2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

You've Got Mail (YGM): Building Missing Asynchronous Communication Primitives

Benjamin W. Priest*†, Trevor Steil*‡, Geoffrey Sanders*, Roger Pearce*
*Center for Applied Scientific Computing; Lawrence Livermore National Laboratory
†Thayer School of Engineering; Dartmouth College
‡School of Mathematics; University of Minnesota
benjamin.w.priest.th@dartmouth.edu; steil016@umn.edu; sanders29@llnl.gov; rpearce@llnl.gov

***Abstract*— The Message Passing Interface (MPI) is the de facto standard for message handling in distributed computing. MPI collective communication schemes where many processors communicate with one another depend upon synchronous handshake agreements. This results in applications depending upon iterative collective communications moving at the speed of their slowest processors. We describe a methodology for bootstrapping asynchronous communication primitives to MPI, with an emphasis on irregular and imbalanced all-to-all communication patterns found in many data analytics applications. In such applications, the communication payload between a pair of processors is often small, requiring message aggregation on modern networks. In this work, we develop novel routing schemes that divide routing logically into local and remote routing. In these schemes, each core on a node is responsible for handing all local node sends and/or receives with a subset of remote cores. Collective communications route messages along their designated intermediaries, and are not influenced by the availability of cores not on their route. Unlike conventional synchronous collectives, cores participating in these schemes can enter the protocol when ready and exit once all of their sends and receives are processed. We demonstrate, using simple benchmarks, how this collective communication improves overall wall clock performance, as well as bandwidth and core utilization, for applications with a high demand for arbitrary core-core communication and unequal computational load between cores.**

***Index Terms*—Distributed Computing, Asynchronous Communication, MPI**

I. Introduction

Traditional high-performance computing (HPC) applications, such as solving systems of partial differential equations, are solvable in a scalable manner using the principles of the bulk-synchronous parallel (BSP) model, in which an algorithm proceeds in supersteps. During a superstep, processors perform a portion of computation, followed by a synchronization phase in which processors exchange the data to perform the next superstep. To scale, these algorithms use the regular structure in problems and the large computation to communication ratio.

As HPC grows to include more diverse problems, these assumptions do not necessarily hold. For instance, graph algorithms are characterized by a low ratio of computation to data access, have irregular structures, and are completely data-dependent [1]. Because of this, many modern HPC problems do not lend themselves nicely to a BSP solution.

In this paper, we introduce You've Got Mail (YGM), a pseudo-asynchronous communication protocol built on top of MPI in C++. Programmers using YGM make use of a mailbox abstraction to perform point-to-point and broadcast communications. During the computation, users queue messages into the mailbox. When the mailbox becomes full, the processor enters a communication context in which it sends and receives all of its messages. After sending and receiving its messages, the processor drops back into its computation, potentially while other processors are still communicating. This pseudo-asynchronicity prevents the speed of all processors from being explicitly tied to the slowest processor, while also preventing a slow processor from running out of memory due to an accumulation of unhandled messages.

YGM allows point-to-point messages as well as broadcasts. Asynchronous broadcasts are particularly useful in cases where some amount of state is replicated across processors, such as when using vertex cuts in graph algorithms [2]. In these cases, synchronization can be handled by broadcast operations rooted at a particular processor.

YGM is currently being used to handle communications for several projects at LLNL. YGM has been incorporated into HavoqGT, LLNL's asynchronous graph library [3]. It was also used in the recent Graph500 submission on Sierra at LLNL. The Graph500 uses breadth-first search (BFS) and single source shortest path (SSSP) to benchmark the graph processing capabilities of computer systems [4]. The recent submission used 2048 compute nodes on Sierra (CPU-only, roughly half of the system) to perform a BFS on a graph with 2^{42} vertices, the largest submission to date.

We begin in Section II by describing similar communication layers. In Section IV, we present the important design features of YGM. In Sections V and VI, we present some sample applications to demonstrate the use and scalability of YGM.

II. Related Work

YGM is conceptually similar to Active Pebbles, a programming and execution model that allows programmers to naturally express solutions to irregular, data-driven problems

Priest and Steil performed this work while visiting Lawrence Livermore National Laboratory. Correspondence should be sent to *rpearce@llnl.gov*. Source code for YGM is planned to be released open source and available at http://github.com/LLNL/ygm

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00045

[5]. Active Pebbles uses AM++ [6] as its underlying active message library for communication. AM++ asynchronously sends messages with remote procedure call (RPC) semantics and is built on top of MPI. It provides optimizations through message coalescing and message reduction. Messages are sent in epochs, making computations proceed similarly to traditional BSP algorithms. However, synchronizations are implicitly handled by AM++, and the communication phase can be made to contain the majority of computational work.

YGM provides many of the same characteristics. Programs written using YGM create a mailbox with a function defining the behavior for received messages and a mailbox size. When the mailbox is full, it enters its communication context in which messages are exchanged between mailboxes. The mailbox's receive function is called on incoming messages, which can spawn additional messages, creating data-dependent synchronizations similarly to AM++ and Active Pebbles. Like AM++, YGM provides message coalescing to better utilize network bandwidth.

There are several major differences between YGM and Active Pebbles. YGM is written as a transport layer without remote procedure call semantics. YGM provides message-routing schemes designed to effectively use shared memory for coalescing messages originating from various cores on the same compute node and to minimize the number of compute nodes any individual core must send messages to. These schemes could also be implemented in Active Pebbles.

The last major difference is the inclusion of asynchronous broadcast operations. These broadcasts provide effective methods of lazy synchronization across cores using the underlying asynchronous send operation. In a typical RPC system, the notion of broadcasts is not natural as it would entail performing the same computation at all cores.

There have also been examples of constituent pieces of YGM in software developed for specific applications. HipMer [7], [8], provides a parallel version of *Meraculous* [9], a de novo genome assembler. HipMer's process for identifying frequent k-mers is similar to how we identify high-degree vertices in graphs, and can likely benefit from using YGM. We also see a behavior similar to our mailboxes being used in constructing a distributed de Brujin graph, where each process keeps a buffer of messages to send to every other process, and a buffer is sent once it reaches a specified size. This application could likely benefit from the additional message routing of YGM.

Another instance of YGM features in previous work is in the node aware parallel sparse matrix-vector multiplication (NAPSpMV) [10]. NAPSpMV provides message coalescing at source and destination compute nodes by using knowledge of a system's topology. The exact choice of intermediate destinations for communication in NAPSpMV will depend on the matrix being used in order to provide communication balance. YGM provides a more flexible design that only depends on the topology and is not application dependent. These optimizations could be incorporated by constructing additional MPI communicators.

III. All to All Asynchronous Routing

We focus our analysis on asynchronous variants of MPI's AllToAll collective. Each of these variants adopt a different routing protocol, each of which takes advantage of a logical distinction between *local* and *remote* routing. We use the term *local* to refer to point-to-point communications wherein the source and destination processes reside on the same compute node. We meanwhile use *remote* to refer to the converse situation, where the endpoints of a message exist on different nodes. The key insight to this analysis is that remote communication requires the transmission of messages over a wire, whereas local communication is handled in shared memory of a single machine.

In general, we assume that remote communication is bit-for-bit more expensive in terms of time than local communication. Furthermore, on many HPC interconnects transmitting many small messages is less efficient than bundling larger messages together. Hence, it is desirable to route messages in such a way that remote communications are both bundled and transmitted in as few discrete messages as possible.

MPI's AllToAll and AllToAllV collectives handle the arbitrary transmission of messages within a group. While there are many known methods to boost the efficiency of these collectives, the particulars are implementation dependent [11]. One notable feature of these collectives, irrespective of implementation, is that they are *synchronous* - all engaged processes enter the same program context and execute the AllToAll routine at once. This feature allows implementations to maintain protocol guarantees, such as guaranteeing the send and receive orders match, while optimizing transmission based upon the send distribution and message sizes. However, it also results in issues in applications with an uneven distribution of communication and/or computation load across processes. For example, if one process takes much longer than the others to reach the communication context, the others simply wait on it. Similarly, if one process is the recipient of a large proportion of the total communication in an exchange that reoccurs frequently, then it will fall behind other processes which must then wait on it. This can result in poor resource utilization, both in terms of CPU and bandwidth as many processes are left idle while waiting.

The other extreme is to handle all communication with point to point communication by posting MPI Recvs and Sends for each message generated over the course of the application. In this way, processes wait only upon their immediate communication partners, and so processes that engage in a relatively low volume of communication do not experience delays due to slow or more heavily utilized processors with which they need not communicate. Relying upon fully point-to-point communication gives us *asynchronicity*, but at the expense of potentially increasing remote communication bandwidth requirements.

We advocate a middle ground similar to the non-blocking Mpi_IalltoallV [12]. In our suite of asynchronous collectives, participants can begin the process of sending messages

as soon as they enter the appropriate program context and leave as soon as they have received all messages. These asynchronous collectives also execute routing in local and remote stages. The objective of these routing strategies is to minimize the number of discrete remote messages and channels, improving bandwidth utilization by reducing overhead. They also constrain the maximum number of direct communication partners for each processor, meaning that most messages are received by at least one intermediary during transmission. We evaluate three different routing protocols: *node local* performs a local exchange of messages before a remote exchange, *node remote* performs a remote exchange before a local exchange, and *node local node remote* or *NLNR* performs a local exchange, a remote exchange, and a final local exchange. We describe these protocols in detail below.

Throughout, we assume that there are N nodes participating in the protocol. We identify each node with an offset in $[N]$. Here we use the notation $[z] = \{1, 2, \ldots, z\}$ for $z \in \mathbb{Z}_+$. We will further assume that each participating node holds the same number of cores C, similarly identified with an offset in $[C]$. We assume without loss of generality that N is a multiple of C. We address a core c on node n by the tuple $(n, c) \in [N] \times [C]$.

When discussing remote communication, we will use the term communication *channel* to refer to a partitioning of cores such that the cores in each partition communicate remotely only with other cores in the same partition.

A. Exchange Phases

All of the protocols described below proceed in a series of *exchanges*. An exchange consists of a subset of processes participating in the collective passing of messages about amongst each other. Each member of an exchange may be responsible for further communication to other uninvolved processes in a later exchange. We call these forwarding processes intermediaries. At the end of an exchange phase, we assume that each process holds all outbound messages intended either for it or for one of the exterior processes for which it is an intermediary. In this document we consider two types of exchanges: *local* and *remote*.

A local exchange consists of all of the processes on a compute node. A local exchange phase consists of each participating process's forwarding each message it holds to its destination or intermediary if its destination is remote.

A remote exchange consists of a set of processes, each of which lives on a different compute node. A remote exchange is similar to a local exchange, wherein participants forward each held message to its destinations or intermediary if the destination is a different process on the same node.

These local exchanges could be implemented a few different ways. For example, as they are a partitioning of the global processor set, they could be implemented with ALLTOALLV calls while potentially still gaining some improvements from asynchronicity. We instead implement them as a round of SEND and RECV calls, so that processes can begin working before all of their exchange partners are available. On systems with optimized ALLTOALL implementations, such as IBM BG/Q *Sequoia* at LLNL, we have seen better bandwidth utilization and performance by implementing these exchanges using ALLTOALLV.

B. Node Local

The node local strategy consists of a local exchange on each node, followed by C remote exchanges involving all processes with the same core offset. At the beginning of the local exchange, the process on core (n, c) holds a set of messages to be transmitted. Each message with destination (n', c') is forwarded to (n, c') in a local exchange. At the end of this local exchange, each core (n, c) holds messages with addresses of the form (n', c). At the end of this exchange, each message is held by a process matching the destination's core offset. Fig. 1 depicts a representation of such a local exchange.

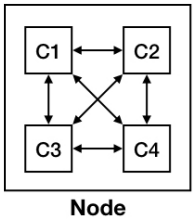


Fig. 1: Example of the 4-core local exchange topology for node local, node remote, and NLNR protocols.

Once a process on (n, c) completes its local communication phase, it enters the program context for a remote exchange. This remote exchange consists of all of the cores with local offset c. Following the local exchange, all messages reside on a process with the correct core offset. Once a process is finished participating in this remote exchange, it has sent all messages routed through it and received all messages sent to it during this round of collective communication. It can safely move on to a different program context, even if others are still working. Fig. 2 depicts a representation of one of these remote exchanges.

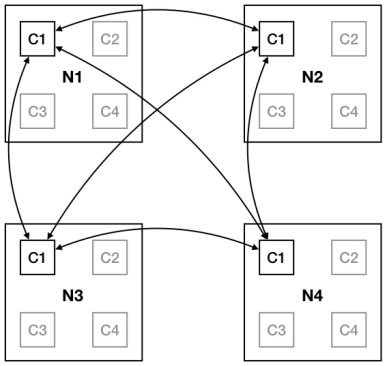


Fig. 2: Example of 4-core, 4-node local exchange topology of core offset 1 for node local and node remote protocols.

In total, the node local protocol consists of N local exchanges and C remote exchanges, which occur in parallel and

may overlap. All messages destined for a particular remote process are accumulated at a single intermediary at each node prior to remote transmission, which potentially allows many small message to be packaged into larger messages, saving on remote overhead.

C. Node Remote

The node remote protocol is very similar to the node local protocol, except its execution happens in reverse. That is, each process participates in a remote exchange with all cores matching its core offset in the first round of communication, followed by a local exchange at each node in the second. For each message held by the process at (n, c) with destination (n', c'), the process forwards the message to (n', c) in this remote exchange. Once all remote exchanges have completed, each message is held on a node matching its destination node offset. A local exchange in the second phase ensures that each message arrives at its destination core.

Fig. 2 and Fig. 1 depict examples of these exchanges in the order prescribed by the node remote protocol, which as stated are the same as the node local protocol, but in reverse. Whereas the node local protocol accumulates all messages to a particular process in a single intermediary before remote transmission, the node remote protocol instead forwards all messages from a particular process destined for the same *node*, allowing for a similar bundling of messages. If the distribution over sender-receiver pairs is roughly uniform in an application, then the two protocols should exhibit roughly equivalent performance. Consider, however, the case of broadcasts. If a process has a message it must send to every other process in the group, then the node local and node remote protocols treat it differently. In the node local protocol, the message must be copied and sent to every on-node core offset before it is then sent over all C remote exchanges to $C * (N - 1)$ remote recipients. In the node remote protocol, it is sent to $N - 1$ remote recipients in a single remote exchange, each of whom then forward it locally during the local exchange phase. So, the node remote protocol can more efficiently handle broadcasts as each one consumes a factor of C fewer remote messages, pushing more of the broadcasting work onto the (typically) faster shared memory. We will see this in greater detail later.

D. Node Local Node Remote (NLNR)

We have detailed two protocols thus far, each of which depend upon C independent communication channels, each including N participating cores. As we have discussed, different applications might prefer either the node local and node remote protocols. However, is it possible to retain qualities of both? The final protocol we discuss in this document we dub node local node remote (NLNR), which does exactly this. The NLNR protocol reduces the number of remote communication channels to the theoretical minimum, while still allowing each node to communicate directly with every other node by eliminating redundancy. A message originating at node n_1 might be transmitted along any of C different remote channels to node n_2 using node local or node remote. In NLNR, there is only one channel connecting each such pair of nodes.

In order to facilitate this reduction in channels, the protocol need occur in three stages: an initial local exchange, a remote exchange, and a final local exchange. It is helpful to visualize the topology of the remote exchange to explain the whole process as the first and second local exchanges serve to coalesce and then distribute messages, respectively. Informally, we structure groups of nodes into "layers" with the same topology as the cores on an individual node. All messages destined for a node with a given offset within its layer then get routed through the core with the same offset within its node. Similarly, a core with a given offset within a node is responsible for receiving messages from all nodes with that same offset within their layers.

More formally, in addition to their node offset $n \in [N]$, we assign to each node a *layer offset* $\ell = n \mod C$. Further, we enforce the rule that $(n, c) \in [N] \times [C]$ is an intermediary for all cores on node n', where $c = n' \mod C$ with corresponding intermediaries (n', c') where $c' = n \mod C$. Fig. 3 depicts such a topology for a single layer, whose connections form a clique. Note that cores with addresses of the form (n, c) where $c = n \mod C$ do not participate in any communication within a layer. These cores only communicate with their corresponding cores in nodes whose layer offsets match their own. Fig. 4 depicts an example inter-layer communication topology for two layers. Intra-layer edges are suppressed for clarity. The edges in Fig. 4 mirror those of Fig. 3 aside from the addition of the self-offset edges missing from Fig. 3. Note that by adding the edges from Fig. 3 to both layers, we have a clique.

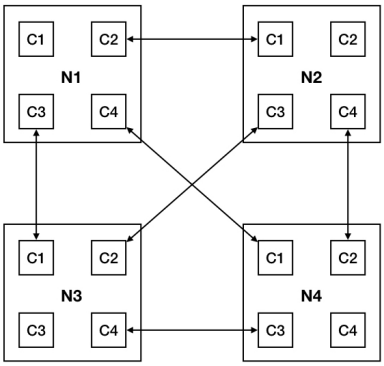


Fig. 3: Example of the remote communication topology within a single layer of 4-core machines in the NLNR protocol.

In the first local exchange on node n, a message from (n, c) with destination (n', c') is forwarded to $(n, n' \mod C)$. In the remote exchange, this message is sent to $(n', n \mod C)$. In the final local exchange, the message arrives at (n', c'). If any of these intermediate cores are the destination, it is received there and not forwarded.

If a process needs to broadcast to all other processes, then it sends this message to each of its local neighbors, who forward it along to each of their remote partners, who in turn distribute

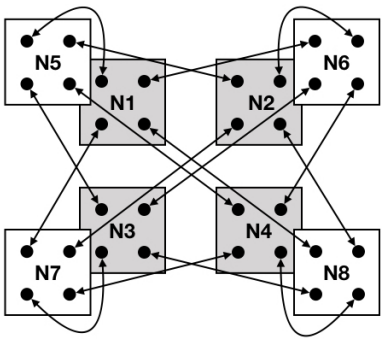


Fig. 4: Example of the remote communication topology between two layers of 4-core machines in the NLNR protocol.

it on each remote node. Like the node remote protocol, a broadcast over NLNR results in $N-1$ remote messages.

While point to point messages are transmitted at most twice in node local and node remote, they might be transmitted up to three times in NLNR. While the local shared memory transmissions are less costly than remote transmissions over a wire, they are still not trivial, so this can result in overhead not seen in the other two protocols. However, in exchange we significantly reduce the number of channels over which remote messages traverse. Recall that node local and node remote each require C communication channels, each of which includes the N cores with matching core offset c for each $c \in [C]$. NLNR, however, requires $\binom{C}{2}+C$ channels, each including $2\frac{N}{C}$ cores (aside from the self-offset channels, which include $\frac{N}{C}$ cores each). Such a channel consists of all the address pairs (n,c), (n',c') where $c = \ell' = n' \mod C$ and $c' = \ell = n \mod C$ for some $(\ell,\ell') \in [C]^2$.

On average, each core must communicate with $\frac{N}{C}$ nodes. Therefore, this configuration is able to minimize the maximum number of nodes any individual core must communicate with.

E. Bandwidth Maximization

Consider a configuration consisting of N nodes with C cores each. We assume N is at least 2 to guarantee remote communications must occur in the routing schemes described in Sections III-B, III-C, and III-D. For the ease of analysis, we will assume message traffic is uniform across all pairs of cores.

First, consider a mailbox with no routing scheme, that is, all cores send messages directly to the destination core. A core in this mailbox will be involved in $(N-1)C$ remote communications. If this core sends a total message volume of V bytes, the average message size sent to each recipient is $\frac{V}{(N-1)C} = \mathcal{O}\left(\frac{V}{NC}\right)$.

For our routing schemes, we can perform a similar analysis. Each core in the Node Local and Node Remote routing schemes communicates remotely with $N-1$ remote cores, giving an average message size of $\frac{V}{N-1} = \mathcal{O}\left(\frac{V}{N}\right)$. Each core in NLNR routing communicates remotely with $\frac{N}{C}$ remote cores because of the organization of nodes into *layers*, giving an average message size of $\mathcal{O}\left(\frac{VC}{N}\right)$.

The difference in average message sizes between routing schemes can have a large impact on mailbox performance on a real network. Fig. 5 shows the network bandwidth for various message sizes in which a single process sends a message to one other process. The available bandwidth increases as message sizes increase with a downward jump as MPI switches from using an eager protocol to a rendezvous protocol at a size of 16KB. For a fixed message volume, we have labeled possible bandwidth values for the above routing schemes using the scaling of average message sizes discussed, assuming a configuration that features 32 cores per node. NLNR being farther to the right on this plot allows it to scale to larger numbers of nodes without increasing the size of mailbox used.

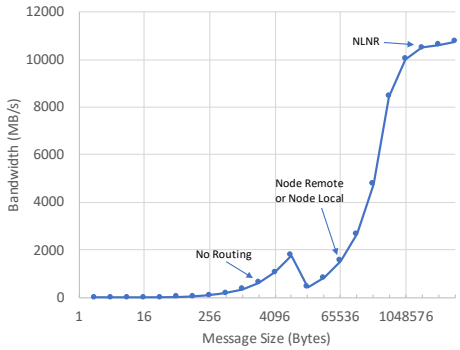


Fig. 5: Network bandwidth between two ranks for various message sizes using MVAPICH 2.3 over the Omni-Path interconnect on *Quartz* at LLNL.

We do not see an asymptotic improvement in the average message size as the number of nodes increases. For all routing schemes, a doubling of the number of nodes results in a halving of the average message size. The constant that gets introduced, however, is C. While this value is constant for a given system, the number of cores per node on new systems has been increasing over time. As the number of cores available increases, the lateral distance between no routing, Node Remote, and NLNR increases, making effective routing even more useful for large numbers of nodes.

IV. Design Features

YGM is designed to provide users with a simple interface for handling communication on distributed systems in general algorithms that may feature data-dependent computations with computation and communication imbalances across cores. Mailboxes are created by designating the behavior upon receiving a message and the message capacity of a mailbox. When a message needs to be sent, a call to either SEND or SEND_BCAST is made, depending on whether the communication is point-to-point or a broadcast operation. Message receiving is asynchronously handled by specifying a callback to YGM, and the receiving core does not need to be aware when it is going to receive a message.

A. Message Coalescing

When sending many small messages, metadata associated to messages can use a significant portion of the network's bandwidth. Message coalescing is a common technique to reduce this bandwidth requirement by bundling several messages together.

The differences in our routing schemes can be thought of as taking advantage of different levels of message coalescing. When messages are sent directly to the destination core, all messages must be coalesced at the core-core level. For Node Local and Node Remote routing, the local exchange allows messages from a single node to a single remote core or a single core to a single remote node can be coalesced. Using NLNR routing, all coalescing available in Node Local and Node Remote routing are available, allowing messages to be coalesced on the node-node level.

B. Termination Detection

Termination detection is handled by tracking when all ranks have completed sending and receiving messages. An algorithm simply calls TEST_EMPTY or WAIT_EMPTY indicating to YGM that the algorithm has completed producing new messages and YGM flushes its pending send buffers, including empty buffers. When empty message buffers are sent by all ranks globally, YGM can terminate. For algorithms that maintain work queues external to YGM, such as LLNL's graph analytics framework HavoqGT, it is necessary to poll YGM completion via TEST_EMPTY. For simplicity, the algorithms described in this paper use WAIT_EMPTY for termination detection.

C. Variable-Length Messages

In many applications, it may be necessary to send messages of different lengths. YGM supports variable-length messages through the use of cereal, a C++ serialization library [13]. Serialization provides a way of packing and unpacking messages on the source and destination cores. Support for C++ standard template library containers is provided by cereal, making it unnecessary for users to implement their own serialization functions in most cases.

V. Experiments

Our experiments were run using LLNL's *Quartz* cluster. Each compute node features 36 cores (2x Intel Xeon E5-2695 v4 @ 2.1GHz) and 128GB DRAM. To test the characteristics of the mailbox under different routing schemes, simple applications were developed that feature large amounts of communication relative to computation.

A. Degree Counting

The first application streams through the edges of a graph to calculate the degree of each vertex. Each vertex is assigned to a particular core for counting, so all edges spawn exactly two messages, each of which corresponds to a single addition. Edges are generated and counted in batches to isolate the time of degree counting from that of edge generation. The psuedocode for degree counting is outlined in Algorithm 1.

Algorithm 1 Degree Counting

```
1:  function DegreeCounting(G)
2:      num_ranks ← YGM.num_ranks
3:      degrees ← Zeros(G.num_vertices/num_ranks)
4:      function RecvFunc(v)                ▷ Callback Function
5:          local_ID ← v/num_ranks
6:          degrees[local_ID] ← degrees[local_ID] + 1
7:      mb ← YGM_mailbox(RecvFunc)
8:      for (u, v) ∈ G do
9:          u_owner ← u%num_ranks
10:         v_owner ← v%num_ranks
11:         mb.Send(u_owner, u)
12:         mb.Send(v_owner, v)
13:     mb.wait_empty()
```

In this example, we have assigned vertices to ranks for counting in a round-robin fashion. For simple uses of YGM such as this, we must define the mailbox's behavior upon receiving a message (lines 4-6), construct the mailbox with this receive callback function (line 7), send messages through the mailbox by providing a destination and message to send (lines 11-12), and then wait for all communications to complete by calling $mb.wait\_empty()$ (line 13). During our loop over G, the mailbox is queueing messages and performing its exchanges each time it reaches capacity. If G is not partitioned evenly across all processors, some processors will exit the loop early. These processors may still be receiving messages and may be necessary as intermediaries during routing, so the call to $mb.wait\_empty()$ keeps them in their communication context until all processors are done sending messages.

B. Connected Components

The second application finds connected components within a graph. To do so, each vertex stores a label that is initialized to its own global ID. For every edge in the graph, a vertex sends its current label to its neighbor. When receiving a label from a neighbor, a vertex stores the minimum of its label and the neighbor's label. This algorithm continues passing over the entire graph until no labels change.

This algorithm terminates with all vertices storing the minimum vertex ID in its connected component. For a graph $G = (V, E)$, this algorithm can take $O(diam(G))$ passes over the graph to complete, where $diam(G)$ is the diameter of G. A pass generates $O(|E|)$ messages, each of which corresponds to $O(1)$ work. By using the Shiloach-Vishkin connected components algorithm, we could find connected components using $O(\log(|V|))$ passes over the graph [14]; however, our implementation was developed to be simple benchmark test of scalability for our mailbox, not the most performant connected components algorithm. A Shiloach-Vishkin implementation could be implemented using YGM.

1) Use of Asynchronous Broadcasts: To test YGM in our connected components algorithm, we generate graphs using

an RMAT generator [15]. RMAT graphs are characterized by degree distributions that approximately follow a power-law distribution [16]. The presence of high-degree vertices can lead to computation and communication imbalances among processors. One way of handling these vertices is through a 2D decomposition of the adjacency matrix [17]. After this decomposition, matrix blocks can become hypersparse, that is, have fewer edges than vertices [18].

Instead, we use delegates to handle the high-degree vertices [2]. With this method, we identify what we consider to be high-degree vertices as delegates. All non-delegates are assigned to cores using a 1D partitioning. Delegate vertices are distributed across all cores with *colocated* edges, i.e., the core responsible for storing a delegate edge is the core assigned to the vertex on the other end of the edge.

When running the connected components algorithm, delegate labels are stored at all cores. After each pass over the graph's edges, the delegate labels are synchronized across all cores. This synchronization can be implemented using the asynchronous broadcasts provided by YGM.

C. Sparse Mat-Vec

The final application we built to demonstrate YGM is a sparse mat-vec (SpMV) that computes the product of a sparse matrix with that of a dense vector, that is, we compute

$$y = Ax$$

given some sparse matrix A and dense vector x. Our implementation stores A in a compressed sparse column (CSC) format and uses delegates with a 1D partitioning of columns across processor cores.

Let $A \in \mathbb{R}^{n \times n}$ and $x, y \in \mathbb{R}^n$. For an index k corresponding to a non-delegated column in A, let $p(k)$ return the ID of the processor responsible for storing column k. x and y are both partitioned so $p(i)$ stores x_i and is responsible for accumulating values into y_i. If the i-th column of A is delegated, then every core gets a local copy of x_i and y_i.

Consider a nonzero entry a_{ij} in A for which the i-th and j-th columns are not delegated. In the course of the SpMV operation, $p(j)$ performs the multiplication $a_{ij}x_j$ and sends the result to $p(i)$ for accumulating into the resulting y_i. Thus each non-delegated edge yields a single multiplication, addition, and message.

In the case that the j-th column is delegated, $p(i)$ stores the nonzero a_{ij}. Because the j-th column is delegated, $p(i)$ also contains a copy of x_j and is able to perform the multiplication and addition without sending a message. In the case that the i-th column is delegated, $p(j)$ keeps a local copy of y_i to which it adds $a_{ij}x_j$, again avoiding a communication. At the end of the computation, all delegated entries in y are combined using an ALLREDUCE operation.

To illustrate how SpMV is implemented with YGM in a simple 1D partitioning, pseudocode is shown in Algorithm 2. This pseudocode does not explicitly depend on the distribution of columns across processors. It only assumes x and y are distributed in the same manner as the columns of A.

Algorithm 2 SpMV

```
1:  function SpMV(A, x)
                          ▷ A stored in distributed CSC format
                          ▷ x distributed same as columns of A
2:      y ← Zeros(x.length)
3:      function RecvFunc(index, value)          ▷ Callback
4:          y[index] ← y[index] + value
5:      mb ← YGM_mailbox(RecvFunc)
6:      for col ∈ A.cols do
7:          col_ind ← col.ID
8:          for (row_ind, val) ∈ col.nonzeros do
9:              prod ← x[col_ind] * val
10:             dest ← Owner(row_ind)
11:             mb.Send(dest, row_ind, prod)
12:     mb.wait_empty()
13:     return y
```

VI. Results and Discussion

Here we present the results of running the applications described in Section V. For these experiments we compare the performance of the Node Local, Node Remote, and NLNR routing schemes presented in Section III. We also include a routing scheme in which all messages are sent directly to their destinations as a baseline for comparison. This scheme is labeled as "NoRoute".

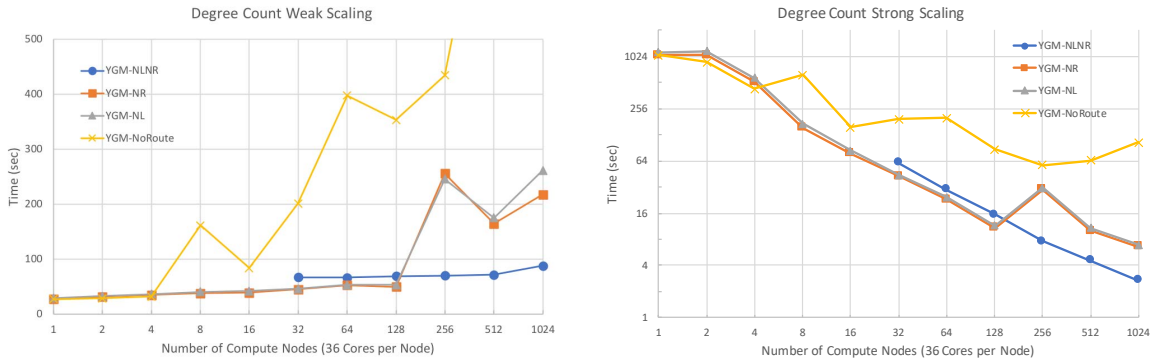
In these tests, NLNR routing was not used for less than 32 compute nodes. At 32 compute nodes, our test system which features 36 cores per node is able to almost completely build a notional layer in NLNR. Up until this point, all remote communications are routed through a small subset of cores. In this situation, Node Remote is a more appropriate choice of routing scheme.

A. Degree Counting

To test the use of degree counting, Erdős-Rényi graphs were generated with edge endpoints uniformly sampled. These graphs were chosen to highlight performance when communication and computation are relatively balanced across all cores. Edges were produced and counted in batches to isolate the time of degree counting from that of edge generation. Figure 6 shows the scaling properties of our degree-counting application using the routing schemes in YGM.

We see that the NoRoute mailbox scales very poorly past 4 compute nodes. Even with coalescing, a mailbox without additional routing makes poor use of network bandwidth. Each core must split its messages to be sent to all other $P - 1$ cores on the system. This provides limited opportunities for coalescing to occur.

Node Local and Node Remote routing demonstrate good strong and weak scaling up to 128 compute nodes. Past a certain point, these mailboxes face a similar scalability issue to that of the No Routing mailbox. Consider a single core in a system. If an additional compute node is added, that core has one additional core it must communicate with in its remote



(a) (b)

Fig. 6: Scaling of degree counting application using YGM. Weak scaling (a) performed using 2^{28} vertices per node and 2^{32} edges per node. Strong scaling (b) performed using 2^{32} vertices and 2^{37} edges. All edges are sampled uniformly. Mailboxes are constructed with a fixed mailbox buffer size of 2^{18} for all numbers of nodes.

channel. Keeping the mailbox size fixed, we eventually reach a point where coalescing loses its effectiveness as we add additional compute nodes.

It is worth noting that in degree counting with uniformly sampled edges, every pair of cores is expected to send the same number of messages as all other pairs of cores. This makes broadcasts and delegate vertices unnecessary to keep computation and communication balanced across cores. Under these circumstances without broadcasts, we expect to see Node Local and Node Remote routing performing very similarly, which agrees with our results.

NLNR routing demonstrates excellent scalability out to 1024 compute nodes. For a system with C cores per compute node, an additional C compute nodes must be added to the system before each existing core sees an increase in the size of remote communication channels due to the organization of nodes into layers. Thus, the effectiveness of coalescing is maintained much longer for NLNR routing than the previous routing schemes.

In Fig. 6, we see that for moderate numbers of compute nodes where Node Local, Node Remote, and NLNR routing are scaling well, Node Local and Node Remote routing provide better absolute performance than NLNR. This difference in performance shows the effect of the additional local exchange phase in NLNR routing.

B. Connected Components

To test our simple Connected Component algorithm, we generated RMAT [15] graphs following the Graph500 [4] methodology for both weak and strong scaling experiments. These graphs were chosen because their skewed degree distributions provide computation and communication imbalances, and allow us to incorporate broadcasts to synchronize delegate vertices. Fig 7 shows the scaling properties of the connected components application. We again see poor scaling without routing. Node Local and Node Remote tend to outperm NLNR for fewer than 128 compute nodes. Beyond that point, NLNR tends to exhibit better scaling, with the notable exception of at 1024 nodes in the weak scaling study.

It is important to note the number of broadcasts in weak scaling is not fixed. As the size of the graph increases, the number of vertices with a degree higher than a given threshold will increase as well. The increase in the number of delegates will cause a larger number of broadcasts during delegate synchronization.

In order to offset the growth in the number of delegates, we scaled the delegate threshold with the same scaling as the expected largest degree in the RMAT graphs. This helped to offset the number of broadcasts, but as seen in Fig. 7a, there is still an appreciable increase as graph sizes increase.

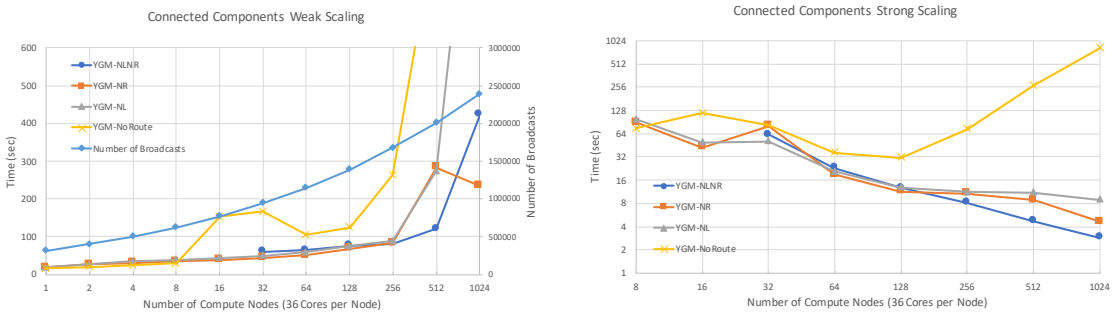
The connected components implementation was designed to demonstrate the performance under heavy broadcast usage. As such, delegate thresholds were chosen to give a larger number of delegates than would typically be desired.

C. Sparse Mat-Vec

Figure 8 gives the scaling results for the SpMV application. Figures 8a and 8c show NLNR providing the best scaling among the included routing schemes.

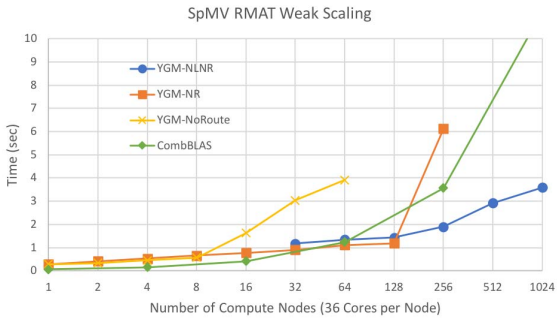
For comparison, we included results of SpMV operations using CombBLAS [19]. CombBLAS provides a distributed graph library using linear algebra primitives. CombBLAS uses a 2D partitioning for matrices stored using compressed sparse column (CSC) or doubly compressed sparse column (DCSC) formats [18], [20]. It is important to note that the operation we are performing is a sparse matrix-dense vector product. One of CombBLAS's strengths comes from its efficient implementation of sparse matrix-sparse vector products [21] which has widespread use in expressing graph algorithms. Our goal with these comparisons is not necessarily to beat CombBLAS performance. CombBLAS is much more sophisticated than our simple SpMV application using YGM and delegates.

Figure 8a shows YGM and CombBLAS SpMV's applied to RMAT graphs (scale 24-32), with the YGM implementation



(a) (b)

Fig. 7: Scaling of connected components application using YGM. Weak scaling (a) performed using 2^{26} vertices per node and 2^{30} edges per node. Growth in the number of broadcast operations is also shown for weak scaling. Strong scaling (b) performed using 2^{30} vertices and 2^{34} edges. Edges generated using RMAT-Graph500 generator. Mailboxes are constructed with a fixed mailbox size of 2^{18} for all numbers of nodes.

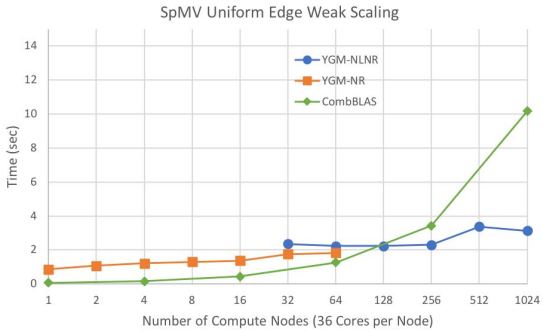


(a)

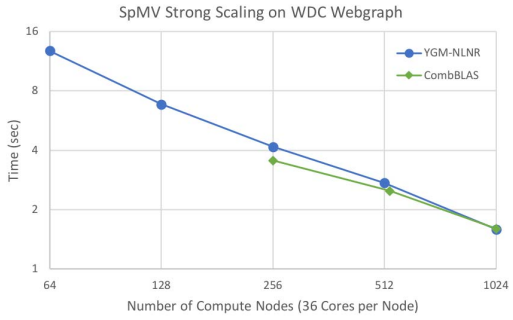
Delegate growth in RMAT weak scaling

Nodes	Number of Delegates
1	101544
2	114849
4	130134
8	154179
16	189445
32	237969
64	303046
128	378275
256	503537
512	653775
1024	848405

(b)



(c)



(d)

Fig. 8: Scaling of sparse mat-vec using YGM and CombBLAS. Weak scaling experiments in (a) used RMAT graphs (parameters 0.57, 0.19, 0.19, 0.05) with 2^{24} vertices per compute node and an edge factor of 16. Growth in the number of delegates used in YGM of RMAT weak scaling experiments shown in (b). Weak scaling experiments in (c) used uniformly generated edges (RMAT with parameters 0.25, 0.25, 0.25, 0.25) with 2^{24} vertices per compute node and an edge factor of 16. Strong scaling results for YGM SpMV implementation using WebDataCommons 2012 webgraph (d). YGM mailbox sizes for weak scaling experiments were 2^{18}. WebDataCommons strong scaling used a mailbox size of $2^{10} \times N$ where N is the number of compute nodes.

making use of delegates to partition high-degree vertices. CombBLAS outperforms YGM's SpMV for relatively small numbers of compute nodes. Scaling past about 64 compute nodes, we see YGM begins to perform better than CombBLAS, with NLNR routing providing the best results for very large number of compute nodes.

While the YGM performance appears promising using RMAT graphs, it is not immediately clear whether this performance is due to the use of YGM as a communication layer or the use of delegate vertices to handle the scale-free nature of the graphs. Figure 8c shows the same test run without using delegates in the YGM implementation and using a graph generated by setting the parameters for the RMAT generator to 0.25, 0.25, 0.25, and 0.25. This will give a graph similar to an Erdős-Rényi graph with uniformly sampled edges. In this case, we see a larger gap between CombBLAS and YGM performance at small numbers of nodes. We do however see the same scaling behavior for the SpMV implementations, allowing YGM to outperform once again for large numbers of nodes without the use of delegates.

Figure 8d shows strong scaling results for the YGM SpMV on the WDC 2012 Webgraph, a large, real-world, scale-free graph [22] featuring 3.5 billion vertices and 128 billion edges. For this experiment, mailbox sizes had to be scaled along with the number of compute nodes. Without increasing the mailbox size, message sizes decreased to the point of preventing scaling of the SpMV. With this modification, we see very similar performance and scaling between YGM and CombBLAS.

VII. Ongoing & Future Work

We are currently investigating techniques to build a hybrid MPI+threads version of YGM. Our current MPI-only implementations require multiple on-node memory copies during each routing step; a hybrid MPI+threads approach can eliminate multiple on-node memory copies. Performance evaluations of a hybrid YGM will be included in future versions of this report. For future work, we are considering building GraphBLAS [23] on top of YGM.

VIII. Conclusion

In this work, we have presented YGM, a pseudo-asynchronous communication layer that makes use of effective routing through local and remote exchanges to provide scalability to large numbers of nodes. YGM has been in use for graph processing applications at LLNL through its inclusion in HavoqGT. Here we presented several simple applications of YGM to demonstrate its scalability. While YGM is effective for graph algorithms, it is more broadly applicable to data-dependent problems that may feature imbalances in compuation and communication across cores.

IX. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-770057). Experiments were performed at the Livermore Computing facility.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, no. 01, pp. 5–20, 2007. [Online]. Available: https://doi.org/10.1142/S0129626407002843

[2] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Supercomputing*, 2014, pp. 549–559. [Online]. Available: https://doi.org/10.1109/SC.2014.50

[3] "Havoqgt." [Online]. Available: https://github.com/LLNL/havoqgt

[4] J. Ang, B. Barrett, K. Wheeler, and R. Murphy, "Introducing the graph 500," *Cray User's Group (CUG)*, 2010.

[5] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: Parallel programming for data-driven applications," in *Proceedings of the International Conference on Supercomputing (ICS '11)*. New York, NY, USA: ACM, 2011, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995934

[6] J. J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine, "Am++: A generalized active message framework," in *Par. Arch. and Comp. Tech.*, Sep. 2010, pp. 401–410.

[7] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de brujin graph construction and traversal for de novo genome assembly," in *Supercomputing*, 2014, pp. 437–448.

[8] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "Hipmer: an extreme-scale de novo genome assembler," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–11.

[9] J. Chapman, I. Ho, S. Sunkara, S. Luo, G. Schroth, and D. S. Rokhsar, "Meraculous: De novo genome assembly with short paired-end reads," *PLoS ONE*, vol. 6, no. 8, p. e23501, 2011.

[10] A. Bienz, W. D. Gropp, and L. N. Olson, "Node Aware Sparse Matrix-Vector Multiplication," p. arXiv:1612.08060, Dec 2016.

[11] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of mpi collective communication on BlueGene/L systems," in *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. New York, NY, USA: ACM, 2005, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088183

[12] "MPI_Ialltoallv." [Online]. Available: https://www.mpich.org/static/docs/v3.2/www3/MPI_Ialltoallv.html

[13] W. S. Grant and R. Voorhies, "cereal - a c++11 library for serialization," 2017. [Online]. Available: http://uscilab.github.io/cereal/

[14] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *J. Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0196677482900086

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," *Proceedings of the SIAM Conference on Data Mining (SDM)*, vol. 6, 2004.

[16] C. Seshadhri, A. Pinar, and T. G. Kolda, "An in-depth analysis of stochastic kronecker graphs," *Journal of the ACM*, vol. 60, 02 2011.

[17] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Supercomputing*, 2011, pp. 65:1–65:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063471

[18] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–11.

[19] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011. [Online]. Available: https://doi.org/10.1177/1094342011403516

[20] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *IN SPAA*, 2009, pp. 233–244.

[21] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 688–697, 2017.

[22] "Web data commons webgraph," 2012. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/

[23] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 643–652.