Can Network-Offload based Non-Blocking Neighborhood MPI Collectives Improve Communication Overheads of Irregular Graph Algorithms?

K. Kandalla[1], A. Buluç[2], H. Subramoni[1], K. Tomko[3], J. Vienne[1], L. Oliker[2], 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 Berkeley National Laboratory*
Berkeley, California
{abuluc, loliker}
@lbnl.gov

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

***Abstract*—Graph-based computations are commonly used across various data intensive computing domains ranging from social networks to biological systems. On distributed memory systems, graph algorithms involve explicit communication between processes and often exhibit sparse, irregular behavior. Minimizing these communication overheads is critical to cater to the graph-theoretic analyses demands of emerging "big data" applications. In this paper, we explore the challenges associated with improving the communication overheads of a popular 2D Breadth First Search (BFS) implementation in the CombBLAS library. This BFS algorithm relies on two common MPI collectives, MPI_Alltoallv and MPI_Allgatherv to exchange data between processes. Since they are blocking operations, their communication overheads account for about 20% of the overall run time and limit the scalability and performance of the 2D BFS algorithm. We propose to re-design the BFS algorithm to leverage MPI-3 non-blocking, neighborhood collective communication operations to achieve fine-grained computation/communication overlap. We also leverage the CORE-Direct network offload feature in the ConnectX-2 InfiniBand adapter from Mellanox to design highly efficient and scalable non-blocking, neighborhood Alltoallv and Allgatherv collective operations. Our experimental evaluations show that we can improve the communication overheads of the 2D BFS algorithm by up to 70%, with 1,936 processes.**

***Keywords*-MPI-3 non-blocking collectives, 2D BFS Algorithms, Collective Offload and InfiniBand**

I. Introduction

The Message Passing Interface (MPI) [1] is a widely used programming model for High Performance Computing applications. MPI defines a set of collective operations that are used to communicate data between a group of participating processes. Owing to their ease of use and portability, these operations are commonly used. The current MPI-2.2 standard defines collective operations to be blocking and this limits the overall performance and scalability of parallel applications. These operations are also not a scalable way of representing irregular, sparse communication patterns [2]. These limitations have spurred interest in the design of both non-blocking and neighborhood collective communication operations in the upcoming version, MPI-3. 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. Also, in order to fully leverage the benefits of non-blocking communication, scientific applications need to be re-designed to overlap compute tasks with the non-blocking collective operations.

InfiniBand is a popular switched interconnect standard being used by almost 41% of the Top500 Supercomputing systems [3]. Since InfiniBand is so widely used, efficient support of non-blocking collectives in MPI implementations on InfiniBand is critical. Mellanox has introduced network offload features in their ConnectX-2 and ConnectX-3 [4] adapters. Using this feature, generic lists of communication tasks can be offloaded to the network interface [5]. Such an interface eliminates the need for the host processor to progress collective communication and provides a low-level mechanism to design non-blocking collectives.

Graphs are one of the most ubiquitous models in analytical workloads. They are powerful representations of many types of relations and process dynamics and are used in a variety of scientific and engineering fields like cyber security, social networks, financial applications, etc. Common graph-theoretic problems arising in these application areas include identifying and ranking important entities, detecting anomalous patterns, finding tightly interconnected clusters, and so on. The solutions to these problems typically involve classical algorithms for problems such as finding spanning trees, shortest paths, biconnected components and flow-based computations. On distributed memory parallel systems, graph algorithms involve explicit communication between processes. To cater to the graph-theoretic analyses demands of emerging "big data" applications, it is essential to minimize or hide these communication overheads to speed up the underlying graph problems.

II. Motivation

Computational scientists have an excellent understanding on mapping regular numerical scientific applications to current generation supercomputing systems. The challenges involved in re-designing regular numerical applications to leverage non-blocking MPI collectives have also been explored [6], [7], [8]. In contrast, little is known about the de-

*This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25755; National Science Foundation grants #CCF-0916302, #OCI-0926691 and #CCF-0937842.

sign, parallel performance and programmability trade-offs of data-intensive graph algorithms. The challenges associated with improving the communication overheads in irregular graph algorithms are also not obvious [9]. The recently created Graph500 List [10] is an attempt to rank supercomputers based on their performance on data-intensive applications. Breadth First Search (BFS) is chosen as the first representative benchmark in this suite. BFS on distributed-memory systems involves explicit, irregular communication between processes, and the distribution (or partitioning) of the graph among processors also impacts performance.

In this paper, we consider the top-down BFS algorithm in the Combinatorial BLAS [11] library (CombBLAS), which is a distributed memory parallel graph library offering a set of linear algebra primitives specifically targeting graph analytics. CombBLAS uses a 2D distribution of its graphs and vectors. The BFS implementation of CombBLAS relies on MPI as the communication substrate and uses common MPI collective communication primitives, such as MPI_Alltoallv and MPI_Allgatherv to exchange data across the row and column communicators. Since the communication pattern is irregular, the standard versions of MPI_Alltoallv and MPI_Allgatherv may not be ideally suited to represent the communication pattern in CombBLAS. Moreover, since they are blocking operations, their communication overheads also grow as we scale up the number of processes. Neighborhood collectives have been proposed to allow each process to specify exactly those peers with which it explicitly exchanges data. This mainly reduces the memory overheads involved in dealing with sparse communication patterns and improves scalability [2], [12]. However, in order to minimize the overheads of collective operations, we need to re-design these graph algorithms to leverage non-blocking collectives. Moreover, the data dependency between the collective operations and the compute functions may not be directly amenable for overlap. These factors lead us to an important problem, *How can we re-design the 2D BFS algorithm in CombBLAS in a scalable manner to efficiently overlap the communication overheads of the collective operations?*

Applications can embed the neighborhood information of any arbitrary communication pattern within a communicator via standard MPI function calls, such as, MPI_Graph_Create(). The neighborhood collective operations can easily use this information to implement the specified communication patterns. Essentially, neighborhood collectives can also be used to implement generic collectives algorithms, such as the pair-wise exchange, or the ring-exchange. This feature can also be used to decompose a collective algorithm across multiple phases, by specifying only a part of the communication graph of the overall collective algorithm within the graph communicator, corresponding to each phase. If an application can be re-designed to split the compute tasks across multiple steps, we could pipeline and overlap the compute and communication phases. We explore this concept in the 2D BFS algorithm in CombBLAS, to achieve fine-grained communication/computation overlap. However, host-based non-blocking collectives may not always deliver good overlap and network-offload-based designs have been demonstrated to be very effective [6], [13]. Hence, in our work, we design non-blocking neighborhood collectives based on InfiniBand's network offload technology. We then leverage these operations to improve the communication overheads of the 2D BFS algorithm in CombBLAS by up to 70% with 1,936 processes. Our MPI-level designs are integrated into the MVAPICH2-1.8 [14] software stack. MVAPICH2 is a high performance MPI library over InfiniBand and RoCE networks. It is being used by more than 1,930 organizations world-wide in 68 countries. To summarize, we address the following important challenges:

1) What are the obstacles to re-designing irregular graph algorithms, such as, the 2D BFS, to achieve fine-grained computation/communication overlap by leveraging non-blocking, neighborhood collective communication operations?
2) Can we design network-offload-based solutions for non-blocking neighborhood collectives and use them to implement common collective patterns such as the Alltoallv and Allgatherv?
3) What are the potential benefits and trade-offs of such a co-design between the 2D BFS algorithm and the MPI libraries?

III. Background

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

A. InfiniBand and ConnectX-2 Network Interface

InfiniBand QDR network cards and switches can deliver 32 Gbps end-to-end bandwidth and about 1.0 to 1.5 μs latency. Along with all of the standard InfiniBand features, the ConnectX-2 and ConnectX-3 [4] network adapters from Mellanox offer a network offloading feature called CORE-Direct [5]. 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.

B. CORE-Direct Support in MVAPICH2

Previously, we proposed network-offload-based designs for various collectives, such as, MPI_Alltoall, MPI_Bcast and MPI_Reduce and MPI_Allreduce and described the communication protocols we use for small and large messages with the CORE-Direct interface in MVAPICH2 software library [6], [13]. We pre-post buffers to minimize the latency of small message exchanges. For large messages,

we either rely on InfiniBand's *Receiver-Not-Ready* (RNR) feature, or use a software-based credit-control mechanism that is similar to the designs proposed in [5].

C. Breadth First Search (BFS) Algorithms

Given a distinguished "source vertex" s, Breadth-First Search (BFS) systematically explores the graph G to discover every vertex that is reachable from s. A path from vertex s to t is defined as a sequence of edges (u_i,u_{i+1}), $0 \leq i < l$, where u_0 = s and u_l = t. The length of a path is the sum of the weights of edges. We use $d(s, t)$ to denote the distance between vertices s and t, or the length of the shortest path connecting s and t. BFS implies that all vertices at a distance k (or "level" k) from vertex s should be first "visited" before vertices at distance k + 1. The distance from s to each reachable vertex is typically the final output. In this paper, we consider the top-down BFS algorithm in the Combinatorial BLAS [11] library (CombBLAS).

IV. Designing Network-Offload-based Non-blocking Neighborhood MPI Collectives

We extend the concept of neighborhood collectives and treat them as "building-blocks" to "compose" generic collective operations. This allows applications and MPI libraries to decompose generic blocking/non-blocking collective operations, such as Alltoallv and Allgatherv, across multiple phases, along with the scalability benefits of existing neighborhood collectives. These building-blocks can be used by applications to achieve fine-grained communication/computation overlap and need to be designed very efficiently. By default, MVAPICH2 uses the pair-wise exchange algorithm to implement large message Alltoallv communication pattern and the ring algorithm to implement large message Allgatherv operations. In this section, we discuss our proposed designs for network-offload based, non-blocking neighborhood variants of AlltoAllv and Allgatherv collective operations for medium and large messages by directly using either of the large message protocols (Section III-B).

Suppose a process exchanges non-zero counts of data elements to X processes in a communicator of size Y, ($X \leq Y$), during an Alltoallv operation. A basic neighborhood communicator for this process consists of the communication graph that corresponds only to the X processes. We propose to divide the original communication graph into multiple sub-graphs. The size of each sub-graph is configurable and each of them can be represented within different neighborhood communicators. Applications can issue multiple non-blocking neighborhood Alltoallv operations with each of these communicators to implement the pairwise exchange algorithm in multiple phases. Hence, the number of steps and the overall communication volume of our proposed designs are identical to that of the default non-blocking MPI_Alltoallv operation. But, we have split the communication operation into multiple phases and each of these phases can be overlapped with compute tasks in a pipelined manner. On modern processors, the overheads associated with handling 0's in sendcnt/recvcnt arrays in an MPI_Alltoallv operation is negligible. However, the memory overheads of using sendcnt/recvcnt arrays, consisting mostly of 0's is prohibitive. Our designs directly inherit the memory scalability merits of neighborhood collectives, because the size of each of the sendcnt/recvcnt arrays will never exceed the number of neighbors in each communication sub-graph, even if these array contain 0's. Also, we create the neighborhood communicators for both the collective operations only once, instead of making several calls to MPI_Graph_Create during each iteration of the 2D BFS algorithm to eliminate the 0's in the sendcnt/recvcnt arrays.

A. Non-Blocking Neighborhood AlltoAllv:

Our network-offload based non-blocking neighborhood solution for the AlltoAllv communication pattern is based on the conventional pair-wise exchange algorithm. Our designs also allow applications to vary the size (or "neighborhood-size") of each of these phases, for finer control over the number of communication steps. For example, consider a communicator of size 64 processes and a neighborhood-size of 8, we create eight different neighborhood communicators and will make eight calls to the non-blocking neighborhood_Alltoallv function. In each of these calls, our proposed MPI-level designs will create task-lists to implement the pair-wise exchange algorithm on the specific neighborhood communicator. Once the task-lists have been posted to the network adapter, it progresses the collective operation independently and requires no CPU intervention. For collectives like Alltoallv, it is not advisable to have multiple concurrent instances of asynchronous send/recv operations, as it can cause contention. Our network-offload-based designs use the hardware-based "wait" tasks in each of the task-lists to split them into multiple "windows", whose sizes are dynamically tunable. The network adapter executes the wait-tasks independently and processes each window, one at a time.

B. Non-Blocking Neighborhood Allgatherv:

Our network-offload-based non-blocking, neighborhood solution for the Allgatherv communication pattern is based on the conventional ring exchange algorithm. Even in this case, applications can choose specific neighborhood-sizes to create neighborhood communicators. For each call to Ineighborhood_Allgatherv, we implement the ring algorithm on the specific communicator. Each process in the ring algorithm communicates only with its left and right neighbors, but uses different send/recv offsets in each iteration. We use the neighborhood information embedded in the communicator to compute the send/recv offsets, for each iteration. Varying the neighborhood-sizes will lead to different

number of communication steps within each call to Ineighborhood_Allgatherv. However, with the ring algorithm, we need to use a window-size of 1, to ensure data consistency. We also explored a simpler "linear" algorithm to implement the Ineighborhood_Allgatherv operation. However, we found such a design to be highly inefficient. Hence, we use only the ring approach for Allgatherv, for the rest of this paper.

V. Re-designing the 2D BFS Algorithm in CombBLAS for Overlap

Algorithm 1 gives the high-level pseudocode of the 2D BFS algorithm that is shown to be more scalable than the 1D BFS algorithms [15]. This algorithm implicitly computes the "breadth-first spanning tree" by returning a dense parents array. The inner loop block performs a single level traversal of the spanning tree. f, which is initially an empty sparse vector, represents the current frontier. t is a sparse vector that holds the temporary parent information for the current iteration only. For a distributed vector v, the syntax v_{ij} denotes the local n/p sized piece of the vector owned by the $P(i,j)$th processor. The syntax v_i denotes the hypothetical n/p_r or n/p_c sized piece of the vector collectively owned by all the processors along the i'th processor row $P(i,:)$ or column $P(:,i)$. The syntax $\otimes$ denotes the matrix-vector multiplication operation on a special semiring where scalar addition is a minimum operation and scalar multiplication just propagates the second operand, $\odot$ denotes element-wise multiplication, and overline represents the complement operation. ALLGATHERV($f_{ij}, P(:,j)$) syntax denotes that subvectors f_{ij} for j = 1, .., p_r is accumulated at all the processors on the jth processor column. Similarly, ALLTOALLV($t_i; P(i,:)$) denotes that each processor scatters the corresponding piece of its intermediate vector t_i to its owner, along the ith processor row.

Algorithm 1 Default 2D BFS algorithm [15].

Input: A: undirected graph represented by a boolean sparse adjacency matrix, s: source vertex id.

Output: π: dense vector, where $\pi[v]$ is the predecessor vertex on the shortest path from s to v, or -1 if v is unreachable.

1: **procedure** BFS_2D(A, s)
2: $f(s) \leftarrow s$
3: **for** all processors $P(i,j)$ **in parallel do**
4: **while** $f \neq \emptyset$ **do**
5: TRANSPOSEVECTOR(f_{ij})
6: $f_i \leftarrow$ ALLGATHERV($f_{ij}, P(:,j)$)
7: $t_i \leftarrow A_{ij} \otimes f_i$
8: $t_{ij} \leftarrow$ ALLTOALLV($t_i, P(i,:)$)
9: $t_{ij} \leftarrow t_{ij} \odot \overline{\pi_{ij}}$
10: $\pi_{ij} \leftarrow \pi_{ij} + t_{ij}$
11: $f_{ij} \leftarrow t_{ij}$

The data dependency between the collective operations and the compute functions in Algorithm 1 is not very conducive for overlap. For example, the t_i sparse vector is populated in line 7 just after f_i is assembled by the ALLGATHERV step and is used immediately in the ALLTOALLV step. Also, it is not feasible to decouple the start and end steps of these collective operations across two successive iterations of the while block. We present our re-designed 2D BFS algorithm in Algorithm 2, which relies on non-blocking, neighborhood-collectives to decompose the collective operations across multiple phases in an asynchronous manner. The compute functions are also decomposed into multiple phases, accordingly. During initialization of the 2-BFS kernel, we create the required graph communicators for Allgatherv and Alltoallv collectives. This is a one time operation and does not cause additional overheads to our proposed 2D BFS algorithm. The notation $P(:,(j_1,j_2))$, where $0 \leq j_1, j_2 \leq p_r$ denotes that each call to a neighborhood collective operates on a specific group of neighbors on the column communicator. Similarly, The notation $P((j_1,j_2),:)$, where $0 \leq j_1, j_2 \leq p_c$ denotes that each call to a neighborhood collective operates on a specific group of neighbors on the row communicator. Since the sparse vectors are only partially filled during each iteration, we use the notation $f_{i,k}$,$t_{i,k}$, $t_{ij,k}$ and $\pi_{ij,k}$ to refer to the parts of these vectors that were populated during the kth iteration. The non-blocking, neighborhood collective functions also accept the "k" parameter so that we choose the corresponding neighborhood communicator for that call. As shown in Algorithm 2, we perform the first phase of both of these collectives in a blocking manner. In the subsequent iterations, we issue the current non-blocking neighborhood collective and overlap it with the compute function on the data received in the previous iteration. Finally, we perform the compute functions on the data received in the last phase of the non-blocking neighborhood collective. Such a design allows us to achieve fine-grained overlap to efficiently hide the communication overheads of the collective operations. The ALLTOALLV step involves two back-to-back calls to MPI_Alltoallv() operation. In Algorithm 2, we initiate both the non-blocking operations during NBR_IALLTOALLV and our MPI-level designs can handle both of them in a completely asynchronous manner.

Our re-designed algorithm requires additional processing to copy and sort the data to ensure correctness. The first sorting step is done immediately after the Neighborhood-Allgatherv operation, to prepare the $f_{i,k-1}$ vector. Depending on its neighborhood-size, the Neighborhood-Allgatherv operation may fill parts of the receive buffer in a non-contiguous manner. Hence, we need to pack the data into a contiguous buffer and sort the data. Next, we issue the $t_{i,k} \leftarrow A_{ij} \otimes f_{i,k-1}$ operation in multiple steps. Now, the t_i vector also needs to be processed and sorted before we perform the ALLTOALLV operation. Similarly, the t_{ij} vector is also populated across multiple steps and also requires a

Algorithm 2 Overlapped 2D BFS algorithm.

Input: A: undirected graph represented by a boolean sparse adjacency matrix, s: source vertex id

Output: π: dense vector, where π[v] is the predecessor vertex on the shortest path from s to v, or -1, if v is unreachable

1: **procedure** BFS_2D_OVERLAP(A, s)
2: $f(s) \leftarrow s$
3: **for** all processors $P(i,j)$ **in parallel** **do**
4: **while** $f \neq \emptyset$ **do**
5: TRANSPOSEVECTOR(f_{ij})
6: $k \leftarrow 0$
7: $f_{i,k} \leftarrow$ NBR_IALLGATHERV($f_{ij}, k, P(:,(j_1,j_2))$)
8: MPI_WAIT
9: **for** $k \leftarrow 1$ to $num\_allgatherv\_comm$ **do**
10: $f_{i,k} \leftarrow$NBR_IALLGATHERV($f_{ij}, k, P(:,(j_1,j_2))$)
11: SORT($f_{i,k-1}$)
12: $t_{i,k-1} \leftarrow A_{ij} \otimes f_{i,k-1}$
13: MPI_WAIT
14: SORT($f_{i,k-1}$)
15: $t_{i,k-1} \leftarrow A_{ij} \otimes f_{i,k-1}$
16: $k \leftarrow 0$
17: SORT($t_{i,k}$)
18: $t_{ij,k} \leftarrow$ NBR_IALLTOALLV($t_i, k, P((j_1,j_2),:)$)
19: SORT($t_{i,k+1}$)
20: MPI_WAIT
21: **for** $k \leftarrow 1$ to $num\_alltoallv\_comm$ **do**
22: $t_{ij,k} \leftarrow$ NBR_IALLTOALLV($t_i, k, P((j_1,j_2),:)$)
23: $t_{ij,k-1} \leftarrow t_{ij,k-1} \odot \overline{\pi_{ij,k-1}}$
24: $\pi_{ij,k-1} \leftarrow \pi_{ij,k-1} + t_{ij,k-1}$
25: SORT($t_{i,k}$)
26: SORT($t_{ij,k-1}$)
27: MPI_WAIT
28: $t_{ij,k-1} \leftarrow t_{ij,k-1} \odot \overline{\pi_{ij,k-1}}$
29: $\pi_{ij,k-1} \leftarrow \pi_{ij,k-1} + t_{ij,k-1}$
30: SORT($t_{ij,k-1}$)
31: $f_{ij} \leftarrow t_{ij}$

sorting step. As shown in Algorithm 2, we overlap these three sorting steps with the neighborhood collective calls. We discuss the overheads of these sorting functions in Section VII. We also evaluate the impact of a Heap-Merge sort and an IntegerSort algorithm on the overheads of the three sorting functions.

VI. EXPERIMENTAL EVALUATION

A. Experimental Setup

We use two experimental clusters for evaluating our proposed designs.

ClusterA: This cluster has 120 compute nodes, each node is based on the dual-socket, quad-core Intel Xeon E5630 (Westmere) architecture operating at 2.53 Ghz with 12 MB L3 cache per socket and 12 GB main memory with Gen2 PCI-Express bus. They are equipped with MT26428 QDR ConnectX-2 HCAs with PCI-Ex interfaces. The operating system used is RHEL 6.1 (2.6.32-131.0.15.el6 kernel) with 1.5.3-3 OFED version and GNU 4.6.3 compilers.

ClusterB: We use the Hyperion cluster [16] at Lawrence Livermore National Lab for our larger scale evaluations. This cluster has about 242 compute nodes, based on the dual-socket, quad-core Intel Xeon E5530 (Nehalem) architecture, operating at 2.40 GHz, with 8 MB L3 cache per socket and 12 GB main memory with Gen2 PCI-Express bus. The operating system is RHEL 5.5 (2.6.18-107.chaos kernel), with 1.5.3-3 OFED version and GNU 4.1.2 compilers.

For graph application runs, we use synthetic graphs based on the R-MAT random graph model [17]. R-MAT is a recursive graph generator that creates networks with skewed degree distributions and a very low graph diameter. We set $edgefactor$ to 16, and the R-MAT parameters a, b, c, and d to $0.59, 0.19, 0.19$ and 0.05 respectively. An R-MAT graph of scale N has 2^N vertices and approximately $edgefactor \cdot 2^N$ edges. These parameters are identical to the ones used for generating synthetic instances in the Graph500 BFS benchmark.

B. Benchmark Suite

Latency: We modify OSU Micro-Benchmarks with our proposed network-offload-based neighborhood collectives to implement the MPI_Allgatherv and MPI_Alltoallv operations on MPI_COMM_WORLD. We report the average latency of the different implementations across various system sizes and message lengths.

Overlap Benchmark: In order to implement the global Allgatherv or Alltoallv operations, we may need to make several calls to their non-blocking neighborhood versions, with different neighborhood communicators. Since each of these calls can be overlapped with compute, we first measure the average time taken to complete each round of these collectives. We then introduce a timer loop on the CPU that runs for exactly the same duration. We time the entire operation and measure the overlap percentage we can achieve through our network-offload-based designs.

With both of these benchmarks, we also study the impact of the neighborhood-size and the window sizes on the communication performance and overlap.

C. Communication Latency

In Figure 1(a), we compare the average communication latency of the default MPI_Alltoallv operation with the time required to complete the Alltoallv exchange through the neighborhood-based Alltoallv operations. We observe that the communication performance of our proposed network-offload based, neighborhood Alltoallv is very similar to that of the default MPI_Alltoallv operation in MVAPICH2. In Figure 1(b), we do a similar study with MPI_Allgatherv operation. In this case, we see that the network-offload based

version has a higher communication cost, when compared to the default MPI_Allgatherv operation in MVAPICH2. On the other hand, if we disable the shared-memory channel for the intra-node communication steps of MPI_Allgatherv, we see that the communication latency of this operation is similar to that of our network-offload based designs. By default, MVAPICH2 maps processes to cores in a block-ordered manner. Since these benchmarks use MPI_COMM_WORLD, in each step of the ring algorithm, we only have two transfers going over the network and the rest of the transfers happen intra-node. In our network-offload based designs, the intra-node communication steps happen through network-loopback and we cannot use the shared-memory channels. We would like to note that the communication latency of both the neighborhood-based collectives remain the same, even as we vary the neighborhood-sizes. Also, as explained in Section IV, with Neighborhood-based Alltoallv, we see a small degradation in the performance when we use a window size of 8.

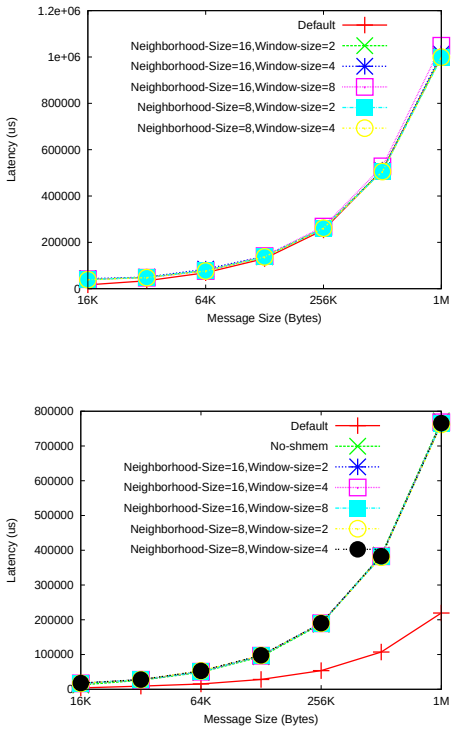

Figure 1. Communication Latency (256 processes, ClusterA): (a) Alltoallv and (b) Allgatherv

D. Communication/Computation Overlap

In this section, we use our overlap benchmark to evaluate the amount of overlap that our network-offload-based neighborhood collectives can offer. In Figures 2(a) and (b), we see that for various message sizes, our designs can offer near-perfect overlap for both neighborhood-based Allgatherv and Alltoallv operations. We also observe that the overlap percentage remains the same, even as we vary the neighborhood-sizes. For brevity, for both these benchmarks, we report results with 256 processes on ClusterA.

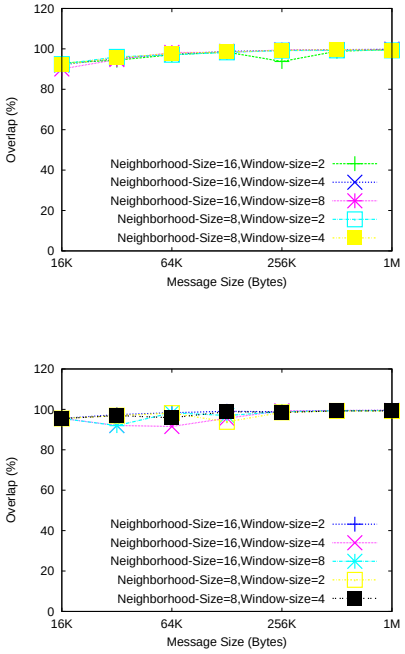

Figure 2. Communication Overlap (256 processes on ClusterB): (a) Alltoallv and (b) Allgatherv

VII. 2D BFS Evaluation

A. Potential for Overlap

In this section, we profile the various key operations performed in each iteration of the default 2D BFS algorithm in CombBLAS with 729 processes, in Cluster A, with various scales. In Table I, we observe that the combined communication overheads account for about 20% of the overall time spent in each iteration of the 2D BFS algorithm. We also observe that the t_i step in line 7 (Algorithm 1) is the most expensive compute function in each iteration of the 2D BFS because its "work" (its computational complexity summed over all processors) [18, Chapter 27] for an entire BFS is linear in the number of edges. In comparison, the time spent in t_{ij} and π_{ij} steps in line 9 and line 10 is lower since the work of these operations are proportional to the number of vertices, which is typically $10-100\times$ less for realistic graphs. As discussed in (Algorithm 2), in our proposed 2D BFS algorithm, we overlap the Allgatherv collective with the t_i step and the Alltoallv collective with the t_{ij} and π_{ij} steps. We observe that we have enough potential to overlap both the collective operations and efficiently hide their communication overheads. For example, with scale=29, we see that the Allgatherv overhead is only about 0.15s, whereas the t_i time is about 0.801. The Alltoallv overheads are also almost comparable with the times being spent in the t_{ij} and π_{ij} steps and we expect our proposed designs to

Table I
APPLICATION RUN TIME, 729 PROCESSES, CLUSTERA (SECONDS)

Scale	26	27	28	29
t_i update	0.0890	0.1860	0.3770	0.801
Allgatherv	0.0214	0.0394	0.0750	0.142
t_{ij}, π_{ij} updates	0.0242	0.0473	0.0902	0.176
AlltoAllv	0.0244	0.0392	0.0676	0.152
Mean Iteration Time	0.1927	0.3660	0.7080	1.816

lead to smaller overheads for both Alltoallv and Allgatherv, through efficient communication/computation overlap.

B. 2D BFS Run-Time Comparison

In this section, we compare the mean time of each iteration of the 2D BFS operation of the default implementation that uses blocking collectives and our proposed version that relies on neighborhood, network-offload based non-blocking Allgatherv and Alltoallv operations. We report results on both platforms, with various system sizes and scales.

In Figures 3(a), (b) and (c), we observe that our proposed 2D BFS algorithm with overlap incurs significantly smaller communication overheads. The communication overheads are much smaller as we scale up both the number of processes and the input scale. For example, with scale=28 and 729 processes, the communication overheads of the collective operations are almost fully overlapped. Similarly, in Figures 4(a), (b) and (c), we observe that about 70% of the communication overheads are overlapped, with larger number of processes, on ClusterB. However, we also observe that our re-designed algorithm incurs additional costs associated with the sorting operations and this leads to a slight degradation in the overall run-time.

C. Communication Overheads analysis

In this section, we take a closer look at the communication overheads with the default and our modified BFS algorithms, with 729 and 1,936 processes, on Clusters A and B, with varying input scales. In Figures 5(a) and (b), we compare the communication overheads of the two BFS algorithms. As we vary the input scales, we see that the overheads of both the collectives are significantly lower in our proposed BFS algorithm. With 1,936 processes on ClusterB, with scale=29, the communication overheads have improved by up to 70%.

In Figure 6(a) and (b), we study the communication overheads of the neighborhood collective operations, for various input scales and neighborhood-sizes, with 729 processes on ClusterA. We observe that the communication overheads marginally increase, as we increase the neighborhood-sizes. However, in Figures 1(a) and (b), we observed that the latency of our proposed network-offload based collective operations remained stable, even as we varied the neighborhood-sizes. In our 2D BFS algorithm, since there are several row and column communicators in the process grid, each network adapter card could be handling multiple concurrent Allgatherv and Alltoallv operations. Whereas, in the benchmark case, we were only doing one global Alltoallv or Allgatherv operations at any given time. Hence, we believe that as we increase the neighborhood sizes, the contention in the network could also increase.

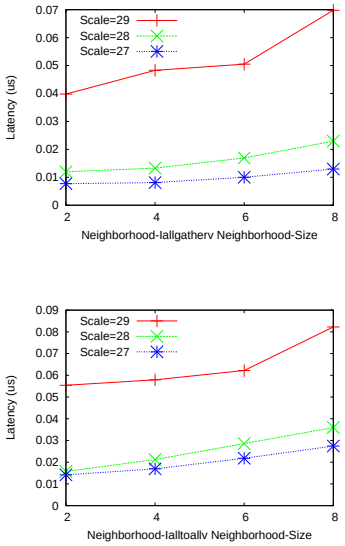

Figure 6. Communication Overheads with 729 Processes on Cluster A: (a) Neighborhood_Iallgatherv and (b) Neighborhood_Ialltoallv

D. Sort Overheads analysis

In Table II, we study the sorting overheads incurred by our proposed 2D BFS algorithm, as we vary the neighborhood-sizes of both the collectives, on ClusterB, with 1,936 processes and scale=29. We observe that the $f_{i,k}$ overhead is the lowest when the neighborhood-size of Ineighbor_Allgatherv is the smallest. As discussed in Section V, with smaller neighborhood-sizes for Allgatherv with the ring algorithm, a process may receive data from contiguous neighbors in a given round of the Neighborhood_Iallgatherv operation. In such cases, we can skip sorting the data. However, as we increase the degree, it is likely that a process may receive from a non-contiguous list of peers and we need to sort and pack the buffers, before we can proceed further. On the other hand, the sorting overheads of the $t_{ij,k}$ vectors get better as we increase the neighborhood-size of the Neighborhood_Ialltoallv operation. This is because we use the heap-merge-sort operation to sort the different segments of the $t_{ij,k}$ vectors. We also observe that the $t_{i,k}$ sorting overheads are fairly stable across various degrees. However, as we observed in Section VII-C, the communication overheads are marginally lower with smaller degrees. Hence, it is necessary to pick the right degrees.

In Figure 5(c), we evaluate the trade-offs of different sorting mechanisms on the three sort functions, with 729 Processes, scale=29 on ClusterA. As discussed in Section V, we compare the Heap-Merge sort algorithm with the basic IntegerSort algorithm, for all the three sort functions, across various neighborhood-sizes. The X-axis labels are of

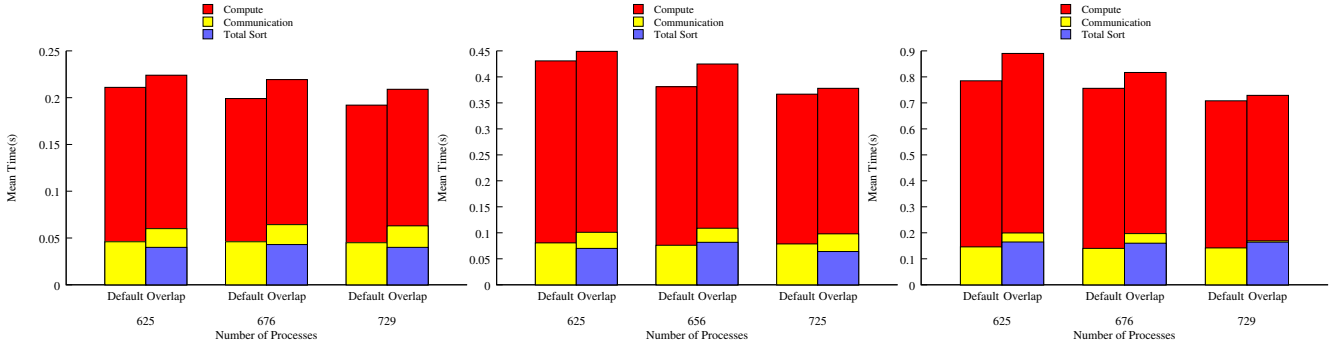

Figure 3. 2D BFS Mean Time Comparison with 729 Processes on Cluster A: (a) Scale=26, (b) Scale=27 and (c) Scale=28

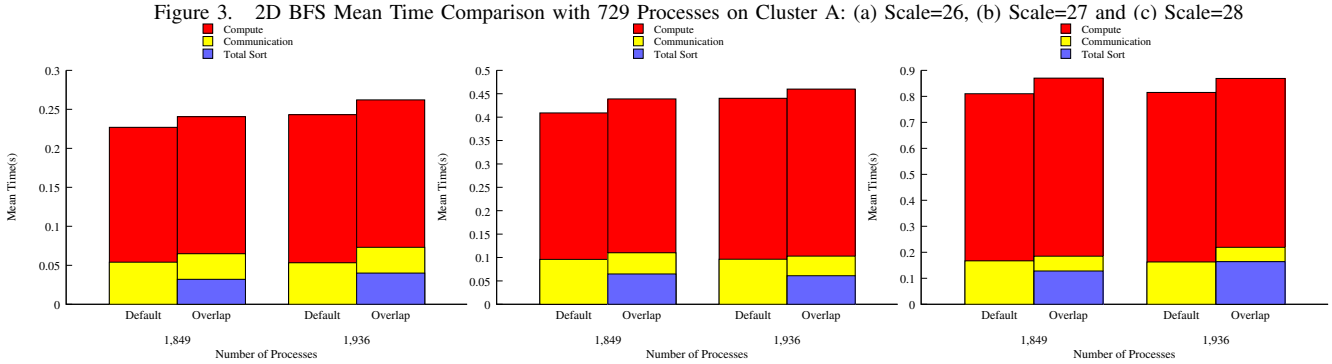

Figure 4. 2D BFS Mean Time Comparison with 1,936 Processes on Cluster A: (a) Scale=27, (b) Scale=28 and (c) Scale=29

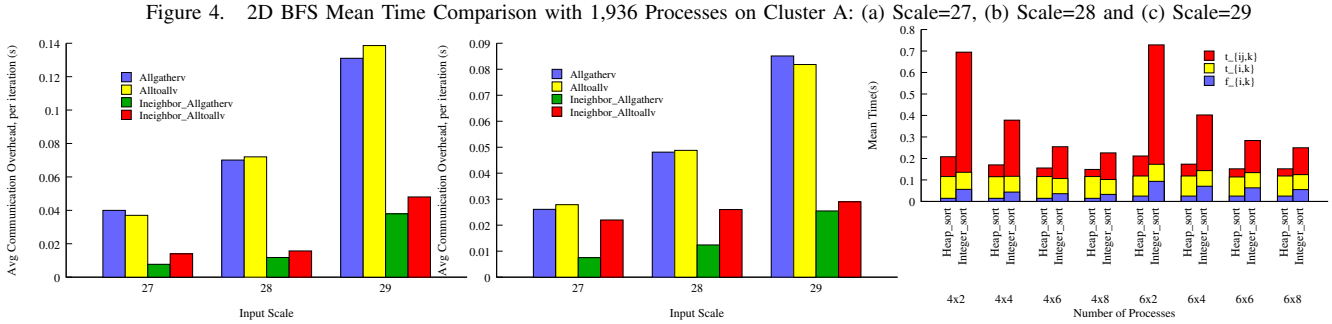

Figure 5. Communication Overheads Comparison:(a) ClusterA (729 Procs), (b) ClusterB (1,936 Procs) and (c) Sorting Overheads Comparison on ClusterA (729 Procs)

the form "axb", where a represents the neighborhood-size of Ineighbor_Allgatherv and b represents that of Ineighbor_Alltoallv. We observe that Heap-Merge sort algorithm is best suited for the $f_{i,k}$ operation and sorting the $t_{ij,k}$ vectors. However, the IntegerSort algorithm does better for sorting the $t_{i,k}$ vectors. In our proposed 2D BFS algorithm, we use a hybrid design that uses both the Heap-Merge sort and the IntegerSort algorithms.

VIII. Related work

Queue-based serial BFS algorithms [18] are very commonly used. Agarwal et al. [19] provides a shared-memory parallelization of the queue-based algorithm on multi-core architectures. Other shared-memory approaches include the lock-free bag-based implementation of Leiserson and Schardl [20], and the synchronization efficient implementation of Xia and Prasanna [21]. Bader et al. [22], proposed Multi-threaded BFS Algorithms on massively multi-threaded architectures. Algorithms and implementations for distributed memory have also been widely studied [15], [23], [24], [25]. Bhatele et al. [26] proposed heuristic based solutions for mapping irregular communication patterns on mesh topologies. Hemmert et al. demonstrate the benefits of using triggered operations and counting events provided by the Portals 4.0 message passing interface [27]. In this paper, we propose network-offload-based designs for non-blocking neighborhood Alltoallv and Allgatherv collectives and re-design the 2D BFS algorithm in CombBLAS to achieve communication/computation overlap.

IX. Conclusion

In this paper, we designed network-offload-based non-blocking, neighborhood collectives and demonstrated that the neighborhood collective operations can be composed to design generic Alltoallv and Allgatherv communication patterns. Our MPI-level designs are scalable beyond 1,936 processes and can offer near-perfect communication/computation overlap, while delivering comparable communication performance as the default host-based implementations. We studied the challenges in re-designing the 2-D

Table II
AVG SORTING OVERHEAD ANALYSIS: 1,936 PROCESSES, SCALE=29 ON CLUSTER B (SECONDS PER ITERATION)

Alltoallv-Degree	2			4			6			8		
Allgatherv-Degree	$f_{i,k}$	$t_{i,k}$	$t_{ij,k}$	$f_{i,k}$	$t_{i,k}$	$t_{ij,k}$	$f_{i,k}$	$t_{i,k}$	$t_{ij,k}$	$f_{i,k}$	$t_{i,k}$	$t_{ij,k}$
2	0.003	0.071	0.103	0.003	0.067	0.059	0.003	0.065	0.046	0.003	0.065	0.034
4	0.012	0.071	0.103	0.012	0.067	0.059	0.012	0.065	0.046	0.012	0.065	0.034
6	0.021	0.070	0.102	0.021	0.067	0.059	0.021	0.065	0.045	0.021	0.065	0.034
8	0.030	0.072	0.104	0.030	0.068	0.059	0.030	0.066	0.040	0.030	0.065	0.034

BFS algorithm in the CombBLAS library for overlapping the collective operations with compute tasks. The default BFS algorithm has a strict data dependency between the communication and compute functions. Hence, in our proposed BFS algorithm, we leverage non-blocking, neighborhood based Alltoallv and Allgatherv communication operations, to achieve fine-grained communication/computation overlap. Our experimental evaluations show that, with 1,936 processes, we are able to improve the communication overheads of the 2D BFS algorithm, by up to 70%. However, our proposed 2D BFS algorithm required additional sorting operations to ensure correctness. We explored various sorting algorithms to minimize the performance impact of these additional steps and our design uses a hybrid of the heap-merge sort algorithm and an integer-sort algorithm. We are currently exploring designs to eliminate these sorting overheads, which would lead to more than 15% benefits in the overall run-time of the 2D BFS algorithm. In particular, we are working on a complete re-design of the data structures used in the 2D algorithm that allows input and output vectors to stay unsorted.

X. ACKNOWLEDGMENTS

We would like to thank the Hyperion System administrators (hyperion-admins@llnl.gov) for their support.

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 J. L. Traeff, "Sparse Collective Operations for MPI," in *IPDPS, HIPS'09 Workshop*, 2009.

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

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

[5] I. Rabinovitz, P. Shamis, R. L. Graham, N. Bloch, and G. Shainer, "Network Offloaded Hierarchical Collectives Using ConnectX-2's CORE-direct Capabilities," in *EuroMPI*, 2010.

[6] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. R. de Supinski, and D. K. Panda, "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers," in *IPDPS*, 2012.

[7] 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," in *SC'06*, 2006.

[8] T. Hoefler, P. Gottschling and A. Lumsdaine, "Leveraging Non-blocking Collective Communication in High-performance Applications," in *SPAA'08*, 2008.

[9] A. Lumsdaine, D. Gregor, B. Hendrickson and J.W. Berry, "Challenges in Parallel Graph Processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[10] "The Graph 500 List," http://www.graph500.org.

[11] A. Buluç and J. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *IJHPCA*, vol. 25, no. 4, pp. 496–509, 2011.

[12] T. Hoefler, F. Lorenzen, and A. Lumsdaine, "Sparse Non-Blocking Collectives in Quantum Mechanical Calculations," in *Euro-PVM/MPI'08*, 2008.

[13] 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*, 2011.

[14] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/.

[15] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *SC'11*, 2011.

[16] Hyperion Linux Cluster, https://www.llnl.gov/news/.newsreleases/2008/NR-08-11-04.html/.

[17] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SDM04*, 2004.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[19] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable Graph Exploration on Multicore Processors," in *SC'10*, 2010.

[20] C.E. Leiserson and T.B. Schardl, "A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)," in *SPAA'10*, 2010.

[21] Y. Xia and V.K. Prasanna, "Topologically Adaptive Parallel Breadth-First Search on Multicore Processors," in *PDCS'09*.

[22] D.A. Bader and K. Madduri, "Designing Multi-threaded Algorithms for Breadth First Search and ST-Connectivity on the Cray MTA-2," in *ICPP'06*, 2006.

[23] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. V. Çatalyürek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in *SC'05*.

[24] N. Edmonds, J. Willcock, T. Hoefler, and A. Lumsdaine, "Design of a Large-Scale Hybrid-Parallel Graph Library," in *International Conference on HPC, Student Research Symposium*, Goa, India, 2010.

[25] G. Cong, G. Almasi and V. Saraswat, "Fast PGAS Implementation of Distributed Graph Algorithms," in *SC'10*, 2010.

[26] A. Bhatele and L.V. Kale, "Heuristic-Based Techniques for Mapping Irregular Communication Graphs to Mesh Topologies," in *Proc. of the 2011 IEEE Int'l Conf. on High Performance Computing and Communications, HPCC 2011*.

[27] 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.