LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-502793

Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers

K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, D. K. Panda

October 4, 2011

IEEE IPDPS 2012
Shanghai, China
May 21, 2012 through May 25, 2012

Disclaimer

Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers*

K. Kandalla[1], U. Yang[2], J.Keasler[2], T. Kolev[2], A. Moody[2], H. Subramoni[1], K. Tomko[3], J. Vienne[1], and D. K. Panda[1]

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

[2] *Lawrence Livermore National Laboratory*
Livermore, California
{yang11, keasler1, kolev1, moody20}@llnl.gov

[3] *Ohio Supercomputer Center*
Columbus, Ohio
{ktomko}@osc.edu

***Abstract*—Scientists across a wide range of domains increasingly rely on computer simulation for their investigations. Such simulations often spend a majority of their run-times solving large systems of linear equations that require vast amounts of computational power and memory. It is hence critical to design solvers in a highly efficient and scalable manner. Hypre is a high performance, scalable software library that offers several optimized linear solver routines and pre-conditioners. In this paper, we study the characteristics of Hypre's Preconditioned Conjugate Gradient (PCG) solver algorithm. The PCG routine is known to spend a majority of its communication time in the MPI_Allreduce operation to compute a global summation during the innerproduct operation. The MPI_Allreduce is a blocking operation whose latency is often a limiting factor to the overall efficiency of the PCG solver routine, and correspondingly the performance of simulations that rely on this solver. Hence, hiding the latency of the MPI_Allreduce operation is critical towards scaling the PCG solver routine and improving the performance of many simulations.**

The upcoming revision of MPI, MPI-3, will provide support for non-blocking collective communication to enable latency-hiding. The latest InfiniBand adapter from Mellanox, ConnectX-2, enables offloading of generalized lists of communication operations to the network interface. Such an interface can be leveraged to design non-blocking collective operations. In this paper, we design fully functional, scalable algorithms for the MPI_Iallreduce operation, based on the network offload technology. To the best of our knowledge, this is the first such design to be presented in the literature. Our designs scale beyond 512 processes and we achieve near perfect communication/computation overlap. We also re-design the PCG solver routine to leverage our proposed MPI_Iallreduce operation to hide the latency of the global reduction operations. We observe up to 21% improvements in the run-times of the PCG routine, when compared to the default PCG implementation in Hypre. We also note that about 16% of the overall benefits are due to overlapping the Allreduce operations.

***Keywords*-MPI-3 non-blocking collectives, Conjugate Gradient Solvers, Collective Offload, InfiniBand**

*This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749, #DE-FC02-06ER25755 and contract #DE-AC02-06CH11357; National Science Foundation grants #CCF-0621484, #CCF-0702675, #CCF-0833169, #CCF-0916302 and #OCI-0926691; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Obsidian, Advanced Clustering, Appro, QLogic, and Sun Microsystems.This work was also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

I. Introduction

The fastest supercomputing systems currently offer sustained peta-flop performance and are allowing scientists to scale their parallel applications to tens of thousands of processors. The Message Passing Interface (MPI) [1] has been a popular programming model for High Performance Computing applications for the last couple of decades. MPI defines a set of collective operations that are used to communicate data among a group of participating processes. Owing to their ease of use and portability, they are commonly used across various applications. The current MPI Standard 2.2, defines the collective operations to be blocking, i.e. the application has to wait until the collective call completes. This limits the overall performance and scalability of various scientific parallel applications. In addition, blocking collective operations are also prone to system noise which directly impacts the performance of parallel applications [2], [3]. This has spurred interest in the design of non-blocking collective communication operations in MPI and the upcoming version of MPI, MPI-3, defines non-blocking collective communication operations.

InfiniBand is a very popular switched interconnect standard being used by almost 41% of the Top500 Supercomputing systems [4]. Since InfiniBand is so widely used, efficient support of non-blocking collectives in MPI implementations on InfiniBand is critical. Mellanox recently introduced network offload features in their ConnectX-2 [5] adapter. Using this feature, generic lists of communication tasks can be offloaded to the network interface [6]. Such an interface eliminates the need for the host processor to progress communication and provides a low-level mechanism which can be leveraged to design non-blocking collective communication algorithms. However, in order to leverage the full benefits of this low-level mechanism, MPI libraries must be designed in a highly efficient manner.

II. Motivation

Application scientists increasingly rely on large scale simulation to perform their scientific explorations. This enables study of scenarios that are infeasible or impractical to study by experiment. Several such applications are known to rely heavily on popular solver routines, such as the Preconditioned Conjugate Gradient (PCG), to solve large

systems of sparse linear equations. The efficiency of the solver routine is extremely critical and strongly affects the overall run-times of scientific simulations. In this paper, we use *Hypre* [7], a high performance, scalable, open source library that implements several preconditioners and solver algorithms, including the PCG. When the PCG solver routine is used with the diagonal scaling preconditioner, it is known to spend a considerable fraction of its communication time in MPI_Allreduce, during the inner-product operation, which is a significant performance bottleneck. Hence, hiding the latency of the MPI_Allreduce operation is critical towards improving the efficiency of the PCG Solver routine. In this paper, we re-design the PCG Solver routine, to leverage our proposed MPI_Iallreduce operation to hide the latency of the global reductions by overlapping it with the compute phases of PCG. Our studies show that we can improve the run-times of PCG by up to 21% when compared to the default PCG implementation in Hypre, about 16% of the overall benefits are due to overlapping the Allreduce operations.

A high performance implementation of a non-blocking interface for collective operations would ideally be expected to deliver near-perfect communication/computation overlap. While the benefits of non-blocking collectives are obvious at a high-level, the real benefits offered by intelligent MPI designs are likely to be the key driver for acceptance of this interface by the application community. For example, to ensure high overlap capabilities, it is necessary to minimize the role of the host-processors in progressing the collective operations. Simplistic designs of non-blocking collectives requiring progressing the MPI library explicitly by CPU intervention, e.g. calling MPI_Test [8], offsets much of the benefit of non-blocking communication. Similarly, if threads within the library are used for progression, the application performance can be hurt by interrupt processing, thread scheduling and other such factors, [9]. It is also critical for a non-blocking collective interface to ensure performance portability, i.e, the benefits of using non-blocking collectives should not be tightly coupled to system architecture and network speeds. In this context, real benefits of non-blocking collectives can only be achieved with corresponding network support [10], [11]. In order to extract the maximum benefit from non-blocking collectives, application developers may have to re-engineer their codes for communication/computation overlap. *Such a co-design between the applications and the MPI libraries can potentially lead to significant improvements in application run-times*.

In this paper, we integrate our designs into the MVAPICH2 [12] software stack, which is a popular MPI implementation for InfiniBand, iWARP and RoCE technologies. MVAPICH2 is currently used by more than 1,700 organizations in 63 countries worldwide. We list the important contributions of this paper below:

1) We propose fully functional designs for the MPI_Iallreduce operation, which leverages the network offload features offered by the ConnectX-2 network interface.
2) We study the various factors that could potentially affect the overlap capabilities of our network offload-based MPI_Iallreduce operation.
3) Linear solvers typically spend a significant amount of their MPI time in the global Allreduce operations. In this work, we re-design the Preconditioned Conjugate Gradient Solver in Hypre, to leverage our MPI_Iallreduce operation.
4) We show that our MPI_Iallreduce designs reduce the impact of system noise on the PCG Solver.

III. Background

In this section we give the necessary background information for our work.

A. InfiniBand and ConnectX-2 Network Interface

Current generation InfiniBand QDR network cards and switches can deliver 36 Gbps end-to-end bandwidth and about 1.0 to 1.5 μs latency. The ConnectX-2 [5] network interface is the latest adapter from Mellanox. Along with all of the standard InfiniBand features, it offers a new network offloading feature called CORE-Direct [13]. Using this feature, arbitrary lists of send, receive and wait operations can be created. These lists can then be posted to a work-request queue to be further processed by the network card. The network adapter independently executes it and eliminates the need for the host processor to progress the communication tasks. Using such task-lists, non-blocking collective operations may be designed by upper-level libraries.

B. Offloading compute operations with ConnectX-2

Unlike collectives such as MPI_Bcast and MPI_Alltoall, the MPI_Allreduce operation also performs a few basic math operations, such as MPI_MAX, MPI_SUM, etc. In order to design MPI_Iallreduce in a truly non-blocking manner, it is desirable to offload the compute phases of the Allreduce operation to the network interface. Such a design could lead to higher overlap capabilities, when compared to designs which may require the host processors to intervene and perform the math operations. Apart from supporting communication offload, the ConnectX-2 interface also allows MPI libraries to create and post task-lists comprising of calc-requests. The calc operation needs to be specified as a part of a send work request element. Upon execution, the result of the calc operation for the given set of operands will be sent to the peer process. However, if a process that is posting such an operation also requires the result, it is necessary to do a network loop-back operation to retrieve the data from the network interface. The current generation CX-2 interface has the limitation that it can only support binary calc operations of scalar values. Solver routines commonly do global Allreduce operations on just one double. So, it is possible for MPI libraries to leverage the CX-2 feature for

such applications. However, to offload reductions on vector data, we may require more advanced hardware support.

C. Allreduce Algorithms in MVAPICH2

State-of-the-art open-source MPI implementations, such as MPICH2 [14], Open-MPI [15] and MVAPICH2 [12] use optimized algorithms to improve the latency of blocking collective operations. MVAPICH2 implements multi-core aware, shared-memory based algorithms for blocking collective operations. The processes that are within a compute node are grouped within a "shared memory communicator". One process per node is designated as a leader and participates in a "leader communicator" which contains leaders from all nodes. We first do a shared-memory based reduction within each compute node to accumulate the data at the leader process. This is followed by an inter-leader Allreduce operation, which may either be implemented through the Recursive-Doubling algorithm, or a tree-based Reduce-Bcast algorithm. Finally, the leader processes do a shared-memory broadcast to complete the Allreduce operation.

D. Impact of System Noise

Several researchers have demonstrated the impact of system noise on the performance of parallel applications [2], [3]. The impact of noise is higher at larger scales, because the delays tend to get propagated across various tasks in the job. Hoefler et al, quantified the impact of noise on various host-based collective operations in [2] and concluded that MPI_Allreduce based on the recursive-doubling algorithm is very sensitive to system noise. However, with network based implementations, the network can independently execute the schedules, with little intervention from the host processors. Such designs have the potential to reduce the impact of system noise on the performance of applications.

E. Hypre

Hypre is an open-source, high performance and scalable package of parallel linear solvers and preconditioners. Hypre is designed to leverage the notion of conceptual interfaces, which expose the various solver routines to users in a modular fashion[16]. Such a design significantly eases the coding burden for application developers and may also be used to provide additional application information to the solver routines. The solvers in Hypre are robust, numerically stable and scalable [17]. Its object model is more generic and flexible when compared to many state-of-the-art solver packages [7] and it may also be used as a framework for algorithm development. In this paper, we focus specifically on the Preconditioned Conjugate Gradient solver routine, which uses the diagonal scaling preconditioner.

IV. Designing Offload-based Allreduce Algorithms

We described the communication protocols we use for small and large messages with the CORE-Direct interface in [10], [11]. We pre-post buffers to minimize the latency of small messages and we rely on InfiniBand's *Receiver-Not-Ready* (RNR) feature for large messages. In this section, we discuss our proposed designs for the network-offload based algorithms for the Allreduce operation.

As discussed in Section III-B, if a process posts a calc-request requires the result of the operation, it is necessary to do a network-loop-back operation. This may affect the latency of the network offload based MPI_Iallreduce operation, because the loop-back operation is expensive. Also, a NIC-level wait-task gets triggered when there is a completion event on the Completion Queue(CQ), it is associated with. Since the network interface does not offer hardware-based tag-matching, there is no way for the NIC to know the source of the data packet. This could potentially lead to race conditions and may even lead to incorrect results during the MPI_Iallreduce operation. To address this problem, it is necessary to use distinct InfiniBand Queue Pair's (QP) and CQ for each pair of processes involved in the Allreduce operation. We limit the size of these CQ's to control the memory foot-print of our designs. We also selectively poll only the specific CQ's, where we are expecting data to arrive, instead of exhaustively polling on all the available CQ's. Such an approach allows us to control the polling overheads.

We now propose our network-offload based designs for both the recursive-doubling and the tree-based reduce-bcast algorithm and explore design-level choices to minimize the performance impact of the network loop-back operation.

A. Network Offload based Recursive-Doubling Algorithm

We describe the network-offload based recursive-doubling algorithm in Figures 1(a) and (b). As shown in Figure 1(a), the recursive-doubling algorithm across N processes requires $(logN)$ iterations. At the start of the algorithm, each process copies its input buffer into an *accumulator-buffer*. In each iteration, a process chooses a specific peer, sends the data in the accumulator-buffer, receives the peer's data, performs the math operations on the data in the accumulator and the data it received from the peer. The result of the calc operation is written back into the accumulator at the end of each step. The recursive-doubling algorithm requires that every process has the updated data at the end of each iteration. Hence, a network offload based implementation for the recursive-doubling algorithm will require each process to do a loop-back operation at the end of every iteration of the algorithm. It is not possible to hide this overhead, because the next step of the algorithm cannot be started before the loop-back operation is completed. Since there are (logN) steps in a recursive-doubling algorithm, each process executes (logN) loop-back operations, for a given MPI_Iallreduce operation.

The other challenge is to ensure data dependency across multiple iterations of the algorithm as the task-lists are created during the MPI_Iallreduce operation, even before the collective has started. In [11], we proposed addressing a

similar problem with the network-offload-based MPI_Ibcast operation. Since each process uses a distinct queue-pair for all of its peers, we pre-post registered buffers on all these queue-pairs. We also mirror the list of pre-posted buffers inside the MVAPICH2 library to monitor the flow of the algorithm and handle the data consistency issues. In Figure 1(b), we demonstrate our network-offload-based recursive-doubling algorithm for Rank0, with a communicator comprising of 4 processes. Rank0 maintains 4 lists, $L00$, $L01$, $L02$ and $L03$, which always mirror the buffers that are currently pre-posted on each of the QP's. We name the buffers on list $L0i$, as $bi0$, $bi1$ and so on. In the first iteration, Rank0 copies the data in the input buffer of the MPI_Iallreduce operation into buffer $b00$ on the list $L00$ and enqueues a wait-task for a recv completion from Rank1 on the specific CQ it has for Rank1. The data arriving from Rank1 will be guaranteed to be in the buffer $b10$, on the list $L01$. Next, Rank0 performs the network-loop-back operation by enqueueing a calc-send task to itself. The network interface will place the result of this operation back at the buffer $b00$, since this is the head of the list $L00$. During the second iteration, Rank0 enqueues a wait-task for the expected recv-completion from Rank2. The data arriving from Rank2, will be available in the buffer $b20$, on the list $L02$, once the network card generates the corresponding recv-completion Finally, Rank0 enqueues a calc-send task to itself, by supplying the data in $b00$ and $b20$ as the operands for the calc operation. The result of this calc operation will be available in the buffer $b01$. Before exiting the MPI_Iallreduce operation, Rank0 also de-queues these buffers, in preparation for the next MPI_Iallreduce operation. When the application executes the MPI_Wait operation on the request corresponding to the MPI_Iallreduce operation, if all the communication tasks have been completed, the process can copy the data out of buffer $b01$ into the "recvbuf" of the MPI_Iallreduce operation.

B. Network Offload based Reduce-Bcast Algorithm

In Figures 2(a), (b) and (c), we describe the steps involved in our network-offload based Reduce-Bcast algorithm. Suppose we consider an intermediate process Pi, of the generic k-nomial reduce tree. This process needs to handle receiving the data from k children processes, creating $k-1$ calc operations to itself (through network loop-back) and 1 calc-send operation to its parent. As shown in Figure 2(a), for a binomial tree, an intermediate process is required to do only one calc-self-send operation, but the communication tree will be tall and thin. However, as we increase the degree of the tree, the number of calc-self-send operations also increase, while the communication tree becomes shorter and denser. For example, with 16 processes, the reduction operation with degree=4, takes 8 steps to finish, whereas with degree=2 and 3, we only require 7 steps. We also observe that any tree-based algorithm leads to a fairly imbalanced communication graph. Such an imbalance could help in overlapping the overhead of the loop-back operation at the intermediate processes, which was not possible with the recursive-doubling algorithm. For example, in Figure 2(a), process $P5$, waits for data to arrive from process 6. On receipt of this message, $P5$ executes the calc-send operation to itself. At the same time process $P7$ also receives data from its child process and executes a calc-send task to $P5$. If processes $P5$ and $P7$ are sufficiently synchronized, it is possible that by the time the data from $P7$ arrives at $P5$, $P5$ is done with the calc-send to itself, hence potentially hiding the overhead of loop-back operation. However, if process $P5$ is delayed in starting the MPI_Iallreduce operation, it may not be possible to hide the cost of the loop-back operation. Once the reduce phase is done, the root of the reduce-tree broadcasts the final result across all the processes. This phase is also offloaded to the network interface, hence, eliminating the need for any intervention from the host processors. Every process chains the task-lists corresponding to the reduce and the broadcast steps, during the MPI_Iallreduce operation itself, which allows us to completely offload the reduce-bcast algorithm.

C. MPI_Iallreduce Design Choices

A simple approach to designing a network-offload based MPI_Iallreduce operation is to ignore the node-level topology and consider the communicator as a flat structure. If there are N processes in the communicator, each process communicates with $(logN)$ peers. However, such a design may suffer from a relatively high communication latency. We refer to such a design as the "Flat" scheme. Another option in designing a network-offload based MPI_Iallreduce operation could be to use the shared-memory approach for the intra-node communication and use the network-offload channels for the inter-node transfers. We refer to such a design as the "Two-Level" scheme. We use the shared-memory-based reduce operation to implement the first intra-node reduction step, and the shared-memory-based broadcast operation to implement the final intra-node broadcast phase. We may choose to use either the network-offload-based recursive-doubling algorithm or the reduce-bcast algorithm for the inter-leader step of the MPI_Iallreduce operation. Since the communication latency of the shared-memory channel is very small, we believe that such a design allows for lower communication latency. However, a limitation of such a design is that it introduces skew across processes, since the leader processes on each node do more work than the rest of the processes on the same node, during the MPI_Iallreduce step. During the MPI_Wait step, the non-leaders are required to synchronize with their leader process, to get the final result of the Allreduce. Such a synchronization may not be very appealing, because the goal of a non-blocking collective interface is to hide the synchronization overheads. Also, since the final intra-node

```
mask=0x1;
while(mask < pof2) {
   dst = rank ^ mask
   send-task to (dst)
   wait-for-recv from (dst)
   calc-send-task (rank)   (Loop-Back)
   wait-for-recv from (rank)
   mask <<= 1
}
```

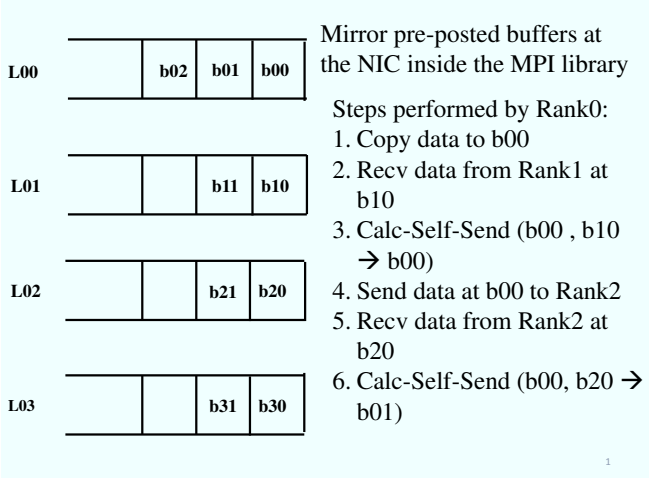

Figure 1. Recursive Doubling Algorithms for MPI_Iallreduce: (a) Offload-based Recursive-Doubling pseudo code (b) Offload-based Recursive-Doubling design

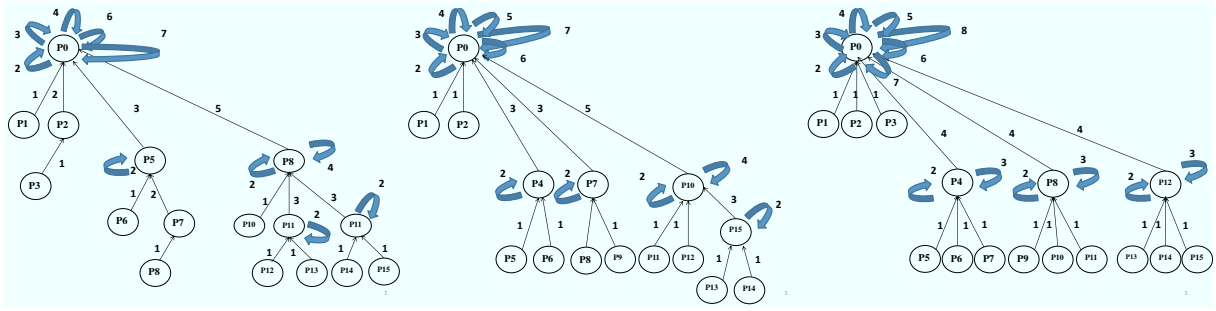

Figure 2. Network Offload-based Reduce-Bcast Algorithms for MPI_Iallreduce: (a) Degree=2, (b) Degree=3, (c) Degree=4

step requires copying the data from the leader processes to the rest of the non-leader processes, it may also lead to evicting the application's data from the caches. However, the "Flat" scheme may alleviate these problems, because it does not require any process synchronization during the MPI_Wait operation and each process is only required to the copy the result of the Allreduce step back into the user-buffer, leading to fewer conflicts in the caches.

V. Designing the Preconditioned CG for Overlap

In this section, we first describe the basic PCG algorithm, which is used in the Hypre software library. We then discuss our implementation of a common variant of the PCG algorithm, within Hypre. Finally, we propose our Overlap-PCG algorithm, which uses non-blocking inner-product operations to achieve communication/computation overlap.

A. Basic PCGSolve Algorithm

The Preconditioned Conjugate Gradient Solver routine is commonly used to solve systems of linear equations of the form $Ax = B$, if A is symmetric and positive definite. The CG method is often used in combination with a preconditioning step, which generates a different matrix C, which is an approximation of A, so that $Cy = Z$ is easier to solve, when compared to $Ax = B$. In Figure 3, we include the pseudo-code for the PCG Solver routine in the Hypre library. We observe that we do three calls to the inner-product function in each iteration of the PCG Solve routine, to update the $sdotp$, $gamma$ and the i_prod variables. The inner-product operation relies on the MPI_Allreduce operation to calculate the global summation value. In each iteration of the PCGSolve routine, we also call the Matvec function, which implements the boundary exchange and the local $hypre\_CSRMatrixMatvec$ operation. The boundary exchange phase uses MPI_Isend, MPI_Irecv and MPI_Waitall operations and is overlapped with the $hypre\_CSRMatrixMatvec$ function. We also observe that the loop in its current form has a strict data dependency and is not very amenable for overlapping the inner-product functions. For example, if we consider the $sdotp$ variable, it gets updated during an inner-product operation and its value is used in the very next step. For the rest of this paper, we refer to this version of the PCG, as "PCG-Algorithm1".

B. PCGSolve Algorithm Variant

In Figure 6, we discuss a variant of the PCG algorithm, which was proposed by authors in [18]. This version, is inherently very similar to the basic PCG Routine described in Figure 3, and has the same numerical stability. However. we also observe that this variant offers the flexibility to use the result of the inner-product functions at a later point in time. For example, the result of the sdotp inner-product is not needed until after the X_Axpy routine is done. We refer to this version of the PCG routine as "PCG-Algorithm2". This

```
x = initial guess, p = 0, beta = 0
r = b - Ax
Solve C * p = r
gamma = inner-prod(r, p)
while( not converged )
    Matvec (A, p, s)                    /* s = A*p */
    sdotp = inner-prod(s, p)
    alpha = gamma/sdotp
    gamma_old = gamma
    x = x + alpha*p                     /* X_Axpy*/
    r = r - alpha*s                     /* X_Axpy*/
    Solve C * s = r                    /*DiagScale*/
    gamma = inner-prod(r, s)
    i_prod = inner-prod(r, r)
    if(i_prod / bi_prod)
        /* Convergence Test */
        if(converged)
            break
    beta = gamma/gamma_old
    p = s + beta*p                      /* P_Axpy */
```

Figure 3. PCG-Algorithm1: Basic Preconditioned Conjugate Gradient Solver Algorithm in Hypre

variant of the PCG also requires a slightly modified version of the preconditioner routine. We describe the differences in the two preconditioner routines in Figures 4 and 5. In Hypre, PCG-Algorithm1, uses the DiagScale preconditioner, described in Figure 4, which uses indirect addressing to access the elements of the A_data array. For PCG-Algorithm2, we create the L and the L^{-T} arrays at the start of the solver routine, and we use these arrays, in the DiagInvScale preconditioner, described in Figure 5. In our case, the matrix L, is the square-root of the diagonal elements of matrix A, and therefore $L = L^T$. We would like to note that, the DiagInvScale routine reads data sequentially from the A_data array, which could lead to better cache behavior. We also observe that the DiagInvScale preconditioner involves floating-point multiplication operations, whereas the DiagScale routine requires floating-point division, which is computationally more expensive. We also observe that the gamma-innerproduct step in PCG-Algorithm2 reads data from the same vector t. In PCG-Algorithm1, we use vectors r and s to compute gamma. This may lead to fewer bus transactions and better cache behavior. Due to both these factors, we expect PCG-Algorithm2 to perform better than PCG-Algorithm1, even though both of them use blocking Allreduce operations.

```
hypre_ParVector *y = (hypre_ParVector *) Hy;
hypre_ParVector *x = (hypre_ParVector *) Hx;
double *A_data = hypre_CSRMatrixData(hypre_ParCSRMatrixDiag(A));
int *A_i = hypre_CSRMatrixI(hypre_ParCSRMatrixDiag(A));
for (i=0; i ¡ local_size; i++)
    x_data[i] = y_data[i]/A_data[A_i[i]];
```

Figure 4. DiagScale Preconditioner

```
hypre_ParVector *y = (hypre_ParVector *) Hy;
hypre_ParVector *x = (hypre_ParVector *) Hx;
double *A_data = L^-1
for (i=0; i ¡ local_size; i++)
    x_data[i] = y_data[i]*A_data[i];
```

Figure 5. DiagInvScale Preconditioner

```
X = initial guess, P = 0, beta = 0
P_prev = 0, w = 0, v = 0, t = 0
r = b - Ax
C = L.L^T
t = L^-1*r                          /* DiagInvScale */
gamma = inner-prod(t ,t)
while( not converged )
    w =L^-T*t                       /* DiagInvScale */
    p = w + beta*p_prev             /* P_Axpy */
    s = A * p                       /* Matvec */
    sdotp = inner-prod(s, p)
    x = x + alpha*p_prev            /* X_Axpy*/
    alpha = gamma/sdotp
    r = r - alpha*s                 /* R_Axpy*/
    i_prod = inner-prod(r, r)
    t = L^-1*r                      /* DiagInvScale */
    gamma_old = gamma
    gamma = inner-prod (t,t)
    beta = gamma/gamma_old
    if(i_prod / bi_prod)
        /* Convergence Test */
        if(converged)
            break;
```

Figure 6. PCG-Algorithm2: Modified Preconditioned Conjugate Gradient Solver Algorithm

C. PCG Algorithm with Overlap

We leverage PCG-Algorithm2 and re-design it to overlap the inner-product operation with independent compute tasks, as described in Figure 7. We design a non-blocking interface for the inner-product function, $init - inner - product$, which can be used to initiate the inner-product operation and return immediately. We can perform some of the other independent compute tasks of the solver routine and wait for the completion of the inner-product by using the $wait - inner - prod$ routine. The init-inner-product function initiates the non-blocking Allreduce operation through our network-offload-based MPI_Iallreduce operation. The $wait - inner - product$ function calls the MPI_Wait operation to wait on the corresponding non-blocking Allreduce operation. In our proposed variant, we have overlapped each of the three inner-products with either the DiagInvScale routine or the X_Axpy operation. For the rest of the paper, we refer to this algorithm as "PCG-Overlap".

VI. Experimental Evaluation

A. Experimental Setup

Each of our compute nodes have eight Intel Xeon cores running at 2.53 Ghz with 12 MB L3 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-2 HCAs with PCI-Ex interfaces. We used a 171-port Mellanox QDR switch, with 11 leafs, each having 16 ports. 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 5.4 (Tikanga), with the 2.6.18-164.el5 kernel version. OFED version 1.5.1 is used on all machines, and the OpenSM version is 3.3.7.

```
X = initial guess, P = 0, beta = 0
P_prev = 0, w = 0, v = 0, t = 0
r = b - Ax
C = L.L
t = L^-1*r                          /* DiagInvScale */
gamma = inner-prod(t ,t)
while( not converged )
    w =L^-1*t                       /* DiagInvScale */
    gamma = wait-inner-prod(t, t)   /*finish gamma inner-product*/
    beta = gamma/gamma_old
    p = w + beta*p_prev             /* P_Axpy */
    s = A * p                       /* Matvec */
    init-inner-prod(s, p)           /*start sdotp inner-product */
    x = x + alpha*p_prev            /* X_Axpy*/
    sdotp = wait-inner-prod(s, p)   /*finish sdotp inner-product*/
    alpha = gamma/sdotp
    r = r - alpha*s                 /* R_Axpy*/
    init-inner-prod(r, r)           /*start i_prod inner-product */
    t = L^-1*r                      /* DiagInvScale */
    i_prod = wait-inner-prod(r, r)  /*finish i_prod inner-product*/
    gamma_old = gamma
    init-inner-prod (t,t)           /*start gamma inner-product */
    if(i_prod / bi_prod)
        /* Convergence Test */
        if(converged)
            break;
```

Figure 7. PCG-Overlap: Overlapping the Innerproducts in the Preconditioned Conjugate Gradient Solver Algorithm

B. Benchmark Suite

In this paper, we use modified versions of the OSU Micro-Benchmarks, which are a part of the MVAPICH2 software package. We measure the average latency of the different implementations of the Allreduce operation across various system sizes. We report communication latency averaged across all the processes, across 1000 iterations and three different runs.

Overlap Benchmark: In this benchmark, we perform floating point matrix-matrix operations by invoking the cblas_dgemm function supported by the Intel MKL Library (10.2.1.017), between the MPI_Iallreduce and the MPI_Wait operations. We measure the overall time required for completion and compute the GFLOPS rating for the given case and compare it against the theoretical peak FLOPS rating for our system. In the first experiment, we fix the message size and vary the matrix size N gradually between the values 10 and 3K and we measure the average throughput. We do a global barrier between two iterations, to ensure that all the processes are synchronized at the start of an iteration.

C. Communication Latency

In Figure 8(a), we compare the latency of the default blocking algorithm in MVAPICH2, with our proposed network offload based designs. We fix the message size to be constant at 1 double as we perform the global summation operation, and we vary the number of processes. We can see that the latency of our proposed network offload based designs are higher than that of the default implementation available in MVAPICH2. We attribute this higher latency to expensive network-loop-back operations, which need to be performed during the execution of the algorithm. As discussed in Section IV, for the recursive-doubling algorithm, every process needs to do a loop-back operation at the end of each iteration. With the tree-based approaches, this issue might be alleviated to an extent, because the number of loop-back operations per process will depend on the degree of the tree. However, the tree-based schemes also require the offload-based broadcast to complete. In Figure 8(a), we compare the latency of the default MPI_Allreduce operation in MVAPICH2, with our proposed network-offload based designs. We observe that the latency of the "Flat" scheme is significantly higher than the rest of the designs. Among the "Two-Level" designs, we observe that both the recursive-doubling (Offload-RD) and the tree-based Reduce-Bcast (Offload-Red-Bcast) are similar. For the tree-based approach, we also vary the degree of the reduce-tree, denoted by Offload-Red-Bcast-2 (for degree 2), Offload-Red-Bcast-3 (for degree 3) and Offload-Red-Bcast-4 (for degree 4). We observe that the latency of the Reduce-Bcast approach is the lowest when the degree is 2. As we increase the degree of the tree, intermediate processes might have to do more number of loop-back operations, before they send the data to their parent processes. We also varied the degree of the tree for the offload-bcast algorithm and we observed that we get the best latency, when the degree of the bcast-tree is 4.

In Figures 8(b), (c) and (d), we further analyze the communication latency of our network offload based designs. We specifically measure the average time required for a process to return from the MPI_Iallreduce and the overhead of the MPI_Wait operations, without attempting any overlap. We can see that the average latency of the MPI_Iallreduce operation is significantly lower with the tree-based design than the recursive-doubling algorithm. This is expected, because the size of the task-list posted by an intermediate process is smaller, when compared to the task-list posted by any process in the recursive-doubling algorithm. Also, for this analysis, we consider the tree-based algorithm with degree-2, since it delivers better latency when compared to the algorithm with higher degrees.

D. Overlap/Throughput Analysis

In this section, we use our throughput benchmark to study the impact on the throughput of the DGEMM operation, when it is overlapped with different variants of the MPI_Iallreduce operation. As indicated in Section IV-C, for collectives with small messages, process skew can play a significant role in affecting the benefits achievable through computation/communication overlap. In Figure 9(a), we compare the measured throughput of the CBLAS-DGEMM operation, when it is overlapped with the MPI_Iallreduce based on either the Two-level-Recursive-Doubling scheme, or the Two-level-Reduce-Bcast algorithm, or the Flat-Recursive-Doubling scheme, being performed with 256 processes. We also compare the measured through-

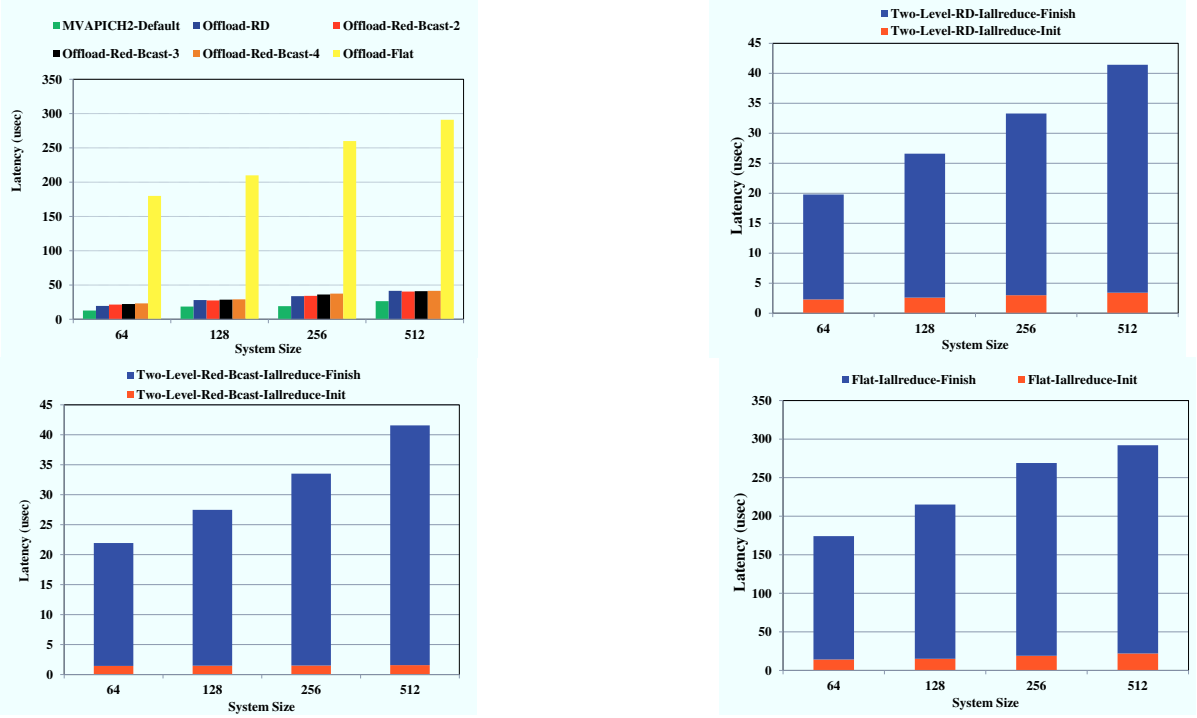

Figure 8. (a) Communication Latency; Latency analysis for the (b) Recursive-Doubling Algorithm, (c) Reduce-Bcast Algorithm (d) Flat Algorithm

put with the peak-throughput of the DGEMM operation, for the given problem size. We can observe that across all the MPI_Iallreduce variants, we are able to achieve throughput which is very close to the peak throughput.

In Figure 9(b), we study the communication overhead of the MPI_Iallreduce operation, when overlapped with the DGEMM operation, as we increase the problem size. We can observe that the communication overheads of the two-level approaches increase as we increase the DGEMM problem size. However, the overhead of the Flat-Recursive-Doubling approach remains nearly constant. We believe that this could either be due to process skew, synchronization inside MPI_Wait and the cache effects, as discussed in Section IV-C. We observe that the variation between the min and the max compute times, across all the processors, for the same DGEMM problem size is as high as 10%. With two-level schemes, if a leader process gets delayed in the DGEMM operation, all the non-leader processes in that node have to wait inside the MPI_Wait operation, which naturally shows up as higher overheads due to MPI_Wait. However, with the Flat approach, every process does the same amount of work inside the MPI_Iallreduce operation. Also, within the MPI_Wait operation, processes only need to poll for completions, without having to synchronize with any other process. We believe that the Flat approach could potentially achieve better benefits through overlap, because they do not introduce skew and may require fewer cache accesses. We also expect that the overall benefits of the flat approach to be higher, if their communication latency could be optimized further, through advanced hardware designs.

Table I
APPLICATION RUN TIME (SECONDS)

Operation		PCG-Algorithm1	PCG-Algorithm2
matvec		39.72	40.29
Inner-Prod Time	sdotp	3.65	4.80
	gamma	9.26	3.09
	iprod	1.23	1.19
Avg Time		14.14	9.08
precond		7.36	3.60
axpy		3.81	3.86
vector_scale		0.98	0.97
solver-time		66.78	59.21

VII. PCG SOLVE PERFORMANCE

A. Potential for Computation/Communication Overlap in PCG Routines

In Table I, we report the average time required to do the different operations for PCG-Algorithm1 and PCG-Algorithm2. For this experiment, we consider 256 processes, and 216000 unknowns per process (-n 60 60 60). We can see that Matvec function accounts for most of the time. We also observe that the precond and the gamma-innerproduct steps in PCG-Algorithm2 are much faster, as discussed in Section V. Both the solvers run for 951 iterations. In each iteration of PCG-Algorithm2, make two calls to the DiagInvScale function, both of which are overlapped by the inner-products. We expect the amount of compute to overlap, per call to Iallreduce to be about 1.8 msec. Since the latency of even the most expensive MPI_Iallreduce operation is about 250μs, with 256 processes, we believe there is enough potential for computation/communication overlap.

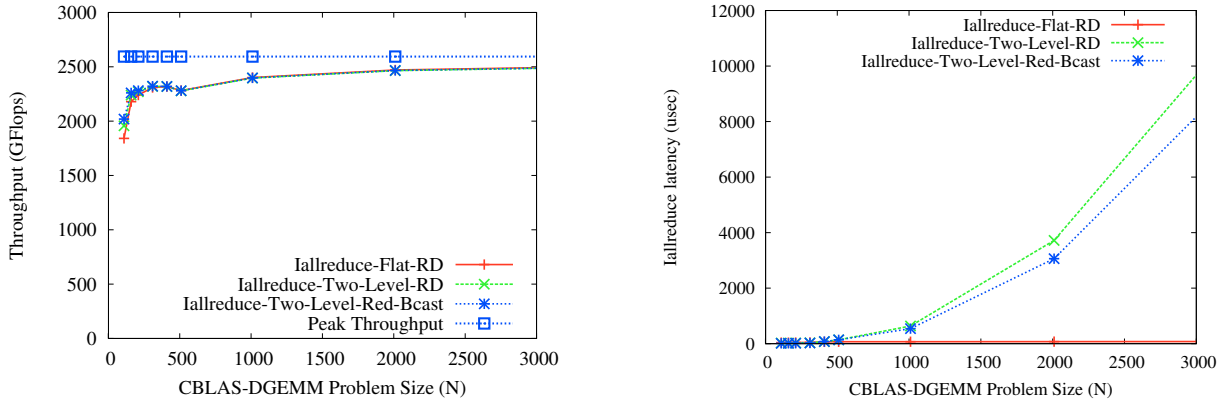

Figure 9. (a) CBLAS-DGEMM throughput, (b) Communication overhead of the MPI_Iallreduce operation

B. PCG Solver Run-Time Comparison

In this section, we study the run-times of the different algorithms for the PCG Solver, across different system sizes and varying number of unknowns. PCG-Algorithm1 and PCG-Algorithm2 use the blocking inner-product function, which uses the regular MPI_Allreduce operation in MVAPICH2. Since our re-designed PCG algorithm in Figure 7 uses a non-blocking inner-product function, we use either one of our proposed MPI_Iallreduce designs. We use a slightly adapted version of the ij.c driver program, which is available in Hypre, to invoke the PCG solver for the 27pt Laplace problem. The 27pt Laplace problem solves a Laplace-like problem with a 27-pt stencil, i.e., each row has an average of 27 non-zeros. We vary the number of processes from 64 through 512 and study the run-times of the different PCG Solver algorithms. Similarly, in Figures 10 (b), (c) and (d), we vary the number of unknowns per process by using the (-n 40 40 40), (-n 60 60 60), (-n 80 80 80) and (-n 10 10 100) run-time options respectively.

In Figure 10(a), we fix the number of unknowns of the Laplace problem by using the -n option as (-n 40 40 40), which leads to 64,000 number of unknowns per process. With 64 processes, the run-time of PCG-Algorithm1 is about 6.8 (s), whereas that of the PCG-Algorithm2 is about 6.1(s). Our proposed PCG algorithm, Overlap-PCG which uses non-blocking inner-products through our network offload based MPI_Iallreduce operation has a run-time of about 6(s) for both the recursive-doubling and the reduce-bcast schemes. However, with the flat scheme, Overlap-PCG(Flat), the run-time is about 5.4(s), which is about 11.5% better than the PCG-Algorithm2 algorithm and about 20.5% better than PCG-Algorithm1. With higher number of processes, say 512, Overlap-PCG(Flat)'s run-time is about 11.1(s), whereas the default PCG-Algorithm1 requires about 14.2(s) and modified PCG-Algorithm2 takes about 12.2 (s).

We observe a similar trend in Figure 10(b), with 216,000 unknowns per process (-n 60 60 60). With 64 processes, the Overlap-PCG(Flat) algorithm, which uses the flat MPI_Iallreduce schemes has a run-time of about 27.93(s), when compared to the PCG-Algorithm2's run-time of 33.3(s) (an improvement of about 16.1%) and PCG-Algorithm1's run-time of 35.6(s) (an improvement of about 21.6%. The run-time of the Overlap-PCG(Flat) is also better than that of the Overlap-PCG(Two-Level-RD) and the Overlap-PCG(Two-level-Red-Bcast) by about 5%. With 512 processes, the Overlap-PCG(Flat) scheme delivers an improvement of about 5.7% when compared to PCG-Algorithm2 and about 13.6% when compared to PCG-Algorithm1.

As we further increase the number of unknowns per process to 512000, in Figure 10(c), Overlap-PCG(Flat) does about 7% better than PCG-Algorithm2 and about 21.5% better than PCG-Algorithm1, with 64 processes. With 512 processes, Overlap-PCG(Flat) performs about 4.5% better than PCG-Algorithm2 and 13% better when compared to PCG-Algorithm1.

We observe that, across various problem sizes, the benefits of overlapping the inner-product appear to be the highest with the Flat MPI_Iallreduce design, than the Two-Level-RD or the Two-Level-Red-Bcast schemes. This is consistent with the results we observed in Figure 9(b). Despite the fact that Flat scheme has very high communication latency, if there is enough compute to overlap, it could potentially deliver better overlap. This is because the Flat scheme is resilient to process skew and cache-friendly. However, we also observe that the benefits are the maximum with 64 processes and the the overall benefits appear to diminish, as we increase the number of processes. This could be attributed to the fact that the latency of the Flat scheme continues to increase, as the number of processes in the MPI_Iallreduce operation increases. We would again like to note that, if next generation hardware interfaces support better features for offloading global reduction operations, without requiring expensive loop-back operations, we could expect to see

higher efficiency, for the Overlap-PCG algorithm, even at higher scales.

C. PCG Solver Run-Time Analysis

In this section, we report the break-up of the run-time between application-level compute and MPI communication. In Figures 11(a), (b) and (c), we consider the case with 64 processes, which showed the maximum improvements and analyze its communication overheads, as we vary the number of unknowns per process. We can see that most of the run-time for the 27pt laplacian problem accounts for the compute phases. MPI_Allreduce dominates the bulk of the communication time, probably indicating that most of the boundary exchange communication is efficiently overlapped by the local Matvec operations. In Figure 11(a), we analyze the run-times across the different PCG and Allreduce algorithms, as we keep the number of unknowns per process constant, at 64,000. The PCG-Algorithm1 and PCG-Algorithm2 versions use blocking MPI_Allreduce and we can see that the overhead of the Allreduce operation in higher in these cases. The Allreduce overheads appear to be smaller with the Overlap-PCG-Red-Bcast and the Overlap-PCG-RD cases, implying that we are seeing benefits through our proposed network offload based designs. We also observe that the overhead of the Allreduce operation is very negligible with the Overlap-PCG-Flat case, which seems to indicate that most of the Allreduce time is effectively hidden. In Figures 11(b) and (c), we repeat the same study, as we increase the number of unknowns per process. We can see that with the Overlap-PCG-Flat scheme, the Allreduce overhead continues to remain significantly smaller than the other alternatives. With better hardware support, we expect that the Overlap-PCG routine achieves better efficiency through completely hiding the latency of the Allreduce operations.

D. Impact of System Noise of PCG Run-Times

In this section, we study the impact of system noise on the performance of PCG Solver routines, by considering PCG-Algorithm2 and the Overlap-PCG algorithm based on the Flat MPI_Iallreduce operation. We believe that this is a fair comparison, because our earlier set of experiments indicate that PCG-Algorithm2 performs better than PCG-Algorithm1, even though both of them use blocking MPI_Allreduce operations. And the Overlap-PCG algorithm delivered achieved better speed-up with the Flat MPI_Iallreduce operation. We rely on a simple daemon that performs matrix-matrix multiplication operations that can be used to inject noise with durations and frequencies and we schedule this daemon on each core on all the nodes. In this experiment, we use 256 processes, fix the number of unknowns per process for the PCG Solver as 216,000. We vary the noise duration from about 50 μs to about 200 μs and the noise frequency between 20Hz and 1KHz. We expect the noise to affect the performance of the compute phases of both the solver routines in a similar manner. However, since PCG-Algorithm2 uses blocking MPI_Allreduce operations, we believe that its communication times could be affected to a greater extent than the Overlap-PCG algorithm, which uses network-offload based MPI_Iallreduce operation. This is mainly due to the fact that the host processors are not required to progress the Allreduce operations, with network-offload based solutions. In Figure 12, we compare the performance degradations of both the PCG algorithms, as we vary the noise duration and frequency. We can observe that the relative performance degradation of the PCG-Algorithm2 is higher, when compared to the Overlap-PCG version, as the noise becomes longer and more frequent. For example, with the extreme case, the performance of PCG-Algorithm2 degrades by as much as 37%, whereas the Overlap-PCG version degrades by about 30%. We expect the effects of noise to be stronger, at larger scales. We shall include experimental results with 512 and 1K core processors, in the final version of the paper.

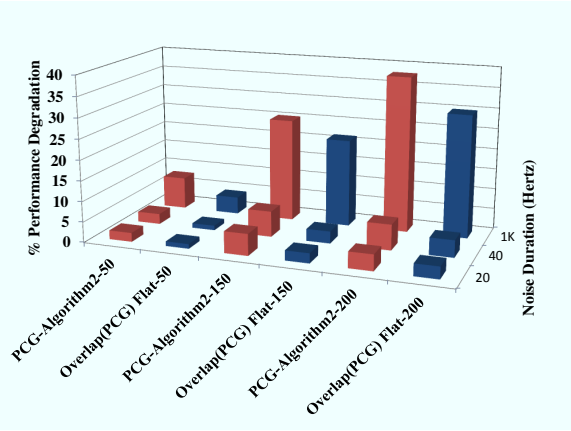

Figure 12. Performance degradation with Noise Injection. PCG-Algorithm2 Vs Overlap-PCG(Flat)

VIII. Related work

Improving computation and communication overlap in parallel applications has traditionally been a topic of great interest [19]. Sancho et al. [20] study the benefits of using dedicated processors for progressing the Allreduce operation and study the benefits with of overlapping the Allreduce operations in POP, a weather modeling application. Improving the efficiency of the Conjugate Gradient Solvers is a widely studied problem [21]. In [22], Hoefler et al. tried to optimize the CG Solver using the CG method by Hestenes and Stiefel[23]. However, the authors noted that they were unable to resolve the data dependency necessary to overlap the Allreduce operations. In our work, we leverage the PCG algorithm variants proposed by Demmel et al. [24] and extend their work to achieve communication/computation overlap through non-blocking implementations of the inner-product operations, which use our network-offload based MPI_Iallreduce operations.

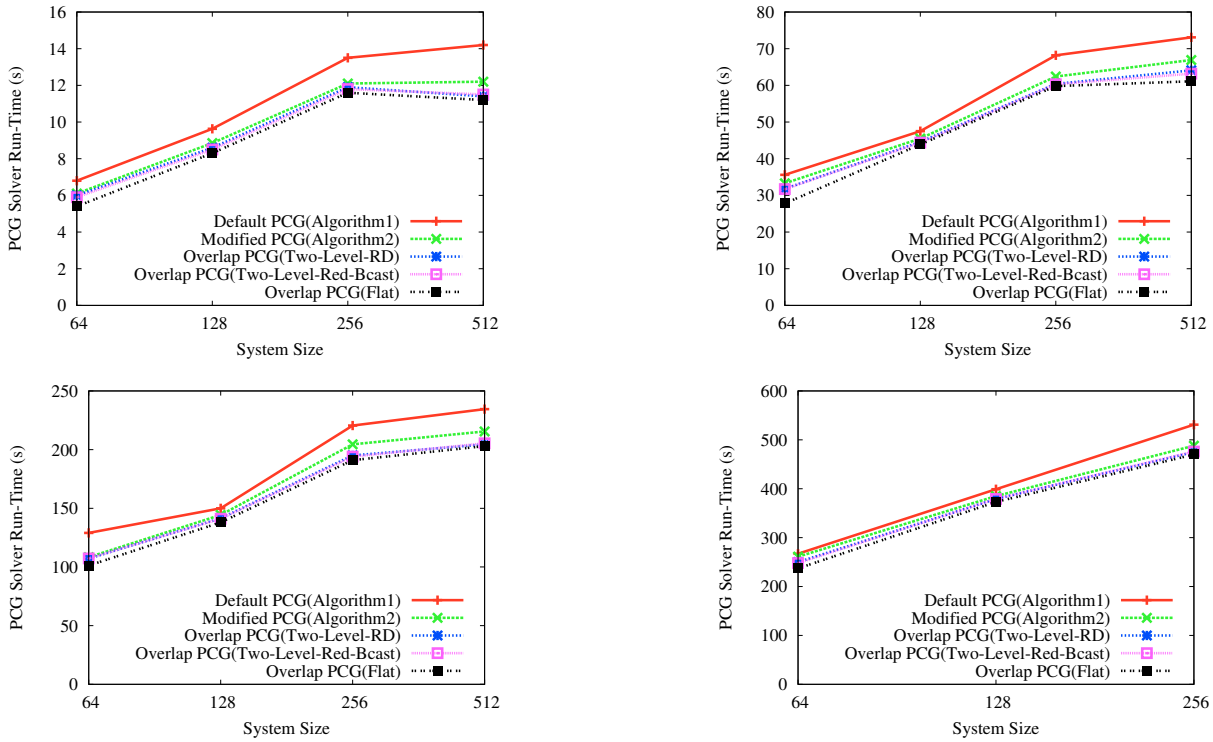

Figure 10. PCGSolve Run-Time Comparisons (a) -n 40 40 40, (b) -n 60 60 60, (c) -n 80 80 80 and (d) -n 100 100 100

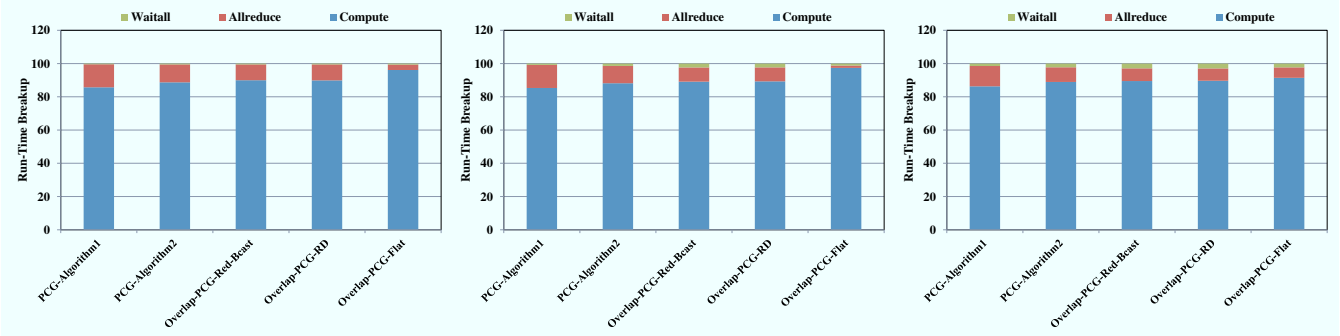

Figure 11. PCG Run-Time Analysis with 64 processes (a) -n 40 40 40, (b) -n 60 60 60 and (c) -n 80 80 80

Hoefler et. al. proposed using host based techniques for designing non-blocking collective operations [8]. However, host based techniques, offer limited performance portability and may not deliver complete overlap. In [25], Hemmert et. al. demonstrate the benefits of using triggered operations and counting events provided by the Portals 4.0 message passing interface. Additionally, Beckman at. al. [26] studied the impact of noise on the performance of collectives by injecting noise. We use a subset of these parameters in our experiments. Graham et. al. reported early experiences with the CORE-Direct software API in [13]. Subramoni et. al. proposed communication primitives for blocking collective operations with the CORE-Direct in [6]. In [10], we designed a scalable network offload based MPI_Ialltoall implementation and demonstrated up to 23% improvement with a parallel 3D FFT library. In [11], we proposed network-offload based designs for the MPI_Ibcast operation and studied the benefits of achieving communication/computation overlap with the HPL benchmark. In this paper, we propose efficient non-blocking designs for the Allreduce operation that scales beyond 512 processes and study the benefits with Preconditioned Conjugate Gradient Solvers in the Hypre software library. We also study the benefits of using network based collectives with system noise. In both [10] and [11], we observed that our network-offload based solutions offer significantly better performance benefits than host-based non-blocking solutions, such as libNBC. Hence, in this work, we focus more on the different design choices for network offload based MPI_Iallreduce and understanding their behavior with the PCG solver algorithms.

IX. Conclusion

In this paper, we designed fully functional, scalable algorithms for the MPI_Iallreduce operation, based on the network offload technology. We showed that we are able to scale our designs to more than 512 processes and we achieve near perfect communication/computation overlap. We also re-designed the PCG solver routine to leverage our proposed MPI_Iallreduce operation to hide the latency of the global reduction operations. Our proposed Overlap-PCG algorithm does up to 21% better than the default PCG implementation in Hypre, about 16% of these benefits are derived through hiding the latency of the global reductions. All of our current work was based on the CX-2 InfiniBand network interface from Mellanox. We believe that the benefits of our proposed approaches could be higher with better hardware support for offloading reductions to the network. We plan to include results with the next generation interface, CX-3, for the final version of the paper. In the future, we wish to explore the benefits of hiding the latency of the Allreduce operations with other solver routines in Hypre. It could also be interesting to study the benefits of our work with real scientific applications, which use Hypre's solvers.

References

[1] MPI Forum, "MPI: A Message Passing Interface," in *www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf*.

[2] T. Hoefler and T. Schneider and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[3] Petrini, Fabrizio and Kerbyson, Darren J. and Pakin, Scott, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'03)*, ser. SC, 2003.

[4] Top500, "Top500 Supercomputing systems," Oct 2010, .

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

[6] H. Subramoni, K. Kandalla, S. Sur and D K. Panda, "Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine," in *HotI'18*, 2010.

[7] R.D. Falgout, J.E. Jones, and U.M. Yang, " The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners," in *chapter in Numerical Solution of Partial Differential Equations on Parallel Computers, A.M. Bruaset and A. Tveito, eds., Springer-Verlag, 51 (2006), pp. 267-294. UCRL-JRNL-205459.* .

[8] T. Hoefler, J. 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.

[9] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?" in *Cluster*, 2008.

[10] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur and D. K. Panda, "High-Performance and Scalable Non-Blocking All-to-All with Collective Offload on InfiniBand Clusters: A Study with Parallel 3D FFT ," in *ISC*, June,2011.

[11] K. Kandalla, H. Subramoni, J. Vienne, K. Tomko, S. Sur and D. K. Panda, "Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: A Case Study with HPL ," in *Hot Interconnects*, August, 2011.

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

[13] 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 *CAC*, 2010.

[14] Argonne National Laboratory, "MPICH2: High-Performance MPI Implementation," http://www.mcs.anl.gov/research/projects/mpich2/.

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

[16] R.D. Falgout, J.E. Jones, and U.M. Yang , "Conceptual Interfaces in hypre," in *Future Generation Computer Systems, Special Issue on PDE Software, UCRL-JC-148957*, 2006, pp. 239–251.

[17] A.H. Baker, R.D. Falgout, Tz.V. Kolev, and U.M. Yang , "Scaling hypre's Multigrid Solvers to 100,000 Cores," in *to appear in High Performance Scientific Computing: Algorithms and Applications - A Tribute to Prof. Ahmed Sameh, M. Berry et al., eds., Springer. LLNL-JRNL-479591.*

[18] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, "Parallel Numerical Linear Algebra," in *Society for Industrial and Applied Mathematics*. SIAM, 1997.

[19] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications," ser. Proceedings of the 2006 ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM, 2006.

[20] J. C. Sancho, D. J. Kerbyson, and K. J. Barker, "Efficient offloading of collective communications in large-scale systems," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 169–178, 2007.

[21] A. Chronopoulos and C. Gear, "On the Efficient Implementation of Preconditioned S-Step Conjugate Gradient Methods on Multiprocessors with Memory Hierarchy," in *Parallel Computing, 11 (1989)*, 2008, pp. 37–53.

[22] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations," *Elsevier Journal of Parallel Computing (PARCO)*, vol. 33, no. 9, pp. 624–633, Sep. 2007.

[23] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, Dec. 1952.

[24] James W. Demmel, Michael T. Heath, Henk A. van der Vorst, "LAPACK Working Note 60, UT CS-93-192, Parallel Numerical Linear Algebra," http://www.netlib.org/lapack/lawnspdf/lawn60.pdf.

[25] K. Hemmert, B. Barrett, and K. Underwood, "Using Triggered Operations to Offload Collective Communication Operations," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6305, pp. 249–256.

[26] P. Beckman, K. Iskra, K. Yoshii, S.Coghlan, A.Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, pp. 3–16, March 2008.