High Performance Implementation of MPI Derived Datatype Communication over InfiniBand

Jiesheng Wu, Pete Wyckoff[†], and Dhabaleswar K. Panda

Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

[†]Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Technical Report
OSU-CISRC-10/03-TR58

High Performance Implementation of MPI Derived Datatype Communication over InfiniBand *

Jiesheng Wu† Pete Wyckoff‡ Dhabaleswar Panda†

†Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

‡Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

MPI derived datatype is a powerful method to define arbitrary collections of noncontiguous data in memory and to enable noncontiguous data communication in a single MPI function call. It can be expected that MPI derived datatype could become a key aid in application development. In practice, however, the poor performance of many MPI implementations with derived datatypes forces users to resort to packing and unpacking data in contiguous buffers manually. This usage actually defeats the purpose of having derived datatype in the MPI standard.

In this paper, we systematically study two main types of approach for MPI datatype communication: Pack/Unpack-based approaches *and* Copy-Reduced approaches *on the InfiniBand network. We focus on overlapping packing, network communication, and unpacking in the Pack/Unpack-based approaches. We use RDMA operations to avoid packing and/or unpacking in the Copy-Reduced approaches. We design four schemes to improve performance of datatype communication.*

We implement and evaluate three schemes based on one of MPI implementations over InfiniBand. Performance results of a vector micro-benchmark demonstrate that latency is improved by a factor of up to 3.4 and bandwidth by a factor of up to 3.6 compared to the current datatype communication implementation, which is derived from MPICH. Collective operations like MPI_Alltoall are demonstrated to benefit. A factor of up to 2.0 improvement has been seen in our measurements of those collective operations on a 8-node system. We also notice that these schemes perform differently in different cases. It is feasible that an implementation can incorporate multiple schemes and choose an appropriate scheme given a datatype operation to achieve the best performance. Techniques discussed in this paper can be applied to other domains such as file and storage systems to support efficient noncontiguous I/O access.

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052, #CCR-0204429, and #CCR-0311542.

1 Introduction

The MPI (Message Passing Interface) Standard [21] has evolved as a de facto parallel programming model for distributed memory systems. MPI has a number of features that provide both convenience and high performance. One of the important features is MPI *derived datatype*. Derived datatype provides a powerful and general way to describe arbitrary collections of noncontiguous data in memory in a compact fashion. The MPI standard provides run time support to create MPI derived datatypes and use them in other functions, such as regular message passing functions, performing remote memory access (RMA), and I/O operations.

Typically, MPI derived datatypes allow users to have concise representations of many commonly used data layouts [11, 26]. An example is given in Section 3.2. It can be expected that MPI derived datatype could become a key aid in application development. In practice, however, the poor performance of many MPI implementations with derived datatypes [5, 11, 27, 30] becomes a barrier to using derived datatypes. A programmer often prefers packing and unpacking noncontiguous data manually even with considerable effort. Therefore, it would not be surprising that there is no datatype communication in either the NAS benchmarks [4] or the ASCI benchmarks [20]. On the other hand, noncontiguous communication occurs commonly in many applications, such as (de)composition of multi-dimensional data volumes [3, 8], fast Fourier transform, and finite-element codes [5]. Thus, it is very important to provide efficient MPI datatype communication in MPI implementations.

MPI datatype communication involves datatype processing, and noncontiguous data communication (in this paper, unless stated otherwise, we refer datatype to noncontiguous datatype). In many networks which only support transfer of contiguous data blocks, packing data into and unpacking data out of a contiguous buffer are usually used for noncontiguous data communication. There are several potential ways to improve MPI datatype communication accordingly: *Improve datatype processing system* [11, 15, 26]; *Optimize packing and unpacking procedures* [5, 11]; *Take advantage of network features to improve noncontiguous data communication* [30, 33]. In this paper, we focus on the last two areas based on the InfiniBand network.

We systematically study two main types of approach for MPI datatype communication: *Pack/Unpack-based approaches* and *Copy-Reduced approaches* on the InfiniBand network. In the first type of approach, to reduce the impact of pack and unpack costs on the performance of datatype communication, we propose a new technique called *Buffer-Centric Segment Pack/Unpack (BC-SPUP)* to pipeline the three steps in a datatype communication: packing, network communication, and unpacking. This technique offers potential overlaps between packing, network communication, and unpacking. Particularly, InfiniBand provides comparable bandwidth to system memory copy bandwidth, which makes more sense to overlap these three steps. Therefore, the pack/unpack costs visible to applications are reduced effectively. In addition, this technique also reduces dynamic memory allocation and deallocation, memory registration and deregistration by using pre-registered pack/unpack buffers as much as possible.

In Copy-Reduced approaches, the main idea is to use InfiniBand remote direct memory access (RDMA) operations to transfer noncontiguous data in a datatype message directly without packing and/or unpacking. We design three schemes: *RDMA Write Gather with Unpack (RWG-UP)*, which avoids packing on the sender side; *Pack with RDMA Read Scatter (P-RRS)* , which avoids unpacking on the receiver side; and *Multiple RDMA Writes (Multi-W)*, which avoids both packing and unpacking and achieves zero-copy datatype communication.

We design the aforementioned four schemes. We identify their design issues and provide solutions to these issues. We also implement and evaluate three of them: BC-SPUP, RWG-UP, and Multi-W based on MVAPICH [24, 19], an MPI implementation over InfiniBand. In this paper, we make the following contributions:

1. Memory copies in datatype communication have significant impact on the InfiniBand network which offers high bandwidth comparable to memory bandwidth. The proposed Buffer-Centric Segment Pack/Unpack scheme effectively overlaps packing, network communication, and unpacking; and reduces pack/unpack costs visible to applications.
2. RDMA Gather/Scatter functionality can be used to transfer datatype messages efficiently by reducing memory copies. It allows multiple blocks to be transferred in one single operation. This not only reduces the total startup costs, but also increases network utilization. Our proposed RDMA Write Gather with Unpack scheme takes advantage of both RDMA Gather Write and segment unpacking. The unpack cost is masked effectively for large datatype messages.
3. Using multiple RDMA writes to transfer a datatype message is very efficient due to zero-copy with condition that each block size is large enough. Otherwise, the total startup costs and the low network utilization of small messages offset the benefit of zero-copy.
4. Memory registration and deregistration on networks with RDMA capabilities add a new dimension to datatype communication. Our scheme to register and deregister datatype message buffers permits efficient use of RDMA operations for datatype communication.
5. We implement and evaluate BC-SPUP, RWG-UP and Multi-W schemes based on MVAPICH. Significant improvement has been achieved in both point-to-point and collective datatype communication. These schemes perform differently in different cases. A combination of these schemes can be deployed in an MPI implementation. An appropriate scheme can be chosen dynamically with respect to the datatype characteristics.

The rest of the paper is organized as follows. We first give a brief overview on InfiniBand in Section 2. Section 3 presents an overview of MVAPICH, its datatype communication, and a motivating example that illustrates problems and potential improvements in the current implementation. Section 4 describes two pack/unpack approaches for datatype communication. Section 5 describes three copy-reduced approaches. We describe the feasibility to combine all schemes in one implementation and to choose one approach dynamically in Section 6. In Section 7, we highlight some implementation details. The performance results are presented in Section 8. We examine some related work in Section 9 and draw our conclusions and discuss possible future work in Section 10.

2 Overview of InfiniBand

The InfiniBand Architecture [13] defines a System Area Network for interconnecting both processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are

connected to the fabric by Host Channel Adapters and Target Channel Adapters, respectively.

An abstraction interface for HCAs is specified in the form of InfiniBand *Verbs*. The interface of TCAs is vendor implementation specific and not defined in the InfiniBand architecture. In the Verbs abstraction, a queue-based communication model is used. A Queue Pair consists of two queues: a send queue and a receive queue. The send queue holds instructions (*descriptors*) to transmit data and the receive queue holds instructions, that describe where received data is to be placed. The completion of send and receive requests is reported through Completion Queues (CQs). Once a request is finished, a completion queue entry can be generated in the associated CQ. Flexible completion notification mechanisms are provided.

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the receiver first posts a receive descriptor into a receive queue. Then, the sender posts a send descriptor into a send queue to initiate data transfer. In channel semantics, there is a one-to-one match between send and receive descriptors. Multiple send and receive descriptors can be posted and consumed in FIFO order.

In memory semantics, Remote Direct Memory Access (RDMA) write and read operations are used. RDMA operations are one-sided. The initiator can write data into and read data from a buffer on the remote node transparently. *Write Gather* and *Read Scatter* are supported in RDMA operations. RDMA write operation can gather multiple data segments together and write all data into a contiguous buffer on the remote side in one single operation. While RDMA read operation can read data from a contiguous buffer on the remote side into several local buffers. These gather and scatter features are very useful to transfer noncontiguous data. RDMA Write with Immediate data is also supported. With Immediate data, a RDMA Write operation consumes a receive descriptor and then can generate a completion entry to notify the remote node of the completion of the RDMA Write operation.

3 Datatype Communication in MVAPICH

In this section, we first describe MVAPICH, an MPI implementation over InfiniBand [24, 19], including its basic communication protocols and datatype communication. Then we present a motivating example to demonstrate performance problems of datatype communication in MVAPICH and discuss possible ways to improve datatype communication performance. Note that the MVAPICH datatype communication path is mostly same as that in the standard MPICH implementation [10], these problems are actually general issues in all MPICH implementations over RDMA-capable networks, such as MVICH [17] and MPICH-GM [23].

3.1 Overview of MVAPICH

MVAPICH is a high performance MPI implementation on InfiniBand. Its design is based on MPICH [10] and MVICH [17]. The InfiniBand communication interface is implemented as a communication device in the MPICH ADI layer [29]. There are two basic protocols in MVAPICH: *Eager* and *Rendezvous*. In Eager protocol, a message is transferred eagerly to a receiver's internal buffer regardless of whether a receive operation had been issued or not. In this protocol, data is first copied into an internal buffer on the sender side. Then it is transferred to an internal buffer on the receiver side. Later, data is copied from the receiver internal buffer into the application buffer. The Eager protocol is used to transfer small and control messages.

In the Rendezvous protocol, the sender and the receiver first perform handshake to synchronize with each other. This synchronization ensures that a matched receive operation has been issued before data transfer. User buffers on both sides are registered and related information is exchanged in the handshake procedure. A zero-copy implementation of the Rendezvous protocol is implemented using RDMA Write operations. The Rendezvous protocol is used to transfer large messages.

MVAPICH is publicly available and has been widely used in many labs, universities, and companies. In the current MVAPICH, we have not exploited the design space for MPI derived datatype communication over InfiniBand. The MVAPICH datatype communication path is derived from MPICH and MVICH without change. Figure 1 shows the communication paths for both small and large datatype messages in the current MVAPICH. Small datatype messages are packed into a temporary buffer (*pack buffer*) first, then copied into an internal buffer of the Eager protocol. On the receiver side, the message is first copied into a temporary buffer (*unpack buffer*) from an internal buffer, then unpacked into the user buffer. To transfer a large datatype message, both sides allocate pack and unpack buffers dynamically. The sender packs data into a pack buffer and then RDMA writes data into the receiver's unpack buffer. The receiver unpacks data into the user buffer. Zero-copy messaging happens only between the pack and unpack buffers. In both protocols, pack and unpack buffers are allocated and freed dynamically. This pack/unpack scheme is a generic datatype communication mechanism deployed in many MPI implementations.

3.2 A Motivating Example

Many MPI implementations perform poorly with derived datatypes [5, 11, 27, 30, 25]. We use a vector datatype ping-pong latency test to demonstrate performance problems in

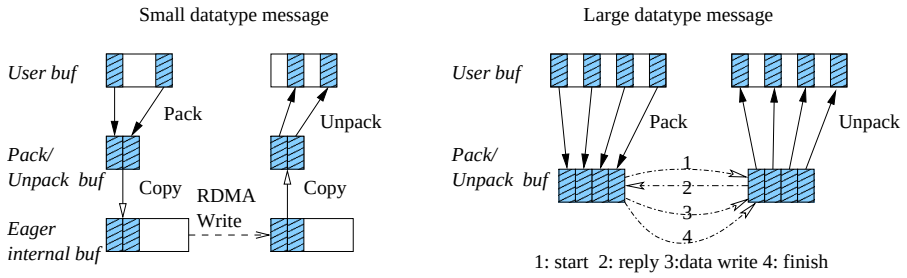

Figure 1. Datatype Communication in MVAPICH

the generic MPICH datatype communication implementation and particular issues arisen in MVAPICH over InfiniBand.

Suppose we want to send one or more columns in a two-dimensional 128×4096 integer array from one process to another process. There are several potential schemes. The first scheme builds a derived datatype using `MPI_Type_vector(128, x, 4096, MPI_INT, &newtype)`, where x is the number of columns, and then to use this *newtype* in `MPI_Send()` and `MPI_Recv()`. The second scheme performs manual pack and unpack and only sends contiguous messages. The third scheme transfers each contiguous block one by one using individual MPI calls. We refer these three schemes as *Datatype*, *Manual*, and *Multiple* schemes, respectively.

One particular issue with the Datatype scheme on InfiniBand network is that dynamic pack and unpack buffers may incur on-the-fly memory registration and deregistration in each datatype operation. It is possible that different pack and unpack buffers are used in different datatype operations. We call this case as *Datatype plus registration and deregistration* (*DT + reg* for short in Figure 2).

Figure 2 shows a log-log plot of the ping-pong latencies of the aforementioned cases with variable numbers of columns in MVAPICH. As a reference, the latency results for transferring the same size of contiguous data, termed as *Contig*, are also shown.

Several observations can be made. First, no more than *one quarter* of contiguous communication performance is achieved in any scheme. All schemes except *Multiple* have two memory copies on top of contiguous communication. Second, *Manual* performs a little better than *Datatype*. This is because of datatype processing overhead. Third, *Datatype plus registration and deregistration (DT+reg)* is much slower than *Datatype*. Fourth, *Multiple* performs a little better when the block size is large enough. Though there are no copies, the total cost of all operations, one per row, and the low network utilization due to small message sizes, degrade the zero-copy benefit when the block size is small. Each individual MPI call needs to pay its protocol overhead separately. In addition, it requires much effort from programmers to handle all communication details.

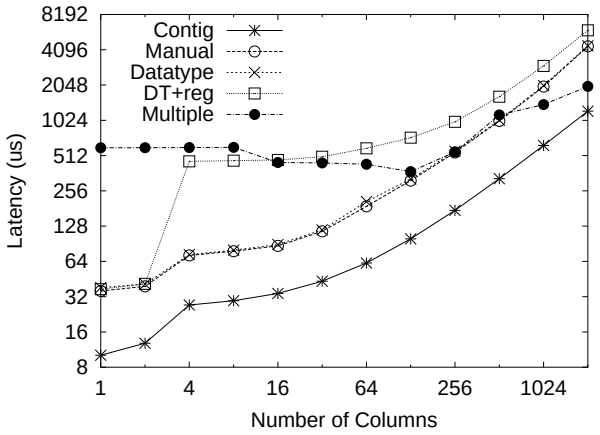

Figure 2. Vector Datatype Transfer Latency Comparison between Different Ways.

Therefore, the poor performance of datatype communication comes from: (1) memory copy; (2) memory registration and deregistration; (3) datatype processing; (4) total startup costs of operations; and (5) low network utilization due to small message sizes. This example motivates us to redesign the datatype communication path in MVAPICH.

We consider the following two types of approach to improve MPI datatype communication over InfiniBand: *Pack/Unpack-Based approaches* and *Copy-Reduced approaches*. The first type of approach is based on the pack/unpack mechanism. We propose a Buffer-Centric Segment Pack/Unpack scheme, which is presented in Section 4. Copy-Reduced approaches center around reducing memory copies by eliminating packing, unpacking, or both. We design three schemes: *RDMA Write Gather with Unpack*, *Pack with RDMA Read Scatter*, and *Multiple RDMA Writes*. Details are discussed in Section 5.

4 Pack/Unpack-Based Approaches

In this section, we first describe the basic Pack/Unpack scheme. Then we propose a Buffer-Centric Segment Pack/Unpack scheme (*BC-SPUP*) and discuss its potential benefits and design issues. This scheme provides overlap between packing, network communication, and unpacking and avoids on-the-fly memory registration and deregistration as much as possible. In Section 5, we discuss three Copy-Reduced approaches, which eliminate packing, unpacking, or both using RDMA operations.

4.1 Basic Pack/Unpack Scheme

MPI datatype communication based on the pack/unpack mechanism can be divided into the following five steps: *datatype processing, packing, communication, datatype processing, and unpacking*. The basic pack/unpack scheme follows these five steps in a straightforward manner. Datatype processing and packing are usually integrated together, so do datatype processing and unpacking. This scheme has been deployed in many MPI implementations.

This scheme is easy to implement. Messages seen by the underlying communication interfaces are always in contiguous regions of memory. It also has the minimum requirement on the datatype processing routines. No partial processing is needed [11, 26] because packing and unpacking are performed on a whole datatype message. However, the performance of this scheme is poor. First, it incurs intermediate copies. Usually two additional copies are required. Second, dynamic pack and unpack buffers incur many memory allocation and deallocation operations. Third, for networks which require that buffers be registered before communication, dynamic pack and unpack buffers may incur on-the-fly memory registration and deregistration. Fourth, packing, communication, and unpacking are serialized.

To overcome these problems, we propose a scheme, called *Buffer-Centric Segment Pack/Unpack (BC-SPUP)*, in the following subsection.

4.2 Buffer-centric Segment Pack/Unpack

The BC-SPUP scheme is designed to overcome the performance problems in the basic pack/unpack scheme using two techniques. First, it uses pre-allocated buffers for pack and unpack operations. These buffers are also optimized and ready for communication, such as aligned on page boundary and registered for RDMA operations. Second, this scheme breaks a datatype message into several segments and applies the basic pack/unpack processing to each segment.

The BC-SPUP scheme has the following potential advantages. First, it avoids dynamic memory allocation and deallocation, reducing system overheads [7]. Second, it eliminates memory registration and deregistration on pack/unpack buffers. Third, these buffers can be well aligned for better communication performance. Fourth, this scheme has the potential to overlap packing, network communication, and unpacking. Figure 3 shows this overlap.

Note that the buffer-centric segment pack/unpack scheme still needs two copies, one for each side. However, most of the copy costs are not visible due to overlap between packing, communication, and unpacking. This scheme can be also applied to other communication interfaces, such as the regular shared-memory interface and networks providing shared memory semantics such as SCI [30]. The pack/unpack buffers can be allocated from some shared regions between a sender and a receiver.

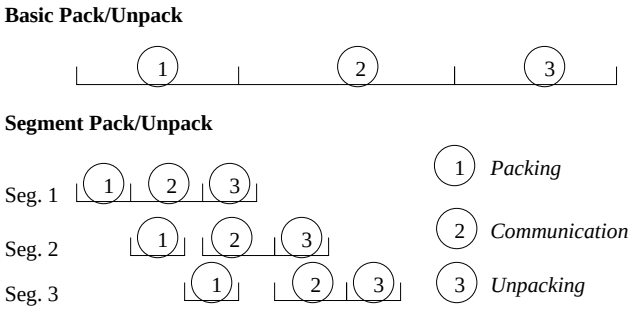

Figure 3. Overlap between packing, network communication and unpacking in the Buffer-Centric Segment Pack/Unpack scheme.

4.3 Design Issues in BC-SPUP

The Buffer-Centric Segment Pack/Unpack scheme shows very attractive potential benefits over the basic Pack/Unpack scheme. However, several issues need to be addressed for this scheme to be used in MPI implementations to achieve high performance for datatype communication.

4.3.1 Pipelining Sender Processing

One of the main objectives in the BC-SPUP scheme is to overlap host processing (including datatype processing and packing/unpacking) and communication. This overlap is achieved by breaking a large datatype message into several segments and pipelining the host processing and communication of each segment. This pipelining at the sender has a pronounced effect when operating in the context of a Rendezvous send. In this case it is not possible to start the next message from the application point of view, thus pipelining within this message shows substantial improvement. In the case where the application submits multiple small messages instead of large ones, these will be pipelined independently of our choice of datatype processing. However, to enable a sender to perform packing on any part of

a datatype message, partial datatype processing is required. Partial datatype processing allows us to start and stop the processing of a datatype at nearly arbitrary points. Several techniques [26, 15] have been proposed to provide partial processing on MPI datatypes. Another issue is to choose the segment size. Given a datatype message, the segment size should be chosen to provide good overlap and to utilize high network bandwidth as well.

4.3.2 Pipelining Receiver Processing

A receiver must know when each segment arrives to pipeline. This requires a notification per segment for a datatype message. A sender can use RDMA Write with Immediate data to send each segment. Then a receive descriptor is consumed and a completion entry is generated into a CQ. However, this approach requires that multiple receive descriptors have been posted for the coming segments. In another way, the send can write a flag to a specified location in the receiver memory. The receiver then can check this flag to detect the arrival of a segment. This approach can avoid pre-posting receive descriptors and possible flow control. However, it requires one more RDMA operation per each segment or techniques discussed in [19]. In our design, we choose RDMA Write with Immediate data.

4.3.3 Handling Buffer Limit

The size of pre-registered pack/unpack buffer pool should be limited to an appropriate size. We could have a per-connection buffer pool to simplify buffer management; however, it will occupy a significant amount of physical memory and limit scalability. It is desirable that the pack/unpack buffer pool in each process be used to communicate with any remote process. In case of burst communication, the pack/unpack buffer pool may be used up. If all pack buffers are used up, a sender can stop sending messages and wait for completion of previous operation for pack buffers. If all unpack buffers are used up, a receiver can delay response to the sender and then stall the communication until unpack buffers are available. Another solution to this issue is to allocate extra pack/unpack buffers when they are used up. These buffers can be added into the pack/unpack buffer pool. When the total size exceeds some threshold, some of these extra pack/unpack buffers may be deregistered. We choose the second solution. When pack/unpack buffers are used up, we fall back to the dynamic pack/unpack allocation and registration as in the basic pack/unpack scheme.

5 Copy-Reduced Approaches

A common feature in the Pack/Unpack-based approaches described above is that data is copied between user buffers and pack/unpack buffers. Copy-reduced approaches are proposed to reduce or avoid these copies. In this section, we describe three design schemes: *RDMA Write Gather with Unpack*, which avoids packing on the sender side; *Pack with RDMA Read Scatter*, which avoids unpacking on the receiver side; and *Multiple RDMA Writes*, which avoids both packing and unpacking.

5.1 RDMA Write Gather with Unpack

In RDMA Write Gather with Unpack (*RWG-UP*) scheme, only unpack buffers are assigned on the receiver side. A sender uses RDMA Write with Gather operations to write multiple contiguous blocks of a datatype message from its user buffers directly into the receiver's unpack buffer. Then, the receiver unpacks data into its user buffers. Figure 4 shows this scheme. Thus, packing (i.e. one copy) is eliminated on the sender side. Since a relatively large number of blocks can be gathered in one RDMA Write operation (for example, the current Mellanox SDK supports 64 blocks), the total number of RDMA Write operations needed to transfer the whole datatype message is reduced significantly. Therefore, the total startup overhead of all RDMA operations is reasonably low. In addition, information about a receiver's unpack buffer for performing RDMA Write is simple.

This scheme can easily achieve segment unpack as discussed in the BC-SPUP scheme in Section 4.2 to mask the memory copy cost on the receiver side. The sender can break a large message into several segments. Each time it uses RDMA Write with Gather to send a segment and drives the receiver to unpack the incoming segment.

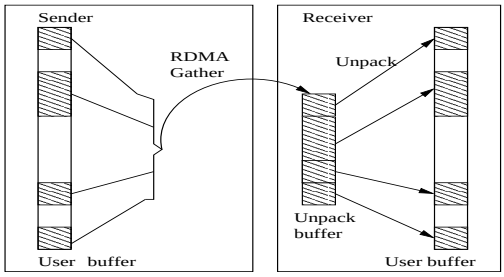

Figure 4. RDMA Write Gather with Unpack Scheme.

Similarly, as mentioned in the BC-SPUP scheme, segment unpack requires partial datatype processing and an appropriate segment arrival mechanism. To enable RDMA Write operations on a datatype message, the sender needs to register all contiguous pieces of the datatype message. Unlike registering and deregistering a contiguous buffer, memory registration and deregistration on datatype message buffers are more complicated. We discuss how to achieve efficient memory registration and deregistration on datatype message buffers in details in Section 5.4.1.

5.2 Pack with RDMA Read Scatter

As shown in Figure 5, the Pack with RDMA Read Scatter (*P-RRS*) scheme follows the exactly opposite procedure of the above RDMA Write Gather with Unpack scheme. The sender packs a datatype message into a pack buffer, then it asks the receiver to read it directly using RDMA Read operations. The receiver can scatter what it reads into multiple blocks of its datatype message buffer in one single operation.

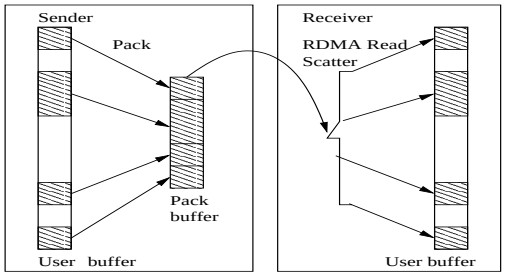

Figure 5. RDMA Read Scatter with Pack Scheme.

This scheme has almost the same requirements as mentioned in the RWG-UP scheme. However, this scheme is a little more costly to pipeline packing and communication. The sender can have segment pack, however, when a segment is available, a control message must be sent to the receiver to trigger the receiver's RDMA Read operation. Another difference is that RDMA Read performance is always lower than that of RDMA Write [31]. This scheme may be useful for asymmetric datatype communication: the sender's datatype is contiguous and the receiver's datatype is noncontiguous. For communication with noncontiguous datatypes on both sides, this scheme will be less efficient than the above RWG-UP scheme. Due to these observations, we do not implement this scheme in our implementation.

5.3 Multiple RDMA Writes

The Multiple RDMA Writes scheme (*Multi-W*) writes each contiguous piece of a datatype message directly into the receiver's buffer, as shown in Figure 6. This scheme can achieve zero-copy datatype messaging. There are two requirements. First, all contiguous blocks of user datatype message buffers must be registered. Second, the sender should be aware of the layouts of contiguous blocks in the receiver user buffer. That means the receiver must send the sender not only its buffer information, but also its datatype information. Then, the sender can decide the source and destination buffers for each RDMA Write operation according to these information.

One of the disadvantages in this scheme is the number of RDMA operations may be large. Another disadvantage

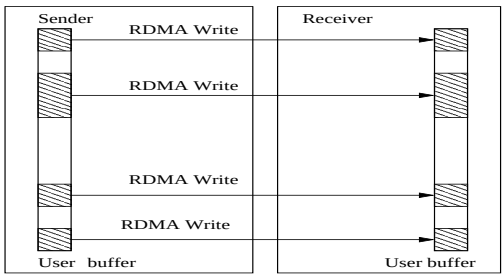

Figure 6. Multiple RDMA Writes Scheme.

is that network utilization may be low because the message sizes in RDMA writes are limited to the sizes of contiguous blocks. However, it can be expected that when these sizes are reasonably large, this scheme can achieve good performance because of zero-copy messaging. The third disadvantage is that some datatype information may be complicated. We discuss how a receiver can send its datatype information to a sender efficiently in Section 5.4.2.

5.4 Design Issues in Copy-Reduced Approaches

Copy-Reduced approaches show very attractive potential benefits due to reduced memory copies. However, several issues need to be addressed for these schemes to be used in MPI implementations to achieve high performance for datatype communication. We address the following two important issues: *reducing memory registration and deregistration overhead* and *handling receiver's datatypes*.

5.4.1 Reducing Memory Registration Overhead

Reducing overheads of memory registration and deregistration on datatype message buffers is a common issue in all three Copy-Reduced schemes. Techniques such as Pin-down cache [12] and FMRD [32] mainly deal with registration and deregistration of contiguous buffers. There is another complication in registering datatype message buffers due to data noncontiguity in these buffers.

There are two simple schemes to register datatype message buffers. The first scheme registers only contiguous blocks. A large number of buffer registration and deregistration events occur and result in high overheads. The second scheme registers the whole buffer which covers the datatype message, including gaps between two contiguous blocks. This scheme reduces the number of registration and deregistration operations at the cost of registering more space. If the total size of the gaps is large, this scheme also becomes costly.

Therefore, a tradeoff should be made between the number of registration and deregistration operations and the total size of memory space to be registered and deregistered. In [33], we proposed an efficient approach, *Optimistic Group Registration (OGR)* to register a list of arbitrary buffers. OGR also deals with the situation in which

gaps between two buffers may not really have been allocated by the application. Thus, it is a more general case than the case of MPI datatype message buffers.

The Optimistic Group Registration scheme first groups buffers into several memory regions. A cost model is used to achieve the tradeoff between the number of operations and the buffer size to be registered and deregistered. Details of the model can be found in [33]. Large gaps which nulls any benefit over individual registration are filtered out. Then, it registers each region independently. The effectiveness of this scheme has been demonstrated in [33] as well.

5.4.2 Handling Receiver's Datatypes

A unique issue arises in the Multiple RDMA Writes scheme: handling the receiver's datatype on the sender side. This issue results from three facts. First, MPI datatypes have only local semantics. Datatypes defined in one process cannot be used in another process. Second, the layouts of data in both sides could be different between a matched send and receive operation. Third, a sender must be able to figure out the address of each contiguous block in a receiver's datatype message buffer and its related registration information such as protection keys before performing RDMA write operations. Therefore, the receiver should send its datatype representation and related registration information to the sender before data transmission. Such information can be sent along with other control messages (if small) or as a separate message. Note that in heterogeneous systems, this issue becomes even more complicated. Our discussion is based on homogeneous systems.

An MPI datatype can be represented by a linear list of <*offset, length*> tuples. Each tuple describes a contiguous block of the datatype by its length and by its offset related to the lower bound. Other light-weight representation formats such as type tree [15] and dataloop [26] can be used here to reduce the size of datatype representation messages. To avoid sending datatype representation for each operation, a datatype cache mechanism [14] can be used. This cache mechanism was proposed by Träff *et al* in the context of performing MPI-2 [22] one-sided communication. The basic idea is to restructure an internal datatype when a datatype representation is received, and store it in a cache table on the sender side. This cache is indexed by the receiver's global rank and the index of a datatype. Thus, each datatype information is sent only once in the first communication operation using that type. Only the type index is needed to be sent in the following functions which use the same datatype. We extend this cache mechanism to handle datatype free and datatype index reuse. In case the receiver frees a datatype, and reuses the type index for a new datatype, the receiver is responsible for sending the new datatype representation. To achieve this, we associate a version number with each type index. When a type index is reused, its version number increases by one. The receiver can detect the version number change and then send the new datatype presentation to the sender. The sender simply replaces the obsolete datatype in its cache with the new one.

6 Choosing an Appropriate Approach

We discussed four main approaches to transfer datatype messages in the last two sections. These approaches have their own advantages and disadvantages. Accordingly, they offer different performance in different situations. An interesting question is: *Given a datatype communication, can we choose the best approach to perform data transfer?* In this section, we discuss the feasibility of choosing an appropriate approach for a given datatype message communication.

The first choice is to choose which type of approach: Pack/Unpack-based approaches or Copy-Reduced approaches. This choice is related to whether the costs of memory registration and deregistration on user buffers are comparable to the memory copy costs and how frequently these buffers are reused among multiple operations. For small messages, it is acceptable to go with the basic pack/unpack scheme. For large messages, without extra information, the MPI implementation cannot make a perfect choice over this. Our experience [18] shows that many applications use only several buffers for all communication, and the costs of memory registration and deregistration can be amortized among multiple operations. Thus, it may be beneficial to go with Copy-Reduced approaches. It is also helpful if we can make use of `MPI_Info` objects to notify the MPI implementation of buffers on which the application has many communication operations. This can help to decide whether to register these buffers or not.

The second choice is which particular Copy-Reduced approach may be more efficient. The handshake between a sender and a receiver in the Rendezvous protocol as mentioned in Section 3.1 can help make a choice. An important metrics in many common cases is the average size and the median size of all contiguous blocks. If these two sizes are large enough (e.g. several KBytes), the Multiple RDMA Writes scheme is a good candidate. Otherwise, RDMA Write Gather with Unpack and Pack with RDMA Read Scatter may be more suitable. For example, the receiver can decide whether it allocates an unpack buffer or not according to its datatype data layouts. Then, the sender can choose Multiple Writes or RDMA Write Gather with Unpack accordingly.

7 Implementation Details

We integrated the Buffer-centric Segment Pack/Unpack (BC-SPUP), RDMA Write Gather with UnPack (RWG-UP), and Multiple RDMA Write (Multi-W) schemes into

MVAPICH [19, 24]. In this section, we discuss some implementation details. First, we discuss the implementation of small datatype messages using the Eager protocol. Then we describe the implementations of BC-SPUP, RWG-UP, and Multi-W in the Rendezvous protocol.

7.1 Small Datatype Messaging

As mentioned in Section 3.1, small messages are first copied into the Eager protocol internal buffers in the MPI layer. Unlike the original implementation which uses extra pack/unpack buffers, we use the Eager protocol internal buffers as pack and unpack buffers. When a small datatype message is ready to be sent, we pack it directly to the Eager protocol internal buffers. Small datatype messages are also received in the Eager protocol buffers. The receiver then unpacks data into user buffers. Figure 7 shows small datatype message communication in our implementation. Compared to the original small datatype communication as shown in Figure 1, two copies are reduced. In addition, no pack/unpack buffers are needed.

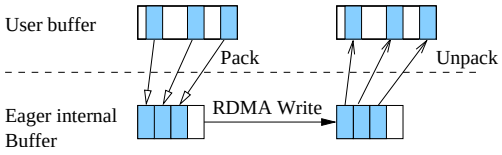

Figure 7. Small Datatype Message Transfer in Eager Protocol.

7.2 Implementing BC-SPUP

We allocate and register one large pack buffer and one large unpack buffer during MPI initialization time, each with a size of 20 MBytes. These buffers are divided into a list of segment buffers with a maximum supported segment size (128 KBytes). Large datatype messages use these buffers. We statically define a segment size for a given message size. For example, if the message size is equal to or larger than 1 MBytes, the segment size is 128 KBytes. A simple rule to choose the segment size is to have at least two segments if the message size is equal to or larger than 16 KBytes. If the message size is less than 16 KBytes, all data will be sent in one segment. Tuning on the segment size is quite important; however, as a proof-of-concept implementation, we simplify the selection here. When the segment size is decided, the sender first packs data into a segment buffer; then it RDMA writes data into the receiver's unpack buffer. In case there is no pack or unpack buffers available, we allocate and register extra buffers. We implemented partial datatype processing mentioned in [26].

7.3 Implementing RWG-UP

In our RWG-UP implementation, an unpack buffer with size of 20 MBytes is allocated and registered during the MPI initialization. This buffer is divided into a list of segment buffers with the maximum supported segment size (128 KBytes). The sender registers the user buffer using the Optimistic Group Registration scheme (implementation details can be found in [33]). When a receiver receives a "rendezvous start" message, it tries to assign a list of segment buffers from the unpack buffer. If there is not enough buffer available on the receiver side, the receiver is responsible to allocate and register extra buffers. Then the receiver sends a "rendezvous reply" message with segment unpack buffer information to the sender. The sender then initiates RDMA Write with Gather operations to put data into the receiver's segment unpack buffers. Immediate data is used in RDMA Write operations to drive the receiver to perform unpacking when a segment arrives.

7.4 Implementing Multi-W

In this scheme, the sender first sends a "rendezvous start" message to the receiver. It then goes to register the user datatype message buffer. When the receiver receives a "rendezvous start" message and a matched receiver operation is issued, it registers the receive buffer and sends related buffer information and datatype representation information to the sender. After the sender receives this message, it performs RDMA Write operations. Registration and deregistration are implemented in the same way as in the implementation of RWG-UP scheme. We use a light weight representation of datatype as mentioned in [26]. We have two implementations to post a list of RDMA write descriptors. One uses the standard post function to post each descriptor one by one. Another uses an extended interface [1] to post a list of descriptors to the same send queue in one operation.

8 Performance Results

This section presents performance results from a range of benchmarks on our implementations of different schemes in MVAPICH. We attempted to find some standard benchmarks such as the NAS benchmarks [4] and the ASCI blue benchmarks [20], in which we wished datatype communications were used. Unfortunately no noncontiguous datatype communication is used in these benchmarks yet, perhaps due to the limited performance achieved in most applications. SKaMPI [25] provides benchmark for MPI derived datatypes. The test datatypes are synthetic and most parameters are defined by users. For simplicity, we developed our own benchmarks, with intention to capture typical usage of derived datatypes. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes, or 1024×1024 bytes.

In this section, we first compare point-to-point latency and bandwidth, and performance of collective operations in different implementations. Then, we quantify effects of sev-

eral design choices on the performance of datatype communication, including segment unpack, list descriptor post and buffer usage.

8.1 Experimental setup

Our experimental testbed consists of a cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_17_0000-rc12-build-001. We used the Linux RedHat 7.2 operating system.

8.2 Vector Micro-benchmark

We evaluated the same example described in Section 3.2 in three new implementations: BC-SPUP, RWG-UP, and Multi-W. Unless stated otherwise, in our tests, segment unpack was enabled in the RWG-UP scheme and list descriptor post was enabled in the Multi-W scheme. In this benchmark, a certain number of columns in a two-dimensional 128×4096 integer array are transferred between two processes. These columns can be represented by a vector datatype shown in Section 3.2. The number of columns varies from 1 to 2048.

Figure 8 compares ping-pong latencies in different implementations, including the current MVAPICH datatype implementation ("Generic"). BC-SPUP performs better than the Generic scheme consistently. It gives a factor of 1.5 improvement over the Generic scheme for large datatype messages. RWG-UP performs better than the Generic scheme in most cases, except that the size of contiguous block is too small for RDMA operations to achieve good performance. It gives a factor of up to 1.8 improvement over the Generic scheme. Multi-W offers a factor of 3.4 improvement when the number of columns is large. When the size of contiguous blocks is small, Multi-W performance degrades significantly.

Figure 9 compares their bandwidth. In the bandwidth test, the same vector datatype is used. The sender pushes 100 consecutive datatype messages and then waits for a reply from the receiver when all messages have been received. Both BC-SPUP and RWG-UP give a factor of 1.2–2.0 improvement over the Generic scheme. Multi-W gives a factor of 1.4–3.6 improvement over the Generic scheme when the number of columns is larger than 64. Similarly, when the number of columns varies from 4 to 64, Multi-W performance degrades a lot because of the large number of RDMA Write operations and the small message size in each operation.

When the number of columns is 1 or 2, the datatype message follows the Eager protocol and has the same communication path in all BC-SPUP, RWG-UP and Multi-W schemes. Thus, the performance is identical. Compared to the Generic scheme, two copies are saved as shown in Figure 7. Thus, there is perceivable improvement over the Generic scheme.

8.3 Performance of MPI_Alltoall

Collective datatype communication can benefit from high performance point-to-point datatype communication provided in our implementations. We noticed that some collective operations such as `MPI_Bcast` perform explicit pack and unpack operations in their implementation when noncontiguous datatype communication occurs [28], these collective operations will not benefit from the performance improvement of point-to-point datatype communication achieved in our implementations. Many of others, which still use point-to-point noncontiguous datatype communication in their implementation [28], can benefit from our implementations.

We designed a test to evaluate `MPI_Alltoall` performance with derived datatypes. In the previous tests, all block sizes are same. In this test, we designed a structure datatype in which the size of its contiguous blocks is different. The datatype is designed as shown in Figure 10: the block size varies from one integer to x integers. The gap (empty blocks in the plot) between two blocks equals to the size of the first block.

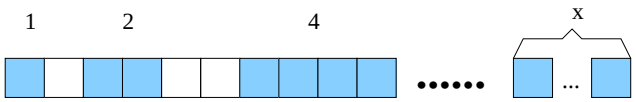

Figure 10. A Struct Datatype.

Figure 11 compares MPI_Alltoall performance of four datatype communication implementations. We use 8 processes. The number of integers in the last block varies from 2048 to 131072, as shown in the x-axis. The block sizes increase exponentially from 4 bytes to the largest block size from the first block to the last block. For example, when the number of integers in the last block is 8192, the block sizes vary from 4 bytes to 32768 bytes. We can see that all BC-SPUP, RWG-UP and Multi-W schemes outperform the Generic scheme. BC-SPUP gives an improvement factor of minimum 1.2, maximum 1.5, and average 1.3. RWG-UP gives an improvement factor of minimum 1.2, maximum 1.4, and average 1.3. Multi-W gives an improvement factor of minimum 1.8, maximum 2.1, and average 2.0. For this datatype, it can be observed that Multi-W is a good choice.

Measurements for other collective operations, which

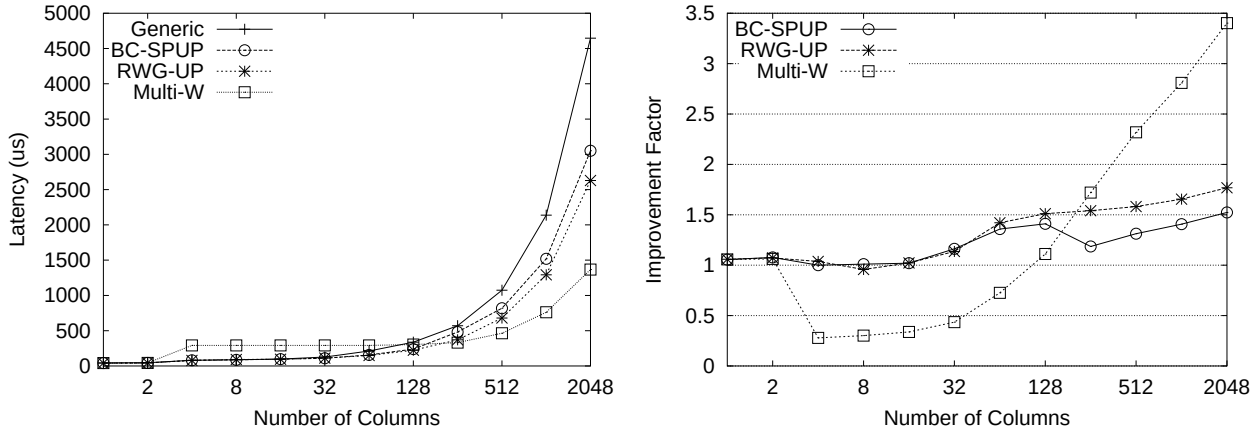

Figure 8. Latency Comparison.

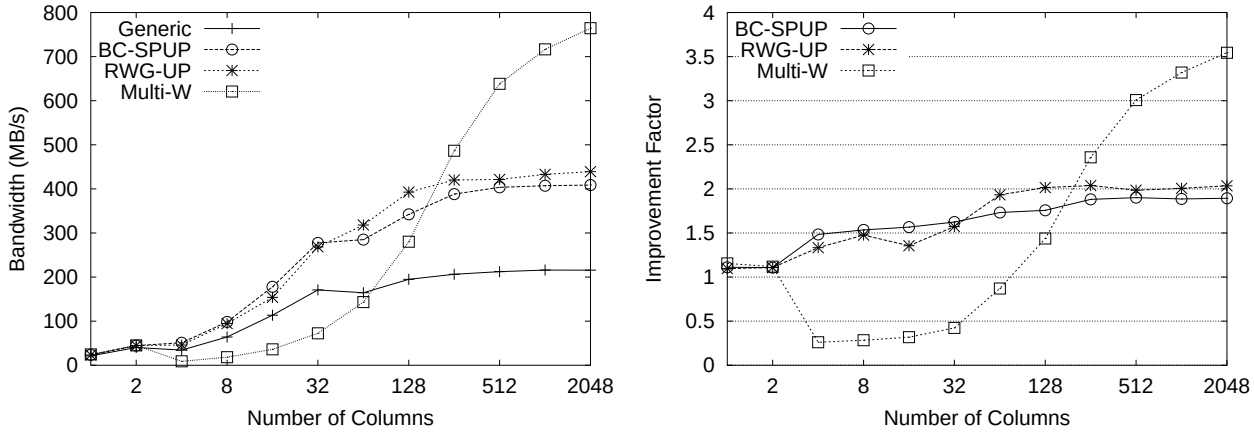

Figure 9. Bandwidth Comparison.

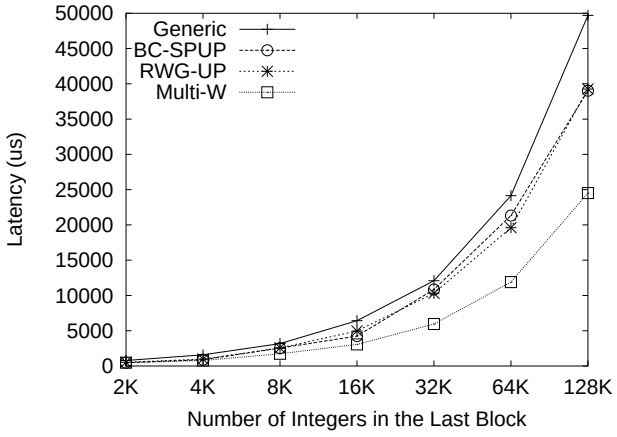

Figure 11. MPI_Alltoall Performance.

could not be presented in this paper due to space limitation, have shown similar results as we observed in the test of MPI_Alltoall.

8.4 Effects of Segment Unpack

To quantify the effects of segment unpack in the RWG-UP scheme, we disabled the unpack trigger in each segment. Then, the receiver begins to unpack until the whole datatype message arrives. It can be expected that the segment unpack gives us better performance due to the overlap between communication and unpacking. We used the aforementioned vector bandwidth test to quantity the effects of segment unpack. Figure 12 shows that a factor of 1.3 improvement in bandwidth can be achieved using the segment unpack.

8.5 Effects of List Descriptor Post

As mentioned in Section 7.4, we have two methods to post a list of RDMA write descriptors: single post many times and list post once. We evaluated the vector band-

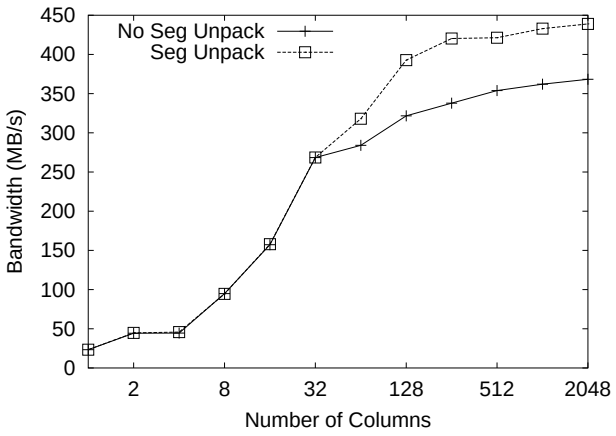

Figure 12. Effects of Segment Unpack.

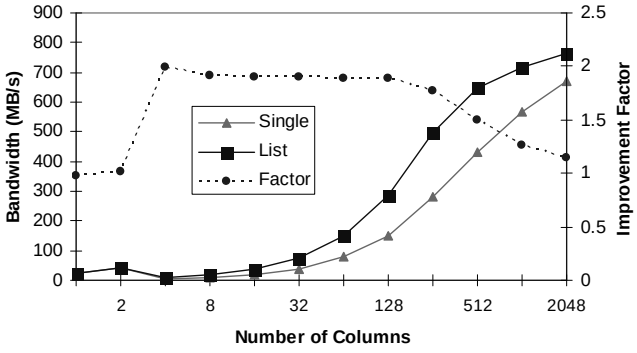

Figure 13. Effects of List Descriptor Post.

width test on these two methods of the Multi-W scheme. Figure 13 shows that the list post offers improvement with a maximum factor of 2.0 and a minimum factor of 1.2 over the single post. The average improvement factor is 1.6. This has two implications. First, posting descriptor is costly and we expect InfiniBand vendors to further optimize it. Second, list descriptor post is a good extension and should be supported.

8.6 Effects of Buffer Usage

Buffer usage has a great impact on MPI communication performance [18]. Performance achieved in schemes we discussed has different dependency with buffer usage, for either application buffers or MPI internal buffers. Particularly, as shown in Section 3.2, the pack/unpack buffers used in the Generic scheme are dynamically allocated and potentially change in datatype communication operations. It is highly probable that memory registration is necessary for each datatype communication. The BC-SPUP scheme uses a pre-registered buffer pool to reduce the impact of dynamic memory allocation and registration to some extent. In the RWG-UP scheme, the unpack buffer is also pre-registered. However, both schemes will need to stall communication when the buffer pool is used up or register more buffers. Both RWG-UP and Multi-W schemes register user buffers. Their performance heavily depends on user buffer usage patterns. For example, if an application keeps using different buffer in each operation, the registration becomes necessary. To show the buffer usage effect, we conducted the vector latency test in a worst scenario. That is, if a scheme needs an internal buffer, the internal buffer is allocated, registered, and deregistered on-the-fly; if a scheme uses user buffers directly, the user buffers are different, dynamic registration and deregistration are included.

Figure 14 shows the worst latencies for each scheme. When the number of columns is less than 512, both RWG-UP and Multi-W schemes perform very poor. This is because they need to register and deregister the whole user array (registering each block is even more costly in this case [33]), while Generic and BC-SPUP only register and deregister buffers with the real data size. The memory registration and deregistration costs dominate their performance. When the number of columns increases, the difference in the costs of registration and deregistration between these schemes decreases. While the memory copy costs catch up, both RWG-UP and Multi-W perform better than Generic due to reduced memory copies. In this test, BC-SPUP always performs better than Generic. Since they both have same registration and deregistration costs, the benefits completely come from the overlap between packing, communication, and unpacking due to the segment pack/unpack technique.

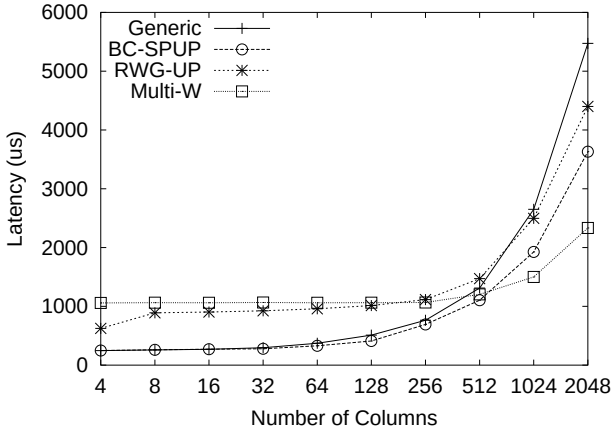

Figure 14. Latency Comparison in the Worst Case of Buffer Usage.

9 Related Work

There are three potential areas to improve MPI datatype communication: *Improving datatype processing system,*

Optimizing packing and unpacking procedures; *To take advantage of network features to improve noncontiguous data communication*. There have been some successful work in each area.

In [11], Gropp *et al.* have provided a taxonomy of MPI derived datatypes according to their memory access patterns and described how to efficiently implement these patterns. Their work focuses on the pack/unpack mechanism. Träff *et al.* have described a technique, called flattening on the fly, for improving the performance of derived datatypes [15]. Their method aims to have a light-weight representation for datatypes using a stack-based approach. Their approach minimizes the use of expensive recursive function calls to parse a derived datatype in the MPI implementation and improves the performance of packing/unpacking. Ross *et al.* [26] have designed a reusable datatype-processing component for the MPICH2 implementation [2]. This component provides three key characteristics: simplified type representation; support for partial processing of datatypes; and separation of type parsing from action to perform on data. Thus, this component can be used in our implementations to further improve performance as our future work.

Byna *et al.* [5] have presented a technique to optimize datatype packing performance. In their work, optimized packing algorithms are automatically selected with respect to the architecture-specific parameters and the datatype memory access patterns. Recently, MPICH2 [2] has begun to deploy segment pack and unpack in its implementation. The Los Alamos Message Passing Interface (LA-MPI) system [9] has used shared memory regions as pack and unpack buffers in its datatype communication path.

In [30], Worringen *et al.* have presented a direct copy technique to improve performance of datatype communication. Their work takes advantage of remote memory operations provided in the SCI network to avoid one intermediate copy. In [33], we have demonstrated the benefits of using RDMA Gather/Scatter operations to support noncontiguous file access in PVFS over InfiniBand. Only the case in which buffers on the I/O server side are always contiguous is discussed. In this paper, we deal with a more general case and more design alternatives.

There is also other I/O work related to MPI datatype. Ching *et al.* [6] have used datatype to present noncontiguous file access information and then to reduce the number of requests and the size of request messages. Worringen *et al.* [16] have described a listless I/O to improve noncontiguous file access. Their work centers around efficient datatype processing and packing/unpacking.

None of these previous work discusses and analyzes the benefits of segment pack and unpack. Issues to register and deregister pack/unpack buffers on RDMA-capable networks are not addressed yet. Furthermore, using RDMA operations to reduce memory copies in datatype communication and its associated design issues are also not addressed in these previous work.

10 Conclusions and Future Work

In this paper, we systematically study two main types of approach for MPI datatype communication: *Pack/Unpack-based approaches* and *Copy-Reduced Approaches* on the InfiniBand network. Along the first type of approach, to reduce the impact of pack/unpack costs on the performance of datatype message communication, we propose a technique called *Buffer-Centric Segment Pack/Unpack* This scheme provides overlap between packing, communication and unpacking in datatype communication. It also avoids dynamic memory registration and deregistration in common cases.

Along the second type of approach, we propose three schemes: RDMA Write Gather with Unpack, Pack with RDMA Read Scatter, and Multiple RDMA Writes. The main idea behind these schemes is to use RDMA operations to reduce and/or eliminate packing and unpacking in datatype communication.

We implement and evaluate three of these four schemes: Buffer-centric Segment Pack/Unpack, RDMA Write Gather with Unpack, and Multiple RDMA Writes in MVAPICH over InfiniBand. Performance results of a vector micro-benchmark demonstrate that latencies are improved by a factor of 1.5-3.4 and bandwidth by a factor of 2.0-3.6 for many cases compared to the current datatype communication implementation, which is derived from MPICH. Collective operations on datatype messages also benefit from these schemes. A factor of up to 2.0 improvement over the original implementation has been seen in our measurements on a 8-node system.

We show that segment pack and/or segment unpack are quite helpful to achieve overlap between host processing and network communication. The implication of this finding is that the pipelining technique becomes more powerful in systems with networks such as InfiniBand which can provide comparable performance to that of memory system. In the RDMA Write Gather with Unpack scheme, we show that segment unpack gives us a factor of 1.3 improvement over that without segment unpack.

We also notice that these schemes perform differently in different cases. We believe it is feasible to choose an appropriate one to fit a given datatype communication to achieve the best performance. This selection is also possible within different parts of a single datatype message. We are currently working in this direction.

We plan to have more tests and performance tuning in our current implementations of these schemes and then release them publicly.

Acknowledgments

We would like to thank David Ashton, Darius Buntinas, Rob Ross and Neill Miller at Argonne National Laboratory for MPI derived datatype discussion with us. We are also thankful to nowlab fellows, Dr. Hyun-Wook Jin, Jiuxing Liu, Pavan Balaji, Weikuan Yu, Amith Mamidala, Weihang Jiang, Adam Wagner, Karthikeyan Vaidyanathan, Sayantan Sur, Gopalakrishn Santhanaraman, Sundeep Narravula, Abhinav Vishnu, Savitha Krishnamoorthy, and Ranjit Noronha for their helpful comments on the paper draft.

References

[1] Mellanox Technologies. `http://www.mellanox.com`.

[2] Argonne National Laboratory. MPICH2 Release 0.94. `http://www-unix.mcs.anl.gov/mpi/mpich2/`.

[3] M. Ashworth. A Report on Further Progress in the Development of Codes for the CS2. In *Deliverable D.4.1.b F. Carbonnell (Eds), GPMIMD2 ESPRIT Project, EU DGIII, Brussels*, 1996.

[4] D. H. Bailey, E. Barszcz, L. Dagum, and H. Simon. NAS Parallel Benchmark Results. Technical Report 94-006, RNR, 1994.

[5] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[6] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[7] P. Ezolt. A Study in Malloc: A Case of Excessive Minor Faults. In *Proceedings of the 5th Annual Linux Showcase and Conference*. USENIX Association, 2001.

[8] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[9] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. Minnich, C. E. Rasmussen, L. Dean Risinger, and M. W. Sukalski. A Network-Failure-tolerant Message-Passing system for Terascale Clusters. In *Proceedings of the 2002 International Conference on Supercomputing*, June 2002.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

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

[12] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th Int. Parallel Processing Symposium*, March 1998.

[13] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24, 2000.

[14] J. L. Träff, H. Ritzdorf and R. Hempel. The Implementation of MPI–2 One-sided Communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.

[15] J. L. Träff, R. Hempel, H. Ritzdorf and F. Zimmermann. Flattening on the Fly: Efficient Handling of MPI Derived Datatypes. In *PVM/MPI 1999*, pages 109–116, 1999.

[16] J. Worringen, J. L. Träff, and H. Ritzdorf. Fast Parallel Non-Contiguous File Access. In *Supercomputing 2003: The International Conference for High Performance Computing and Communications*, Nov. 2003.

[17] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. `http://www.nersc.gov/research/FTG/mvich/index.html`, August 2001.

[18] J. Liu, B. Chandrasekaran, J. W. W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand Myrinet and Quadrics. In *Supercomputing 2003: The International Conference for High Performance Computing and Communications*, Nov. 2003.

[19] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing*, June 2003.

[20] Los Alamos National Laboratory. The ASCI Blue Benchmarks. `http://www.llnl.gov/asci-benchmarks/`.

[21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[22] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[23] Myricom Inc. MPICH-GM Software. `http://www.myricom.com/scs/index.html`.

[24] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html`, January 2003.

[25] R. Reussner, J. L. Träff, and G. Hunzelmann. A Benchmark for MPI Derived Datatypes. *Lecture Notes in Computer Science*, 1908:10+, 2000.

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

[27] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A Case Study in Application I/O on Linux Clusters. In *SC2001*, Nov. 2001.

[28] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. In *EuroPVM/MPI*, Oct. 2003.

[29] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Oct. 27–31, 1996.

[30] J. Worringen, A. Gaer, F. Reker, and T. Bemmerl. Exploiting Transparent Remote Memory Access for Non-Contiguous- and One-Sided-Communication. In *Workshop on Communication Architecture for Clusters 2002 (in conjunction with IPDPS)*, April 2002.

[31] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. Technical Report, OSU-CISRC-0/03-TR, April 2003.

[32] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[33] J. Wu, P. Wyckoff, and D. K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.