Regularizing Irregularly Sparse Point-to-point Communications

Oguz Selvitopi
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
roselvitopi@lbl.gov

Cevdet Aykanat
Computer Engineering Department
Bilkent University
Ankara, Turkey
aykanat@cs.bilkent.edu.tr

ABSTRACT

This work tackles the communication challenges posed by the latency-bound applications with irregular communication patterns, i.e., applications with high average and/or maximum message counts. We propose a novel algorithm for reorganizing a given set of irregular point-to-point messages with the objective of reducing total latency cost at the expense of increased volume. We organize processes into a virtual process topology inspired by the k-ary n-cube networks and regularize irregular messages by imposing regular communication pattern(s) onto them. Exploiting this process topology, we propose a flexible store-and-forward algorithm to control the trade-off between latency and volume. Our approach is able to reduce the communication time of sparse-matrix multiplication with latency-bound instances drastically: up to 22.6× for 16K processes on a 3D Torus network and up to 7.2× for 4K processes on a Dragonfly network, with its performance getting better with increasing number of processes.

KEYWORDS

Point-to-point communications, irregular communications, process topology, virtual topology, store-and-forward, latency

ACM Reference Format:
Oguz Selvitopi and Cevdet Aykanat. 2019. Regularizing Irregularly Sparse Point-to-point Communications. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356187

1 INTRODUCTION

The time a parallel application spends in communication on distributed memory systems is affected by many factors. Apart from the underlying network topology and hardware, which can effectively provide a practical value for the latency and the bandwidth cost of transmitting a message, the computation and communication characteristics of the application are crucial for its scalability.

The communication operations may exhibit a certain degree of regularity, which one can take advantage of by realizing them via MPI collectives [3, 4, 10, 14, 17, 21] in an easy and efficient manner. For example, in stencil applications a process communicates with a well-defined set of a few other processes, or in the case of global collectives each process participates in gathering, scattering, reducing, etc. data they possess. In the presence of a high variation among the communicating processes, the communication operations become irregular and a certain subset of processes communicate with relatively more processes compared to other remaining processes. Consider a scenario where a single process communicates with more than half of the processes in the system via point-to-point (P2P) messages, while the remaining processes communicate only with a few processes. In such a scenario, this single process has the potential of rendering the whole application unscalable. Moreover, using collectives under similar scenarios may not always prove feasible in terms of efficiency. The goal of this work is to improve the performance of such scenarios, where the communication patterns are sparse and irregular, and there is a high variance among the number of processes each process communicates with.

The sparse and irregular communication operations are often manifested with a high imbalance in communicated message counts. Figure 1 illustrates three such example matrices from a sparse matrix-vector multiplication on 256 processes. In all three instances, there are a few processes that send out more P2P messages compared to other processes. This is reflected as a large difference between the average message count (indicated with a dashed line) and the maximum message count (indicated with a solid line). Such instances exhibit high overall latency and easily become latency-bound when the communicated messages have small sizes, i.e., no more than a few kilobytes.

We propose a structured way of performing sparse and irregular communication operations by organizing processes into a regular structure called virtual process topology (VPT). By utilizing this VPT, we restrict the processes that can directly communicate with each other and impose a regular communication pattern onto otherwise irregular communication operations. Our ideas for forming the VPT are inspired by the k-ary n-cube networks. The two fundamental differences between the proposed VPT and these networks are: (i) our VPT is on the software level and oblivious to the underlying networking, and (ii) the neighborhood definitions in these structures are different. We propose a novel store-and-forward algorithm to realize the communication operations for a given set of processes (along with the data they want to send) and the VPT these processes are organized into. Our methodology can be implemented as an alternative communication pattern in an MPI distribution.

The organization of processes into the proposed VPT and performing communications on this VPT enable a trade-off between the maximum message count (which is related to the total latency cost) and the communication volume (which is related to the bandwidth cost). An important parameter in forming a VPT for a given set of processes is its dimension. A low-dimensional VPT results

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356187

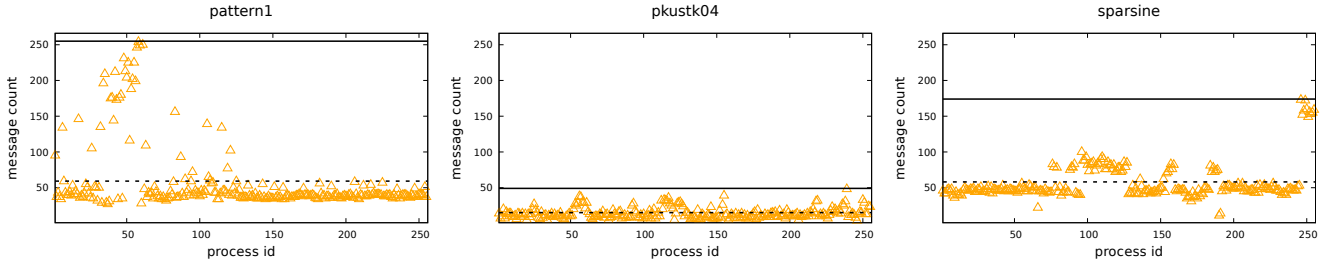


Figure 1: Message counts of 256 processes during sparse matrix-vector multiplication. The straight horizontal line is the maximum message count and the dashed line is the average message count. See Table 1 for the properties of these three matrices.

in a higher maximum message count and lower communication volume than a high-dimensional VPT. Hence, by varying the VPT dimension, our methodology is able to control the trade-off between the latency and bandwidth costs. The upper bounds on the maximum message count attained by the proposed VPT vary from linear to logarithmic complexities, which offer a wide range of choices in the control of total latency cost. The wide breadth of cases encompassed by our methodology provides a powerful mechanism to trade important performance metrics in communication to get the best performance. Our contributions are summarized as follows:

(1) We propose a novel virtual process topology to regularize sparse and irregular communication operations. We give process neighborhood definitions in this VPT and discuss how to form VPTs of various dimensions.
(2) We propose a store-and-forward algorithm to realize the communication operations on a given VPT. We describe in detail how messages are communicated in the VPT and give illustrative examples to clarify characteristics of the VPT.
(3) We analyze the proposed store-and-forward algorithm in terms of its maximum message count, communication volume, and buffer usage. We give upper bounds for each of these in order to examine different aspects of our algorithm.
(4) We test the proposed methodology on 22 latency-bound sparse matrix-vector multiplication instances that are difficult to scale. We analyze our algorithm's performance in terms of several important performance metrics and parallel runtime on up to 16K processes.

The rest of this paper is organized as follows. Section 2 introduces the terminology and states the addressed problem. In Section 3, we give our algorithm to perform communication operations on a given VPT. We analyze our algorithm's performance in terms of three important metrics in Section 4. Section 5 describes how we form the VPT. Section 6 evaluates the proposed methodology. We give our related work in Section 7 and conclude in Section 8.

2 TERMINOLOGY & PROBLEM STATEMENT

Our focus is on a distributed parallel processing setting, where there are K processes, denoted with $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$, which communicate with each other via message passing. We consider a communication scenario, in which each process has a set of messages that it needs to send to a subset of processes, denoted with $SendSet(P_i) \subseteq \mathcal{P}$. The message that needs to be sent from P_i to P_j is denoted with m_{ij}. This is a very common scenario prevalent in many parallel applications. We assume K is a power of 2, however, our methodology and algorithms can easily be extended where this is not the case. We use h, i, j, and ℓ for process indices (k is reserved for the definitions about VPT).

A VPT organizes K processes into a special structure and is characterized by its dimension, the sizes of these dimensions, and the process neighborhood definition. An n-dimensional VPT is denoted by $T_n(k_1, k_2, \ldots, k_n)$, where dimension $1 \leq d \leq n$ is of size k_d and $K = k_1 \times k_2 \times \ldots \times k_n$. We use T_n and omit the parentheses in the notation when the dimension sizes are irrelevant to the subject discussions. For dimension indices, we use c and d. Each process P_i in the VPT is identified by a vector $\langle P_i^n, P_i^{n-1}, \ldots, P_i^1 \rangle$ of n coordinates, where P_i^d is a number with radix k_d, i.e., $P_i^d \in \{1, 2, \ldots, k_d\}$. P_i and P_j have the same coordinate in dimension d if $P_i^d = P_j^d$ and they are said to be neighbors if they differ only in a single coordinate. The coordinate they differ in is the dimension they are neighbors in. Hence, P_i and P_j are neighbors in dimension d if

$$P_i^d \neq P_j^d \text{ and } P_i^c = P_j^c \text{ for } 1 \leq c \neq d \leq n.$$

In dimension d, there are a total of K/k_d process groups, each of which contains k_d processes. Hence, each process has $k_d - 1$ neighbors in dimension d and the processes in the same group differ only in the dth coordinate. We use the function $v(P_i, d)$ to denote the neighbors of P_i in dimension d:

$$v(P_i, d) = \{P_j : P_i^c = P_j^c \text{ for } 1 \leq c \neq d \leq n\}.$$

Figure 2 illustrates 64 processes organized into a VPT $T_3(4, 4, 4)$. The figure illustrates the neighbors of $P_i = \langle 3, 2, 3 \rangle$ in each dimension with different colors. The processes $P_h = \langle 3, 2, 1 \rangle$, $P_j = \langle 1, 2, 3 \rangle$, and $P_\ell = \langle 3, 4, 3 \rangle$ are neighbors of P_i in dimensions three, one, and two, respectively. The figure has links between processes shown by faded lines, which are included for discussing our method's relation to k-ary n-cube networks. These links do not indicate the actual neighborhood information.

The neighborhood definition in our VPT is quite different from the neighborhood definitions in common regular-structured applications such as stencils. Consider the neighbors of a process P_i in Figure 2 for a 7-point stencil. Each process would have two neighbors along each dimension, whose respective coordinates differ only by one from P_i. In our VPT, P_i is neighbor to all the processes that are along the same dimension with it and the coordinates of

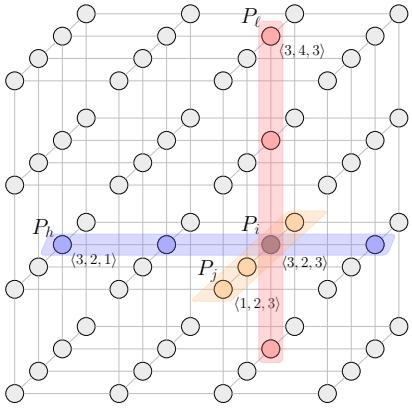


Figure 2: 64 processes organized into a 4×4×4 virtual process topology with $n = 3$, i.e., $T_3(4, 4, 4)$. The neighbors of P_i in each dimension are indicated with different colors.

these neighbors can differ by more than one. The distinct neighborhood definition in our VPT necessitates a custom communication algorithm to realize P2P messages, which we discuss in Section 3.

2.1 Virtual process topology versus k-ary n-cube networks

The way we organize the processes for communication can be considered to be similar to the topology of the k-ary n-cube networks. Our ideas are indeed motivated by the principles governing these networks, however, there are certain differences. We describe these differences in order to get a better understanding of the virtual process topology we use by relating it to a well-studied subject.

The first and foremost difference is the context where the topologies are utilized: networking is on the hardware level and the topology we form for the processes on the other hand, is virtual and on the software level. Our method organizes a number of parallel processes (i.e., MPI tasks) into a virtual structure that resembles to the structure of k-ary n-cube networks. Our method is oblivious to the underlying network of the parallel system and takes advantage of certain characteristics of a virtual topology in order to improve the communication performance.

The second crucial difference is the neighborhood definition. In k-ary n-cube networks, two distinct nodes P_i and P_j are neighbors in dimension d if (i) $P_i^d \neq P_j^d$, (ii) $P_i^c = P_j^c$ for $1 \leq c \neq d \leq n$, and (iii) $|P_i^d - P_j^d| = 1$ or $|P_i^d - P_j^d| = k_d - 1$ (including the wrap-around links). In our VPT, however, the neighborhood definition is relaxed by involving only the first two cases, hence, the k_d processes in each of K/k_d groups are all neighbors. In a k-ary n-cube network, each of the K/k_d groups is a 1D torus, whereas in our virtual process topology, the nodes in each of these groups are "completely connected" in terms of networking. Therefore, in a k-ary n-cube network, each node has two neighbors in dimension d, while in our process topology each process has $k_d - 1$ neighbors. We should note that the neighborhood definition of VPT $T_{\lg_2 K}(2, \ldots, 2)$ becomes equivalent to the neighborhood definition of 2-ary $\lg_2 K$-cube networks (i.e., hypercubes). Figure 3 illustrates the neighbor processes in three different dimensions of a VPT $T_3(4, 4, 4)$.

The last difference is the sizes of the dimensions. In k-ary n-cube networks, the size of each dimension is the same and equal to k, and

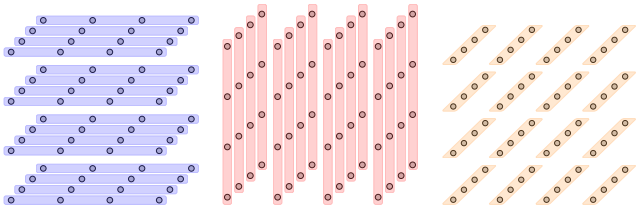


Figure 3: The neighborhood of communication operations executed in 3 stages.

there are a total of k^n nodes. In our VPT, the sizes of the dimensions can be different and there are a total of $K = \prod_d k_d$ processes. This provides more freedom in the organization of processes and it is important because in the software level having each dimension the same size can be too restrictive. In this sense, our method allows more irregular topologies as the processes for a certain VPT dimension n can be organized in different ways.

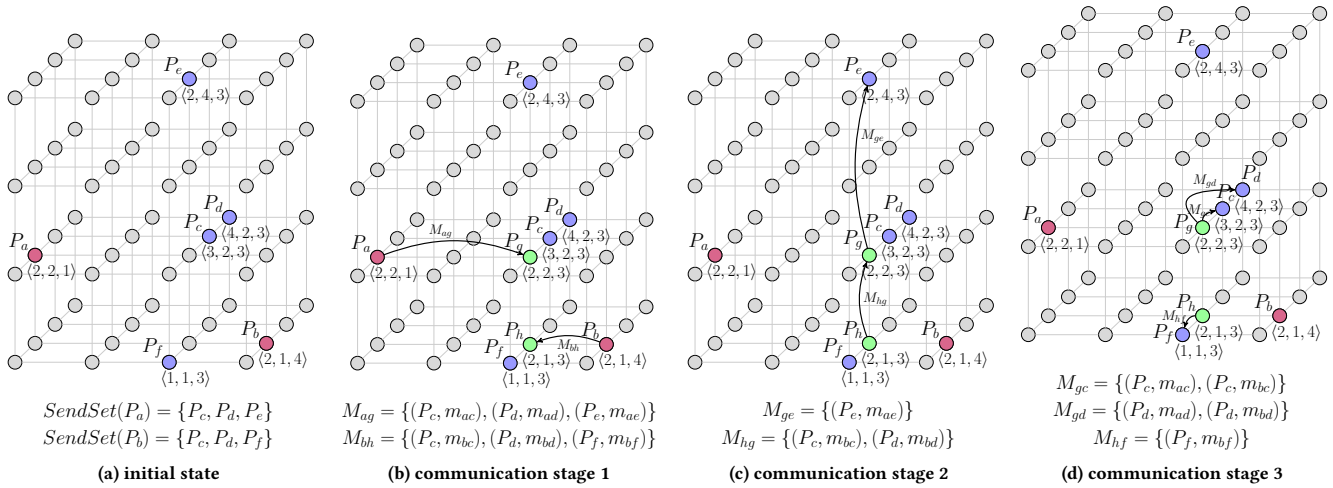
2.2 Addressed problem

As mentioned earlier, we consider a scenario where a set $\mathcal{P}$ of processes are involved in communicating with each other. A straightforward and common approach is to assume no structure in the process topology and allow each process to communicate with each other, i.e., each P_i simply sends a P2P message to the processes in $SendSet(P_i)$. We address the problem of improving the communication performance of this scenario by assuming the described VPT. As opposed to the straightforward approach, using a VPT T_n with $n > 1$ disables the direct communication between certain processes and necessitates a methodology based on storing and forwarding messages. In other words, some processes need to communicate *indirectly* with the help of the processes they can directly communicate with. We propose an algorithm to perform the communications between processes under the described process topology, sketch the details of the proposed algorithm, analyze its performance and discuss certain implementation issues.

We consider this as a black-box operation called by each process, which simply provides their data to be sent along with the VPT. Instead of using a pair of primitives such as `MPI_Send`/`MPI_Recv` or `MPI_Put`/`MPI_Get`, each process passes their data and processes they want to communicate this data to a procedure, which then handles the communication by taking the process topology into account using the same primitives. The described topology actually encompasses the case where each process is allowed to directly communicate with any other process. This is the case with T_1, i.e., there is a single dimension and each process is neighbor to all other processes. In that sense, our algorithm generalizes how processes can communicate in a structured manner and on the extreme end where there is no structure, it corresponds to being able to directly send out P2P messages to any process.

3 A STORE-AND-FORWARD ALGORITHM

In an n-dimensional VPT T_n, the communication between processes is executed in n stages. The communication operations for a process P_i in stage $1 \leq d \leq n$ start after P_i receives all its messages from the previous communication stages. Without loss of generality, we assume that the communications related to the first dimension are



(a) initial state (b) communication stage 1 (c) communication stage 2 (d) communication stage 3

Figure 4: Three stages of communication in a VPT $T_3(4, 4, 4)$ for $SendSet(P_a) = \{P_c, P_d, P_e\}$ and $SendSet(P_b) = \{P_c, P_d, P_f\}$. The source and destination processes are respectively illustrated in red and blue. Since P_a and P_b cannot directly communicate with the processes in their $SendSet$s, their messages are forwarded via their neighbors, which are illustrated in green.

executed in the first stage, the ones related to the second dimension are executed in the second stage, etc. We restrict the processes that can communicate with each other in a stage. In stage d, each process P_i is allowed to communicate only with its $k_d - 1$ neighbors in dimension d, i.e., the processes in $v(P_i, d)$. P_i may or may not communicate with these neighbors depending on what it has to send in its buffers. Since there may be processes in $SendSet(P_i)$ that are not neighbors of P_i in any dimension, for sending data to such processes P_i needs the help of the other processes. This necessitates a *store-and-forward* scheme, in which the messages that need be communicated between non-neighbor processes are stored and forwarded by a well-defined set of intermediate processes.

Before describing the complete algorithm, we first focus on how a single message is communicated between two processes. Consider a message m_{ij} with its source P_i and destination P_j. Depending on where these two processes are in the VPT, m_{ij} may need to visit multiple hops before reaching P_j. To communicate m_{ij}, P_i checks its coordinate in the first dimension, P_i^1, and if it is different than P_j^1, it forwards this message to one of its neighbors in the first dimension, which is the process with the coordinates $\langle P_i^n, \ldots, P_i^2, P_j^1 \rangle$. Otherwise, i.e., if $P_i^1 = P_j^1$, P_i keeps the message because it does not need to communicate it in this stage. Let P_ℓ be the process that has m_{ij} after the communication in the first stage completes (which is either P_i or $\langle P_i^n, \ldots, P_i^2, P_j^1 \rangle$). P_ℓ then repeats the same process for $d = 2$, and either forwards or stores m_{ij}. This is repeated until m_{ij} reaches its destination. Hence, at stage d, the process that has m_{ij}, P_ℓ, has to decide whether to forward m_{ij} by comparing its dth coordinate with the dth coordinate of the destination process:

$$\text{forward } m_{ij} \text{ to } \langle P_\ell^n, \ldots, P_\ell^{d+1}, P_j^d, P_\ell^{d-1}, \ldots, P_\ell^1 \rangle \text{ if } P_\ell^d \neq P_j^d,$$
$$\text{store } m_{ij} \text{ if } P_\ell^d = P_j^d.$$

In other words, if P_ℓ and P_j differ in coordinate d, P_ℓ forwards m_{ij} to its neighbor in dimension d whose dth coordinate is the same with that of P_j. Otherwise, it does not communicate the message in this stage. By this logic, it can be easily seen that m_{ij} is forwarded in stage d if $P_i^d \neq P_j^d$, and the number of times it gets forwarded is equal to $|\{d : P_i^d \neq P_j^d\}|$, i.e., the number of coordinates they differ in, which is the Hamming distance. This process of communicating a message in the VPT can be considered to be similar to the e-cube routing for hypercubes [20], and more generally to dimension-ordered deterministic routing.

Consider a process P_i and its $SendSet(P_i)$. Let the coordinates of the processes in $SendSet(P_i)$ differ from those of P_i only after dth coordinate, i.e., $\{P_\ell \in SendSet(P_i) : P_i^n \neq P_\ell^n, \ldots, P_i^d \neq P_\ell^d, P_i^{d-1} = P_\ell^{d-1}, \ldots, P_i^1 = P_\ell^1\}$. All the relevant messages in this scenario have to be first communicated to some neighbor of P_i in dimension d, say P_j, in order to reach their destination. Hence, the communication between P_i and P_j is actually a message that contains a list of smaller messages, which we refer to as *submessages*. Each submessage is simply a two-tuple that consists of the id of the destination process and the message destined for that process. Note that there is a *single* actual message that is communicated between P_i and P_j, but it contains a number of submessages that will be sorted out by P_j. In fact, P_j may also receive messages from its other neighbors in dimension d, which may inherently contain other submessages, and it may also possess submessages from the messages that are received in previous communication stages. To make a distinction between a message and a submessage, we denote the direct message between P_i and P_j with M_{ij}, and the submessage with source P_i and destination P_j as (P_j, m_{ij}). The submessages are always contained within messages, and these messages are communicated between neighbors in distinct stages of the communication.

Figure 4 illustrates two processes $P_a = \langle 2, 2, 1 \rangle$ and $P_b = \langle 2, 1, 4 \rangle$ that need to send data to $SendSet(P_a) = \{P_c, P_d, P_e\}$ and $SendSet(P_b) = \{P_c, P_d, P_f\}$, respectively. Both processes need to send their data with the help of their neighbors in the first communication stage. For P_a this neighbor is $P_g = \langle 2, 2, 3 \rangle$ and for P_b it is $P_h = \langle 2, 1, 3 \rangle$. Observe that the message M_{ag} sent from P_a to P_g consists of three submessages (P_c, m_{ac}), (P_d, m_{ad}), and (P_e, m_{ae}). The message M_{bh} sent from P_b to P_h also consists of three submessages

Algorithm 1: Store-and-Forward

Input: P_i, $SendSet(P_i)$, $T_n(k_1, k_2, \ldots, k_n)$
Output: List L of messages

▷ Initialize buffers
1 Let fwbuf be a list of size n
2 **for** $d \leftarrow 1$ **to** n **do**
3 Let fwbuf$[d]$ be a list of size k_d

▷ Process my send list
4 **for** $P_j \in SendSet(P_i)$ **do**
5 $d \leftarrow \text{argmin}_{c \le n}\, P_i^c \neq P_j^c$
6 fwbuf$[d][P_j^d] \leftarrow$ fwbuf$[d][P_j^d] \cup (P_j, m_{ij})$

▷ Communication in d stages
7 **for** $d \leftarrow 1$ **to** n **do**
8 Allocate stbuf to receive messages in this stage
  ▷ Send out messages to my neighbors
9 **for** $P_j \in \nu(P_i, d)$ **do**
10 **if** fwbuf$[d][P_j^d] \neq \emptyset$ **then**
11 Form M_{ij} from the submessages in fwbuf$[d][P_j^d]$
12 Send M_{ij} to P_j
13 Wait for messages from my neighbors
  ▷ Process received messages
14 **for** $M_{ji} \in$ stbuf **do**
  ▷ Scatter the submessages into their buffers
15 **for** $(P_\ell, m_{h\ell}) \in M_{ji}$ **do**
16 $c \leftarrow \text{argmin}_{d < c' \le n}\, P_i^{c'} \neq P_\ell^{c'}$
17 fwbuf$[c][P_\ell^c] \leftarrow$ fwbuf$[c][P_\ell^c] \cup (P_\ell, m_{h\ell})$

▷ Gather messages that belong to P_i
18 $L = \emptyset$
19 **for** $d \leftarrow 1$ **to** n **do**
20 $L \leftarrow L \cup$ fwbuf$[d][P_i^d]$
21 **return** L

(P_c, m_{bc}), (P_d, m_{bd}), and (P_f, m_{bf}). After these messages are received by P_g and P_h, they process the submessages in them to determine which stage they will be forwarded in. For P_g, the submessage (P_e, m_{ae}) will be forwarded in the second stage while the submessages (P_c, m_{ac}) and (P_d, m_{ad}) will be forwarded in the third stage. For P_h, the submessages (P_c, m_{bc}) and (P_d, m_{bd}) will be forwarded in the second stage while the submessage (P_f, m_{bf}) will be forwarded in the third stage. Another important point to note is that a message sent by a process can contain submessages that it received in earlier stages. For instance, the message M_{gc} sent from P_g to P_c in the third stage consists of submessages that P_g received from P_a in the first stage and P_h in the second stage.

Algorithm 1 presents a high-level description of the proposed store-and-forward communication scheme for a given VPT T_n. The algorithm focuses on the operations as performed by $P_i \in \mathcal{P}$. P_i first initializes its buffers in lines 1-3. The buffer fwbuf$[d][x]$ holds the submessages that will be forwarded in stage d to neighbor of P_i whose coordinate d is equal to x, i.e., $P_j^d = x$. Then, P_i traverses the processes in $SendSet(P_i)$ and concatenates each submessage into the respective buffer (lines 4-6). The first stage that m_{ij} will be communicated is given by the index of the smallest coordinate that P_i and P_j differ in. The communication operations are performed in d stages between lines 7-17. In stage d, P_i communicates with a subset of its neighbors in $\nu(P_i, d)$ and examines the submessages in received messages. Using the buffers corresponding to stage d, P_i sends out a message to each P_j if the respective buffer is not empty (lines 9-12). It then examines the received messages from its neighbors in $\nu(P_i, d)$. For each received message M_{ji}, it examines the submessages in them by finding which communication stage they will be forwarded in and putting them in their respective locations in fwbuf (lines 14-17). For a submessage $(P_\ell, m_{h\ell})$ received in stage d, the stage it will be forwarded is given by the smallest coordinate greater than d that P_i and P_ℓ differ in. The data that belongs to P_i is given by the submessages in the locations fwbuf$[d][P_i^d]$ for $1 \le d \le n$ (lines 18-21).

In Algorithm 1, when two submessages $(P_\ell, m_{i\ell})$ and $(P_\ell, m_{j\ell})$ that originate from distinct processes P_i and P_j but destined for the same process P_ℓ arrive at an intermediate process P_h, they will be put into the same forward buffer and in the rest of the algorithm they will always be communicated within the same messages. Dual of this case, when two submessages (P_j, m_{ij}) and $(P_\ell, m_{i\ell})$ that originate from the same process P_i but destined for the distinct processes P_j and P_ℓ arrive at an intermediate process P_h, they will be put into different forward buffers and in the rest of the algorithm they will always be communicated within distinct messages.

In a certain communication stage, the processing of submessages in the received messages involves putting each submessage in its respective buffer from which it will be forwarded in later stages. In this operation, the submessages in a received message are scattered across multiple forward buffers. This operation is illustrated in Figure 5. A buffer that will be used for communication in stage d may be filled with the submessages from any stage earlier than d. After a buffer is used for communication in stage d, it cannot be further filled with the submessages that arrive in further stages.

It is interesting to examine certain cases in the proposed VPT. For a given number of processes K, where K is a power of 2, if we use $n = 1$, Algorithm 1 simply corresponds to each process communicating with each other directly. This is no different than P_i sending a direct P2P message to each process in $SendSet(P_i)$. Since $k_d > 1$ for each dimension, the largest T_n we can get is $n = \lg_2 K$, and $k_d = 2$ for all d. In this case, each process can communicate with exactly one process in each stage. In network terminology, this is equivalent to hypercubes, where each node is connected to exactly one node in each dimension.

4 ANALYSIS

The end goal of using a virtual process topology is to provide a flexible medium where one can easily achieve a trade-off between the total latency (i.e., message count) and the bandwidth (i.e., the amount of data communicated) costs by controlling the dimension of the topology. For a fixed number of processes, increasing the VPT dimension in the proposed store-and-forward scheme increases the number of times a submessage gets forwarded. On the other hand, since $K = k_1 \times k_2 \times \ldots k_n$, increasing the VPT dimension for a fixed number of processes will result in smaller dimension sizes,

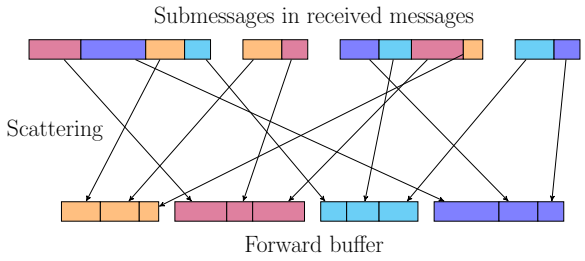


Figure 5: Scattering of submessages in received messages to their corresponding buffers.

which means fewer messages communicated at each stage. Hence, a low-dimensional VPT in our method results in higher total latency and lower bandwidth cost compared to a high-dimensional VPT. In this section, we analyze the performance of the proposed store-and-forward scheme in terms of (i) maximum message count, (ii) volume of communicated data, and (iii) buffer sizes. We analyze the worst-case scenario where each process has data to send to every other process, i.e., $|SendSet(P_i)| = K - 1$ for each P_i. For our discussions, we assume that $k_1 = k_2 = \ldots = k_n = k$ and each process needs to send the same amount of data s. Note that under these assumptions $K = k^n$ and for a fixed value of K the greatest value that n can get is $\lg_2 K$, which occurs when $k = 2$.

In a communication stage d, each process can talk to its $k - 1$ neighbors. Hence, the maximum message count at any stage is equal to $k - 1$. Since there are n stages, the maximum message count in the overall store-and-forward scheme is $n(k - 1)$. If we have a single stage of communication (i.e., $n = 1$), the maximum message count is $K - 1$, i.e., $O(K)$. Keeping the number of processes fixed, for $n = 2$, we get a maximum message count of $2(\sqrt{K} - 1)$, which is $O(K^{1/2})$ and asymptotically smaller than $O(K)$. For $n = 3$, this reduces to $O(K^{1/3})$, and etc. On the most extreme case, where we have $k = 2$ and $n = \lg_2 K$, the maximum message count reduces to $O(\lg_2 K)$. Hence, by varying the VPT dimension for a specific number of processes, it is possible to obtain a wide spectrum of different asymptotic bounds on the maximum message count, and hence on the total latency cost. These bounds range between linear and logarithmic complexities, with a wide range of sub-linear complexities between them.

Compared to directly communicating messages between processes, our store-and-forward scheme with $n > 1$ can increase the communication volume as the submessages need to be forwarded. We focus on the volume of data needed to communicate the messages that initially originate from P_i, i.e., m_{ij} for $P_j \in \mathcal{P} - \{P_i\}$. As all processes are assumed to have the same *SendSet* in our analysis, the analysis performed for P_i applies to all processes. In the case of direct communication (i.e., a VPT of dimension 1), the communication volume due to P_i is equal to $V = s(K - 1)$, where each message has the same size s. A loose upper bound on volume can easily be obtained by assuming each submessage gets forwarded in every stage, which gives nV for a VPT of dimension n. However, it is possible to derive the exact number of times a message gets forwarded and hence find exact communication volume incurred in the store-and-forward scheme. A submessage with source $P_i = \langle P_i^n, P_i^{n-1}, \ldots, P_i^1 \rangle$ and destination $P_j = \langle P_j^n, P_j^{n-1}, \ldots, P_j^1 \rangle$ gets forwarded by the number of coordinates these two processes differ in. Hence, since we assume P_i has data to send to every other process, we can count how many coordinates P_i differs from each of them. There are $(k-1)^\ell \binom{n}{\ell}$ processes that differ by $1 \le \ell \le n$ coordinates from P_i. The submessages destined for these $(k-1)^\ell \binom{n}{\ell}$ processes gets forwarded ℓ times. Hence, the volume incurred in forwarding submessages of P_i is:

$$V = s \sum_{\ell=1}^{n} (k-1)^\ell \binom{n}{\ell} \ell.$$

For all processes, this quantity is simply multiplied by the number of processes K. If we compare the loose upper bound and this quantity, for example for $K = 256$ and T_4, while the ratio between the loose bound and the volume in direct communication is 4, the ratio between the derived quantity and the volume in direct communication is 3.01. For T_8 these two values are 8 and 4.02, and for T_2 they are 2 and 1.88.

We next analyze the buffer size requirements of the processes in the store-and-forward scheme. In the worst-case scenario, initially, each process has a submessage for every other process. Hence, there are a total of $K(K-1)$ submessages, each of size s. The submessages at P_i after the communication at stage d completes are given by each submessage with source P_j and destination P_ℓ that satisfies the following two conditions: (i) the first d coordinates of the destination process P_ℓ are equal to the first d coordinates of P_i and (ii) the last $n - d$ coordinates of the source process P_j are equal to the last $n - d$ coordinates of P_i. The submessages in P_i's buffers at the beginning of stage d are given by

$$\begin{aligned}\{(P_\ell, m_{j\ell}) : & P_\ell \in SendSet(P_j) \text{ and} \\ & P_\ell^1 = P_i^1, \ldots, P_\ell^{d-1} = P_i^{d-1} \text{ and} \\ & P_j^d = P_i^d, \ldots, P_j^n = P_i^n\}.\end{aligned}$$

In the latter condition, there are k^d processes whose last $n - d$ coordinates are equal to the last $n - d$ coordinates of P_i. Regarding the former condition, there are k^{n-d} submessages whose destination's first d coordinates are equal to the first d coordinates of P_i. Multiplying these two quantities, we get $k^d k^{n-d} = k^n = K$ submessages at process P_i after stage d completes. Since P_i does not send a message to itself in stage d, there is one fewer submessage, i.e., $K - 1$. Therefore, the buffer size required at any communication stage at a process is bounded by $s(K - 1)$. Observe that when we multiply the number of submessages present at P_i, $K - 1$, with the number of processes K, we get the total number of submessages $K(K - 1)$ being shuffled around the VPT at any communication stage.

5 FORMING VIRTUAL PROCESS TOPOLOGY

For a given number of processes, there are several ways to organize them into the virtual topology described earlier. There are two important parameters in this respect: the dimension of the VPT, i.e., n, and the organization of processes in that VPT, i.e., how we determine $k_1, k_2, \ldots, k_n$ for a given n. In Section 4, we explained how we can attain a trade-off between maximum message count and communication volume by varying n. These discussions assumed each dimension has the same size. This is not required by our

algorithm and the proposed store-and-forward scheme allows sizes of these dimensions to be different from each other.

Recall that we assume K to be a power of two. For a given K and n, we have $k_1 \times k_2 \times \ldots \times k_n = K$ and $k_d > 1$ for $1 \leq d \leq n$. Note that since K is a power of two, each of the n dimensions is also a power of two. To keep the maximum message count as small as possible, the values $k_1, k_2, \ldots, k_n$ should be close to each other. The maximum message count in communication stage d is $k_d - 1$ and in all stages this makes up to $\sum_d (k_d - 1)$. Keeping this observation in mind, we propose a simple scheme to form a VPT that is optimal in terms of maximum message count.

For a given K, the values n can take range from 1 to $\lg_2 K$. For the first ($\lg_2 K \mod n$) dimensions, we set their sizes to $2^{\lfloor \lg_2 K/n \rfloor + 1}$. For the remaining $n - (\lg_2 K \mod n)$ dimensions, we set their sizes to $2^{\lfloor \lg_2 K/n \rfloor}$. This scheme distributes the processes among the dimensions as close as possible and ensures that no two dimension sizes differ by more than a factor of 2. Hence, it provides the best attainable overall maximum message count.

Although we aim to attain the lowest upper bound on the maximum message count in the VPT formation, this may not be always desirable since it is likely to cause more forwarding. For a fixed VPT dimension, it is also possible to achieve a trade-off between message count and volume by varying how we set the size of each dimension. If we distribute the processes in such a way that each dimension ends up with close sizes (like the scheme above), we attain good maximum message count at the expense of increasing the likelihood of messages getting forwarded. On the other hand, if the dimension sizes have high variance, this results in worse maximum message count but it decreases the likelihood of messages getting forwarded. This is because a process in the former case has fewer neighbors than it has in the latter case. We do not explore this trade-off in our work since we can already obtain a similar trade-off by adjusting the VPT dimension.

6 EXPERIMENTS

6.1 Setup

We test the proposed store-and-forward scheme within a distributed sparse matrix vector multiplication (SpMV). We chose SpMV for our tests because it is a very common kernel. Moreover, testing SpMV allows us to evaluate our algorithm's impact in a realistic setting. We utilize a row-parallel SpMV algorithm, which consists of a communication phase followed by a computational phase: the processes first communicate the input vector elements by sending/receiving P2P messages and then they compute the output vector elements through local SpMV operations. The proposed approach is not restricted to any kind of partitioning and it is basically applicable to any scenario where a number of processes interchange P2P messages. The communication phase is implemented in two different ways:

(1) **BL**: The baseline scheme in which the P2P messages are exchanged without any regularization. In other words, processes plainly issue simple sends and receives and do not aim to do anything specific to address latency. This corresponds to the VPT T_1.

(2) **STFW**: The proposed store-and-forward scheme in which the communication operations are realized via VPTs of dimension $n > 1$. For a given number of processes K, since our algorithm embodies $\lg_2 K - 1$ different VPT dimensions (excluding T_1, which corresponds to the baseline), we use a suffix to indicate this. As a result, we use STFWn to abbreviate our scheme, where n is the VPT dimension $1 < n \leq \lg_2 K$.

Table 1: Properties of the matrices used in the experiments.

		Number of		Row/column degree		
Matrix	Kind	rows/cols	nonzeros	max	cv	maxdr
cbuckle	structural mechanics	13 681	676 515	600	0.16	0.044
msc10848	structural eng.	10 848	1 229 778	723	0.42	0.067
fe_rotor	undirected graph	99 617	1 324 862	125	0.29	0.001
sparsine	structural eng.	50 000	1 548 988	56	0.36	0.001
coAuthorsDBLP	co-author network	299 067	1 955 352	336	1.50	0.001
net125	optimization	36 720	2 577 200	231	0.95	0.006
nd3k	2D/3D problem	9 000	3 279 690	515	0.26	0.057
GaAsH6	chemistry problem	61 349	3 381 809	1646	2.44	0.027
pkustk04	structural eng.	55 590	4 218 660	4230	1.46	0.076
gupta2	linear programming	62 064	4 248 286	8413	5.20	0.136
TSOPF_FS_b300_c2	power network	56 814	8 767 466	27742	6.23	0.488
pattern1	optimization	19 242	9 323 432	6028	0.78	0.313
SiO2	chemistry problem	155 331	11 283 503	2749	4.05	0.018
human_gene2	gene network	14 340	18 068 388	7229	1.09	0.504
coPapersCiteseer	citation network	434 102	32 073 440	1188	1.37	0.003
mip1	optimization	66 463	10 352 819	66395	2.25	0.999
TSOPF_FS_b300_c3	power network	84 414	13 135 930	41542	7.59	0.492
crankseg_2	structural eng.	63 838	14 148 858	3423	0.43	0.054
Ga41As41H72	chemistry problem	268 096	17 488 476	702	1.53	0.003
bundle_adj	computer vision prb.	513 351	20 208 051	12588	6.37	0.025
F1	structural eng.	343 791	26 837 113	435	0.52	0.001
nd24k	2D/3D problem	72 000	28 715 634	520	0.19	0.007

cv: coefficient of variation. **maxdr**: maximum degree ratio (i.e., max degree divided by the number of rows/columns).

The test matrices are row-wise partitioned by using PaToH [5]. Using a partitioner reduces the communication overheads in SpMV, which is a common technique to improve the parallel performance. We test for five different number of processes, $K \in \{32, 64, 128, 256, 512\}$. For STFW, this implies $2 \leq n \leq 9$. For a different set of experiments involving large-scale communication analysis in Section 6.5, we utilize a 4K, 8K, and 16K processes. All codes are implemented in C and use MPI for communication.

For parallel runs, we use a BlueGene/Q system, on which a node consists of 16 PowerPC A2 CPUs with 1.6 GHz clock frequency and 16 GB memory. The nodes of this system are connected with 5D torus chip-to-chip network. We also evaluate the communication time performance of BL and STFW in Section 6.4 on a Cray XC40 system, in which a node consists of two 16-core Intel Haswell CPUs with 2.3 GHz clock frequency and 128 GB memory. The nodes in this system are connected with Cray Aries interconnect in Dragonfly network topology. For large-scale communication analysis in Section 6.5, we use this system and yet another system - a Cray XK7 machine with a 3D torus network and Cray Gemini interconnect. A node in the latter system consists of a single AMD Opteron Interlagos CPU with 2.2 GHz clock frequency and 32 GB memory. The parallel runtimes reported in the following sections are the averages of 100 SpMV iterations.

The sparsity pattern of the matrix has a considerable effect on the characteristics of the communication. Since the proposed algorithm is tailored for latency-bound communications, we select a subset of symmetric matrices that are expected to reflect this in BL. Such matrices often have dense rows/columns and they are quite irregular. A combination of these factors causes high latency. For our evaluations in Sections 6.2, 6.3, 6.4, we utilize the top 15 test matrices in Table 1. For the large-scale communication analysis

Table 2: Comparison of schemes in terms of six different metrics and four different process counts.

K	Scheme	mmax	mavg	vavg	time (usec) comm	time (usec) SpMV	buffer size (KB)
64	BL	44.3	22.9	2105	573	1479	34.2
	STFW2	13.3	10.4	3150	375	1360	62.6
	STFW3	8.9	7.6	3879	366	1342	60.3
	STFW4	7.9	6.9	4218	323	1320	55.5
	STFW5	7.0	6.3	4569	330	1328	55.3
	STFW6	6.0	5.5	5022	287	1304	53.9
128	BL	73.9	34.6	1578	674	1099	25.9
	STFW2	20.8	15.5	2332	415	877	49.4
	STFW3	12.9	10.9	2916	375	866	47.9
	STFW4	10.0	8.6	3345	367	869	46.9
	STFW5	9.0	8.0	3554	378	881	44.1
	STFW6	8.0	7.3	3852	328	889	41.8
	STFW7	7.0	6.3	4249	303	860	41.7
256	BL	120.5	50.2	1181	825	1091	20.1
	STFW2	26.5	18.8	1844	439	681	38.8
	STFW3	16.5	13.4	2279	386	631	38.3
	STFW4	11.9	10.1	2736	359	608	37.9
	STFW5	11.0	9.5	2848	383	649	36.1
	STFW6	10.0	8.8	3082	334	632	33.6
	STFW7	9.0	7.9	3336	329	622	34.5
	STFW8	8.0	7.2	3544	322	636	32.3
512	BL	187.6	65.7	872	1223	1349	16.1
	STFW2	41.6	28.0	1348	492	609	31.3
	STFW3	20.1	15.5	1785	395	531	30.9
	STFW4	15.9	13.5	2029	383	522	29.5
	STFW5	13.0	10.8	2257	386	526	30.4
	STFW6	12.0	10.6	2358	368	513	28.4
	STFW7	11.0	9.8	2495	390	531	27.6
	STFW8	10.0	9.0	2682	348	500	26.6
	STFW9	9.0	8.0	2906	338	477	25.0

mmax: maximum message count. **mavg**: average message count. **vavg**: average volume (words).

in Section 6.5, we utilize the bottom 10 matrices in Table 1 (i.e., matrices with more than 10 million nonzeros). All matrices are from SuiteSparse Matrix Collection [8]. The coefficient of variation (cv) in the table captures how irregular the matrix is. The maximum degree (max) and the ratio of the maximum row/column degree to the total row/column count (maxdr) indicate how likely the matrix has a dense row/column.

6.2 Analysis of Performance Metrics

We analyze the behavior of the proposed algorithm in terms of six performance metrics in Table 2: maximum message count (mmax), average message count (mavg), average volume (vavg), the communication time, parallel SpMV time, and buffer size. The values reported for a metric in the table are the geometric averages of the values obtained in that metric for all 15 test matrices. The first two metrics are particularly addressed by STFW, and it is crucial to achieve improvements in both compared to BL to reduce the total latency cost. The third metric, average volume, is also important as STFW causes an increase in it compared to BL. The unit of volume is a word. The maximum message count is in terms of number of messages sent by individual processes. The communication and parallel SpMV time metrics (both taken on BlueGene/Q) reflect

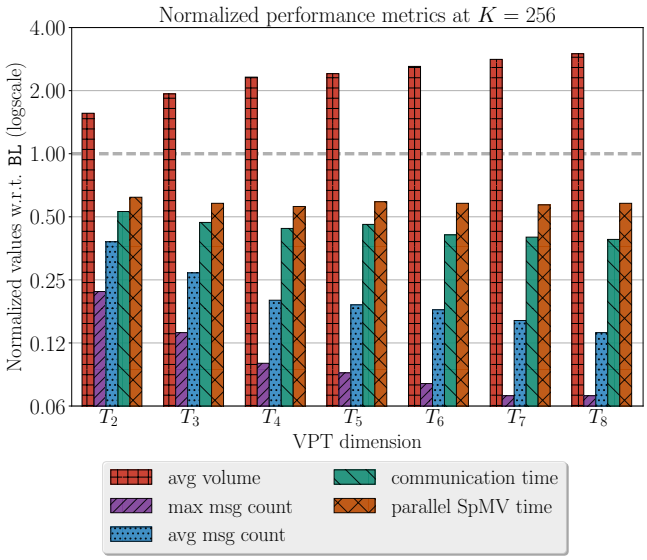


Figure 6: Values in various performance metrics of different VPT dimensions for STFW normalized with respect to the values obtained by BL at $K = 256$. A value $y > 1$ in the figure indicates BL is y times better than STFW, whereas a value $y < 1$ indicates STFW is $1/y$ times better than BL.

whether the proposed algorithm works in practice. We also illustrate these metrics for $K = 256$ in Figure 6. The values obtained by STFW schemes are normalized with respect to those of BL in the figure and they are plotted for each different VPT dimension. The last metric in Table 2 is the buffer size in kilobytes and it includes the sizes of the buffers used to send and receive original messages.

Our algorithm attains drastic improvements in two latency metrics as seen in Table 2. For $K = 64, 128, 256$, and 512, STFW respectively achieves 3.3×–7.4×, 3.6×–10.6×, 4.6×–15.1×, and 4.5×–20.9× smaller maximum message counts compared to BL. STFW also improves the average message count. Although improvements in this metric are not as high as they are in the maximum message count, it still achieves 3×–8× smaller average message counts in the highest VPT dimension at each process count compared to BL. As the VPT dimension gets higher for a specific K, the improvements in these two metrics get higher as STFW tackles the latency overhead more aggressively with increasing VPT dimension. Observe that the difference between the maximum and the average message count decreases with VPT dimension. This is because spreading the communicated messages into more dimensions increases the chances of a message to be forwarded in more stages by reducing the number of directly communicating processes. For example at $K = 64$, while a process can directly communicate with $2(8-1) = 14$ processes for $T_2(8, 8)$, it can only directly communicate with $6(2-1) = 6$ processes for $T_6(2, 2, 2, 2, 2, 2)$. Recall that for a VPT dimension of n, the maximum message count is bounded by $\sum_d k_d - 1$, which is also verified in the table.

The improvements of STFW in latency metrics come at the expense of larger volume values. This is expected as favoring the latency cost metrics at the expense of the bandwidth cost metrics is the gist of our approach. STFW incurs 1.5×–2.4×, 1.5×–2.7×,

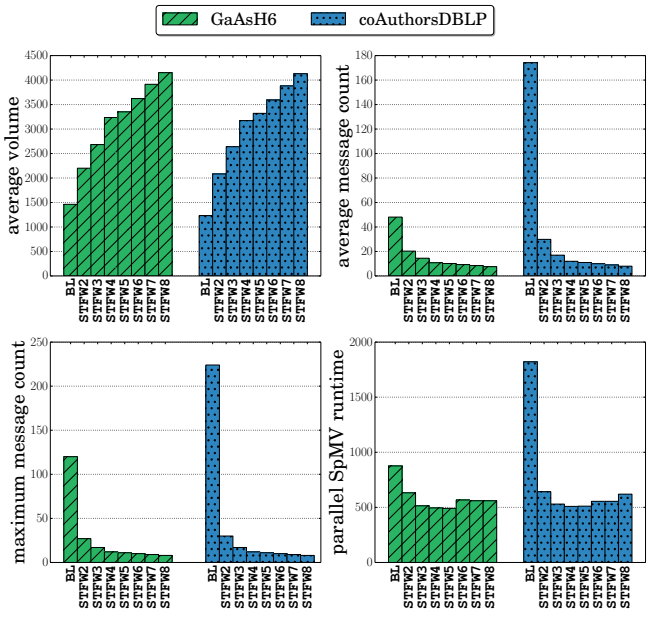


Figure 7: Detailed comparison of the schemes in two matrices GaAsH6 and coAuthorsDBLP at $K = 256$.

1.6×–3.0×, and 1.6×–3.3× more average volume than BL for $K = 64, 128, 256$, and 512, respectively. It is important to note that the rate of improvements achieved by STFW in average message count is higher than the rate of increases caused by STFW in average volume. This can be seen in Figure 6 by comparing the first and the third bar of every dimension: for example for T_5, STFW incurs 2.4× the average volume of BL, while it improves the average message count by a factor of 5.3.

As the instances in our experiments are generally bound by latency, reducing the related cost metrics with STFW greatly helps in reducing both the communication time and the parallel runtime as seen in Table 2. STFW achieves up to 50%, 55%, 61%, and 72% improvement in communication time compared to BL for $K = 64, 128, 256$, and 512, respectively. This is reflected in parallel SpMV runtime as STFW achieves up to 12%, 22%, 44%, and 65% improvement.

It can be said that for a communication time in which the portion of the total latency cost is higher, STFW would be more effective in improving the parallel performance. Figure 7 compares the performance of BL and STFW in two different matrices, for which BL and STFW attain comparable volume statistics (top left figure). However, the matrix coAuthorsDBLP has a higher latency overhead than the matrix GaAsH6 for BL. The improvements of STFW over BL in latency cost metrics are reflected in the parallel SpMV time more prominently in the matrix that is more latency-bound: coAuthorsDBLP. STFW effectively turns the irregular communication patterns into regular patterns, and in doing so makes the total latency cost much more predictable and uniform. As the VPT dimension gets higher, the message communication pattern becomes more regular and the variation between the number of messages communicated by the processes gets close to disappearing.

Table 2 also presents the maximum sizes of the buffers used by the BL and STFW schemes. For BL, the buffer size for a process is the summation of the sizes of the original messages it sends and receives. For STFW, it includes the sizes of the buffers for these original messages as well as the sizes of store and forward buffers in Algorithm 1. The sizes of the buffers utilized by the STFW schemes are always less than twice the sizes of the buffers used by BL. Observe that the sizes of buffers used for communication decrease as VPT dimension gets higher for a specific process count. This is because in low-dimensional VPTs a process is likely to store and forward messages from more processes compared to the high-dimensional VPTs. Also observe that for a specific VPT dimension, as the number of processes get larger, the buffer sizes decrease. This is due to the strong scaling of these instances, which results in message sizes to get smaller with increasing number of processes.

6.3 Effect on Scalability

We investigate how STFW affects scalability by comparing it to BL in Figure 8. In order to make the plots in the figure less crowded, we only focus on even VPT dimensions used for STFW. The parallel SpMV runtime is plotted against five different values of processes $K = 32, 64, 128, 256$, and 512. The figure presents runtime plots of 12 of the 15 test matrices. Note that the smallest number of processes for running STFW6 and STFW8 are 64 and 256, respectively. For this reason, there are no points in the plots for these two schemes for the process counts smaller than those values.

As seen in Figure 8, STFW succeeds in scaling instances that are otherwise unscalable with BL. These instances include matrices such as coAuthorsDBLP, GaAsH6, gupta2, human_gene2, net125, pattern1, sparsine, TSOPF_FS_b300_c2, which are characterized by very high latency overhead. They have a large maximum message count that is close to the process count. For instances that are not as latency-bound as those mentioned, still the latency costs are manifested at large process counts. These instances include matrices such as coPapersCiteseer, fe_rotor, nd3k, pkustk04. They scale somewhat similarly with both BL and STFW up to a certain point. However, then the benefit of using STFW becomes more apparent and they scale better with STFW.

It can be observed from Figure 8 that a low-dimensional VPT (i.e., STFW2) often leads to worse scalability compared to a high-dimensional VPT in instances that have very high latency overhead. However, another important factor in scalability is the increase of volume caused by STFW. If the volume is high, being aggressive in reducing the total latency cost can hurt the scalability. Trying to save a couple of messages per process by increasing the VPT dimension may increase the average message size significantly while leading to minor reductions in the total latency cost. This is best seen in the plot for the matrix TSOPF_FS_b300_c2. This matrix has the largest volume among the matrices in the figure, and is also a latency-bound instance. In this matrix, STFW2 attains better scalability than STFW4, STFW6, and STFW8. Reducing latency together with keeping the increase in volume small leads to a better scalability in this instance.

6.4 Communication on Different Networks

We compare the communication performance of BL and STFW for 128 and 512 processes on two different networks in Figure 9. The communication times are the geometric averages of the values

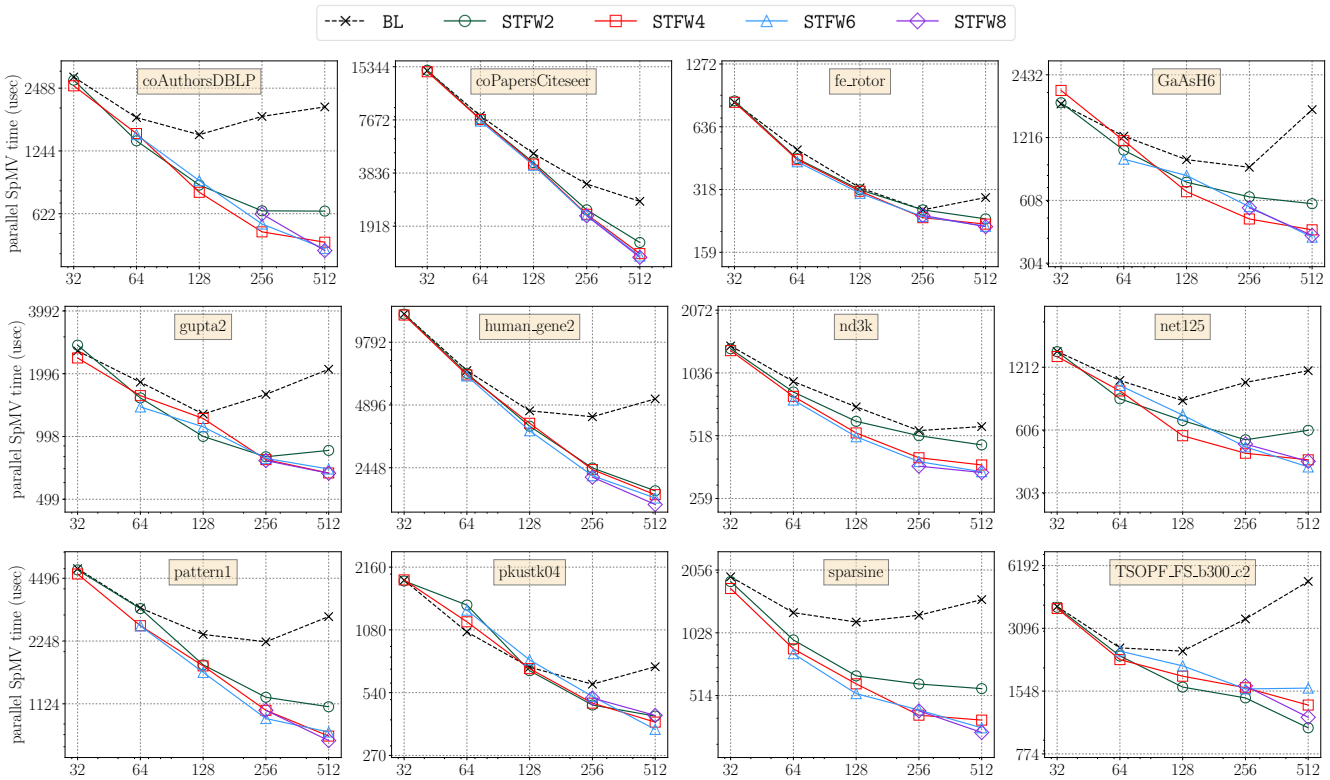


Figure 8: Parallel SpMV runtime comparison on BlueGene/Q for 12 matrices (both axes in logscale).

obtained by running SpMV on 15 test matrices. The communication times obtained for the BlueGene/Q system can also be seen in Table 2. Even though STFW still obtains better parallel SpMV runtime than BL on Cray XC40, we do not report these runtimes as the matrices were too small to scale beyond 64 or 128 processes. Yet, we present the obtained communication times as they illustrate how our method can substantially benefit the communication time on a different network.

In Figure 9, the STFW schemes are able to improve the communication time substantially on both networks. For example on 128 processes, STFW4 achieves 45% and 70% improvement over BL on BlueGene/Q and Cray XC40 systems, respectively. On 512 processes, these improvements increase to 69% and 85%, respectively. The better improvements on Cray XC40 system can be attributed to this network having a larger message start-up time to per-word transfer time ratio, which makes it more latency-bound compared to BlueGene/Q, and hence renders our method's ability to bound message count more effective.

Although our method can effectively offer a trade-off independent of the underlying physical network, the best VPT dimension still depends on the characteristics of the physical network. For a latency-bound network, higher-dimensional VPTs would be more preferable since they reduce the total latency cost more aggressively. On the other hand, for bandwidth-bound networks, lower-dimensional VPTs would be a better choice since they cause less forwarding, and hence less increase in volume.

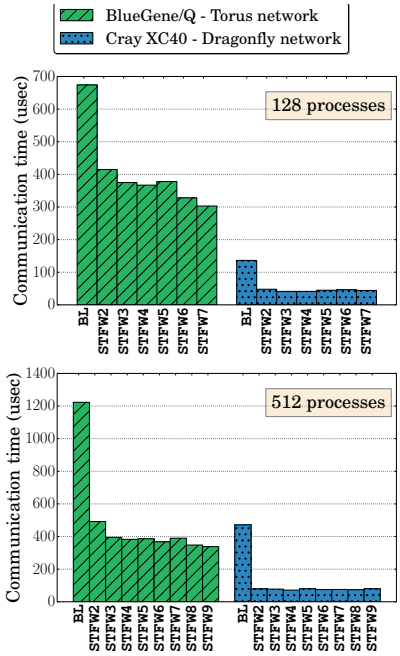


Figure 9: Communication times of BL and STFW on Torus and Dragonfly networks for 128 and 512 processes.

Table 3: Average communication statistics and times on a Cray XK7 system with a 3D Torus network and a Cray XC40 system with a Dragonfly network. The communication times (usec) are presented in the "comm" column.

Cray XK7 (3D Torus)										Cray XC40 (Dragonfly)				
8K processes					16K processes					4K processes				
Scheme	mmax	mavg	vavg	comm	Scheme	mmax	mavg	vavg	comm	Scheme	mmax	mavg	vavg	comm
BL	695.8	123.2	598	4420	BL	1054.9	137.6	425	8220	BL	486.6	105.5	819	1419
STFW2	136.5	58.1	914	590	STFW2	160.6	61.0	683	1109	STFW2	86.1	43.7	1271	294
STFW3	55.2	33.7	1215	361	STFW3	65.1	34.4	887	498	STFW3	39.0	25.0	1682	221
STFW4	34.0	23.4	1473	260	STFW4	41.8	24.3	1064	391	STFW4	26.1	18.1	2024	238
STFW7	18.9	13.9	2037	300	STFW8	20.0	13.9	1568	510	STFW7	17.0	13.0	2594	199
STFW8	18.0	13.7	2081	397	STFW9	19.0	13.9	1601	491	STFW8	16.0	12.8	2663	270
STFW12	14.0	11.4	2494	374	STFW13	15.0	11.4	1917	694	STFW11	13.0	10.9	3098	289
STFW13	13.0	10.4	2644	511	STFW14	14.0	10.5	2017	696	STFW12	12.0	9.8	3312	387

mmax: maximum message count. **mavg**: average message count. **vavg**: average volume (words).

6.5 Large-scale Communication Analysis

We analyze the large-scale communication performance of the compared schemes on two systems with different network types in order to assess the viability of running the proposed algorithm on thousands of processes. In the larger system, a Cray XK7 machine with a 3D Torus network and Gemini interconnect, we evaluate our method for 8192 and 16384 processes. In the smaller system, a Cray XC40 machine with Dragonfly network and Aries interconnect, we evaluate our method for 4096 processes. The evaluation is conducted on 10 matrices that have more than 10 million nonzeros in Table 1. For our scheme, we evaluate a total of seven VPT dimensions for each process count: the lowest three VPT dimensions $(2, 3, 4)$, the middle two VPT dimensions $(\lfloor \lg_2 K/2 \rfloor + 1, \lfloor \lg_2 K/2 \rfloor + 2)$, and the highest two VPT dimensions $(\lg_2 K - 1, \lg_2 K)$. This choice of selection of VPT dimensions aims to keep the discussions in this section simple while trying to cover the different spectra offered by our methodology. We present the geometric averages of the metrics related to communication in Table 3. The overall parallel SpMV time and buffer sizes are not reported as the sole purpose of this section is to analyze communication and the tested instances are almost always dominated by the communication time (more than 90% of the overall parallel SpMV time is spent in performing communication).

As seen in Table 3, our methodology improves the time spent in communication drastically on both systems. On Cray XK7, the communication time is improved by 94% and 95% compared to BL on 8K and 16K processes, respectively, both obtained by STFW4. On Cray XC40, the communication time is improved by 86% compared to BL on 4K processes, obtained STFW7. For BL, the communication time increases by a factor of 1.9 on Cray XK7 when the number of processes increases from 8K to 16K, while it increases by a factor of 1.5 for STFW4. This can be attributed to the better control of the increase in communication statistics by the STFW schemes: for example when the number of processes is increased from 8K to 16K, the maximum message count of BL increases by a factor of 1.52, whereas the maximum message count of STFW4 increases only by a factor of 1.23. If we compare these values to the communication time improvements reported in Table 2 of Section 6.2, it is safe to say that our method becomes more beneficial as the instances get more communication-bound, despite the fact the tested set of matrices are different. Even though these matrices are different, they exhibit similar communication characteristics as they are all irregular and generally latency-bound, whose corresponding properties in Table 1 attest to this fact.

We provide a detailed comparison of all schemes for all of 10 matrices in Figure 10 on the largest tested number of processes, i.e., 16K. In general, the middle VPT dimensions (STFW4, STFW8, STFW9) tend to perform better than the low (STFW2, STFW3) and high (STFW13, STFW14) VPT dimensions. This is because the low VPT dimensions are still often bound by latency and the high VPT dimensions simply cause too many forwarding steps and hence increase the volume significantly.

7 RELATED WORK

We investigate algorithmic techniques on the software level to improve the latency costs by reducing the message counts between processes. MPI collectives contain a rich history in this aspect [3, 4, 10, 14, 17, 21]. It is possible to attain logarithmic bounds on the latency costs using algorithms like bidirectional exchange for collective communication operations such as broadcast, scatter, etc [6]. Furthermore, the popular MPI implementations such as MPICH [1] switch between multiple algorithms depending on the message size to optimize latency or bandwidth costs.

Somewhat related to our work are the sparse neighborhood collectives [11–13, 15] in MPI, with which one can define a restricted set of neighbor processes and perform collectives on them. In our methodology, the neighboring processes in distinct communication stages may or may not communicate with each other depending on what submessages they forward. In other words, our VPT indicates which processes *may* communicate. Hence, it may not be feasible to perform sparse collectives in the case where there are no or only a few communicating neighbor processes.

In MPI, one can define a virtual topology to indicate how processes communicate. The two types of virtual topologies supported by MPI are Cartesian and graph topologies. The provided virtual topology and additional information such as edge weights can effectively be used for the ordering of process ranks in mapping to

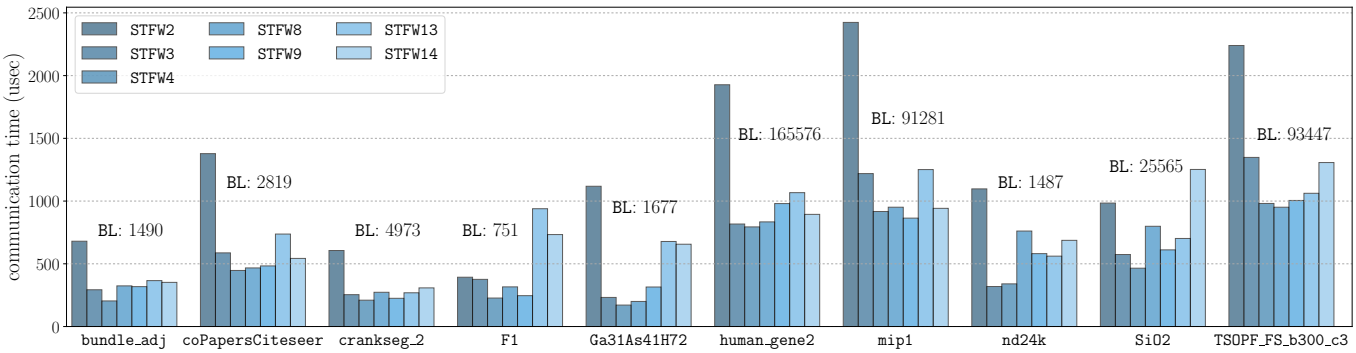


Figure 10: The communication times of ten matrices on 16K processes for different VPT dimensions. The values of BL are reported as text in order to make the analysis of STFW schemes more readable.

physical topology. In our work, we assume that the proposed VPT is completely independent of the physical topology.

It is also possible to reduce the latency costs with a careful distribution of data that will be communicated throughout the application. These often involve a preprocessing phase where the application is modeled with graphs/hypergraphs that are able to capture the communication requirements of the application. There are several works in this direction [2, 9, 18, 19, 22] and they often strive for better modeling of communication costs in the application.

The proposed VPT harbors similarities with the k-ary n-cube networks. The two common switching techniques in networks are the store-and-forward and wormhole switching [7, 16]. Our VPT borrows ideas from store-and-forward switching in the sense that there is no message fragmentation and multiple submessages may need to be stored in and forwarded by multiple processes in their buffers before reaching their destination. The submessages in our VPT refer to the original data the processes want to send, and they are contained in direct larger messages between processes. The important difference between the store-and-forward routing and the proposed store-and-forward scheme for our VPT is that multiple submessages received by a process in earlier communication stages are gathered into a single message to be forwarded in later stages. That is, at any stage of our algorithm, a message communicated between a pair of processes contains multiple submessages, where these submessages possibly differ in their source and destinations.

8 CONCLUSIONS

We proposed an efficient algorithm to perform sparse and irregular communication operations in a distributed setting. We organized the processes into a virtual process topology and this enabled us to control the trade-off between the latency and bandwidth costs. The communication operations under this topology are realized with the proposed store-and-forward algorithm. We further proposed an effective way of forming the virtual process topology. Our experiments on a set of latency-bound instances showed that the proposed methodology can offer significant improvements in the performance of parallel sparse matrix-vector multiplication by reducing its runtime up to 65% on 512 processes.

As future work, we plan on trying out different strategies for mapping processes onto both the virtual process topology and the physical topology. There are two approaches in order to benefit from process and rank ordering. In the first, we can map processes to the VPT with the aim of reducing the communication volume and/or the message count. For the reduction of volume, the basic idea would be to reduce the Hamming distance of the pair of processes that have a large amount of data to send to each other. Although more involved, one can also reduce the message count with a careful mapping of process to the VPT. However, this is more involved because the messages in the VPT are split/joined throughout the communication. In the second, we can benefit from mapping of the processes to the physical topology by trying to keep the processes that communicate large amount of data "close" to each other in terms of physical topology. Closeness can be defined by the mapping of processes where the communication is cheaper, such as on-node communication. Observe that in the first approach, we aim to reduce the communication volume and message count in the VPT, while in the second approach, these two quantities stay fixed and we aim to reduce the time to realize them by exploiting the physical topology.

ACKNOWLEDGMENTS

This work was supported by the Director, Office of Science, U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] 2019. MPICH 3.3 - A portable implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.

[2] Kadir Akbudak and Cevdet Aykanat. 2014. Simultaneous Input and Output Matrix Partitioning for Outer-Product–Parallel Sparse Matrix-Matrix Multiplication. *SIAM Journal on Scientific Computing* 36, 5 (2014), C568–C590. https://doi.org/10.1137/13092589X arXiv:http://dx.doi.org/10.1137/13092589X

[3] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. 2005. Optimization of MPI Collective Communication on BlueGene/L Systems. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 253–262. https://doi.org/10.1145/1088149.1088183

[4] Jehoshua Bruck, Ching-Tien Ho, Eli Upfal, Shlomo Kipnis, and Derrick Weathersby. 1997. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Trans. Parallel Distrib. Syst.* 8, 11 (Nov. 1997), 1143–1156. https://doi.org/10.1109/71.642949

[5] Umit Catalyurek and Cevdet Aykanat. 1999. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Trans. Parallel Distrib. Syst.* 10 (July 1999), 673–693. Issue 7. https://doi.org/10.1109/71.780863

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective communication: theory, practice, and experience: Research Articles.

Concurr. Comput. : Pract. Exper. 19, 13 (Sept. 2007), 1749–1783. https://doi.org/10.1002/cpe.v19:13

[7] William J. Dally and Charles L. Seitz. 1986. The torus routing chip. *Distributed Computing* 1, 4 (01 Dec 1986), 187–196. https://doi.org/10.1007/BF01660031

[8] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[9] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit Çatalyürek. 2015. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *J. Parallel and Distrib. Comput.* 77, 0 (2015), 69 – 83. https://doi.org/10.1016/j.jpdc.2014.12.002

[10] Ahmad Faraj and Xin Yuan. 2005. Automatic Generation and Tuning of MPI Collective Communication Routines. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 393–402. https://doi.org/10.1145/1088149.1088202

[11] Torsten Hoefler, Rolf Rabenseifner, Hubert Ritzdorf, Bronis R. de Supinski, Rajeev Thakur, and Jesper Larsson Träff. 2011. The Scalable Process Topology Interface of MPI 2.2. *Concurr. Comput. : Pract. Exper.* 23, 4 (March 2011), 293–310. https://doi.org/10.1002/cpe.1643

[12] Torsten Hoefler and Timo Schneider. 2012. Optimization Principles for Collective Neighborhood Communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 98, 10 pages. http://dl.acm.org/citation.cfm?id=2388996.2389129

[13] Torsten Hoefler and Jesper Larsson Traff. 2009. Sparse Collective Operations for MPI. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/IPDPS.2009.5160935

[14] Sameer Kumar, Amith Mamidala, Philip Heidelberger, Dong Chen, and Daniel Faraj. 2014. Optimization of MPI Collective Operations on the IBM Blue Gene/Q Supercomputer. *Int. J. High Perform. Comput. Appl.* 28, 4 (Nov. 2014), 450–464. https://doi.org/10.1177/1094342014552086

[15] S. H. Mirsadeghi, J. L. Träff, P. Balaji, and A. Afsahi. 2017. Exploiting Common Neighborhoods to Optimize MPI Neighborhood Collectives. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 348–357. https://doi.org/10.1109/HiPC.2017.00047

[16] L. M. Ni and P. K. McKinley. 1993. A survey of wormhole routing techniques in direct networks. *Computer* 26, 2 (Feb 1993), 62–76. https://doi.org/10.1109/2.191995

[17] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. 2005. Performance analysis of MPI collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*. 8 pp.–. https://doi.org/10.1109/IPDPS.2005.335

[18] Oguz Selvitopi and Cevdet Aykanat. 2016. Reducing latency cost in 2D sparse matrix partitioning models. *Parallel Comput.* 57 (2016), 1 – 24. https://doi.org/10.1016/j.parco.2016.04.004

[19] R.O. Selvitopi, M.M. Ozdal, and C. Aykanat. 2015. A Novel Method for Scaling Iterative Solvers: Avoiding Latency Overhead of Parallel Sparse-Matrix Vector Multiplies. *Parallel and Distributed Systems, IEEE Transactions on* 26, 3 (March 2015), 632–645. https://doi.org/10.1109/TPDS.2014.2311804

[20] Herbert Sullivan and T R Bashkow. 1977. A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I. *SIGARCH Comput. Archit. News* 5, 7 (March 1977), 105–117. https://doi.org/10.1145/633615.810659

[21] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (Feb. 2005), 49–66. https://doi.org/10.1177/1094342005051521

[22] Bora Uçar and Cevdet Aykanat. 2004. Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies. *SIAM J. Sci. Comput.* 25, 6 (2004), 1837–1859. https://doi.org/10.1137/S1064827502410463

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran the proposed store-and-forward-based communication algorithm on a BlueGene/Q parallel system. We tested our algorithm within sparse matrix-vector multiplication. We used the MPICH 2.2 for MPI communications. We also used a Cray XC40 system for communication performance evaluation of the store-and-forward-based scheme. On Cray, MPICH 3.0 was used.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

```
code: https://bitbucket.org/roguzsel/stfw/src/master/
matrices used in the experiments can be accessed at
↪ https://sparse.tamu.edu/
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: IBM BlueGene/Q system with 1.6 GHz PowerPC A2 processors. Cray XC40 system with Intel Haswell 2.3 GHz processors. Cray XK7 system with 2.2 GHz AMD Opteron Interlagos processors.

Operating systems and versions: CNK, lightweight proprietary kernel

Compilers and versions: gcc 4.9.0, icc 18.0.1

Libraries and versions: MPICH 2.2, MPICH 3.0

Key algorithms: Sparse matrix-vector multiplication

Input datasets and versions: Sparse matrices from SuiteSparse Collection (https://sparse.tamu.edu/)