Nonuniformly Communicating Noncontiguous Data: A Case Study with PETSc and MPI

P. BALAJI, D. BUNTINAS, S. BALAY, B. SMITH, R. THAKUR AND W. GROPP

Technical Report:

Argonne National Laboratory (ANL/MCS-P1380-1006)

Nonuniformly Communicating Noncontiguous Data: A Case Study with PETSc and MPI [*] [†]

P. Balaji[‡] D. Buntinas S. Balay B. Smith R. Thakur W. Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
{balaji, buntinas, balay, bsmith, thakur, gropp}@mcs.anl.gov

Abstract

Due to the complexity associated with developing parallel applications, scientists and engineers rely on high-level software libraries such as PETSc, ScaLAPACK and PESSL to ease this task. Such libraries assist developers by providing abstractions for mathematical operations, data representation and management of parallel layouts of the data, while internally using communication libraries such as MPI and PVM. With high-level libraries managing data layout and communication internally, it can be expected that they organize application data suitably for performing the library operations optimally. However, this places additional overhead on the underlying communication library by making the data layout noncontiguous in memory and communication volumes (data transferred by a process to each of the other processes) nonuniform. In this paper, we analyze the overheads associated with these two aspects (noncontiguous data layouts and nonuniform communication volumes) in the context of the PETSc software toolkit over the MPI communication library. We describe the issues with the current approaches used by MPICH2 (an implementation of MPI), propose different approaches to handle these issues and evaluate these approaches with micro-benchmarks as well as an application over the PETSc software library. Our experimental results demonstrate close to an order of magnitude improvement in the performance of a 3-D Laplacian multi-grid solver application when evaluated on a 128 processor cluster.

1 Introduction

Developing large-scale parallel applications and simulations has been a cumbersome and inherently daunting task, owing to the complexity of current generation parallel systems. Accordingly, scientists and engineers rely on high-level software libraries such as PETSc [1, 2], ScaLAPACK [3] and PESSL to ease implementation and minimize the development cycle required to parallelize their application. These high-level software libraries internally use parallel communication libraries such as the Message Passing Interface (MPI) [6] or the Parallel Virtual Machine (PVM) [20] to move appropriate data between processes as needed.

Most scientific applications formulate the physical equations corresponding to problems in discrete numerical forms. Similarly, they represent the domain on which they intend to solve the equation into a grid of data points. Such formulation involves representing the problem domain (1-D, 2-D or 3-D) as a structured grid, unstructured grid or adaptive mesh, and evaluating the discretized equations using the associated data. High-level software libraries assist such a representation by providing abstractions for mathematical operations, data representation and management of parallel layouts of the data as required by the application.

With the high-level libraries managing the data layout internally, it is only intuitive for them to choose to organize the data in representations suitable for performing the library operations optimally. However, this places additional overhead on the underlying communication library in two aspects:

Noncontiguous data layout: Since the overall data layout is optimized for the library operations, the partial data that needs to be communicated to other processes tends to be laid out noncontiguously in memory. For example, representing a 2-D grid as a contiguous vector might be optimal for computation. However, if a process needs to communicate a column of the grid to its neighbor, this partial data is now noncontiguously laid out in memory with a uniform stride.

Nonuniform communication volumes: In grid layouts (e.g., structured grids) several applications communicate only (or mostly) with their immediate neighbors. Depending on the mapping of data to processes in each dimension (e.g., a process could potentially manage a nonsquare region in a 2-D grid) and different discretization models (e.g., star or box-type stencils, as will be described in Section 2), the amount of data communicated to different neighbors can be different. In short, depending on the way the high-level library manages data, a process can communicate vastly different volumes of data with each of the other processes in the system.

In this paper, we analyze the overheads associated with these two aspects (noncontiguity in data layout and nonuniformity in communication volumes), in the context of the PETSc software toolkit over MPI. We describe the issues with current approaches used by MPICH2 [17] (an implementation of MPI) and its derivatives (e.g., MVAPICH2 [10] from Ohio State University) and propose different approaches for handling these issues. Specifically, for the first aspect (nonuniform data processing), we propose a new dual-headed lookup-based design for handling inefficiencies with noncontiguous

[*] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

[†] The authors would like to thank Prof. Panda and his team from the Ohio State University for allowing us access to their 64-node InfiniBand cluster.

[‡] This author's work was also supported in part by the National Science Foundation Grant #0702182.

data processing. For the second aspect (nonuniform communication volumes), we describe multiple designs for different operations that support nonuniform communication volumes (e.g., `MPI_Allgatherv`, `MPI_Alltoallw`).

Apart from proposing and describing the different designs, we also experimentally evaluate each of them to illustrate the inefficiencies associated with existing approaches and the benefits obtainable with our approaches. Next, we evaluate our integrated framework (comprising of all our new designs) with the PETSc toolkit to understand the impact of these designs on high-level software libraries. Finally, we evaluate our framework with a 3-D Laplacian multi-grid solver application using the PETSc library over MPI. Our experimental results demonstrate that most current approaches do not follow any sophisticated mechanisms to enhance the nonuniform data processing and communication and use identical mechanisms as uniform data processing; this leads to significant loss in the performance and scalability of applications. On the other hand, sophisticated schemes specifically tuned for nonuniform data communication can avoid such inefficiencies and achieve high performance. For example, with our framework, we could achieve close to an order of magnitude improvement in the performance of a 3-D Laplacian multi-grid solver application on a cluster of 128 processors.

2 The PETSc Toolkit

The Portable Extensible Toolkit for Scientific Computation (PETSc) was developed in the Mathematics and Computer Science Division at the Argonne National Laboratory. PETSc is a set of software tools for users writing large-scale application code involving solutions to Partial Differential Equations (PDEs). Application domains currently using PETSc include Nanosimulations, Biological sciences, Fusion, Geosciences, Environmental/Subsurface flow, Computational Fluid Dynamics, Wave propagation and others.

Figure 1 describes the abstract components of PETSc. Like other high-level software libraries, PETSc provides a suite of data structures and routines to create vectors, matrices, and distributed arrays (sequential and parallel). It also provides routines for linear and nonlinear numerical solvers to be used in applications written in C, C++, Fortran and Python. Timestepping methods and graphics are incorporated as well.

PETSc utilizes MPI for inter-process communication while providing implicit message passing for the application, i.e., it transparently handles the moving of data between processes without requiring the application to directly call any data transfer routines. This includes handling parallel data layouts (parallel vectors and matrices), communicating ghost point data (data points that are needed for computation, but reside on a different process memory) and others.

2.1 Handling Parallel Data Layouts

As mentioned earlier, PETSc provides mathematical abstractions like matrices, vectors, and utilities like distributed arrays to represent a grid in a parallel manner, thus distributing data objects across processors (Figure 2). However, to evaluate a local function, each process requires its local portion of the data as well as its bordering portions or *ghost positions*, that are owned by the neighboring processes. This data layout pattern presents three interesting challenges in communication: (i) noncontiguous data layout, (ii) nonuniform communication volumes to different processes and (iii) nonuniform set of processes that are communicated with.

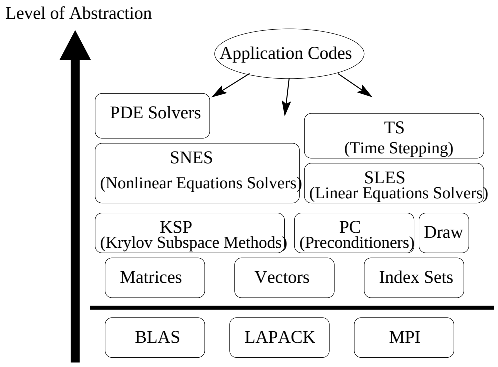

Figure 1: PETSc Architecture

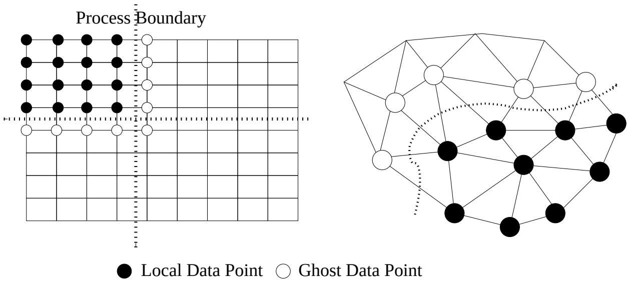

Figure 2: Ghost Points: Bordering portions of process' local data that is used for computation

Noncontiguous data layout: Application data is represented in a PETSc vector object which is a contiguous array on each processor. When this data corresponds to a multidimensional structured grid, the array values are contiguous on the first dimension but strided on the other dimensions. Further, each grid point might have multiple field values (for e.g., pressure, temperature, x-velocity and y-velocity) which get stored interlaced in the PETSc vector. This representation allows PETSc to perform the required mathematical operations in an efficient manner. Now, when the data corresponding to the ghost points needs to be communicated to a neighboring process, PETSc uses MPI to perform this data transfer efficiently. However, from MPI's perspective, the data that needs to be communicated is now laid out in a noncontiguous manner. As we will see in Section 3, handling this efficiently is a non-trivial issue that needs to be addressed.

Nonuniform communication volumes: Depending on the discretization model used, the number of neighbors that need to be contacted by a process and the volume of data that needs to be transferred to each neighbor differs. For example, Figure 3 illustrates two such discretization models, namely box-type stencil and star-type stencil. As shown in the figure, in

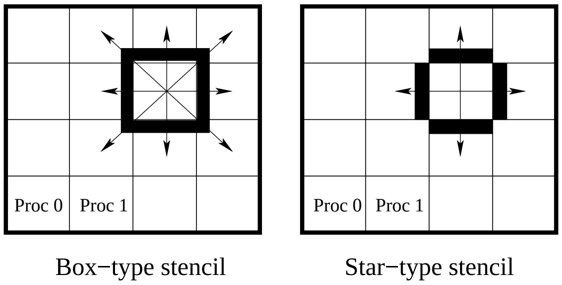

Figure 3: Data Layout Models: Box-type Stencil and Star-type Stencil

a star-type stencil, each process has to communicate with two neighbors in each dimension, i.e., 4 processes in a 2-D grid. The volume of communication in each dimension is the same, but could differ across dimensions (e.g., if a process manages a nonsquare portion of the grid). A box-type stencil is more complicated with respect to the communication pattern. In this model, each process has to communicate with four neighbors in each dimension, two along the sides and two along the corners. The interesting aspect in this communication pattern is that the volume of data communicated to the neighbors that share a side is typically much larger than the volume of data communicated to the neighbors that share only a corner. Further, as the dimensionality of the grid increases, the number of different volumes that need to be communicated could potentially increase quadratically.

Nonuniform set of processes to communicate with: As mentioned earlier, applications solving PDEs typically perform communication only with their neighbors. This is handled with PETSc using MPI collective operations (e.g., MPI_Alltoallw) where each process communicates a non-zero amount of data to its neighbors and zero data to the rest of the processes. This can be considered to be a special case of the previous aspect (nonuniform communication volumes), but is relevant to note separately due to its potential to minimize the impact of skew if designed appropriately. It is to be noted that all processes perform communication with a set of processes during the collective operation, but each process does not communicate with every other process. Thus, if a set of processes are delayed, if appropriately designed, the collective communication operation should not delay a process which does not belong to the set of delayed processes.

3 Understanding upper-layer requirements on MPI

In this section we describe implications of the requirements from upper-layers (such as high-level software libraries) on MPI. Specifically, we analyze MPICH2 (a high-performance open-source MPI implementation from Argonne National Laboratory) and its derivate MPI implementations (e.g., MVAPICH2 over the InfiniBand network from the Ohio State University), and identify design issues that affect its capability to handle the requirements from upper layers.

3.1 Issues with Noncontiguous Data

MPI provides a mechanism to describe noncontiguous memory regions, using MPI *derived datatypes*. Once a derived datatype has been created describing the noncontiguous region, it can be passed as a parameter to MPI communication functions and the MPI library will handle sending the noncontiguous data. In this way, the use of datatypes in an application simplifies coding. The alternative to using MPI datatypes is for the programmer to explicitly pack the noncontiguous data into a contiguous buffer then send that buffer, and on the receiving side, the programmer would have to receive the data into a contiguous intermediate buffer and unpack it.

Let us consider an 8x8 2-D matrix where each element consists of three double-precision floating-point values (Figure 4). The date in the first column of the matrix would be laid out noncontiguously in memory as shown in Figure 5. Figure 6 depicts a datatype that could be used to describe this column. Datatypes are defined recursively, where a new datatype is built up from other datatypes. In this case a single matrix element is described by a *contiguous* datatype of three *doubles*. The column of these elements is described by a *vector*, with a *stride* of eight, of the element datatype.

Although the MPI implementation can use network operations which can directly send from or receive into noncontiguous buffers, e.g., writev in the sockets API, these generally perform well only for datatypes which are *dense*, i.e., which have medium to large segments of contiguous data. For *sparse* noncontiguous datatypes, which consist of many short contiguous segments, it is often more efficient to pack the data into an intermediate buffer before sending (Figure 7). Further, most MPI implementations perform packing of the data and the actual communication from the packed buffer in a pipelined manner to improve performance. Accordingly, an intermediate buffer of a certain size (depending on the pipelining granularity) is allocated and data packed into it.

The arrow in Figure 6 shows how much of the derived datatype has been processed and copied into the intermediate buffer by the MPI implementation and represents the current *context* of the datatype. This context is internally stored by the MPI implementation, so that when the next portion of the datatype is packed (due to the pipelining of packing and communication), the datatype processing can quickly reload the context and continue from there instead of re-searching through the datatype for the appropriate context.

While the context-based datatype processing is efficient in the simple case where there is no additional processing between pipeline events, in real implementations this is rarely the case. For example, before each data packing event, MPICH2 examines the derived datatype to see whether it is sparse or dense in order to decide whether or not to pack the data into an intermediate buffer, i.e., it performs a look-ahead in the derived datatype. Thus, the context of the derived datatype is incremented by the look-ahead amount. In case the datatype turns out to be sparse, data packing needs to be performed from the previous context – this is a problem since our current context is already incremented by the look-ahead amount! Cur-

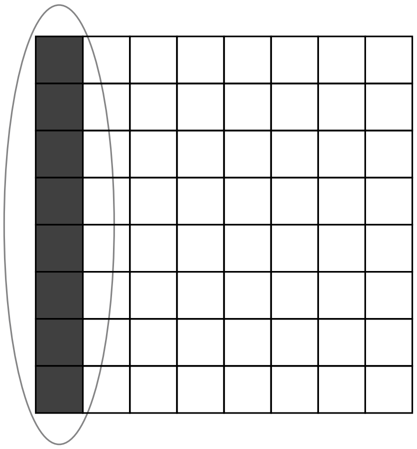

Figure 4: An 8x8 2-D matrix. Each element consists of three doubles.

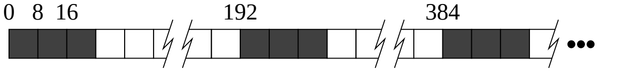

Figure 5: Noncontiguous memory layout of the first column of the matrix.

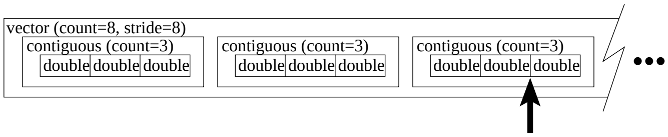

Figure 6: MPI datatype describing the first column of the matrix.

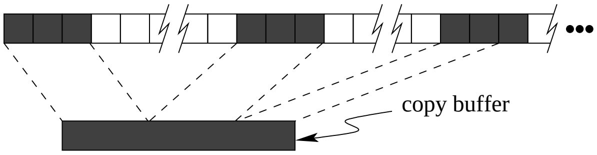

Figure 7: Packing part of the first column of the matrix.

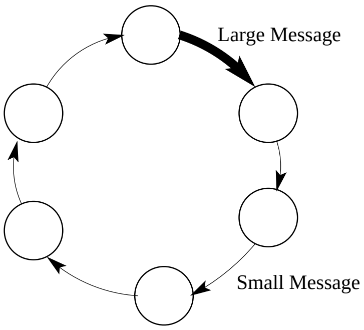

Figure 8: Allgatherv Ring Algorithm for Large Messages – One large message in the communication volume set can sequentialize the entire communication operation

rently, this problem is dealt by re-searching the entire derived datatype for the appropriate context. This search time, however, increases quadratically with the size of the derived data type. In Section 4.1 we present an optimization to the data processing code in MPICH2 to eliminate this overhead.

3.2 Issues with Nonuniform Communication Volumes

A common model in scientific applications is to repeatedly perform a computation phase followed by a communication phase in which all processes communicate at the same time. To support such models, MPI defines collective communication operations in which all processes send and receive portions of data. For instance in the `MPI_Allgather()` operation, each process specifies the data to be sent to all other processes, as well as the buffer into which it will receive the data from each of the other processes. In this operation the amount of data that each process sends is required to be the same, i.e., the volume of data communicated by each process should be uniform. MPI also provides a variant of this operation called `MPI_Allgatherv()` which allows each process to send a different amount of data, i.e., the volume of data communicated by each process could be nonuniform.

MPICH2 collective communication operations have been optimized for uniform communication patterns. However, when there is a large difference between the amount of data that is sent by each process, these operations perform suboptimally. Consider the case where the processes call `MPI_Allgatherv()` and one process has a large amount of data to send, while the others have small amounts of data to send. Because the total amount of data to be sent is large, MPICH2 uses the *ring* algorithm which is optimal for large messages (in the uniform communication volumes case). In the ring algorithm, the processes are arranged in a logical ring and each process receives data from its predecessor and sends data to its successor. If each process were to send the same amount of data, the communication would be well balanced, and each process' capacity would be fully utilized through continuous sending and receiving. On the other hand, in our example, because of the large imbalance of message sizes, most of the time only one process will be sending and one process will be receiving a large message; the other processes will be idly waiting for the next message. Thus, the communication time would be dominated by the large message being passed around the ring. Figure 8 illustrates this problem.

In some collective communication patterns used by scientific applications, each process may have different data to send to different processes. MPI provides the `MPI_Alltoall()` operation to support such a pattern for uniform communication volumes (i.e., each process sends the same volume of data to each of the other processes). `MPI_Alltoallw()` operation is the nonuniform communication volume counterpart for `MPI_Alltoall()`. As a special case, `MPI_Alltoallw()` also provides support for a communication pattern where each process sends and receives data from some processes, but not necessarily from every other process – this is achieved by specifying *zero* communication volume for the other processes in the collective operation. For instance, this operation can be used to perform a nearest-neighbor communication pattern, where processes are arranged logically in a grid and each process exchanges data only with the processes adjacent to it in the grid. Note that in this operation, every process is exchanging data with some processes, hence it is participating in the operations; the process just does not communicate with *every* other process in the system.

Because the MPICH2 collective communication operations are optimized for uniform communication volumes, the MPI_Alltoallw() operation assumes (from a performance stand point) that the same amount of data is sent and received between every two processes. So, cases where the communication volumes are nonuniform might not be handled in the most efficient way. Specifically, MPI_Alltoallw() implementation requires each process to send and receive a message to every other process in a round-robin manner. This approach has two major disadvantages. First, if a process has no data to exchange with another process, sending and receiving *zero* bytes of data (as is with the current implementation of MPICH2 and its derivatives) adds an additional synchronization step between the two processes. For example, in the case of nearest-neighbor communication, each process will only have data to exchange with a few other processes, regardless of the number of processes in the application. Second, let us consider a scenario where process 1 has to send a large message to process 2 and a small message to process 3. Now, in the round-robin if process 2 comes before process 3, the data to be sent to process 2 is processed before that of process 3. In cases where the processing overhead is large (e.g., if the data is noncontiguous), process 3 could be significantly delayed. On the other hand, if the smaller messages are sent out first, this can be avoided. In Section 4.2 we present optimizations to MPICH2 collective communication operations for nonuniform communication volumes.

4 Redesigning MPI Communication

In this section, we describe different approaches to handle the issues pointed out in Section 3. In Section 5, we evaluate our schemes independently as well as with PETSc-based applications to understand the benefits achievable.

4.1 Processing Noncontiguous Data

As described in Section 3.1, approaches used by current MPI implementations to handle noncontiguous data rely on continuous parsing of the derived datatype. If a look-ahead needs to be performed within the datatype to understand if the portions of the datatype are sparse or dense, such continuity is lost. This results in the algorithm having to search the derived datatype at each step to find the position till where it had previously parsed and continue from there. Thus, the searching time needed to find the previous position increases quadratically with the size of the derived datatype.

In this section, we present a new approach to handle such issues with derived datatype processing, namely a dual-context look-ahead approach (figure 9). The primary idea of this approach is to utilize two contexts to parse the datatype instead of one. These two contexts can be thought of as snapshots of the layout of the derived datatype.

The first context is primarily utilized for look-ahead in order to understand the structure of the upcoming portions of the derived datatype. This context rolls forward within the datatype allowing the MPI library to analyze the structure of the datatype and make a decision on the algorithm to be utilized (for sparse and dense derived datatypes). A second important functionality of this context is to maintain a list of data elements that have been parsed by the look-ahead mechanism, i.e., this context maintains pointers to the data and the lengths of each element it has parsed through.

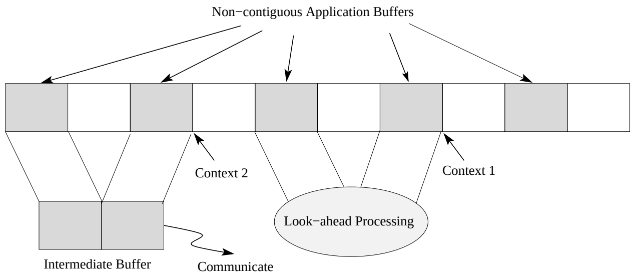

Figure 9: Dual-context Look-ahead Design

The second context is utilized for the actual processing of the datatype, including packing the data if needed and communicating the appropriate data in a pipelined fashion. This context, instead of directly parsing through the actual datatype, first parses through the list of data elements that were created by the first context. Once it has parsed through these elements, it continues parsing the actual datatype from the point where the look-ahead had completed.

There are two primary advantages of this approach. First, this approach is not intrusive into the datatype structures, since only the first context modifies them; the second context only works on the copy created by the first context. This allows the remaining components in the MPI library that rely on datatype processing to be not affected by these alterations. Second, this approach keeps track of the initial context of the datatype and hence completely eliminates the need to re-search the datatype for the appropriate context.

The only disadvantage of this approach is that it has to maintain a copy of the pointers and lengths of the data that the first context parses through. However, it is to be noted that the current implementation only parses through a small number of data elements (e.g., 15); thus this overhead is negligible in even moderately large datatypes.

4.2 Handling Nonuniform Communication Volumes

As described in Section 3.2, current collective operations are highly ill-equipped to deal with applications which use nonuniform communication volumes, i.e., not all processes send or receive the same amount of data. When the difference in the communication volumes is large, several times this results in sequentialization of the communication. In some cases, this can also result in increased skew making the application more sensitive to load imbalance and system noise.

In this section, we propose different designs for two such collective communication operations that deal with nonuniform communication volumes, namely MPI_Allgatherv (Section 4.2.1) and MPI_Alltoallw (Section 4.2.2).

4.2.1 Enhancing MPI_Allgatherv

When the total size of the data that needs to be communicated is large, MPI_Allgatherv currently uses a ring algorithm (which is optimal for uniform communication). At every step, each node forwards the data it received in the previous step to the next node. If one node is sending a large amount of data and the remaining nodes are sending small amounts of data, this would essentially sequentialize the transfer of the large message, thus taking O(N) time for communication, where N is the number of nodes participating in the operation.

On the other hand, if we can quickly identify the communication volumes of all the processes, we can design more sophisticated algorithms for nonuniform communication volumes. Thus, we break down this problem into two subproblems: (i) designing an efficient approach to quickly identify large nonuniformities in the communication volume set and (ii) designing a more efficient algorithm based on the knowledge of the nonuniform communication volumes.

Identifying Nonuniformities in the Communication Volume Set: We formulate this subproblem as an outlier detection problem, i.e., we need to identify if a small subset of the communication volume set falls significantly outside the range of the rest of the communication volume set. We do this by computing the *outlier ratio*, using equation 1, and comparing this to a threshold value.

$$\frac{\texttt{k\_select}(\texttt{COMM\_VOL\_SET}, N)}{\texttt{k\_select}(\texttt{COMM\_VOL\_SET}, N \times \texttt{OUTLIER\_FRACT})} \tag{1}$$

In this equation, COMM_VOL_SET is the set of communication volumes used by each of the processes in the system (note that this information is already available at each process in an MPI_Allgatherv operation), N is the number of processes participating in the communication operation and OUTLIER_FRACT is the fraction of processes that have to be outside the range of volumes encompassing the bulk of the messages in order to called *outliers*.

In this equation, k_select() allows us to determine the k^{th} smallest element of a set of elements. We utilize the algorithm by Floyd and Rivest to evaluate k_select() in linear time. Thus, the entire mathematical formulation described above can be obtained in linear time. It is to be noted that the existing approach parses through the entire communication volume set to identify the total communication volume, which already is linear time. With our current approach, we are increasing the coefficient of the linear time taken, but not its computational complexity.

Designing Efficient Algorithms based on the Communication Volume Characteristics: Based on the knowledge of the communication volumes that is obtained as described above, in cases where some of the communication volumes are significantly larger than the rest, we use two algorithms.

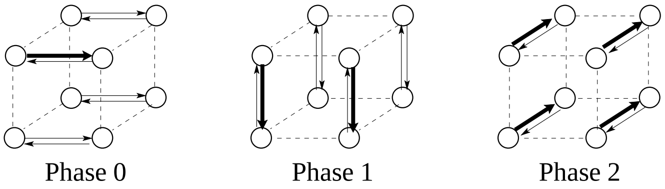
Phase 0 Phase 1 Phase 2

Figure 10: Recursive doubling algorithm for eight processes

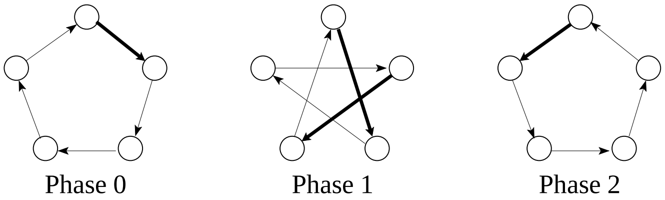
Phase 0 Phase 1 Phase 2

Figure 11: Dissemination algorithm for five processes

The first algorithm is a recursive doubling approach which is used when power-of-two number of processes are participating in the communication. The second algorithm is a variant of the dissemination algorithm [8], and is used for non-power-of-two number of processes. The recursive doubling algorithm proceeds in $\log N$ phases, where in each phase each process exchanges data with another process. With each phase the amount of data exchanged increases, as each pair of processes exchange not only their own data, but the data which they have received in the previous phases. Figure 10 shows the basic communication pattern for the recursive doubling algorithm for eight processes.

The dissemination algorithm proceeds in $\lceil \log N \rceil$ phases. In phase p, each process with rank i sends its data to the process with rank $i + 2^p \mod N$, and receives data from the process with rank $1 - 2^p \mod N$, where N is the number of participating processes. As with the recursive doubling algorithm, the amount of data sent and received in each phase increases from the previous phase. Figure 11 shows the basic communication pattern for the dissemination algorithm.

The main benefit with both these algorithms is that the movement of the large outlier messages to all the processes is carried out simultaneously by multiple process. For example, if we just consider the movement of one large message, the data follows a binomial tree pattern, instead of the sequential pattern in the previous ring algorithm.

4.2.2 Enhancing MPI_Alltoallw

Current algorithms for implementing MPI_Alltoallw, again, do not carefully consider the overheads associated with nonuniform communication volumes. Specifically, if some processes either communicate significantly larger volumes of data that requires preprocessing (e.g., noncontiguous data that requires to be packed) as compared to the others or do not communicate at all, current algorithms perform suboptimally.

In our approach, for each process, we divide the data to be communicated to every other process into bins based on the volume of data to be communicated. If the data that is being communicated requires preprocessing, we process the bins containing small messages first, and then move to the bins containing larger messages. In this approach, since the processing of the data is sequentialized by the host processor, remote processes that communicate only small amounts of data do not have to wait for the larger data messages to be processed. In the existing approach, however, no such prioritization is performed potentially causing the wait times for processes that only communicate small amounts to be higher than required.

Processes to which there is no communication are placed in a separate bin, which is completely exempted. This reduces unnecessary skew in the collective operation. In our implementation, we used three bins – *zero* size messages, small messages (lesser than a threshold) and large messages.

5 Experimental Evaluation

In this section, we evaluate the approaches described in Section 4 and compare them with the existing approaches. We perform microbenchmark-level evaluations of the proposed schemes in Sections 5.2 and 5.3. Evaluation with PETSc benchmarks is performed in Section 5.4 and with the 3-D Laplacian multi-grid solver application in Section 5.5.

5.1 Experimental Testbed

The testbed used in this paper consisted of two clusters:

Cluster 1: 32-node cluster of dual Intel EM64T 3.6GHz processors, 2MB L2 cache, 2GB DDR2 400MHz SDRAM and Intel E7520 (Lindenhurst) chipset. We used RedHat AS4 with the kernel.org kernel 2.6.16.

Cluster 2: 32-node cluster of dual AMD Opteron 2.8GHz processors, 1MB L2 cache, 4GB DDR 400MHz SDRAM and NVidia 2200/2050 chipset. We used RedHat AS4 with the kernel.org kernel 2.6.16.

Network: All 64 nodes in the machines were connected together with Mellanox MT25208 InfiniBand DDR adapters through a 144-port IB switch.

Software: We used the the OpenFabrics Gen2 stack as the underlying IB driver and verbs interface. Above this interface, we used the MVAPICH2 implementation of MPI (which is a derivative of the MPICH2 stack from Argonne National Laboratory) over InfiniBand. Specifically, we used MVAPICH2-0.9.5 as the base case and compared it against our optimizations to it (labeled as MVAPICH2-New).

5.2 Evaluating Noncontiguous Data Processing Overheads

In this section, we evaluate our optimizations to the noncontiguous datatype processing using a benchmark which sends a matrix from one process to another while transposing it. In the benchmark the sender sends the data in column-major order while the sender receives the data in row-major order, effectively transposing the matrix. Because of the noncontiguous layout of the data being sent, the performance of this benchmark is highly dependent on the performance of the MPI implementation's noncontiguous datatype processing.

In Figure 12, we see that as the size of the matrix being transposed increases, the time to perform the transpose increase much faster for the original implementation (labeled MVAPICH2-0.9.5), than for the optimized implementation (labeled MVAPICH2-New). For the 1024x1024 sized matrix, the optimizations give over 85% improvement over the original implementation. This improvement is expected to further increase for larger matrices.

Figure 13 shows the breakdown of the time spent (normalized to 100%) in the transpose benchmark for the current approach and the optimized approach. The figure shows the time spent performing communication, packing the data, and searching for the current location in the datatype. As expected, because the original implementation loses its context within the derived datatype each time it sends a portion of the noncontiguous data, the search time increases dramatically with the size of the matrix. In the optimized implementation we have eliminated the search time altogether by using the dual-context approach, and so the communication dominates the time to perform the benchmark.

5.3 Evaluating Nonuniform Volume Collective Communication

We next evaluate the performance of our optimizations for nonuniform communication patterns using MPI benchmarks.

In the first benchmark, we measure the average latency of `MPI_Allgatherv` when Process 0 sends a large amount of data while the other processes send only one double (Figure 14). The graph on the left shows the latency of `MPI_Allgatherv` for 64 processes as we vary the amount of data that Process 0 sends while the graph on the right shows the latency when Process 0 sends 32 KB of data as we vary the number of processes. For both cases, the latency of the original implementation increases faster than our optimized implementation. We see up to a 20% improvement for 64 processes.

Our next benchmark evaluates the performance of `MPI_Alltoallw` for nonuniform communication. In this benchmark, processes are arranged in a logical ring and each process has a 10x10 matrix of doubles to exchange with its successor and predecessor, but nothing to exchange with other processes.

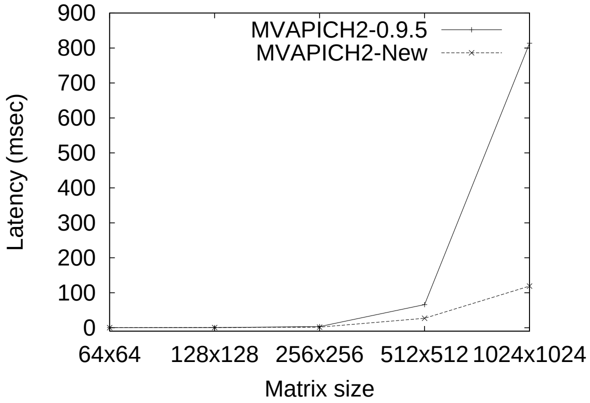

Figure 12: Performance of matrix transpose benchmark

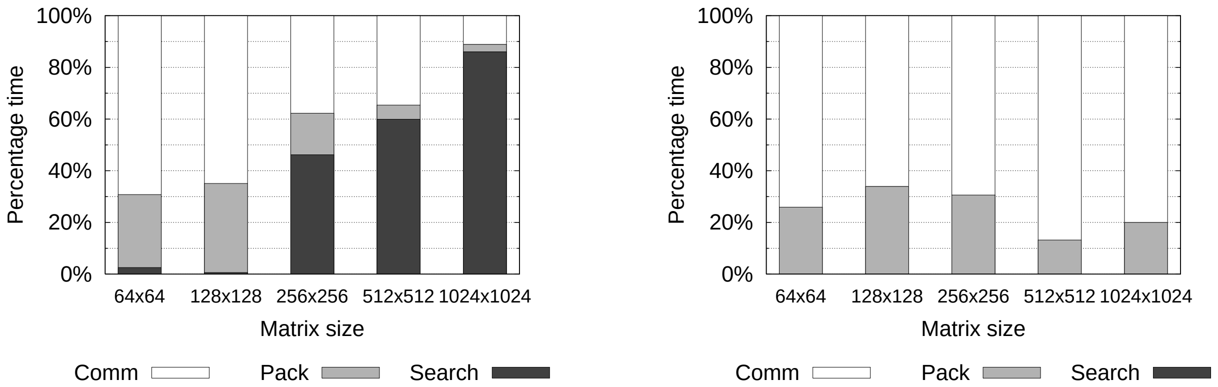

Figure 13: Datatype processing breakup: (a) Current approach and (b) Proposed dual-context look-ahead approach

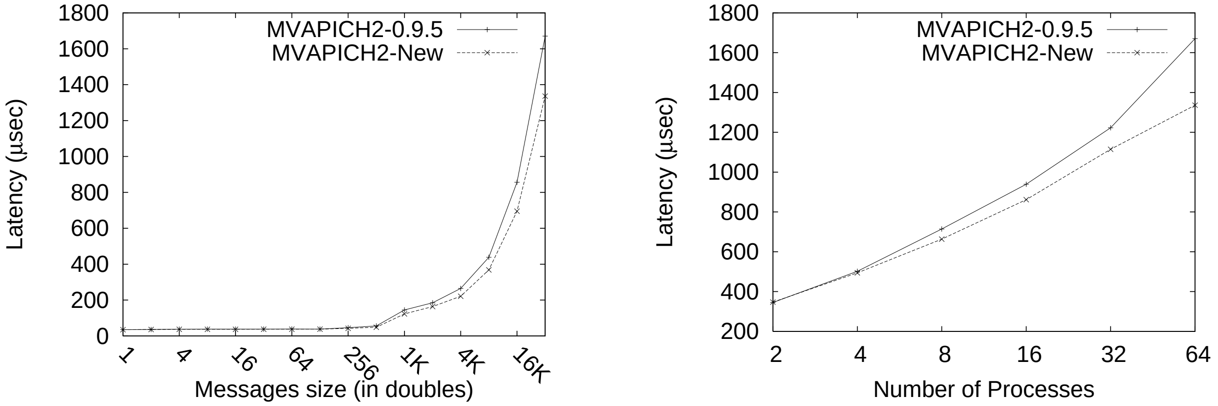

Figure 14: MPI_Allgatherv Performance: (a) With varying problem size and (b) With varying system size

Figure 15 shows the results of this benchmark as the number of processes is varied. We see that the original implementation is much more easily impacted by any minor imbalance between processes as compared to our optimized implementation. Note that we did not add any artificial skew to the benchmark, but given that we used a combination of two different clusters in our evaluation (32 Intel nodes and 32 Opteron nodes), some skew is bound to be present between the processes. For the 128 processor case we see over 88% improvement. The evaluation till 32 processes was done completely on the Opteron cluster – the benefit in this case is about 50%.

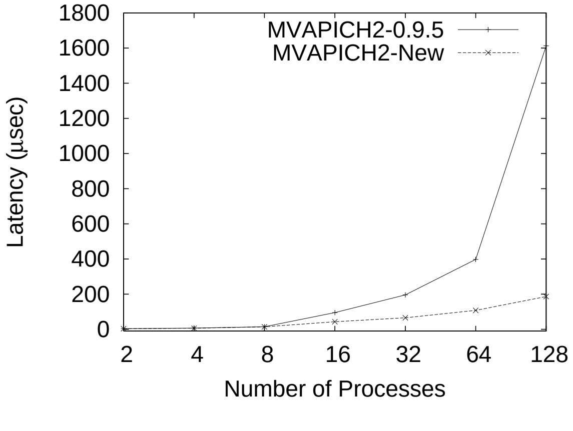

Figure 15: MPI_Alltoallw performance

5.4 PETSc Vector Scatter Benchmark

We also evaluated the benefit of our optimization at the PETSc library-level using a vector scatter benchmark to stress the communication portion of PETSc.

In this benchmark, we create two 1-D grids of elements with one degree of freedom (i.e., each element is comprised of one double precision physical element). These 1-D grids are initially passed over to PETSc to lay them out in a parallel manner across multiple processes. Once laid out, each process scatters the elements in its portion of the first vector to unique portions of the second vector. This benchmark emulates the communication portion of applications that require to operate on multiple 1-D grid structures where the elements of each of the grid structures is distributed across the system.

Because of the poor performance of the original implementation, the PETSc library by default does not use the derived datatypes or collective operations. Rather, it uses a hand-tuned algorithm which explicitly performs the packing of data and individual sends and receives to scatter the vector. Together with this hand-tuned option, PETSc also has a mechanism to use derived datatypes and collective operations for MPI implementations which optimize these routines.

We compared the performance of PETSc for the three implementations: (i) the default hand-tuned implementation (labeled as hand-tuned), (ii) using the implementation that uses derived datatypes and collective operations for the original MPI (MVAPICH2-0.9.5) and (iii) using derived datatypes and collective operations with our optimized MPI (MVAPICH2-New). Figure 16 shows the results of this benchmark as we varied the number of processes. The 1-D grid size is scaled according to the number of processes, so the number of elements managed by each process is constant across the graph. The graph on the left shows the latency of the three implementations, while the graph on the right shows the percentage improvement of the optimized implementation to the original implementation and the hand-tuned implementation. We see that our optimized implementation shows a large improvement over the original implementation as the number of processors increases: over 95% for 128 processors.

The hand-tuned implementation performs slightly better than our optimized implementation (almost 4% at 128 processes). This indicates that there are more optimizations that we can apply to MPI. Also, note that though the hand-tuned implementation does perform better, it requires implementing the packing and communication pattern explicitly, which in general complicates the library communication implementation. By using MPI capabilities (derived datatypes and collective operations), these implementations can be simplified, which may be a desirable trade-off considering the small performance gain from using a hand-tuned algorithm.

5.5 3-D Laplacian Multi-grid Solver

In this section, we evaluate the performance of a 3-D Laplacian multi-grid solver application that is modeled by the following partial differential equation:

$$\bigtriangledown = i.\frac{\partial u}{\partial x} + j.\frac{\partial u}{\partial y} + k.\frac{\partial u}{\partial z} \qquad (2)$$

$$BoundaryConditions : 0 \leq x, y, z \leq 1$$

A grid of size 100x100x100 is used with one degree of freedom. The data grid varies the values of the variants (x, y, z) uniformly across the grid in each dimension. The application itself utilizes PETSc to solve this equation using a multi-grid solver. In our experiments, we evaluated the application by using three levels in the multi-grid to solve the equation.

As shown in Figure 17, the execution time of the application using our optimized implementation continues to decrease even up to 128 processes. However, with the original implementation, the execution time increases after 32 processes. This indicates that the application using the optimized implementation scales better than when using the original implementation. Our optimized implementation gives almost 90% improvement in the execution time for 128 processes. When using the hand-tuned implementation, the application performs better than our implementation by just over 10% for 4 processors. As the number of processors increases, however, this gap decreases, and is less than 3% for 128 processes.

6 Related Work

There has been a lot of previous work on optimizing the performance of noncontiguous data transfers, including packing and unpacking techniques [4, 18, 7] and approaches to utilize intelligent network adapters [23, 24, 19]. However, none of these approaches consider the search time to be a significant bottleneck. But, as we described in this paper, the search time within a datatype increases quadratically with the datatype size and becomes a significant bottleneck for large datatypes. In [18], the authors proposed an efficient mechanism to reduce this search time in simple cases; however, with advanced schemes, especially those involving look-ahead, this scheme loses its benefit. There has also been a significant amount of work in optimizing various collective operations including broadcast [9, 13, 12, 14], reduce [16, 15], allgather [11, 14, 21], alltoall [22], etc. Most of this work has been for uniform volume communication. There has also been some limited work to optimize the performance of collective operations that perform nonuniform volume communication [5]. However, the actual approaches proposed in this literature are to optimize performance based on network topology rather than the variation in the communication volumes between processes. In summary, this paper addresses an important problem and presents various novel designs to tackle them in an efficient manner.

7 Concluding Remarks

In this paper, we analyzed two kinds of overheads (noncontiguous data layouts and nonuniform volumes of communication to different processes) that are typically created as a byproduct of multi-layered structures containing at least the applications, high-level software libraries and the communication libraries. We used the PETSc software library and the MPI communication library as examples of a case study and noticed that current approaches in MPI implementations are tuned to support contiguous and uniform data communication and are ill-equipped to handle noncontiguous and nonuniform communication, especially when used together. We designed sophisticated approaches specifically optimized for nonuniform communication of noncontiguous data and demonstrated significant benefits in some cases. Specifically, our experimental results demonstrate close to an order of magnitude improvement in the performance of the 3-D Laplacian multi-grid solver application on a 128 processor cluster.

In future, we would like to analyze overheads in other high-level libraries and software such as FLASH.

References

[1] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries.

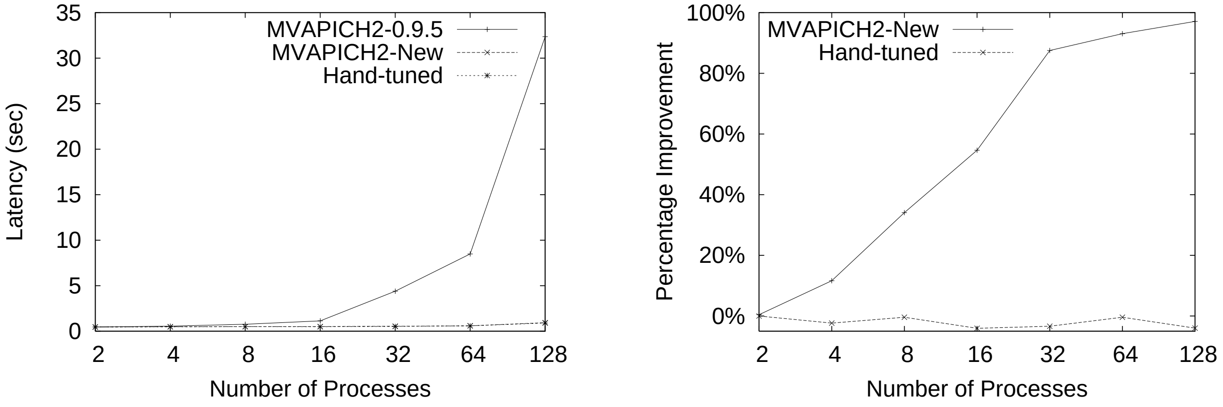

Figure 16: PETSc Vector Scatter Benchmark: (a) Absolute Performance and (b) Percentage Improvement

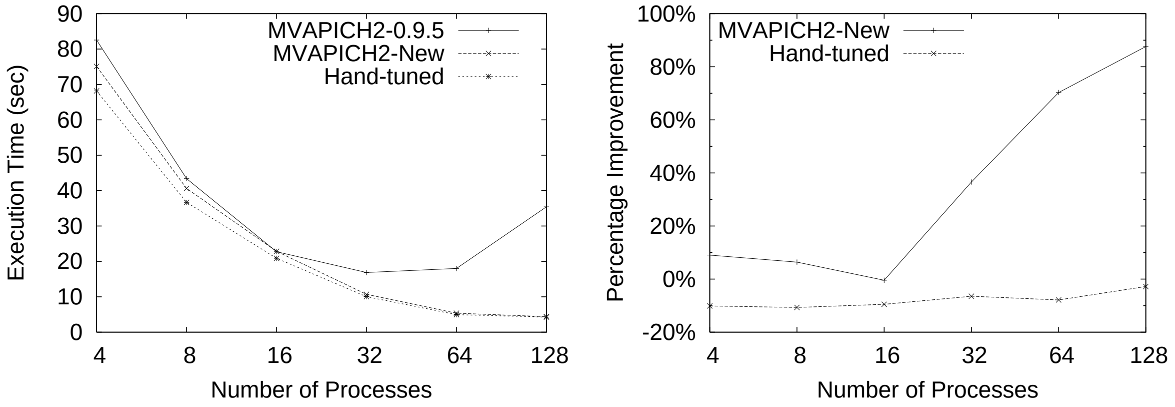

Figure 17: 3-D Laplacian Multi-grid Solver Application Evaluation: (a) Absolute Performance and (b) Percentage Improvement

In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997.

[4] S. Byna, W. Gropp, X.-H. Sun, and R. Thakur. Improving the Performance of MPI Derived Datatypes by Optimizing Memory Access Costs. In *SC*, 2003.

[5] A. Faraj and X. Yuan. Automatic Generation and Tuning of MPI Collective Communication Routines. In *ICS*, 2005.

[6] MPI Forum. MPI: A Message Passing Interface. In *SC*, 1993.

[7] W. Gropp, E. Lusk, and D. Swider. Improving the Performance of MPI Derived Datatypes. In *MPIDC*, 1999.

[8] Han and Finkel. An optimal scheme for disseminating information. In *ICPP*, 1988.

[9] C. T. Ho and S. L. Johnsson. Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes. In *ICPP*, 1986.

[10] W. Huang, G. Santhanaraman, H. W. Jin, Q. Gao, and D. K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *CCGRID*, 2006.

[11] M. Jacunski, P. Sadayappan, and D. K. Panda. All-to-All Broadcast on Switch-Based Clusters of Workstations. In *IPDPS*, 1999.

[12] S. L. Johnsson and C.-T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE TC*, 1989.

[13] Manoj Kumar. Supporting Broadcast Connections in Benes Networks. Technical Report RC 14063, IBM Research, May 1988.

[14] S. Lee and K. G. Shin. Interleaved All-to-all Reliable Broadcast On Meshes and Hypercubes. In *ICPP*, 1990.

[15] A. Mamidala, J. Liu, and D. K. Panda. Efficient Barrier and Allreduce on IBA clusters using Hardware Multicast and Adaptive Algorithms. In *Cluster*, 2004.

[16] A. Moody, F. Petrini, and D. K. Panda. Efficient Reduce Algorithms on the Quadrics Network. In *IPDPS*, 2002.

[17] MPICH2. http://www.mcs.anl.gov/mpi/

[18] R. Ross, N. Miller, and W. Gropp. Implementing Fast and Reusable Datatype Processing. In *EuroPVM/MPI*, 2003.

[19] G. Santhanaraman, J. Wu, and D. K. Panda. Zero-Copy MPI Derived Datatype Communication over InfiniBand. In *EuroPVM/MPI*, Sep 2004.

[20] V. S. Sunderam. PVM: A Framework for Parallel and Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[21] S. Sur, U. Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda. High Performance RDMA Based All-to-all Broadcast for InfiniBand Clusters. In *HiPC*, 2005.

[22] S. Sur, H. W. Jin, and D. K. Panda. Efficient and Scalable All-to-All Exchange for InfiniBand-based Clusters. In *ICPP*, 2004.

[23] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda. Host-assisted Zero-copy Remote Memory Access Communication on InfiniBand. In *IPDPS*, 2004.

[24] J. Wu, P. Wyckoff, and D. K. Panda. High Performance Implementation of MPI Datatype Communication over InfiniBand. In *IPDPS*, 2004.