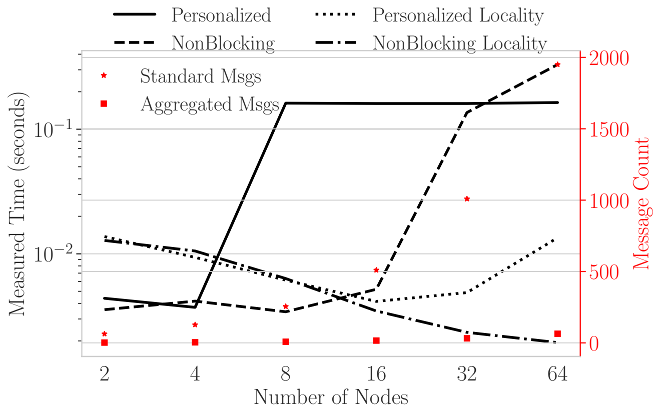


(b) NLR

Fig. 6: The cost of the various `MPIX_`**`alltoall`**`_crs` methods using **OpenMPI**.

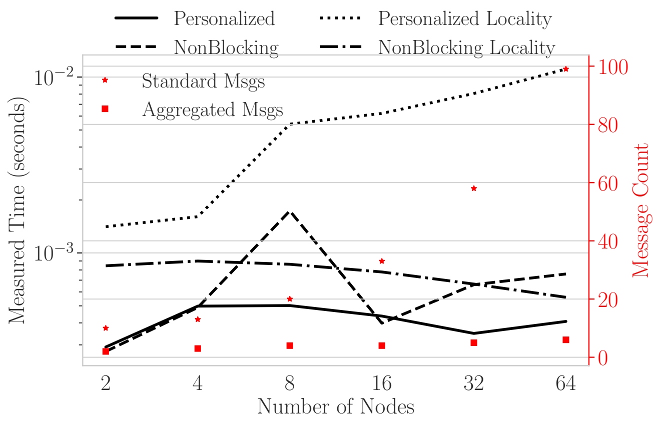


(a) dielFilterV2clx

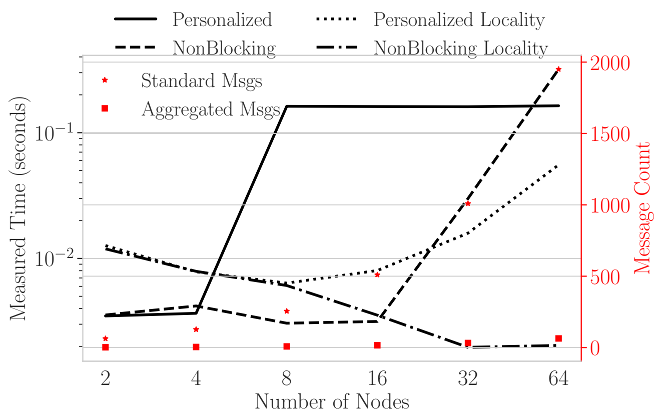


(b) NLR

Fig. 7: The cost of the various `MPIX_`**`alltoallv`**`_crs` methods using **Mvapich2**.



(a) dielFilterV2clx



(b) NLR

Fig. 8: The cost of the various `MPIX_`**`alltoallv`**`_crs` methods using **OpenMPI**.

blocking algorithm within each region. Additional possible optimizations include using an `MPI_Alltoallv` for dense intra-region redistribution. However, as inter-region communication bottlenecks the SDDE, intra-region communication optimizations are beyond the scope of this paper.

Note that the locality-aware methods assume a constant number of processes per node. These algorithms could be extended for variable sized node counts by finding the smallest number of processes on any node, p, and sending inter-node messages to `local_rank` mod p.

V. Results

The section analyzes the performance of the various SDDE algorithms across a subset of the largest 35 matrices from the Suitesparse Matrix Collection with non-zero counts under 25 million. These matrices are small enough to be stored on just two nodes but also large enough to be scaled out to 64 nodes.

Each matrix is partitioned row-wise across all available processes so that each holds an equivalent number of rows. The columns with non-zero values determine communication requirements. For example, if process p holds a non-zero in column n, and process q holds the row n, process p will send a dynamic message to process q during the SDDE.

Two types of SDDE problems are tested.

1) **Constant-Sized SDDE:** If process p holds any non-zero column corresponding to some process q, it will send a dynamic message to q with the number of associated non-zero columns. Therefore, each message within the constant-sized SDDE is a single integer, representing only the size of future steps of communication for the given pattern.
2) **Variable-Sized SDDE:** If process p holds any non-zero column corresponding to some process q, it will send a dynamic message to q with the indices of all associated non-zero columns. Therefore, each message within the variable-sized SDDE is equal to the number of corresponding non-zero columns and contains the indices to be sent during future steps of communication for the given pattern.

Two suitesparse matrices are scaled across nodes, from 2 to 64 nodes, showing the impact of process count on the various SDDE approaches. These matrices are detailed in Table II.

Matrix	# Rows	# Cols,	# Nonzeros
dielFilterV2clx	607,232	607,232	25,309,272
NLR	4,163,763	4,163,763	24,975,952

TABLE II: Matrix dimensions

A. Parallel Environment:

The SDDE algorithms are analyzed on the Quartz supercomputer at Lawrence Livermore National Lab. Each node of Quartz contains two Intel Xeon E5-2695 v4 processors, totaling 36 available CPU cores per node. All algorithms are tested on both system versions of MPI available on Quartz, OpenMPI 4.1.2, and Mvapich2 2.3.7. For simplicity, 32 processes per node are used in each test, allowing for power of 2 process counts.

B. Constant-Sized SDDE

Figures 5 and 6 show the performance of the `MPIX_Alltoall_crs` with OpenMPI and MVAPICH2, respectively, for the two suitesparse matrices detailed in Table II. The costs of the various `MPIX_Alltoall_crs` algorithms are displayed as black lines, while the maximum number of inter-node messages sent by any process are displayed as red dots. The locality-aware SDDE methods communicate only the aggregated number of inter-node messages while the personalized, non-blocking, and RMA approaches all communicate the standard message count. All non-aggregated message sizes are a single integer.

For all matrices, the maximum number of inter-node messages is greatly decreased with aggregation. At the smallest scales, the personalized method often outperforms the others. The overhead of the required `MPI_Allreduce` is minimal

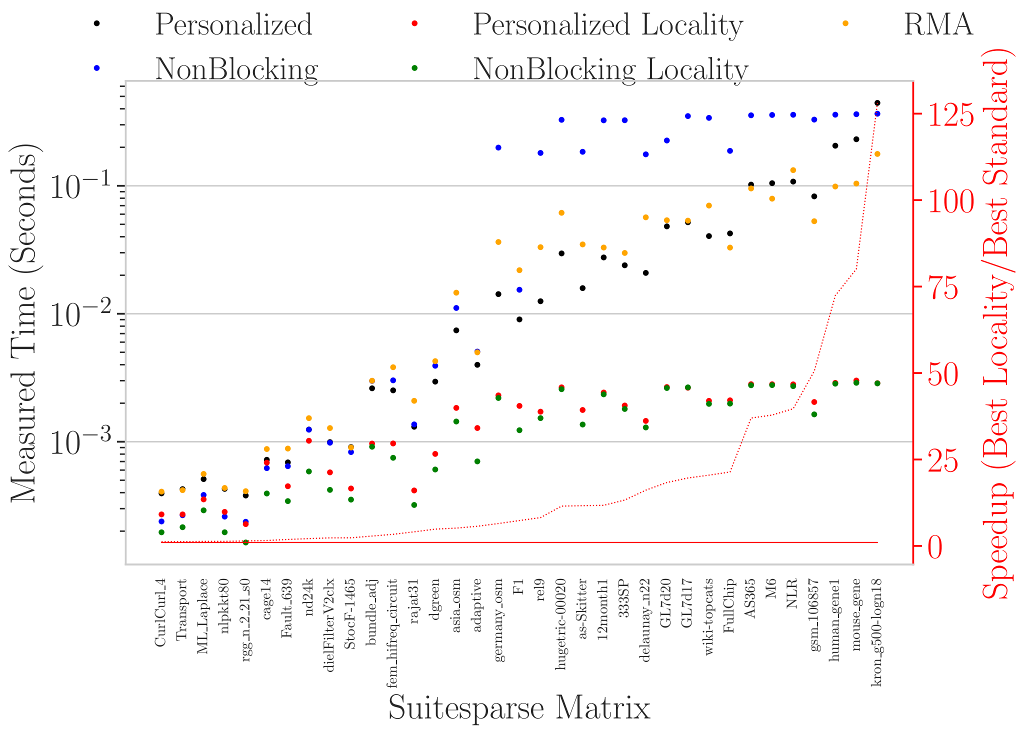


Fig. 9: The cost of the various `MPIX_alltoall_crs` using Mvapich on 64 nodes. The dots represent measured times, the dotted red line shows the speedup of the best locality-aware approach over the best standard approach, and the solid red line shows the baseline speedup of 1.

at these smaller scales. Further, message counts are smaller, reducing benefits of aggregation. As a result, the overhead of aggregating data and sending extra intra-region communication does not pay off at the smallest scales.

At large scales, the locality-aware non-blocking SDDE outperforms the other methods. The non-blocking approaches do not require `MPI_Allreduce` operations, which become expensive at larger scales. Further, aggregation greatly reduces message counts, outweighing overheads associated with concatenating messages and increasing intra-region communication. The performance at all scales varies with MPI implementation.

Figure 9 measures the performance of the various SDDE algorithms on all 35 suitesparse matrices at the largest scale of 64 nodes. The red dotted line shows the speedup of the best locality-aware approach over the best standard approach. The solid red line shows the baseline of 1. All 35 matrices achieve speedup. As the cost of the SDDE increases, the speedup associated with locality-aware SDDE approaches increases accordingly. The constant-sized locality-aware SDDE achieves up to 128x speedup, with an average speedup of 18x across all matrices.

C. Variable-Sized SDDE

Figures 7 and 8 analyze the costs of the various `MPIX_Alltoallv_crs` algorithms using OpenMPI and MVAPICH2, respectively, scaling the two suitesparse matrices detailed in Table II across various process counts. As the variable-sized SDDE methods exchange indices to be communicated in future exchanges, message sizes vary with sparsity pattern. However, aggregation changes message sizes within the SDDE only minimally, adding only metadata of per-process message sizes to the aggregated messages. Both the standard and aggregated message counts are equivalent to those within the `MPIX_Alltoall_crs` tests. Note, there are no RMA results as the RMA approach only applies to `MPIX_Alltoall_crs` methods.

At small scales, the standard approaches outperform their locality-aware counterparts, as aggregation only minimally reduces message counts when few processes are involved. As a result, the overhead of aggregating data and increasing intra-node exchanges outweighs any benefits of aggregated messages.

At larger scales, the locality-aware non-blocking SDDE outperforms the other approaches for NLR. However, the personalized method remains optimal for dielFilterV2clx. The message counts vary drastically between the two sparsity patterns, with dielFilterV2clx requiring only up to 100 messages per process at the largest scale, which NLR requires nearly 2000. Locality-aware aggregation is much more impactful with the larger message counts.

Figure 10 displays the cost of the SDDE algorithms on each of the 35 suitesparse matrices at the largest scale of 64 nodes. The red dotted line shows the speedup of the best locality-

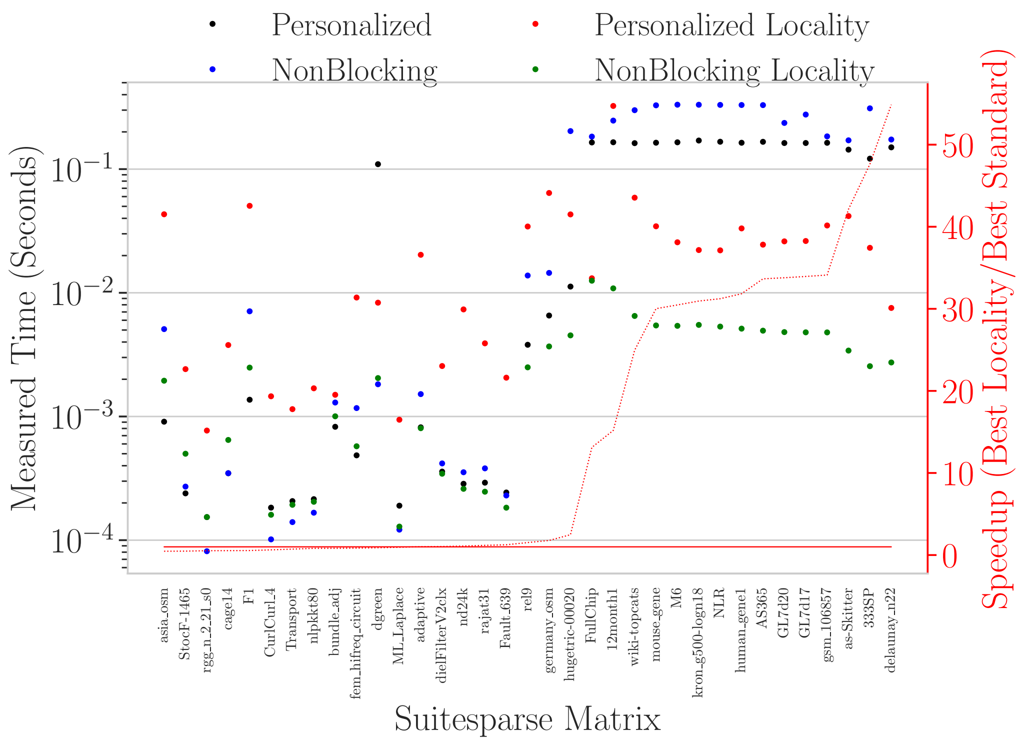


Fig. 10: The cost of the various `MPIX_`**`alltoallv`**`_crs` using Mvapich on 64 nodes. The dots represent measured times, the dotted red line shows the speedup of the best locality-aware approach over the best standard approach, and the solid red line shows the baseline speedup of 1.

aware approach over the best standard approach. The solid red line shows the baseline of 1. Of the 35 matrices, 23 achieve speedup, with a maximum speedup of 54x and an average speedup of 14x.

VI. Conclusions and Future Directions

The optimal sparse dynamic data exchange algorithm varies with communication pattern, problem size, process count, MPI implementation and architecture. As per-region process counts increase, the potential for locality-aware optimizations increases. The impact of locality-aware aggregation increases with the number of original messages to a single region, as they can be aggregated into a single message, greatly reducing inter-region message count. Parallel applications with large irregular communication constraints are expected to achieve significant benefits from locality-aware SDDE algorithms at strong scaling limits. When tested across 64 nodes, with 32 processes per node, the locality-aware approaches achieve up to over 50x speedup over the best non-locality-aware approach for matrices with high message counts, while incurring slowdown for matrices that have minimal communication requirements. This impact is seen for both the `MPIX_Alltoall_crs` and `MPIX_Alltoallv_crs` methods. While this paper did not explore locality-aware aggregation for the RMA method, similar concatenation strategies could be used within `MPI_Puts` to reduce the synchronization overheads as well as communication costs.

Future performance models are needed to dynamically select the optimal SDDE algorithm for given sparsity patterns on each existing and emerging architecture. Furthermore, as heterogeneous architectures increase in prevalence, the SDDE algorithm should be adapted for these systems. There are an increasingly large number of possible paths of data movement on emerging systems, including GPUDirect, copying to a single CPU, and copying portions of data to all available CPU cores, which should be combined with existing algorithms to optimize emerging heterogeneous architectures.

Acknowledgment

This work was performed with partial support from the National Science Foundation under Grant Nos. CCF-2151022 and CCF-2338077, the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966, and under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-854677).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the U.S. Department of Energy's National Nuclear Security Administration.

References

[1] S. M. R. Pichahi, "Improving the performance and scalability of algebraic multigrid," Ph.D. dissertation, The University of Utah, 2021.

[2] A. Bienz, W. D. Gropp, and L. N. Olson, "Reducing communication in algebraic multigrid with multi-step node aware communication," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 547–561, 2020.

[3] G. Shipman, "Cellar," https://github.com/lanl/CELLAR, 2022.

[4] "*hypre*: High performance preconditioners," https://llnl.gov/casc/hypre, https://github.com/hypre-space/hypre.

[5] T. Trilinos Project Team, *The Trilinos Project Website*, 2020 (acccessed May 22, 2020). [Online]. Available: https://trilinos.github.io

[6] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[7] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *SIGPLAN Not.*, vol. 45, no. 5, p. 159–168, jan 2010. [Online]. Available: https://doi.org/10.1145/1837853.1693476

[8] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[9] A. Bienz, D. Schafer, and A. Skjellum, "Mpi advance : Open-source message passing optimizations," in *EuroMPI'23: 30th European MPI Users' Group Meeting*, 2023. [Online]. Available: https://eurompi23.github.io/assets/papers/EuroMPI23_paper_33.pdf

[10] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005. [Online]. Available: https://doi.org/10.1137/S0036144502409019

[11] A. Lazzaro, J. VandeVondele, J. Hutter, and O. Schütt, "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*, ser. PASC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3093172.3093228

[12] R. Keller and R. L. Graham, "Characteristics of the unexpected message queue of mpi applications," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–188.

[13] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating mpi message matching misery," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 281–299.

[14] Y. Ishikawa, K. Nakajima, and A. Hori, "Revisiting persistent communication in mpi," in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 296–297.

[15] M. Hatanaka, A. Hori, and Y. Ishikawa, "Optimization of mpi persistent communication," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 79–84. [Online]. Available: https://doi.org/10.1145/2488551.2488566

[16] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned multithreaded mpi communication," in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds. Cham: Springer International Publishing, 2019, pp. 330–350.

[17] M. G. Dosanjh, A. Worley, D. Schafer, P. Soundararajan, S. Ghafoor, A. Skjellum, P. V. Bangalore, and R. E. Grant, "Implementation and evaluation of mpi 4.0 partitioned communication libraries," *Parallel Computing*, vol. 108, p. 102827, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819121000752

[18] A. Bienz, W. D. Gropp, and L. N. Olson, "Node aware sparse matrix–vector multiplication," *Journal of Parallel and Distributed Computing*, vol. 130, pp. 166–178, 2019.

[19] M. Hidayetoğlu, T. Bicer, S. G. de Gonzalo, B. Ren, V. De Andrade, D. Gursoy, R. Kettimuthu, I. T. Foster, and W.-m. W. Hwu, "Petascale xct: 3d image reconstruction with hierarchical communications on multi-gpu nodes," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[20] B. Priest, T. Steil, G. Sanders, and R. Pearce, "You've got mail (ygm): Building missing asynchronous communication primitives," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 221–230.

[21] J. L. Träff and A. Rougier, "Mpi collectives and datatypes for hierarchical all-to-all communication," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 27–32. [Online]. Available: https://doi.org/10.1145/2642769.2642770

[22] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, and J. Dongarra, "Han: a hierarchical autotuned collective communication framework," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 23–34.

[23] J. L. Träff and S. Hunold, "Decomposing mpi collectives for exploiting multi-lane communication," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 270–280.

[24] A. Bienz, L. Olson, and W. Gropp, "Node-aware improvements to allreduce," in *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, 2019, pp. 19–28.

[25] A. Bienz, S. Gautam, and A. Kharel, "A locality-aware bruck allgather," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 18–26. [Online]. Available: https://doi.org/10.1145/3555819.3555825

[26] G. Chochia, D. Solt, and J. Hursey, "Applying on node aggregation methods to mpi alltoall collectives: Matrix block aggregation algorithm," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 11–17. [Online]. Available: https://doi.org/10.1145/3555819.3555821

[27] Q. Kang, S. Lee, K. Hou, R. Ross, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving mpi collective i/o for high volume non-contiguous requests with intra-node aggregation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2682–2695, 2020.

[28] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*. Cham: Springer International Publishing, 2016, pp. 117–158. [Online]. Available: https://doi.org/10.1007/978-3-319-49487-6_4