Improving Parallel 3D FFT Performance using Hardware Offloaded Collective Communication on Modern InfiniBand Clusters

Krishna Kandalla[1], Hari Subramoni[1], Karen Tomko[2], Dmitry Pekurovsky[3], Nishanth Dandapanthula[1], Sayantan Sur[1] and Dhabaleswar K. (DK) Panda [1]

[1] *Department of Computer Science and Engineering, The Ohio State University* {kandalla, subramon, dandapan, surs, panda}@cse.ohio-state.edu

[2] *Ohio Supercomputer Center,* {ktomko}@osc.gov

[3] *San Diego Supercomputer Center* {dmitry}@sdsc.edu

***Abstract*—Parallel 3D FFT (P3DFFT) is an important component of many scientific computing applications ranging from fluid dynamics, astrophysics and molecular dynamics. One of the main limiting factors of parallel 3D FFT performance and scalability is the time spent in All-to-all communication. The implementation of P3DFFT uses the Message Passing Interface (MPI) as the parallel programming model. MPI is the dominant programming model for past couple of decades, and many scientific applications use it. Of the many communication primitives offered by MPI, collective communications, especially MPI_Alltoall, consume a lot of time for applications using P3DFFT. Hiding the latency of MPI_Alltoall is critical towards scaling libraries such as P3DFFT. The newest revision of MPI, MPI-3, is widely expected to provide support for non-blocking communication to enable latency-hiding. At the same time, popular interconnection networks, such as InfiniBand, have provided support for non-blocking collective communication. For example, the latest ConnectX-2 adapter by Mellanox provides a low-level task-list offload capability. In this paper, we design a non-blocking offloaded collective for Alltoall Personalized Exchange (MPI_Alltoall) using the task-list offload by the ConnectX-2 network adapter. Simultaneously, we re-design the P3DFFT library and a sample application kernel to overlap the MPI_Alltoall operations with application level computation and demonstrate the benefits of using our novel non-blocking Alltoall algorithm. Our experimental evaluation shows that we are able to achieve near perfect overlap of computation and communication (99%) through the use of offload mechanism without any adverse impact on the latency of the MPI_Alltoall operation. We are also able to see an improvement of upto 23% in the overall run-time with the modified P3DFFT kernel when compared to the default blocking version.**

I. Introduction

Across various scientific domains, application scientists are constantly looking to push the envelope of their research by running large scale parallel jobs on supercomputing systems. The need to achieve high resolution results with smaller turn around times has been driving the evolution of supercomputing systems over the last decade. Current generation supercomputing systems are typically comprised of thousands of compute nodes based on modern multi-core architectures and offer a vast array of computing resources. Interconnection networks have also rapidly evolved to offer low latencies and high bandwidths to meet the communication requirements of parallel applications. Together, these systems are allowing scientists to scale their parallel applications across tens of thousands of processes.

Parallel 3-Dimensional Fast Fourier Transform (P3DFFT) is one of the major components used by a wide array of scientific computation applications. P3DFFT is implemented using the Message Passing Interface (MPI) [1] parallel programming model. While several researchers have explored various different programming models [2], [3], MPI has been the dominant programming model for past couple decades and has scaled to the largest parallel machines. The MPI Standard defines a set of collective operations to provide abstractions for various group communication patterns. These operations are very popular among application developers owing to their ease of use and portability. Current generation scientific applications spend a considerable amount of their communication time performing various collective operations [4], [5]. The MPI Forum is working towards a revised specification, MPI-3, that is widely expected to enable non-blocking collective communication. A draft non-blocking collective communication chapter has already been formally read at recent MPI Forum meetings. The aim of including non-blocking collective communication is to enable latency hiding. While communication proceeds in the background, processors are expected to work on computation, thus providing overlap of computation and communication.

The goal of non-blocking collective communication interface is to allow application developers to utilize the computing cycles of the host processors to perform application level tasks while the collective operation progresses in the background. One of the major challenges of attempting overlap is that the MPI communication stack needs to be "progressed" to proceed with a list of communication operations. The progression is typically done by using the host processor which traverses lists of pending communication operations. In order to progress the MPI library, MPI_Test calls can be invoked. In [6], [7], [8], researchers have addressed this problem of overlapping collective communication and computation through the usage of MPI_Test calls. However, authors note that in order to achieve the best possible overlap, it is necessary to use the "right" number of MPI_Test

calls. The application developer needs to guess the right places and right number of calls to MPI_Test to ensure that progress is made. Of course, this is very hard to do in context of real applications due to two major reasons: *i)* the number of test calls varies according to computation time and network speed, both factors change dramatically for various systems, thus, making it virtually impossible to do this in a portable fashion, and *ii)* the computation for the application can be done by a third party library, therefore, the application developer has no way of inserting MPI_Test calls (such is the case with our example of P3DFFT). In [9], authors propose using a separate thread to progress the collective communication in the background while the main application thread performs compute tasks in the foreground. However, authors conclude that such methods are most beneficial if the progression threads could be scheduled on idle compute cores owing to possible resource contention between the compute and communication progress threads.

Given the significance of non-blocking collectives and the limitations of the existing designs, there has been a recent growth of interest in offloading collective operations to the network interfaces. In this regard, InfiniBand hardware vendors such as Mellanox and Voltaire have recently introduced network offload solutions such as ConnectX-2 [10] and Fabric Collective Accelerator (FCA) [11], respectively. In this paper, we leverage the network offload features provided by the ConnectX-2 InfiniBand network adapter to design an efficient non-blocking algorithm for the Alltoall Personalized Exchange (MPI_Alltoall) operation. We have also studied the performance characteristics of the P3DFFT application, which performs several large message MPI_Alltoall exchanges and we have re-designed the application to take advantage of our proposed non-blocking MPI_Alltoall algorithm to achieve communication/computation overlap. Our experimental evaluation shows that we are able to achieve near perfect overlap of computation and communication (99%) through the use of offload mechanism without any adverse impact on the latency of the MPI_Alltoall operation.

In short, we have addressed the following broad challenges in our work:

- In [12], [13], authors have proposed algorithms to optimize blocking collective operations. However, as applications are scaled out, it is necessary to hide the costs of collective operations. In this context, researchers have proposed a few interesting ideas to overlap computation with collective operations. But, is it possible to achieve better overlap by offloading collective operations to the network interface?
- The ConnectX-2 network interface provides network offload features and in [14], [15], [16], researchers have studied the potential benefits of such designs. However, these studies do not deal with data-moving collective operations. Can we design an efficient non-blocking algorithm that leverages the network offload feature for a dense operation like the *Alltoall Personalized Exchange*?
- A non-blocking interface should allow applications to perform multiple concurrent collective operations in an overlapped manner with compute tasks. Is it possible to design such an interface for non-blocking collectives with network offload designs?
- Finally, P3DFFT kernels rely heavily on MPI_Alltoall operations. How can we re-design a parallel 3-D FFT kernel to leverage our proposed non-blocking Alltoall operation? What is the impact of this re-design on performance and scalability of the P3DFFT kernel?

The rest of the paper is organized as follows. In Section II, we give a brief overview of InfiniBand, MPI, and other technologies used in this paper. Section III presents the motivation for our work. Section IV explains the design methodology we followed for implementing hardware offloaded Alltoall operations. In Section V, we describe our modifications to P3DFFT to leverage non-blocking Alltoall operations. Our experimental results and analysis are described in Section VI. Finally we summarize our conclusions and future work directions in Section VII.

II. Background and Related Work

In this section, we discuss the relevant background and related work for our work.

A. InfiniBand and ConnectX-2 Network Interface

InfiniBand has emerged as the popular I/O interconnect standard and almost 41% of the Top500 Supercomputing systems [17] rely on InfiniBand to address the communication requirements of the current generation applications. Current generation InfiniBand network cards and switches can deliver 32 Gbps end-to-end bandwidth (Quad-Data-Rate or QDR) and about 1-1.5 micro-second latency. InfiniBand is also the primary interconnection network on top machines such as Nebulae in China with 120,640 cores, ranking second on the list. The ConnectX-2 [10] network interface is the latest adapter introduced by Mellanox [18]. It offers all the standard InfiniBand features. In addition, it offers a new low-level task-list based offloading feature called CORE-Direct. Using this new feature, upper-level software can form arbitrary lists of send, recv and wait operations and post them to a work-request queue of the network card. The network card is then responsible for carrying out the tasks, freeing the processor effort in periodically checking on the progress of these tasks. Using this task list, collective operations may be designed by upper-level software.

B. Message Passing Interface

The Message Passing Interface (MPI) [1] is one of the popular programming models used for designing parallel applications. The MPI Standard defines various abstractions such as communicator objects, Cartesian topologies along with various communication primitives such as point-to-point, collective and one-sided operations. In our work,

we use the MVAPICH2 [19] software stack, which is a high performance MPI implementation over InfiniBand and RDMA networks.

Point-to-Point Communication Protocols: Most MPI libraries use the eager protocol for short messages and the rendezvous protocol for medium and large messages. However, for networks that offer RDMA features, any network exchange requires the data buffers to be registered with the network interface. In order to amortize the costs associated with registering buffers, the MPI libraries use a few optimizations such as creating a set of buffers and registering them with the network interface during the initialization and maintaining a cache for the dynamically registered user-buffers. When a process is performing an eager send(), it copies the data into one of these pre-registered buffers and posts an *ibv_post_send* operation. At this point, the MPI_Send() operation can return safely because the data has been buffered internally. Similarly, with the eager-recv() operations, the network interface places an incoming message into one of these pre-registered buffers and the host processor copies the newly arrived data into the user buffers. However, with the traditional rendezvous protocol, the processes first participate in a *ready-to-send/clear-to-send* hand-shake phase. Following this hand-shake, the MPI library needs to perform the necessary tag matching operations before initiating the *RDMA-Put* or the *RDMA-Get* operations. The specific protocols used in the MVAPICH2 library and their overlap characteristics were described in detail in [20], [21].

Collective Operations in MPI: The collective operations defined in the MPI specification can be conveniently and portably used by application developers to implement various communication patterns such as *One-to-All*, *All-to-One*, *All-to-All* and synchronization operations. Traditionally, MPI implementations relied purely on point-to-point operations to implement various collective algorithms. The current MPI-2.2 Standard defines collective operations to be blocking operations. However, the MPI-3 Standard [1] will include specifications regarding the usage and implementation of a non-blocking interface for collective operations.

The Alltoall Personalized Exchange operation(MPI_Alltoall) is the most dense collective operation defined in the MPI Standard. Every process that participates in this operation sends and receives a distinct message from every other process in the group. Since this operation is heavily used in the Fast Fourier Transform(FFT) kernels, several researchers have proposed algorithms to optimize its performance [22]. MVAPICH2 uses the hypercube algorithm proposed by Brucks et. al [23] for small messages and the pair-wise exchange algorithm for larger messages. The pair-wise exchange algorithm's cost function grows linearly with respect to the number of processes in the group [24]. Due to the volume of data being exchanged, the performance is also strongly affected by network contention.

C. P3DFFT and its applications

Many applications in areas including Direct Numerical Simulations of Turbulence, astrophysics, and material science rely on highly scalable 3D FFTs [25], [26], [27], [28], [29], [30], [31]. In [32], authors indicate that scaling FFT on systems with core-counts exceeding 10^4 is challenging. In [29], researchers note that the MPI communication overhead is more than 40% on 128 compute cores. In [31], scientists have reported experimental results associated with running 3-D turbulence models on a BlueGene System across 16K compute cores. Any improvements in the 3DFFT kernels can be directly translated into benefits to these end applications.

The Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) library from the San Diego Supercomputer Center (SDSC) is an open source library for carrying out 3D FFTs at large scale [33]. The portable library is written in Fortran 90 with MPI-based parallelization. It leverages the fast serial FFT implementations of either IBM's ESSL or FFTW. P3DFFT uses a 2D, or pencil, decomposition. This overcomes an important limitation to scalability inherent in FFT libraries by increasing the degree of parallelism to up to N^2 and has shown good scalability on 10's of thousands of cores when integrated into a Direct Numerical Simulation (DNS) turbulence application [34]. In this paper we use one of the sample programs provided with the P3DFFT library distribution, *test_sine*, to evaluate the impact of our non-blocking Alltoall with a variant of P3DFFT restructured for latency hiding. The *test_sine* kernel performs a forward transform followed by a backward transform on a set of 3D sine waves and verifies the results.

III. Motivation

With the advent of multi-core compute servers, current generation MPI implementations such as MPICH2 [35], Open-MPI [36] and MVAPICH2 [19], utilize aggressive mechanisms such as multi-core aware [12], [37], [38] and NUMA-aware [39] to optimize the performance of collective operations.

Since collective operations involve a group of processes, the communication cost models [40] suggest that the time required to complete a collective operation is a linear or a logarithmic function of the number of participating processes. In reality, the amount of time required to complete a collective operation could be even higher owing to various factors. As applications are scaled out to larger number of processes, the time required to complete various collective operations will continue to increase. Since these operations are blocking in nature, the performance of the collective operations poses a serious challenge to the scalability of applications. It is hence necessary to design an efficient non-blocking interface for collectives to overlap application-level

computation during the execution of collective operations in order to hide the latency associated them.

A. Current State-of-the-art with Non-Blocking Collectives: LibNBC

In Section I, we briefly discussed the concepts that have been proposed by researchers to overlap application level computation with collective communication. In this section, we describe them in greater depth and differentiate these ideas with our network offload designs.

1) Using Periodic MPI_Test calls for Progress: The libNBC library allows users to initiate a non-blocking Alltoall operation by calling "NBC_Ialltoall" and checking for its completion at a later time by calling "NBC_Wait". In order to progress the communication in between, we need to call "NBC_Test" while performing the compute tasks. If applications are designed in this manner, it is necessary to use the right number of these calls and the right amount of computation to achieve good overlap. We first measure the base latency of the NBC_Ialltoall/NBC_Wait operations without attempting to overlap any computation across 64 processes with a payload size of 8MB. We then run a computation loop between the NBC_Ialltoall and NBC_Wait calls that runs for the same duration as the measured base latency. (We describe our benchmarks and experimental setup in Sections VI-A and VI-B.) In Table I, we vary the frequency with which we call the NBC_Test operations and calculate the overlap % that is achievable. We can see that we get the best overlap for this particular case when we poke the MPI library 1000 times while the host processor is doing compute tasks. This leads us to the following observation:*Across various applications, collectives operations and payload sizes, to achieve the best overlap, it is necessary to determine the right amount of computation to perform and the right frequency of NBC_Test calls to use. This is not a trivial task and tuning the application to achieve the best overlap places a higher burden on the application scientists.*

2) Using a Separate Thread for Progress: LibNBC's threaded progression mode spawns a new thread that progresses the collective schedule in the background while the application performs computation in the foreground. However, utilizing multiple threads leads to resource sharing between the two threads and host processor's cycles should be divided up between these two threads in some manner. In [9], authors note that such designs are most efficient when the progression threads are scheduled on idle cores on the compute nodes. However, in most current generation supercomputing systems, running parallel jobs in a manner that utilizes all the cores across all the nodes leads to good resource utilization and better overall system throughput. Also, utilizing a real time thread to achieve overlap can sometimes be tricky and places a higher burden on the MPI library developers. We were unable to use libNBC's real-time thread progression mode for experiments. For the rest of our paper, we restrict ourselves to using the basic libNBC library with Test() calls to achieve communication progress. (A description of our experimental setup can be found in Section VI-A.)

B. How will Network Offload help Non-Blocking Collectives?

These observations lead us to the following conclusions. In order to provide the best possible overlap between collective communication and application level computation, it is necessary to:

- Minimize the involvement of the host processors' involvement in progressing the collective communication schedules, so that most of the CPU cycles can be utilized towards application level computation.
- Lower the burden on application scientists to tune their applications to use the right amount of computation and the frequency of MPI_Test() calls across various systems.
- Reduce the burden on MPI library developers by minimizing the usage of threaded progression techniques.

In [15], researchers have demonstrated the early designs that leverage the network offload features offered by the ConnectX-2 network interface. The ConnectX-2 interface allows software stacks to create a task-list with various send()/recv() and wait() operations to mimic a collective operation. Once a process creates a task-list and posts it to the network interface, the rest of the communication can be progressed by the network interface with very little intervention from the host processor, which can be used to perform application-level tasks. Researchers have already explored the challenges involved in offloading some of the MPI communication operations such as MPI_Barrier in [14]. In [16], authors demonstrate the design of collective communication primitives and their usage to offload collective communication to the network interface and also demonstrate the % of CPU availability during offload operations.

However, neither of these two papers have demonstrated the utility of the network offload interface with data moving collectives. In this paper, we propose a non-blocking algorithm that leverages the ConnectX-2 network offload interface for the Alltoall Personalized Exchange operation, which is the most dense collective operation.

Table I
PERCENTAGE OF OVERLAP FOR 8MB MESSAGE SIZE WITH 64 PROCESSES

Number of Test Calls	Percentage Overlap
2	8.73
10	21.16
100	36.13
1,000	97.24
100,000	78.95

C. How will Non-Blocking Alltoall Algorithms help FFT Kernels?

The P3DFFT kernel relies on all-to-all exchange in order to transpose the data between different pencil orientations. As already mentioned, all-to-all exchange can be a very expensive operation, stressing the interconnect bisection bandwidth. The baseline version uses blocking all-to-alls, and since in 3D FFT algorithm the volume of data exchanged is large (N^3 elements), the transpose becomes a major bottleneck for performance of the whole kernel. In many of our tests the communication phase takes 50% or more of the total execution time. Apart from using more advanced networks and better implementations of (blocking) MPI_Alltoall algorithms, a fruitful strategy appears to hide latency associated with the exchange by overlapping communication and computation. Since the MPI standard currently does not have nonblocking collectives we turn to NBC library as well as our own implementations of MPI_Alltoall using the offload feature and MPI-2 Puts. Our hope is that with a suitable software mechanism and a sophisticated network it will be possible to achieve a high degree of overlap and thus hide most of the communication latency.

IV. Designing Non-Blocking Algorithms with Collective Offload

In Section II-B, we discussed the point-to-point protocols used in MPI libraries for small and large messages. In this section, we discuss the point-to-point protocols that we use with our network offload designs to minimize the host processor intervention. We also describe our non-blocking Alltoall Personalized Exchange Algorithm that leverages the ConnectX-2 network offload feature.

```
MPI_Ialltoall(send_buf, send_count, send_type,
  recv_buf, recv_count, recv_type, comm, request)
{
  /*  register user buffers  */
  /*  Perform local-copy    */
  for(i =1 ; i < comm-size; i++) {
      peer = my_rank ^ i ;
       /* ibv_post_recv */
       /* create a send-task, en-queue it */
       /* create a wait-task , en-queue it*/
       /* update the corresponding request[] entries */
  }
  /* Post the task-list to the network interface */
  return;
}
```

Figure 1. Design of Network Offloaded Non-Blocking AlltoAll Operations

A. Point-to-point Communication Protocols for Overlap with Offload

In our proposed designs, we use a dedicated InfiniBand queue-pair for all offload communication. For small messages, since the copy costs are lower than the registration overheads, we use a protocol similar to the existing eager protocol. For small message MPI_Recv() operations, we post an *ibv_post_recv* operation on this dedicated queue-pair. On arrival of a new message on this queue-pair, the network interface places the data into one of the pre-registered buffers. When the application executes the corresponding MPI_Wait() operation, this data is copied into the user-buffers. For small message MPI_Send operations, we create a task-list entry that corresponds to the send request and enqueue it in a task-list. Once all such send() and wait() (if any) entries are created, the process posts the task-list to the network interface and returns. The network interface will progress each of these operations automatically. When the control enters the MPI library at a later point of time, it can check for completion of various operations and mark the corresponding requests as complete.

For larger messages, as indicated in Section II-B, the traditional rendezvous protocol requires the host processor's intervention, because the tag-matching cannot be offloaded to the network interface, currently. The degree of overlap achievable diminishes if the host processor is required to periodically enter the MPI library to perform various rendezvous protocol related operations. Owing to this reason, we use the following simpler protocol for larger messages. The processes first register the user buffers with the network interface and then post the *ibv_post_recv* operations (if any). Once all the tasks corresponding to the recv() operations have been posted, each process populates the task-list with the send() tasks and the necessary wait() tasks. Once the task-list has been created to mimic the collective algorithm, it can be posted to the network interface. Beyond this step, the host processor is free to return from the collective call and continue with application-level tasks. We need to perform an MPI_Wait() at a later point of time to ensure that the collective operation has completed and the buffers can be re-used again.

The other major challenge concerning the design of an efficient non-blocking interface collectives deals with unexpected messages and their buffering policies. Since process skews affect most applications [41], it is very likely that a particular process is yet to initiate a collective call, when most of the other processes are already performing the operation. Pre-allocating and registering memory buffers to address this problem is expensive and it is not obvious how many such buffers need to be allocated. In this paper, we rely on the *Receiver Not Ready*(RNR) feature offered by the InfiniBand hardware. The InfiniBand network interface internally performs a degree of flow control between communicating processes to determine whether a particular process has already posted its *ibv_post_recv* operation and if it is ready to receive a message. If a process is not ready yet, the network interface corresponding to the origin process continuously probes that of the peer process by sending small control packets until the receiver has posted its corresponding *ibv_post_recv* operation. In our proposed designs, we leverage this feature to design non-blocking

collective algorithms with minimal memory overheads.

B. Designing Non-Blocking Alltoall Personalized Exchange Algorithm

For a collective operation like MPI_Alltoall, it is necessary to control the number of concurrent exchanges done by each process. Given the overall volume of data being exchanged, if we were to perform all the send()/recv() operations at the same time, it will easily choke the network. For larger messages, most MPI libraries use the pair-wise exchange algorithm. If P processes are participating in the MPI_Alltoall operation, each process runs through P iterations performing send()/recv() operations with a distinct peer in each step.

In Figure IV, we describe our network offload implementation of the pair-wise exchange algorithm. Each process registers the user-buffers and posts all the necessary *ibv_post_recv* operations, while queuing up the send() and the wait() tasks in the task-list.

In our algorithm, we also define the term "communication-window" to indicate the number of send() operations that are performed in a given batch of the operation. The task-list is populated as a series of windows, with each window comprising of a fixed number of send() operations and the same number of wait() operations. The wait() operation ensures that the network interface waits until a relevant completion event is picked up from the InfiniBand completion queue. Hence, within each window, by inserting the same number of wait() operations as send() operations, we ensure that the traffic from one batch does not affect the traffic in the next batch. In Figure IV, we demonstrate how a task-list with a communication-window of size 1 is created. We discuss the implications of varying the size of the communication-window in our proposed network offloaded Alltoall algorithm in Section VI-C.

A non-blocking interface for collective operations should also allow application developers to initiate multiple non-blocking operations and should ideally guarantee the progress of these operations, while the host processor is busy with the compute tasks. In our designs, since the network interface progresses the collective operations internally, we also explored the possibility of initiating multiple non-blocking MPI_Ialltoall operations and its impact on performance and overlap. Our modified P3DFFT kernel leverages this feature to overlap two different MPI_Ialltoall operations with compute tasks, at the same time. We have also designed a simple micro-benchmark to evaluate the performance of our network offload designs by varying the number of concurrent non-blocking operations. More details about this benchmark can be found in Section VI-B.

V. Redesigning P3DFFT to Achieve Communication/Computation Overlap

The Cooley-Tukey algorithm for FFT used for 1D FFTs is very efficient computationally, O(NlogN). However its *butterfly* pattern of memory accesses make it a challenge to parallelize. To perform a 3D FFT, the 1D transform must be applied in each of the three dimensions. Two primary strategies are possible in parallelizing the 3D transform:

1) Direct approach: develop a parallel 1D FFT and communicate as necessary to carry out FFT on data that is distributed
2) Transpose approach: Rearrange data prior to each 1D FFT such the the data for the FFT is available locally and a serial 1D FFT can be used

While both methods require expensive communication operations, the commonly used transpose approach affords the opportunity to combine many smaller messages as a larger buffer in a single all-to-all exchange. In order to scale this approach to a large number of processors, a 2D domain decomposition (*pencils*) is used in P3DFFT as shown in Figure 2(b) and is carried out via the steps outlined in Figure 2(a).

If the original data array is distributed as pencils along the X dimension (*X-pencils*), i.e. all data in the X dimension is local, with the Y and Z dimensions split among processors in rows and columns of the 2D processor grid respectively, then in the first of the transposes the Y dimension is gathered to become local while the X dimension is split among the row processors. This involves an all-to-all exchange in rows, which is implemented in the baseline version as MPI_Alltoall over Cartesian sub-communicator *ROW*. The second transpose similarly brings together locally all data for the Z dimension and splits the Y dimension within columns, which involves an all-to-all over communicator *COL*.

Each of the two transposes exchanges a total of N^3/P elements. In applications of 3DFFT this implies a rather substantial message sizes, and therefore performance is bandwidth-bound. The row transpose typically takes much less time than the column one since the tasks in ROW communicator fall in the same node (or on a few adjacent nodes), so the network is not used or used little for this data exchange.

The *test_sine* kernel is a simple driver for the P3DFFT library (http://code.google.com/p/p3dfft). The baseline version simply calls a forward and backward transform repeated over a number of iterations as outlined in Figure 3.

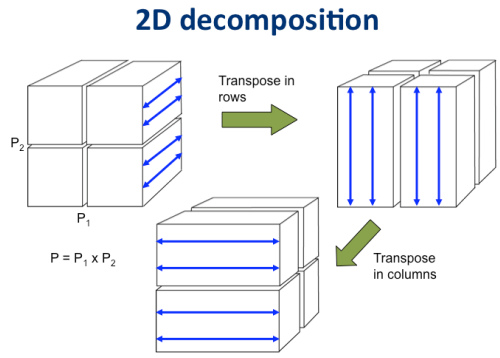
```
/* B initialized to 3D sine */
do for n iterations
    /* Forward transform */
    call p3dfft_ftran_r2c (B,A)
    /* normalize by size of grid */
    A = A/(nx · ny · nz)
    /* Backward transform */
    call p3dfft_btran_c2r (A,B)
end do
/* Compare resulting B with expected result */
```

Figure 3. Algorithm for the main loop in test_sine FFT kernel.

In many applications of 3D FFT it is necessary to transform several independent arrays (variables) at a time.

```
1D FFT in x
transpose x and y
1D FFT in y
transpose y an z
1D FFT in z
```

(a) Algorithm sequence for the forward transform in the default version of P3DFFT



(b) Transpose and 1D FFT operations for the 2D domain decomposition

Figure 2. Design of Forward Transform in 3DFFT and 2D Decomposition

When designing a version enhanced with overlap of communication and computation we chose to avoid splitting the transposes into smaller chunks. Instead we keep the bulk transposes in place by overlapping communication and computation stages for different variables. Thus the arrays A and B in the above loop now will get an extra dimension representing the number of variables, which is also passed to the forward and backward FFT routines. The forward FFT routine is restructured as shown in Figure 4. The backward routine is similarly restructured.

```
call trans_xy(beg(1,1),buf(1,1),1)
call init_exchange_col(buf(1,1),1)
prev = 1
curr = 2
do j=2,nv
    call trans_xy(beg(1,j),buf(1,curr),curr)
    call init_exchange_col(buf(1,curr),curr)
    call complete_exchange_col(end(1,j-1),prev)
    prev = 3 - prev
    curr = 3 - curr
enddo
call complete_exchange_col(end(1,nv),prev)
```

Figure 4. Algorithm for the forward transform in the redesigned multi-variable, pipelined, overlapped version

Here the loop index j runs over the variables that need to be transformed. Arrays beg and end are the original and final arrays for FFT transform. The first index in this example runs over the entire XYZ volume, while the second index is the variable count. An intermediate work array buf's first index likewise represents the data volume and the second index represents one of the two communication windows over which an exchange is occurring.

We start with trans_xy for variable 1 (V_1) which performs the first three stages of the 3D FFT algorithm, namely transform in X, row transpose, and transform in Y. Then we initiate (post) an all-to-all exchange in the COL communicator for this variable and let the nonblocking communication proceed. Assuming it does not interfere with the CPUs, the latter can work on XY transform of the second variable. This is the overlap of computation for XY transform of V_2 with network communication for V_1. After posting the exchange for V_2 we are then ready to complete the exchange for the first variable with a call to complete_exchange_col, which is essentially a Wait call followed by a transform of the resulting array (now in shape of Z-pencils) in Z dimension, thus completing the algorithm for V_1. Meanwhile the non-blocking exchange for variable V_2 is ongoing behind the scenes, thus achieving an overlap of FFT in Z for V_1 with Column transpose for V_2. The cycle continues for the rest of the variables, always keeping at least one exchange current in the background with the help of two communication windows (indices prev and curr). This design is only one of many possible (and work on this continues), it should be kept in mind that in practice the column transpose dominates the row transpose since the latter occurs among contiguously numbered tasks which are typically placed on cores of the same node. Assuming the size of ROW communicator is not too large, the row transpose occurs entirely within the nodes. This decision is consistent with the current trend of increasing cores per node on large system, therefore we chose not to overlap the row transpose with computation. However it is overlapped with the column transpose as part of trans_xy routine.

To summarize, we have developed a version of 3D FFT kernel which includes overlap of communication with computation without sacrificing the appealing large sizes in messages. This version allows us to study latency hiding with various non-blocking collective communication protocols. For this paper we consider three non-blocking replacements for the MPI_Alltoall in the Column transpose: i) NBC_iAlltoall, ii) MPI2 one-sided calls (put with fence synchronization), and iii) our network offloaded Alltoall algorithm.

VI. Experimental Results

We detail the results of our experimental evaluation in this section. All tests were run on a quiet system without any background traffic.

A. Experimental Setup

Each node of our test-bed has eight Intel Xeon cores running at 2.40 Ghz with 8192 KB L2 cache. The cores are organized as two sockets with four cores per socket. Each node also has 12 GB of memory and Gen2 PCI-Express bus. They are equipped with MT26428 QDR ConnectX HCAs with PCI-Ex interfaces. We used a 36-port Mellanox QDR switch to connect all the nodes. Each node is connected to the switch using one QDR link. The HCA as well as the switches use the latest firmware. The operating system used is Red Hat Enterprise Linux Server release 5.3 (Tikanga), with the 2.6.18-128.7.1.el5 kernel version. OFED version 1.5.1 is used on all machines, and the OpenSM version is 3.1.6.

B. Benchmark Suite

In this paper, we use modified versions of the OSU Micro-Benchmarks [42]. We use the *osu_alltoall* benchmark to measure the latency of the blocking MPI_Alltoall operations for various message sizes. We extend this benchmark to measure the overlap with non-blocking versions of Alltoall operations as shown in Figure 5. For various message sizes, we first measure the *base-latency* required to perform the non-blocking operation without inserting any computation. Then, we initiate the non-blocking Alltoall operation and run a compute task to span the duration of the base-latency. For the libNBC case, we need to rely on making NBC_Test calls from within the compute loop at periodic intervals to progress the communication in the background. However, with our network offload design, since the network progresses the collective operation in the background, it is not necessary to perform any Test() operations. Once the compute loop completes, we measure the time spent within the MPI_Wait() operation and we measure the total time required to perform the communication and computation operations in an overlapped manner. We then determine, the amount of time the host processor spent performing the compute tasks and the overlap %, as shown in Figure 5.

```
start_timer(base-latency);
Ialltoall()
Wait()
end_timer(base-latency);
/* calculate base-latency */
start_timer(overlap-latency)
Ialltoall()
while(timer <= base-latency) {
    /* compute and update timer */
    if(timer == time_slice) {
        /* If Test calls are required */
        start_timer(test-overhead)
        Test();
        end_timer(test-overhead)
        /* update overheads due to Test() */
    }
}
start_timer(wait-overhead)
Wait()
end_timer(wait-overhead)
end_timer(overlap_latency)
/* update overheads due to Wait() */
overheads = wait_overhead + test_overhead
compute_time = overlap_latency - overheads
overlap_ratio = compute_time / overlap_latency
```

Figure 5. Overlap Benchmark

C. Performance of Offload Alltoall with Different Communication Window Sizes

In Figures 6 (a) and (b), we study the performance characteristics of the network offloaded Alltoall operation by varying the number of concurrent send operations (communication-window), for 32 and 64 processes. As we can see from Figure 6 (a), the length of the offload window does not make a significant impact with smaller number of processes. However, as the number of participants in an Alltoall operation increases, we can see that the length of the offload window has a clear impact on the communication latency. The size of the window essentially controls the number of network of outstanding communication operations for each process. Having a window that is too small would lead to under-utilization of the network bandwidth and a higher communication latency. A window that is too large will lead to heavy network contention and also results in poor communication performance. As Figure 6 (b) depicts, we get the best performance with a window of size six. For all further experiments, we use this optimized window size of six.

D. Communication Performance Comparison: Latency

In Figures 7 (a) and (b), we compare the basic latency of our proposed network offloaded Alltoall algorithm with libNBC's NBC_Ialltoall operation and the existing MPI_Alltoall algorithm used in the MVAPICH2 library, for 32 and 64 processes. Since the MPI_Alltoall operation is a blocking call, with both libNBC and our proposed design, we initiate the Alltoall operation and immediately enter the wait() call, without attempting to overlap any compute tasks. As we can see, the offload scheme offers performance comparable to the default alltoall scheme. The libNBC scheme on the other hand, performs worse than both the default and the offload schemes. This could be attributed to a different communication schedule that is used by the libNBC library.

E. Computation/Communication Overlap

In Figure 8, we compare the amount of communication/computation overlap achievable through our proposed network offloaded Alltoall algorithm and libNBC's NBC_Ialltoall operation. As indicated in Section III-A2, we were unable to use libNBC's real-time thread option for our work. Hence, for all overlap comparison studies, we rely on progressing the communication through calling the NBC_Test() function. Since the amount of overlap achievable through the NBC_Test() option depends directly on the frequency with which we call this function, for every

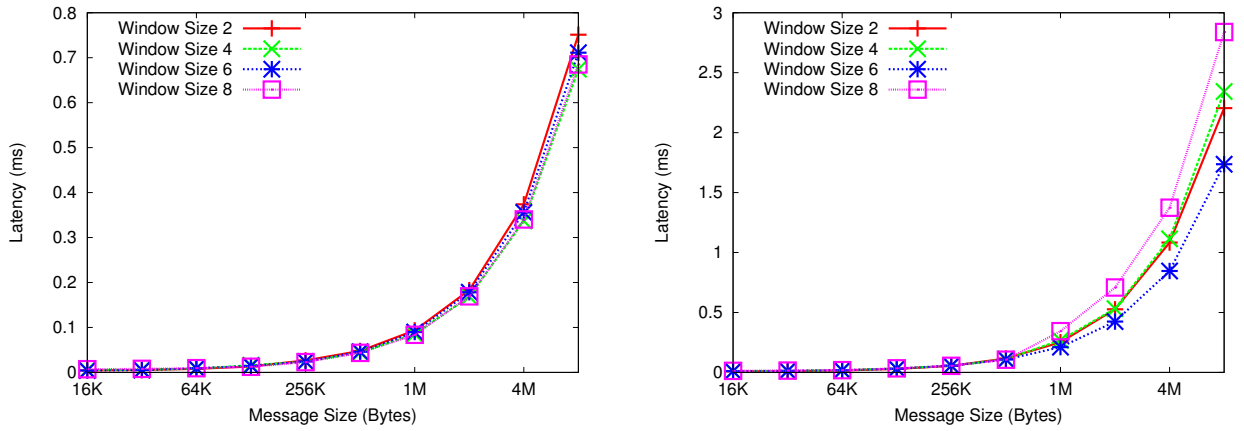


Figure 6. Impact of Varying Communication Window on Performance of Alltoall Algorithm for (a) 32 Processes and, (b) 64 Processes

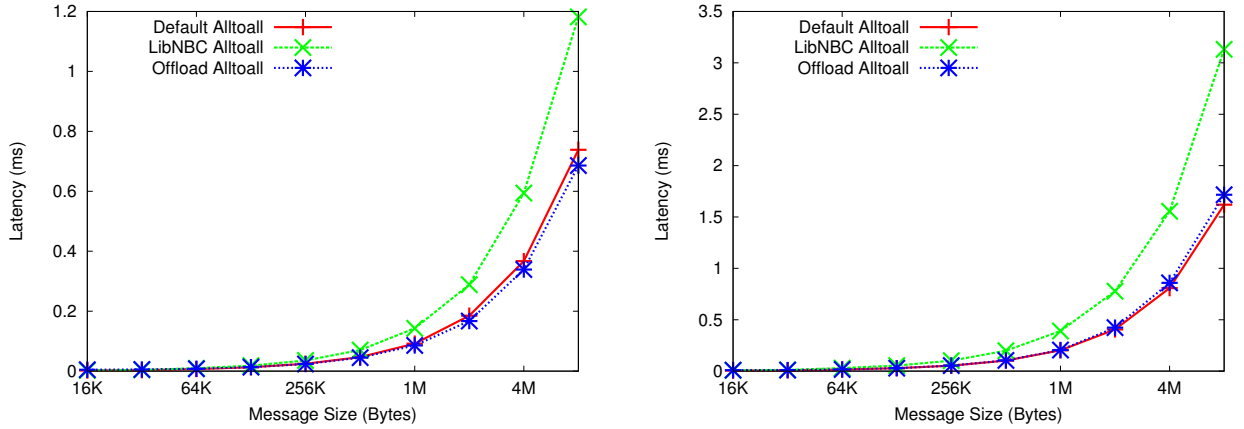


Figure 7. Latency Performance of Different Alltoall Implementations for (a) 32 Processes and, (b) 64 Processes

message size, we experimented by varying the number of NBC_Test calls from 2 to 10,000. For brevity, we choose to demonstrate the overlap (%) for 2, 1,000 and 10,000 NBC_Test calls for each message size. Figure 8 shows the percentage of overlap obtained with libNBC and our proposed offloaded versions of Alltoall for 32 processes. With libNBC, we observe that we get the best overlap when we invoke the NBC_Test function about 1000 times to progress the collective communication. If the number of Test() calls are too low, very little progress is made during the compute phase and most of the communication will be performed in the Wait() operation. On the other hand, if the number of Test() calls are too high, the overheads associated with calling the Test() function increases and limits the amount of time the host processor can spend on performing compute tasks. However, with the network offload approach, since the network interface deals with progressing the entire collective communication with no intervention from the host processor, we expect the entire communication time to be overlapped with compute tasks. In Figure 8, we can see that the overlap percentage is about 99% for all message sizes with 32 processes. However, we would like to note that for larger system sizes, since the size of the task-list grows, the performance of the offload engine in the current generation ConnectX-2 begins to drop and we observed about 90% communication/computation overlap with 64 processes. This could be a limitation of the current firmware of the network adapter, as it is still under active development. We are in touch with the vendor about this issue and an updated firmware might be available in the next few months. We will present the updated results with the new firmware in the final version of the paper.

F. Impact of Overlapping Computation and Communication on Latency

In Figures 9(a) and (b), we study the the impact of overlapping computation and communication on the aggregate latency of the alltoall benchmark for 32 and 64 processes, respectively. As we can see, with the offloaded scheme, overlapping computation and communication has little or no impact on the aggregate latency, because we are able to achieve high overlap. With libNBC, even though we are using the right number of Test() calls to get the best overlap possible, since the base latency of performing the Alltoall operation is poorer, we see larger completion times.

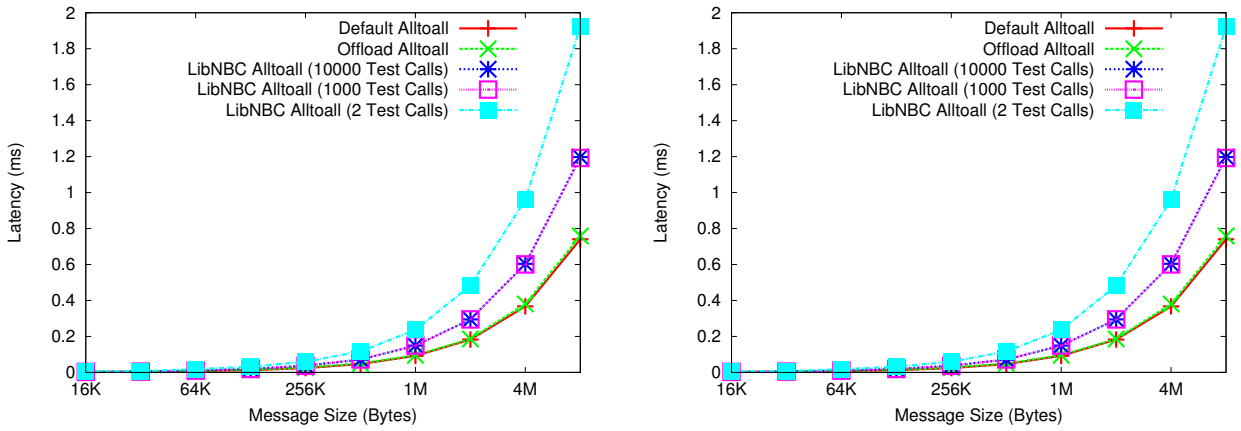


Figure 9. Impact of Overlapping Computation and Communication on Latency of Alltoall Operations for (a) 32 Processes and, (b) 64 Processes

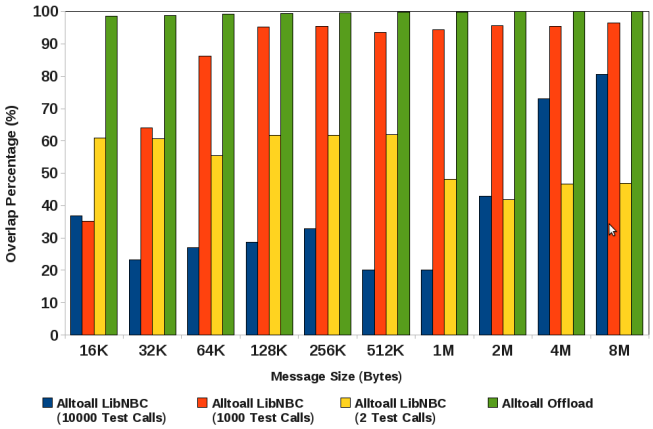


Figure 8. Percentage Overlap with Different Alltoall Implementations for 32 Processes

G. Performance with Multiple Concurrent Alltoall Operations

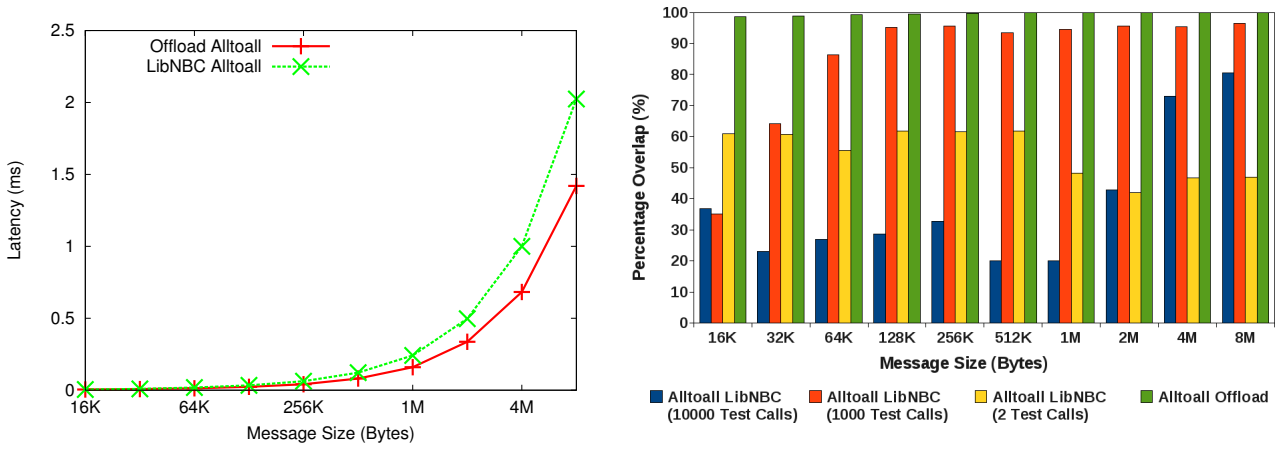
In Section IV-B, we indicated that a non-blocking interface for collectives should preferably allow application developers to initiate multiple concurrent collectives and overlap them with compute tasks. We discussed that our proposed network offload designs allow us to initiate more than one Alltoall operations and the network interface progresses each of these on its own. In Figures VI-G and 10(b), we compare the aggregate latency (communication + computation) and overlap of our proposed network offload designs with libNBC, with 2 concurrent Alltoall operations across 32 processes. We can observe that even in this case, our network offload approach performs better for all message sizes. Even with 2 concurrent Alltoall operations, we see better aggregate latency and better overlap ratios, when compared to the libNBC. We believe that this is a strong contribution to the community because having the flexibility to overlap multiple different collective operations at the same time with compute tasks, while still delivering good communication performance and better overlap can significantly impact the run-times of parallel applications.

H. P3DFFT Kernel Performance Comparison

To evaluate how our network offload Alltoall operation can be utilized to improve the performance of applications which require many 3D FFT operations we carried out a small study with the P3DFFT Sine kernel. As described in Section V we replaced the two most expensive Alltoall operations in the P3DFFT Sine kernel (namely the column wise transposes which occur in both the forward and backward transforms) with non-blocking alternatives, for various 3D FFT problem sizes. In Table II, we compare the application run-times with the base blocking version, one-sided version, the re-designed P3DFFT kernel with overlapped collective communication with libNBC and our proposed network offloaded MPI_Alltoall. The Offload Alltoall is consistently the best performing, outperforming both other non-blocking approaches and reducing overall runtime from 7.3%-23.5% when compared with blocking Alltoall as shown in Table II. We are not very sure at this point of time as to why the performance difference drops as the data-set size increases. We are looking into this problem. Also, LibNBC does not fair well in this comparison, that is because we have not incorporated NBC_test calls into the P3DFFT library. The computation work that is overlapped with the all to all is carried out by highly tuned FFT libraries. The computation loop must be restructured to make several individual calls to the FFT library in order to accommodate the NBC_test calls, a design we plan to evaluate for system in which offload can not be implemented.

VII. Conclusion

In this paper, we design a non-blocking offloaded collective for Alltoall Personalized Exchange (MPI_Alltoall) using the task-list offload by the ConnectX-2 network adapter. Simultaneously, we re-design the P3DFFT library and a sample application kernel to overlap the MPI_Alltoall operations with application level computation and demonstrate the benefits of using our novel non-blocking Alltoall algorithm. Our experimental evaluation shows that we are able to achieve



(a) Alltoall Latency

(b) Overlap Percentage

Figure 10. Performance with Multiple Concurrent Alltoall Operations between 32 Processes

Table II
P3DFFT SINE KERNEL RUN TIME

Data-Set Size	Offload (s)	libNBC (s)	One-Sided (RMA) (s)	Default (s)	Percentage Improvement (%)
512	1.56	2.08	2.01	2.04	23.5
600	2.94	3.78	3.66	3.49	15.7
712	5.03	6.54	6.29	5.68	11.44
800	7.12	9.20	8.80	7.68	7.29

near perfect overlap of computation and communication (99%) through the use of offload mechanism without any adverse impact on the latency of the MPI_Alltoall operation. We are also able to see an improvement of upto 23% in the overall run-time with the modified P3DFFT kernel when compared to the default blocking version.

The P3DFFT library is widely used by many scientific applications. We plan to contribute an efficient implementation of P3DFFT appropriate for modern InfiniBand clusters connected with ConnectX-2 network adapters so that these application users may leveraging the non-blocking Alltoall collective developed in this work. In the future, we plan to continue working in this direction. With the P3DFFT kernel, we plan to overlap the intra-node communication phases with computation operations in a manner similar to our proposed designs. We aim to develop a suite of collective operations that are offloaded efficiently using CORE-Direct. We also plan to evaluate the impact of non-blocking collectives on end applications, such as those that use P3DFFT, on larger clusters connected via ConnectX-2 as such systems become available.

REFERENCES

[1] MPI Forum, "MPI: A Message Passing Interface," in *Proc. of Supercomputing*, 1993.

[2] P. Mehrotra and J. Van Rosendale and H. Zima, "Solving Irregular Problems with High Performance Fortran," in *Third Working Conference on Massively Parallel Programming Models*, Nov. 1997, pp. 2–11.

[3] John Mellor-Crummey and Laksono Adhianto and William Scherer III, "A New Vision for Coarray Fortran," in *The Third Conference on Partitioned Global Address Space Programming Models*, Oct. 2009, p. 377.

[4] J. Vetter and F. Mueller, "Communication Characteristics of Large-scale Scientific Applications for Contemporary Cluster Architectures," vol. 63, no. 9, 2003, pp. 853–865.

[5] H. Karimabadi, H. X. Vu, D. Krauss-Varban, Y. Omelchenko, "Global Hybrid Simulations of the Earths Magnetosphere," vol. 359, 2006, p. 256.

[6] T. Hoefler, J. M. Squyres, W. Rehm, and A. Lumsdaine, " A Case for Non-blocking Collective Operations ," in *Frontiers of High Performance Computing and Networking . ISPA 2006 Workshops, Lecture Notes in Computer Science*, vol. 4331/2006, 2006, pp. 155–164.

[7] T. Hoefler and P. Gottschling and A. Lumsdaine, "Leveraging Non-blocking Collective Communication in High-performance Applications," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'08*. Association for Computing Machinery (ACM), Jun. 2008, pp. 113–115.

[8] T. Hoefler and A. Lumsdaine, "Optimizing non-blocking Collective Operations for InfiniBand," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop*, Apr. 2008.

[9] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?" in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[10] Mellanox Technologies, "ConnectX-2 Architecture," http://www.hpcwire.com/features/Mellanox-Rolls-Out-Next-Iteration-of-ConnectX-57046327.html.

[11] Voltaire, http://www.voltaire.com/.

[12] R. L. Graham and G. Shipman, "MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science*.

[13] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and P. D. K., "Designing Multi-leader-based Allgather Algorithms for Multi-core clusters," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009, pp. 1–8.

[14] R. Graham and S. Poole and P. Shamis and G. Bloch and N. Boch and H. Chapman and M. Kagan and A. Shahar and I. Rabinovitz and G. Shainer, "ConnectX2 InfiniBand Management Queues: New support for Network Ofoaded Collective Operations," in *CCGrid'10*, Melbourne, Australia, May 17-20 2010.

[15] R. Graham, S. Poole, P. Shamis, G. Bloch, N. Boch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Overlapping Computation and Communication: Barrier Algorithms and Connectx-2 CORE-Direct Capabilities ," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, Workshop on Communication Architectures for Clusters (CAC) '10*, 2010.

[16] Hari Subramoni, Krishna Kandalla, Sayantan Sur and Dhabaleswar K. Panda, "Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine," in *The 18th Annual Symposium on High Performance Interconnects, HotI 2010 (To appear)*, 2010.

[17] Top500, "Top500 Supercomputing systems," June 2010, http://www.top500.org/lists/2010/06.

[18] "Mellanox Technologies," http://www.mellanox.com.

[19] MVAPICH2, http://mvapich.cse.ohio-state.edu/.

[20] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, "Design and Implementation of MPICH2 over InfiniBand with RDMA Support," *Proceedings of the 18th IEEE International Parallel & Distributed Processing Symposium*, vol. 1, p. 16b, 2004.

[21] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 32–39.

[22] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, "Optimization of All-to-All Communication on the Blue Gene/L Supercomputer ," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008, pp. 320–329.

[23] Jehoshua Bruck and Ching-Tien Ho and Shlomo Kipnis and Derrick Weathersby, "Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems," in *Proc. of the 6th ACM Sym. on Par. Alg. and Arch.*, 1994, pp. 298–309.

[24] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," in *Int. J. of High Perf. Comp. App.*, 2005, pp. (19)1:49–66.

[25] H. Homann and J. Bec and R. Grauer, "DNS of Finite-Size Particles in Turbulent Flows," in *John von Neumann Institute for Computing NIC Symposium*, Feb. 2010, pp. 357–364.

[26] H. Homann, O. Kamps, R. Friedrich, and R. Grauer, "Bridging from eulerian to lagrangian statistics in 3d hydro- and magnetohydrodynamic turbulent flows," *New Journal of Physics*, vol. 11, no. 7, p. 073020, 2009.

[27] G. Grafke, "Numerical simulations of possible finite time singularities in the incompressible Euler equations: Comparison of numerical methods," vol. 237, 2008, pp. 1932–1936.

[28] T. Weidauer et al, "Shallow Moist Convection," *Proceedings of John von Neumann Institute for Computing NIC Symposium*, pp. 373–380, Feb 2010.

[29] S. Laizet, E. Lamballais and J.C. Vassilicos, "A numerical strategy to combine high-order schemes, complex geometry and parallel computing for high resolution DNS of fractal generated turbulence," in *Computers & Fluids*, vol. 39, no. 3, 2010, pp. 471 – 484.

[30] N. Peters and L. Wang and J.P. Mellado and J. H. Gobbert and M. Gauding and Ph. Schafer and M. Gampert, "Geometrical Properties of Small Scale Turbulence," in *John von Neumann Institute for Computing NIC Symposium*, Feb. 2010, pp. 365–371.

[31] J. Schumacher and M. Putz, "Turbulence in Laterally Extended Systems," in *Parallel Computing: Architectures, Algorithms and Applications*, vol. 38, 2007.

[32] J. Bodart, "Large Scale Simulation of Turbulence Using a Hybrid Spectral/Finite Difference Solver," in *in Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, 2009, pp. 473–482.

[33] Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) library, San Diego Supercomputer Center (SDSC), http://code.google.com/p/p3dft.

[34] D.A. Donis, P.K. Yeung and D. Pekurovsky, "Turbulence Simulations on O(10^4) Processors," in *TeraGrid 2008*.

[35] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/.

[36] Open-MPI, http://www.open-mpi.org/.

[37] R. Nishtala and K. A. Yelick, "Optimizing Collective Communication on Multicores," in *HotPar*, 2008.

[38] N. T. Karonis and B. R. de Supinski and I. Foster and W. Gropp and E. Lusk and J. Bresnahan, "Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, May. 2000, p. 377.

[39] A. R. Mamidala, R. Kumar, D. De and D. K.Panda, "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics," in *8th IEEE International Symposium on Cluster Computing and the Grid, 2008, Lyon*, May 2008, pp. 130–137.

[40] R. Thakur and W. Gropp, "Improving the Performance of Collective Operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, 2003, Vol. 2840/2003*, 2003, pp. 257–267.

[41] P. Patarasuk and X. Yuan, "Efficient MPI Bcast across Different Process Arrival Patterns," in *IPDPS*, 2008, pp. 1–11.

[42] The Ohio State University, "OSU MPI Benchmarks," http://mvapich.cse.ohio-state.edu.