Exploring MPI Communication Models for Graph Applications Using Graph Matching as a Case Study

Sayan Ghosh*, Mahantesh Halappanavar†, Ananth Kalyanaraman*, Arif Khan†, Assefaw H. Gebremedhin*

* Washington State University, {sayan.ghosh, ananth, assefaw.gebremedhin}@wsu.edu
† Pacific Northwest National Laboratory, {hala, ariful.khan}@pnnl.gov

***Abstract*—Traditional implementations of parallel graph operations on distributed memory platforms are written using Message Passing Interface (MPI) point-to-point communication primitives such as Send-Recv (blocking and nonblocking). Apart from this classical model, the MPI community has over the years added *other* communication models; however, their suitability for handling the irregular traffic workloads typical of graph operations remain comparatively less explored. Our aim in this paper is to study these relatively underutilized communication models of MPI for graph applications. More specifically, we evaluate MPI's one-sided programming, or Remote Memory Access (RMA), and nearest neighborhood collectives using a process graph topology. There are features in these newer models that are intended to better map to irregular communication patterns, as exemplified in graph algorithms.**

As a concrete application for our case study, we use distributed memory implementations of an approximate weighted graph matching algorithm to investigate performances of MPI-3 RMA and neighborhood collective operations compared to nonblocking Send-Recv. A matching in a graph is a subset of edges such that no two matched edges are incident on the same vertex. A maximum weight matching is a matching of maximum weight computed as the sum of the weights of matched edges. Execution of graph matching is dominated by high volume of irregular memory accesses, making it an ideal candidate for studying the effects of various MPI communication models on graph applications at scale.

Our neighborhood collectives and RMA implementations yield up to $6\times$ speedup over traditional nonblocking Send-Recv implementations on thousands of cores of the NERSC Cori supercomputer. We believe the lessons learned from this study can be adopted to benefit a wider range of graph applications.

***Keywords*—Graph analytics, One-sided communication, MPI-3 RMA, MPI-3 neighborhood collectives, Graph matching.**

I. Introduction

Single program multiple data (SPMD) using message passing is a popular programming model for numerous scientific computing applications running on distributed-memory parallel systems. Message Passing Interface (MPI) is a standardized interface for supporting message passing through several efforts from vendors and research groups. Among the communication methods in MPI, point-to-point Send-Recv and collectives have been at the forefront in terms of their usage and wide applicability. Especially, for distributed-memory graph analytics, Send-Recv remains a popular model, due to its wide portability across MPI implementations and support for communication patterns (asynchronous point-to-point updates) inherent in many graph workloads.

In addition to the classical Send-Recv model, Remote Memory Access (RMA), or one-sided communication model, and neighborhood collective operations are recent advanced features of MPI that are relevant to applications with irregular communication patterns. A one-sided communication model separates communication from synchronization, allowing a process to perform asynchronous nonblocking updates to remote memory. Prior to the recent release of MPI-3 [11], MPI RMA had several limitations that made it unsuitable for one-sided operations in applications. Bonachea and Duell [3] discuss the limitations of MPI-2 RMA compared with contemporary Partition Global Address Space (PGAS) languages.

Data locality and task placement are crucial design considerations to derive sustainable performance from the next generation of supercomputers. Presently, MPI offers enhanced support for virtual topologies to exploit nearest neighborhood communication [18]. To derive the maximum benefit from the topology, there are special communication operations in MPI, namely *neighborhood collectives* (abbreviated as NCL), to collectively involve only a subset of processes in communication. MPI neighborhood collectives advance the concepts of standard collective operations, and build up on decades of past research effort in optimized algorithms for collective operations. We provide an overview of these models in §II.

From an application perspective, graph algorithms have recently emerged as an important class of applications on parallel systems, driven by the availability of large-scale data and novel algorithms of higher complexity. However graph algorithms are challenging to implement [25]. Communication patterns and volume are dependent on the underlying graph structure. Therefore, a critical aspect of optimization in such applications is in designing the granularity of communication operations, and the choice of communication primitives.

Summary of contributions: In this paper, we use graph matching, a prototypical graph problem, as a case study to evaluate the efficacy of different communication models. Given a graph $G = (V, E, \omega)$, a matching M is a subset of edges such that no two edges in M are incident on the same vertex. The weight of a matching is the sum of the weight of the matched edges, and the objective of the maximum weight matching problem is to find a matching of maximum weight. Algorithms to compute optimal solutions are inherently serial and are impractical for large-scale problems, albeit having polynomial time complexity [12, 23]. However, efficient approximation algorithms with expected linear time complexity can be used to compute high quality solutions [10, 29]. Further, several of the half-approximation algorithms can be parallelized efficiently [5, 26]. We use the parallel variant of the *locally-dominant* algorithm in this work to explore the efficacy of three communication models (detailed in §IV). In particular, we implemented the locally-dominant algorithm using Send-Recv, RMA and neighborhood collectives. To the best of our knowledge, it is the first time that these communication schemes have been used for half-approximate matching.

Secondly, we devised a detailed case study for understand-

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00085

ing and characterizing the performance efficacy of different communication models for graph matching under different measures. Our *working hypothesis* is that the newer communication models of RMA and NCL are likely to outperform the classical Send-Recv model for graph matching. The goal of the case study was to test this hypothesis using the three implementations of half-approximate matching.

Summary of findings: While our study demonstrates the general validity of our working hypothesis (at a high level), it also paints a more nuanced picture about the way these individual models differ with input distributions, ordering schemes, and system sizes. More specifically:

- We demonstrate that with our RMA and NCL implementations, one can achieve up to $\mathbf{4.5}\times$ speedup for a billion-edge real-world graph relative to Send-Recv (§V).
- We demonstrate that while RMA is more consistent at delivering high performance, NCL is more sensitive to the type of input graph and to input vertex ordering.
- Besides runtime performance, we also evaluate the energy and memory consumption costs of these different models for graph matching, and show that NCL and RMA significantly reduce these costs as well. Our study makes a case for not viewing any one of these metrics in isolation but to identify models that are likely to achieve *best tradeoffs* under different configurations.

II. Building parallel graph algorithms using MPI-3 RMA and Neighborhood Collectives

A number of graph-based algorithms are iterative in nature, requiring updates to different subset of vertices in each iteration, with some local computation performed on the vertices by the current process. This is referred to as the *owner-computes* model. Implementing such *owner-computes* graph algorithms on distributed-memory using MPI Send-Recv typically requires asynchronous message exchanges to update different portions of the graph simultaneously, giving rise to irregular communication patterns. Algorithm 1 illustrates a generic iterative graph algorithm, representing the *owner-computes* model.

Algorithm 1: Prototypical distributed-memory *owner-computes* graph algorithm using nonblocking Send-Recv for communication. `Compute` function represents some local computation by the process "owning" a vertex.
Input: $G_i = (V_i, E_i)$ portion of the graph G in rank i.

```
1:  while true do
2:    X_g ← Recv messages
3:    for {x, y} ∈ X_g do
4:      Compute(x, y) {local computation}
5:    for v ∈ V_i do
6:      for u ∈ Neighbor(v) do
7:        Compute(u, v) {local computation}
8:        if owner(u) ≠ i then
9:          Nonblocking Send(u, v) to owner(u)
10:   if processed all neighbors then
11:     break and output data
```

Within the realm of the *owner-computes* model, it is possible to replace Send-Recv with other viable communication models. Our distributed-memory approximate matching implementations follow the *owner-computes* model and serve as an ideal use-case to study the impact of different communication models, while being representative of a wide variety of distributed-memory graph algorithms. We briefly discuss two alternative communication methods in this section, namely Remote Memory Access (RMA) and neighborhood collective operations. MPI RMA was presented in the MPI-2 specification (circa 1997). MPI-3 was the next major update to the MPI standard (circa 2012), that introduced neighborhood collectives and included significant extensions to MPI RMA.

Remote Memory Access (RMA): In contrast to MPI two-sided model, MPI RMA extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. To achieve this, every process involved in an RMA operation needs to expose part of its memory such that other processes can initiate one-sided data transfers targeting the memory of any particular process. The exposed memory is referred to as an MPI *window*. After creation of an MPI window, processes can initiate put/get one-sided operations. For synchronization, users need to invoke a *flush* operation, which completes its outstanding RMA operations. Two categories of MPI RMA communication exist: *active* and *passive*. For an active target communication, both origin and target processes are involved in the communication, but for passive target communication, only the origin process is involved in data transfer. We use passive target communication, since it is better suited for random accesses to different target processes [19].

Neighborhood collectives: Distributed-memory graph algorithms exhibit sparse communication patterns, and are therefore ideal candidates for exploring the relatively recent enhancements to MPI process topology interface [18]. Distributed graph topology allows each process to only define a subset of processes (as edges) that it communicates with. Neighborhood collective operations make use of this graph topology, in optimizing the communication among neighboring processes [16, 17, 27]. Fig. 1 represents the process graph topology, mimicking the underlying data distribution (we do not assume fixed neighborhoods). A graph topology can also be augmented with edge weights representing communication volumes between nodes, at the expense of extra memory. However, we only use unweighted process graphs in this paper. In the context of the *owner-computes* model, the primary impact of using a distributed graph topology is reduction in memory consumption, which in this case is expected to be proportional to the subset of processes as opposed to all the processes in the communicating group. We provide supporting information for this reduction in Table VIII.

III. Half-Approximate Matching

We discuss in this section the half-approximate matching algorithm in serial, and in Section IV we discuss its parallelization in distributed-memory.

A. Matching preliminaries

A matching M in a graph is a subset of edges such that no two edges in M are incident on the same vertex. The objective of a matching problem can be to maximize the number of edges in a matching, known as the maximum matching, or to maximize the sum of the weights of the matched edges, known as the maximum weight matching (when weights are associated with the edges). A further distinction can be on the optimality of the solutions computed – optimal or approximate. We limit our scope to half-approx weighted matching

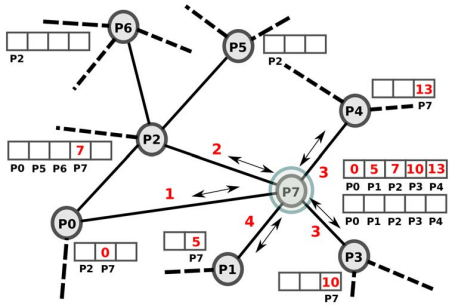
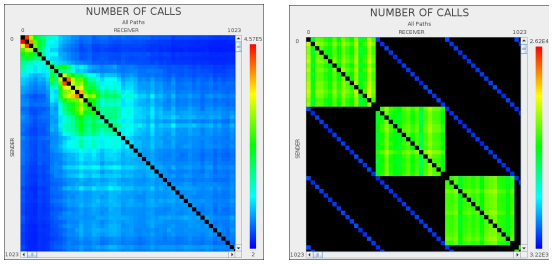


Fig. 1: Subset of a process neighborhood and MPI-3 RMA remote displacement computation. Number of ghost vertices shared between processes are placed next to edges. Each process maintains two $\mathcal{O}(neighbor)$ sized buffers (only shown for **P7**): one for storing prefix sum on the number of ghosts for maintaining outgoing communication counts, and the other for storing remote displacement start offsets used in MPI RMA calls. The second buffer is obtained from `alltoall` exchanges (depicted by arrows for **P7**) of the prefix sum buffer among the neighbors.



(a) MPI calls, graph matching. (b) MPI calls, Graph500 BFS.

Fig. 2: Communication volumes (in terms of Send-Recv invocations) of MPI Send-Recv baseline implementation of half-approx matching using Friendster (1.8B edges) and Graph500 BFS using R-MAT graph of 2.14B edges on 1024 processes. Black spots indicate zero communication. The vertical axis represents the sender process ids and the horizontal axis represents the receiver process ids.

algorithms in this paper. We refer the reader to several classical articles on matching for further details [13, 21, 23].

A simple approach to compute a half-approx matching is to consider edges in a non-increasing order of weights and add them to the matching if possible. This algorithm was proposed by Avis and is guaranteed to produce matchings that are half-approx to optimal matching [1]. However, ordering of edges serializes execution. Pries proposed a new algorithm by identifying *locally dominant edges* – edges that are heavier than all their adjacent edges – without a need to sort the edges [29]. The locally dominant algorithm was adapted to a distributed algorithm by Hoepman [20], and into a practical parallel algorithm by Manne and Bisseling [26] and Halappanavar *et al.* [15]. We build on the work on MatchboxP [5] and implement novel communication schemes. We note that each communication model is a nontrivial implementation and requires significant modifications to the Send-Recv based algorithm. Further, communication patterns generated by matching are distinctly different from available benchmarks such as Graph500, as shown in Fig. 2, which makes it a better candidate to explore novel communication models in MPI-3.

Intuitively, the locally-dominant algorithm works by identifying dominant edges in parallel, adding them to the matching, pruning their neighbors and iterating until no more edges can be added. A simple approach to identify locally dominant edges is to set a pointer from each vertex to its current heaviest neighbor. If two vertices point at each other, then the edge between these vertices is locally dominant. When we add this edge to the matching, only those vertices pointing to the endpoints of the matched edge need to be processed to find alternative matches. Thus, the algorithm iterates through edges until no new edges are matched. This algorithm has been proved to compute half-approx matchings [26]. The algorithm has expected linear time complexity but suffers from a weakness when there are no edge weights (or weights are all equal) and ties need to be broken using vertex ids [26]. However, a simple fix for this issue is to use a hash function on vertex ids to prevent linear dependences in pathological instances such as paths and grids with ordered numbering of vertices.

B. Serial algorithm for half-approximate matching

Algorithm 2 demonstrates the serial half-approximate matching algorithm, based on [26]. There are two phases in the algorithm: in the first phase, the initial set of locally dominant edges in $G = (V, E, \omega)$ are identified and added to matching set M; the next phase is iterative—for each vertex in M, its unmatched neighboring vertices are matched. For a particular vertex v, N'_v represents unmatched vertices in v's neighborhood. The vertex with the heaviest unmatched edge incident on v is referred as v's mate, and this information is stored in a data structure called $mate$. Throughout the computation, $mate$ of a vertex can change as it may try to match with multiple vertices in its neighborhood. We use similar notations for the parallel algorithms as well.

Algorithm 2: Serial matching algorithm.
Input: Graph $G = (V, E, \omega)$.
Output: M set of matched vertices.

1: $mate_v \leftarrow \emptyset \quad \forall v \in V$, $M \leftarrow \emptyset$
2: **for** $v \in V$ **do**
3: $\quad u \leftarrow mate_v \leftarrow \arg\max_{u \in N'_v} \omega_{u,v}$
4: $\quad$ **if** $mate_u = v$ **then**
5: $\quad\quad M \leftarrow M \cup \{u, v\}$
6: **while** true **do**
7: $\quad v \leftarrow$ *some vertex from* M
8: $\quad$ **for** $x \in N'_v$ *where* $mate_x = v$ *and* $x \notin M$ **do**
9: $\quad\quad y \leftarrow mate_x \leftarrow \arg\max_{y \in N'_x} \omega_{x,y}$
10: $\quad\quad$ **if** $mate_y = x$ **then**
11: $\quad\quad\quad M \leftarrow M \cup \{x, y\}$
12: $\quad$ **if** *processed all vertices in* M **then**
13: $\quad\quad$ break

IV. Parallel Half-Approximate Matching

In distributed-memory, we need to communicate information on candidate mates, and a way to express the context of the data that is being communicated. We discuss these communication contexts in detail in the next subsection and subsequently present our distributed-memory implementation of the matching algorithm. But first we describe the strategy we follow for graph distribution.

A. Graph distribution

Our graph distribution is 1D vertex-based, which means each process owns some vertex, and all of its edges. The input graph is stored as a directed graph, so each process stores some vertices that are owned by a different process in order to maintain edge information. For example, if vertex u is owned by process #0 and vertex v is owned by process #1,

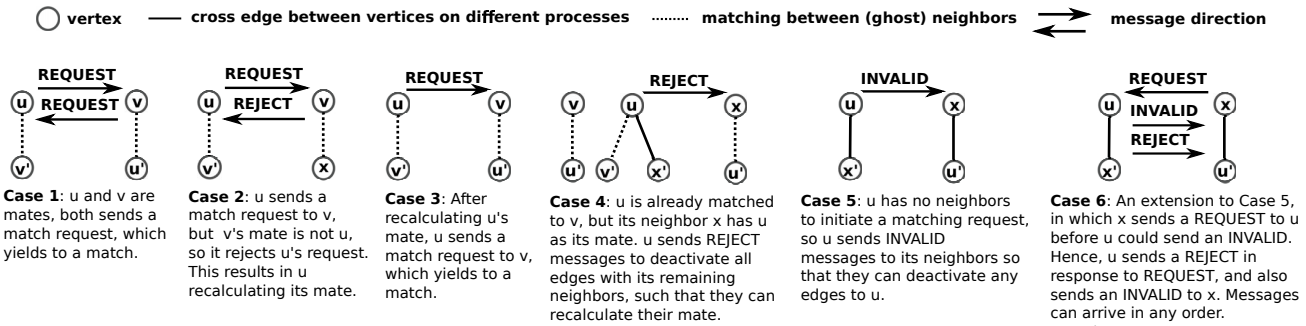


Fig. 3: Communication contexts depicting different scenarios in distributed-memory half-approx matching. If y is a vertex, then y' is its "ghost" vertex.

and there is an edge between u and v, then process #0 stores $u - v'$, and process #1 stores $v - u'$ (where u'/v' are "ghost" vertices, and there is an undirected edge between process #0 and #1 in the distributed graph process topology). We store locally owned vertices and edges using Compressed Sparse Row (CSR) format [9]. The $owner$ function takes a vertex as an input parameter and returns its owning process. This simple distribution may cause load imbalance for graphs with irregular node degree distributions; in §V-C we investigate the impact of graph reordering techniques and load imbalances.

B. Communication contexts

The message contexts are used to delineate actions taken by a vertex to inform its status to its neighbors and to avoid conflicts. For Send-Recv implementation, these contexts are encoded with message tags, and for RMA and neighborhood collective implementations, they are part of the communication data. A vertex is done communicating with its ghost vertices when all its cross edges have been deactivated, and is no longer a part of a candidate set. Fig. 3 demonstrates communication contexts arising from different scenarios as the distributed-memory algorithm progresses.

From these scenarios, we conclude that a vertex may send at most 2 messages to a "ghost" vertex (i.e., an incident vertex owned by another process), so the overall number of outgoing/ incoming messages is bounded by twice the number of ghost vertices. This allows us to precompute communication buffer sizes, making it convenient for memory allocation. Based on the ghost counts, a process can stage incoming or outgoing data associated with a particular neighbor process on discrete locations in its local buffers.

C. Distributed-memory algorithm

Algorithm 3 demonstrates the top-level implementation of our distributed-memory half-approx matching. Similar to the serial algorithm 2, the parallel algorithm also consists of two phases: in the first phase, locally owned vertices attempt to match with the unmatched vertices on its neighborhood (FINDMATE, Algorithm 4), and in the next phase, neighbors of matched vertices in M_i are processed one by one (PROCESSNEIGHBORS, Algorithm 5).

We maintain a per process counter array called $nghosts$ that tracks the number of "active" ghost vertices in its neighborhood (that are still unmatched and available). As the ghost vertices are processed, the count is decremented. When the sum of the content of $nghosts$ array returns 0, it means that a process does not have an active ghost vertex, and is free to exit if it has no pending communication. An `MPI_Allreduce` operation to aggregate ghost counts may also be required for RMA and NCL to exit the iteration.

Algorithm 3: Top-level distributed-memory algorithm.
Input: Local portion of graph $G_i = (V_i, E_i)$ in rank i.

```
1:  mate_v ← ∅   ∀v ∈ V_i, nghosts_i ← ∅, M_i ← ∅
2:  for v ∈ V_i do
3:      FINDMATE(v)
4:  while true do
5:      PROCESSINCOMINGDATA()
6:      v ← some vertex from M_i
7:      if owner(v) = i then
8:          PROCESSNEIGHBORS(v)
9:      if SUM(nghosts_i) = 0 then
10:         break
```

We use generic keywords in the algorithms (such as `Push`, `Evoke` and `Process`) to represent communication/ synchronization/buffering methods used across multiple MPI versions. Table I provides a mapping of those keywords to the actual MPI functions used by a specific implementation.

TABLE I: Description of keywords used in algorithms.

Keyword/Action	Send-Recv	RMA	Neighborhood Collectives
`Push` (mark data for imminent communication)	MPI_Isend	MPI_Put	Insert data into send buffer.
`Evoke` (evoke outstanding communication)	MPI_Iprobe	MPI_Win_flush_all MPI_Neighbor_alltoall	MPI_Neighbor_alltoall MPI_Neighbor_alltoallv
`Process` (handle incoming data)	MPI_Recv	Check data in local MPI window.	Check data in receive buffer.

The FINDMATE function, depicted in Algorithm 4, is used whenever a locally owned vertex has to choose an unmatched vertex with maximum weight (i.e., mate) from its neighborhood. Apart from initiating matching requests ($REQUEST$ communication context), if there are no available vertices in the neighborhood for matching, then it can also eagerly send an $INVALID$ message to all its neighbors, such that they can deactivate the edge (Case #5 from Fig. 3). After a vertex receives an $INVALID$ message from its neighboring vertex, the neighbor can no longer be considered as a potential candidate. Therefore, $INVALID$ messages are broadcast by vertices that cannot be matched with the goal of minimizing futile matching requests.

Edge deactivation involves evicting an unavailable vertex from the neighborhood candidate set of potential mates (for

Algorithm 4: FINDMATE: Find candidate mate of a vertex in rank i.
Input: Locally owned vertex x.

1: $y \leftarrow mate_x \leftarrow \arg\max_{y \in N'_x} \omega_{x,y}$
2: **if** $y \neq \emptyset$ **then** {Initiate matching request}
3: **if** $owner(y) = i$ **then**
4: **if** $mate_y = x$ **then**
5: $N'_x \setminus y$
6: $N'_y \setminus x$
7: $M_i \leftarrow M_i \cup \{x, y\}$
8: **else**{y is a ghost vertex}
9: $N'_x \setminus y$
10: $nghosts_y \leftarrow nghosts_y - 1$
11: **Push**$(REQUEST, owner(y), \{y, x\})$
12: **else**{Invalidate N'_x}
13: **for** $z \in N'_x$ **do**
14: **if** $owner(z) = i$ **then**
15: $N'_x \setminus z$
16: $N'_z \setminus x$
17: **else**
18: $N'_x \setminus z$
19: $nghosts_z \leftarrow nghosts_z - 1$
20: **Push**$(INVALID, owner(z), \{z, x\})$

Algorithm 5: PROCESSNEIGHBORS: Process active neighbors of a matched vertex in rank i.
Input: Locally owned matched vertex v.

1: **for** $x \in N'_v$ **do**
2: **if** $mate_v \neq x$ **then**
3: **if** $owner(x) = i$ **then**
4: $N'_v \setminus x$
5: $N'_x \setminus v$
6: **if** $mate_x = v$ **then**{Recalculate $mate_x$}
7: **FINDMATE**(x)
8: **else**
9: $N'_v \setminus x$
10: $nghosts_x \leftarrow nghosts_x - 1$
11: **Push**$(REJECT, owner(x), \{x, v\})$

e.g., $N'_x \setminus y$ represents evicting vertex y from the candidate set of vertex x). When the unavailable vertex is a ghost, then the $nghosts$ counter needs to be decremented as well.

The PROCESSNEIGHBORS function helps to mitigate potential conflicts in the neighborhood of a matched vertex. After the first phase, multiple vertices (denoted by set X) in the neighborhood of a matched vertex v may list v as a mate. However, since v is already matched and unavailable from the candidate sets, $\forall x \in X, x \notin M$. In that case, PROCESSNEIGHBORS evokes mate recalculation for x if it is locally owned, or sends a $REJECT$ message to the owner of x (Case# 4 of Fig. 3).

PROCESSINCOMINGDATA ensures proper handling of incoming data from another process. Based on the received communication context, relevant action is taken; that involves edge deactivation, along with mate recalculation or successful matching or rejection of a matching request.

D. Implementation of the distributed-memory algorithms

In this section, we discuss implementations of the distributed-memory algorithms using MPI Send-Recv, RMA and neighborhood collectives.

Algorithm 6: PROCESSINCOMINGDATA: Process incoming data in rank i.

1: $flag \leftarrow$**Evoke**()
2: **if** $flag = true$ **then** {received data}
3: $\{x, y, ctx\} \leftarrow$ **Process** incoming data
4: **if** $ctx.id = REQUEST$ **then**
5: $matched \leftarrow false$
6: **if** $x \notin M_i$ **then**
7: $N'_x \setminus y$
8: **if** $mate_x = y$ **then**
9: $M_i \leftarrow M_i \cup \{x, y\}$
10: $matched \leftarrow true$
11: **if** !$matched$ **then** {push $REJECT$ if match not possible}
12: $N'_x \setminus y$
13: $nghosts_y \leftarrow nghosts_y - 1$
14: **Push**$(REJECT, ctx.source, \{y, x\})$
15: **else if** $ctx.id = REJECT$ **then**
16: $N'_x \setminus y$
17: $nghosts_y \leftarrow nghosts_y - 1$
18: **if** $mate_x = y$ **then**
19: **FINDMATE**(x)
20: **else**{received $INVALID$}
21: $N'_x \setminus y$
22: $nghosts_y \leftarrow nghosts_y - 1$

a) MPI Send-Recv implementation: The baseline Send-Recv implementation uses `MPI_Isend` to initiate a nonblocking send operation for communicating with a neighbor process. It uses a nonblocking probe call (i.e., `MPI_Iprobe`) in every iteration, before receiving the message (using `MPI_Recv`), as there is no prior information on incoming message due to irregular communication. The communication context is encoded in the message tags. At present, we do not aggregate outgoing messages, therefore, PROCESSINCOMINGDATA checks for incoming messages and receives them one at a time.

b) MPI-3 RMA implementation: In MPI RMA, a process needs to calculate the memory offset (also referred to as *target displacement*) in the target process's memory in order to initiate a one-sided data transfer. Calculation of the target displacement is nontrivial and error prone. A straightforward way to calculate remote displacement is via a counter held by every process in the communicating group. However, maintaining a distributed counter requires extra communication, and relatively expensive atomic operations.

In Fig. 1, we show an alternate way to precompute remote data ranges for a process graph neighborhood. The amount of data exchanged between two nodes of a process graph is proportional to the number of ghost vertices. A prefix sum on the number of ghosts a process is sharing with each of its neighbor (process) allows a process to logically partition its local outgoing buffer among target processes, to avoid overlaps or conflicts. After performing the prefix sum over the number of ghosts, an `MPI_Neighbor_alltoall` within a process neighborhood informs a process of a unique offset that it can use in RMA calls targeting a particular neighbor. In addition to this scheme, each process would just need to maintain a local counter per neighboring process. There is also no way to determine incoming data size in MPI RMA without an extra communication. Hence, we issue an `MPI_Neighbor_alltoall` on a subset of processes, to exchange outgoing data counts among the process neighborhood. This may lead to load imbalance when the process

neighborhood sizes are disproportionate.

For the RMA version, before the MPI window can be accessed, we have to invoke a *flush* synchronization call, to ensure completion of current outstanding one-sided data transfers at the origin and remote side.

c) MPI-3 neighborhood collectives implementation: We use blocking neighborhood collective operations on a distributed graph topology. Specifically, we use `MPI_Neighbor_alltoall` and `MPI_Neighbor_alltoallv` to exchange data among neighbors. The distributed graph topology is created based on ghost vertices that are shared between processes following our 1D vertex-based graph distribution. Unlike MPI RMA or Send-Recv cases, where communication is initiated immediately, in this case, the outgoing data is stored in a buffer for later communication. Nearest neighborhood communication is invoked using this buffer once every iteration. The idea is to allow data aggregation before initiating collective communication. The outgoing data counts are exchanged among using `MPI_Neighbor_alltoall`, allowing a process to receive incoming data counts and prepare the receive buffer. The performance of neighborhood collectives on our distributed graph topology may be suboptimal in certain cases, especially since we do not make any assumptions about the underlying graph structure.

Although we discuss distributed-memory implementations using half-approx graph matching as a case study, our MPI communication substrate comprising of Send-Recv, RMA and neighborhood collective routines can be applied to any graph algorithm imitating the *owner-computes* model.

V. Experimental evaluation

In this section, we present results and observations from our experimental evaluation using a set of synthetic and real-world graphs with diverse characteristics. In the context of neighborhood collective model, we study the impact of graph reordering using the Reverse Cuthill-McKee (RCM) algorithm [6, 24]. We also provide results from comparing the performance of our implementations with a similar implementation of half-approx matching named MatchBox-P [5]. MatchBox-P uses the MPI Send-Recv model. Since this implementation has limitations on the size and types of inputs it can process, we compare our results with MatchBox-P only for moderate-sized inputs.

A. Notations and experimental setup

Notations. We use the following descriptors in the figures and tables listed in this section to refer to the different variants of our parallel algorithm presented in §IV:

- **`NSR`**: Baseline parallel version using nonblocking MPI Send-Recv.
- **`RMA`**: Uses MPI-3 RMA, internally it also uses neighborhood collectives to exchange incoming data counts among neighbors.
- **`NCL`**: Uses blocking MPI-3 neighborhood collectives.
- **`MBP`**: Nonblocking MPI Send-Recv in MatchBox-P.

In the process graph, each process is represented by a unique vertex or node, and an edge is added to each node that it shares neighborhood with (see Fig. 1). If the degree of a node is high, it participates in a lot of neighborhoods resulting in larger volume of communication. The number of edges in the process graph is represented by $|E_p|$, whereas the total number of edges in the input graph, including the edges connected to ghost vertices is denoted by $|E'|$. The average and maximum node degrees in the process graph are represented by d_{avg} and d_{max}. In this context, standard deviation applies to degrees in the process graph (denoted by σ_d), and edges augmented with ghost vertices (denoted by $\sigma_{|E'|}$). We follow these notations throughout the rest of the paper.

Computing platform. We used the NERSC Cori supercomputer for our experimental evaluations. NERSC Cori is a 2,388-node Cray® XC40™machine with dual-socket Intel® Xeon™E5-2698v3 (Haswell) CPUs at 2.3 GHz per node, 32 cores per node, 128 GB main memory per node, 40 MB L3 cache/socket and the Cray® XC™series interconnect (Cray® Aries™with Dragonfly topology). We use cray-mpich/7.7.0 as our MPI implementation, and Intel® 18.0.1 compiler with `-O3 -xHost` compilation option to build the codes. We use the TAU profiling tool [30] to generate point-to-point communication matrix plots.

Dataset. We summarize the datasets used for evaluation in Table II. We use different types of synthetically generated graphs: Random geometric graphs (RGGs); R-MAT graphs, used in Graph500 BFS benchmark; and, stochastic block partition graphs (based on the degree-corrected stochastic block models). Datasets were obtained from the SuiteSparse Matrix Collection[1] and MIT Graph Challenge website[2].

TABLE II: Synthetic and real-world graphs used for evaluation.

Graph category	Identifier	$\lvert V \rvert$	$\lvert E \rvert$
Random geometric graphs (RGG)	d=8.56E-05	536.87M	6.64B
	d=6.12E-05	1.07B	13.57B
	d=4.37E-05	2.14B	27.73B
Graph500 R-MAT	Scale 21	2.09M	33.55M
	Scale 22	4.19M	67.10M
	Scale 23	8.38M	134.21M
	Scale 24	16.77M	268.43M
Stochastic block partitioned graphs	high overlap,low block sizes (HILO)	1M	23.7M
	—"—	5M	118.7M
	—"—	20M	475.16M
Protein K-mer	V2a	55M	117.2M
	U1a	67.7M	138.8M
	P1a	139.3M	297.8M
	V1r	214M	465.4M
DNA	Cage15	5.15M	99.19M
CFD	HV15R	2.01M	283.07M
Social networks	Orkut	3M	117.1M
	Friendster	65.6M	1.8B

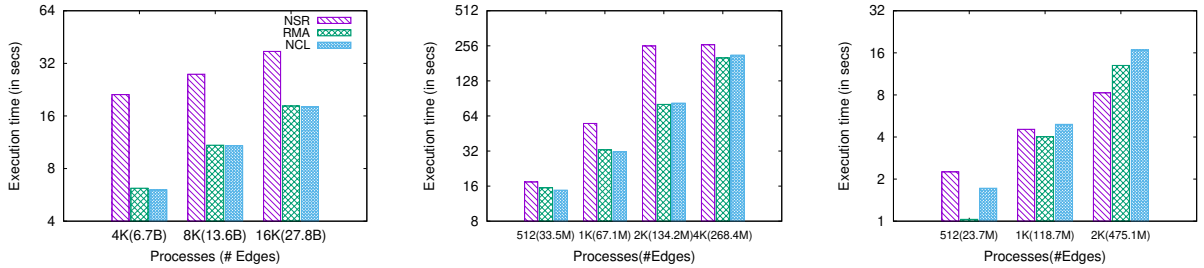
B. Scaling analysis and comparison with MatchBox-P

We present execution time in seconds for different inputs in this section. Data is presented in $\log_2$ scale for both X axis and Y axis. We present both strong scaling and weak scaling results. We first present weak scaling performance of three classes of synthetic graphs in Fig. 4. Our distributed-memory implementation of random geometric graph (RGG) generator is such that any process executing matching on the subgraphs will communicate with at most two neighboring processes. By restricting the neighborhood size to two, we observe $2-3.5\times$ speedup on 4-16K processes for both NCL and RMA versions relative to NSR, for multi-billion-edge RGG (Fig. 4a). In Fig. 4b, we demonstrate the weak scaling performance of moderate sized Graph 500 R-MAT graphs (with $33-268$M edges) on 512 to 4K processes. We observe about $1.2-3\times$ speedup for RMA and NCL relative to NSR.

Fig. 4c demonstrates contrasting behavior, where NSR performs better than NCL and RMA, using a stochastic

[1]https://sparse.tamu.edu

[2]http://graphchallenge.mit.edu/data-sets



(a) Random geometric graphs on 4K-16K processes.

(b) Graph500 R-MAT graphs on 512-4K processes.

(c) Stochastic block-partitioned graphs on 512-2K processes.

Fig. 4: Weak scaling of NSR, RMA, and NCL on synthetic graphs.

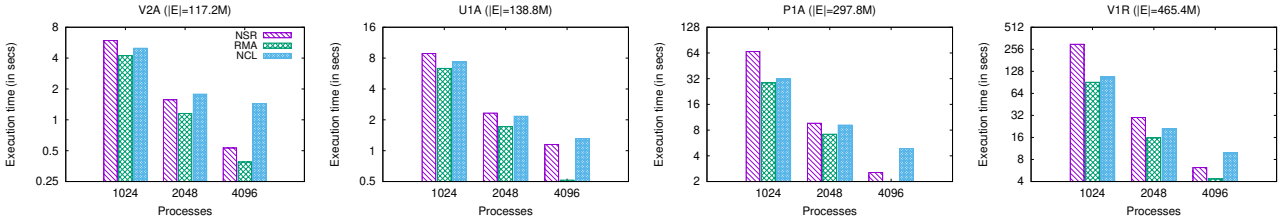


Fig. 5: Strong scaling results on 1K-4K processes for different instances of Protein K-mer graphs.

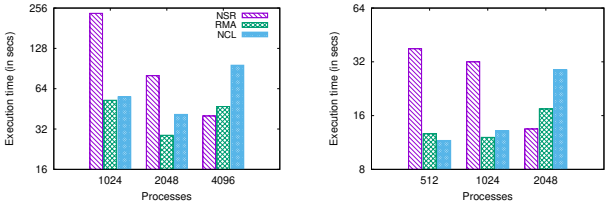
block partitioned graph (450M edges), comprising of clusters of vertices and high degree of connectivity between them. Although NCL scales with the number of processes, NSR performs at least $1.5 - 2.7\times$ better across 512-2K processes. In order to gain better insight on the performance of NCL, we present statistics on the process graph for different number of processes with this input in Table III. Due to high degree of connectivity between processes, NCL/RMA is not efficient for this input (in stark contrast to RGG distribution, where the maximum degree is bounded).

TABLE III: Neighborhood graph topology statistics for stochastic block partitioned graph on 512-2K processes.

p	$\mid E_p \mid$	d_{max}	d_{avg}
512	1.31E+05	511	511
1024	5.24E+05	1023	1023
2048	2.10E+06	2047	2047

We now present performance results from the execution of real-world graphs of moderate to large sizes. For the Protein k-mer graphs in Fig. 5, we observe that RMA performs about $25 - 35\%$ better than NSR and NCL. In some cases, performance of both RMA and NCL was $2 - 3\times$ better than NSR. The structure of k-mer graphs consists of grids of different sizes; when the grids are densely packed, it affects the performance of neighborhood collectives.

Strong scaling results for the two social network graphs, Friendster (1.8B edges) and Orkut (117M edges), are presented in Fig. 6. We observe $2 - 5\times$ speedup for NCL and RMA on 1K and 2K processes, relative to NSR. However, for both the inputs, scalability of NCL and RMA is adversely affected with larger number of processes. Similar to the stochastic block-partition graph, we observe large neighborhood for NCL, as shown in Table IV, resulting in poor performance for NCL. For Friendster, the number of edges connecting ghost vertices ($|E'|$) increase by $4\times$ on 4K processes, whereas for Orkut the increase between 512 and 2048 processes is $14\times$. Since we use blocking collective operations (for RMA/NCL), the degree distribution adversely affects the performance at scale,



(a) Execution times of Friendster on 1K-4K processes.

(b) Execution times of Orkut on 512-2K processes.

Fig. 6: Performance of RMA and NCL on social network graphs.

TABLE IV: Neighborhood graph topology statistics for Friendster and Orkut.

p	$\mid E_p \mid$	d_{max}	d_{avg}	σ_d
Friendster on 2K/4K processes				
2048	2.09E+06	2047	2045	2045.29
4096	8.33E+06	4095	4069	4069.87
Orkut on 512/2K processes				
512	1.30E+05	511	509	509.03
2048	1.84E+06	2047	1797	1808.03

as compared to nonblocking Send-Recv implementation. Consequently, we next present reordering as a potential approach to address this problem (§V-C).

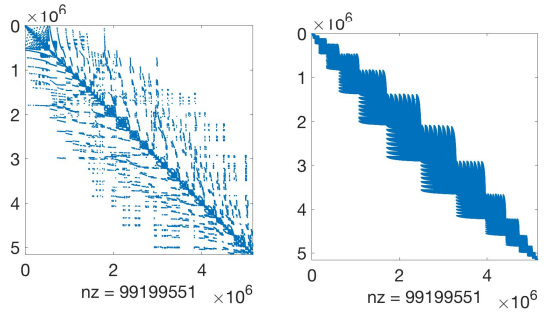
C. Impact of graph reordering

We explore bandwidth minimization using the Reverse Cuthill-McKee (RCM), which can be implemented in linear time and is therefore a practical heuristic for large-scale inputs. The reordered and original matrices that we use in our experiments are presented in Fig. 7.

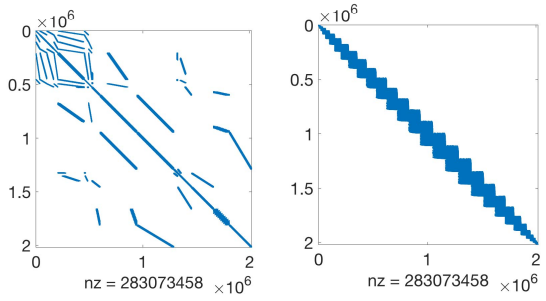
We observe counter-intuitive results with graph reordering, due to a simple 1D vertex-based partitioning of data in our current implementations. We observe that every process experiences an increase in overall communication volume due to an increase in the number of ghost vertices. Table V summarizes this increase by presenting the number of edges augmented by ghost vertices for a given number of partitions (i.e., $|E'|$), for both the original and RCM-based reordered graphs. Overall,

TABLE V: Impact of reordering depicted through the number of edges augmented with the number of ghost vertices for different partitions.

Graph	$\|V\|$	$\|E\|$	Original				RCM			
			$\|E'\|$	$\|E'\|_{max}$	$\|E'\|_{avg}$	$\sigma_{\|E'\|}$	$\|E'\|$	$\|E'\|_{max}$	$\|E'\|_{avg}$	$\sigma_{\|E'\|}$
Cage15 (p = 256)	5.15E+06	9.92E+07	1.88E+08	1.29E+06	7.34E+05	1.64E+05	1.98E+08	8.77E+05	7.74E+05	8.64E+04
HV15R (p = 512)	2.01E+06	2.83E+08	5.62E+08	1.34E+06	1.10E+06	9.29E+04	5.66E+08	1.24E+06	1.11E+06	6.36E+04



(a) Original Cage15. (b) Reordered Cage15.



(c) Original HV15R. (d) Reordered HV15R.

Fig. 7: Rendering of the original graph and RCM reordered graph expressed through the adjacency matrix of the respective graphs (Cage15 and HV15R). Each non-zero entry in the matrix represents an edge between the corresponding row and column (vertices).

we observe $1-5\%$ increase in total and average number of edges for reordered cases due to a balanced number of edges per process. We observe that standard deviation ($\sigma_{|E'|}$) is decreased by $30-40\%$ relative to the original graph distribution. For the same datasets, we summarize the details of process neighborhood in Table VI. We observe that the average node degree (d_{avg}) of RCM-based reordered graphs is about $2\times$ that of the original graphs, and thus, increasing the volume of communication on average. Consequently, NSR suffers a slowdown of $1.2-1.7\times$ for reordered graphs. As illustrated in Fig. 8, NCL exhibits a speedup of $2-5\times$ compared to the baseline Send-Recv version. In Fig. 9, we show the communication profile for the original and reordered variants of HV15R. Although RCM reduces the bandwidth, the irregular block structures along the diagonal can lead to load imbalance. We also note that the two inputs chosen for evaluation have amenable sparsity structure and do not completely benefit from reordering. However, our goal is to show the efficacy of reordering as a good heuristics for challenging datasets in the context of neighborhood collectives.

TABLE VI: Neighborhood topology of original vs RCM reordered graphs.

Graphs	Original				RCM			
	$\|E_p\|$	d_{max}	d_{avg}	σ_d	$\|E_p\|$	d_{max}	d_{avg}	σ_d
Cage15 (p = 256)	3572	58	27.90	9.40	7423	87	57.99	23.91
HV15R (p = 512)	5100	43	19.92	7.38	14403	83	56.26	18.12

In Fig. 8, we also observe NSR performing $1.2-2\times$ better

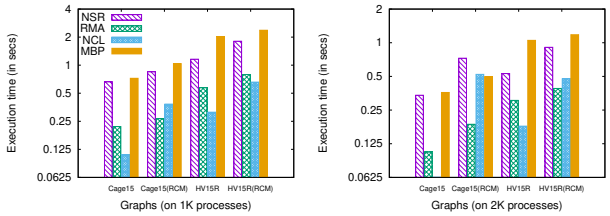
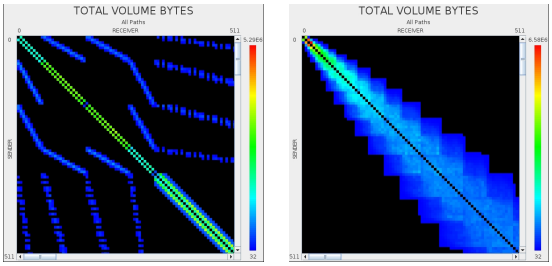


Fig. 8: Comparison of original vs RCM reordering on 1K/2K processes.



(a) Total message volume—original HV15R. (b) Total message volume—reordered HV15R.

Fig. 9: Communication volumes (in bytes) of original HV15R and RCM reordered HV15R. Black spots indicate zero communication. The vertical axis represents the sender process ids and the horizontal axis represents the receiver process ids.

than MBP for large graphs, whereas NCL/RMA consistently outperformed MBP by $2.5-7\times$.

D. Performance summary

We hypothesized that one-sided (RMA) and neighborhood collective (NCL) communication models are superior alternatives to the standard point-to-point (NSR) communication model. But, our findings reveal the trade-offs associated with these versions. In this section, we discuss three aspects to summarize our work: i) Performance of the MPI implementations (Table VII and Fig. 10), ii) Power and memory usage (Table VIII), and iii) Implementation remarks.

Performance of MPI implementations: We use a combination of absolute (summarized in Table VII) and relative (illustrated in Fig. 10) performance information to summarize the overall performance of the three communication models used to implement half-approximate matching. For each input, we list the best performing variant in terms of speedup relative to NSR using data from 512 to 16K process-runs in Table VII. We capture the relative performance using the performance profile shown in Fig. 10. We include data from 50 representative combinations of $(input, \#processes)$ to build this profile. We observe that RMA consistently outperforms the other two, but NCL is relatively close to RMA. However, performance of NSR is up to $6\times$ slower than the other two, but is competitive in about 10% of the inputs.

Power and memory usage: Using three moderate to large inputs on 1K processes (32 nodes), we summarize energy and memory usage in Table VIII. Information is collected using CrayPat [8], which reports power/energy per node and average memory consumption per process for a given system

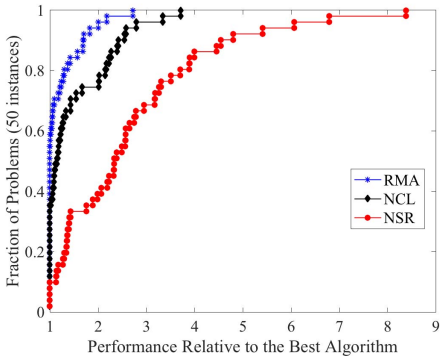


Fig. 10: Performance profiles for RMA, NCL and NSR using a subset of inputs used in the experiments. The X-axis shows the factor by which a given scheme fares relative to the best performing scheme. The Y-axis shows the fraction of problems for which this happened. The closer a curve is aligned to the Y-axis the superior its performance is.

TABLE VII: Versions yielding the best performance over the Send-Recv baseline version (run on 512-16K processes) for various input graphs.

Graph category	Identifier	Best speedup	Version
Random geometric graphs (RGG)	d=8.56E-05	3.5×	NCL
	d=6.12E-05	2.56×	NCL
	d=4.37E-05	2×	NCL
Graph500 R-MAT	Scale 21	2.32×	NCL
	Scale 22	3×	RMA
	Scale 23	3.17×	RMA
	Scale 24	2×	NCL
Protein K-mer	V2a	1.4×	RMA
	U1a	2.2×	RMA
	P1a	2.32×	RMA
	V1r	3.3×	RMA
DNA	Cage15	**6**×	NCL
CFD	HV15R	4×	NCL
Social network	Orkut	3.26×	NCL
	Friendster	**4.45**×	RMA

configuration. In Table VIII, we see that the average memory consumption for NCL is the least, about $1.03 - 2.3\times$ less than NSR, and about $9 - 27\%$ less than RMA for each case. The overall node energy consumption of NSR is about $4\times$ that of NCL and RMA for Friendster. The relative increase in communication percentages of NCL and RMA relative to NSR can be attributed to the exit criteria in the second phase of the algorithm (as described in §IV). For NSR, a local summation on the $nghosts$ array is sufficient to determine the completion of outstanding `Send` operations. However, for RMA and NCL, since processes do not coordinate with each other, it may lead to a situation where a process exits the iteration and waits on a barrier, while another process has a dependency on the process that exited, and is therefore stuck in an infinite loop. To avoid such situations, we have to perform a global reduction on the $nghosts$ array to ascertain completion, which adds to the additional volume in communication. Performance or energy values cannot be taken in isolation and for identifying an approach that provides the best tradeoffs, one needs to compute Energy-Delay Product (EDP). While more testing is needed to that effect, based on the results we observed in this paper, we find that NCL appears to provide a reasonable tradeoff between power/energy and memory usage.

Implementation remarks: Based on our experience in building distributed-memory half-approx matching (which is representative of a broader class of iterative graph algorithms), we posit that the RMA version provides reasonable benefit in terms of memory usage, power/energy consumption and

TABLE VIII: Power/energy and memory usage on 1K processes.

Ver.	Mem. (MB/proc.)	Node eng. (kJ)	Node pwr. (kW)	Comp. %	MPI %	EDP
Friendster (1.8B edges)						
NSR	977.7	2868.04	10.7	61.6	38.4	8.29E+08
RMA	577.4	793.27	9.78	21.4	78.6	1.35E+08
NCL	419.3	740.13	9.65	20.8	79.1	1.27E+08
Stochastic block-partitioned graph (475.1M edges)						
NSR	154.8	485.80	8.18	57.5	42.5	2.88E+07
RMA	196.3	690.41	9.09	7.2	92.8	5.24E+07
NCL	149	593.90	8.82	7.2	92.7	4.00E+07
HV15R (283.07M edges)						
NSR	210.2	154.98	5.95	13.5	86.4	4.04E+06
RMA	116.8	163.97	6.32	4.6	95.3	4.25E+06
NCL	106.9	140.85	6.07	3.2	96.7	3.27E+06

overall scalability. While it is possible to make the Send-Recv version optimal, handling message aggregation in irregular applications is challenging. On the other hand, neighborhood collective performance is sensitive to the graph structure, and mitigating such issues requires careful graph partitioning (§V-C), which in itself is NP-hard. However, due to the ubiquity of RDMA interconnects, it is possible that RMA is better optimized than neighborhood collectives over the graph topology on current Cray systems.

Our distributed-memory half-approx matching code is available for download under the BSD 3-clause license from: https://github.com/Exa-Graph/mel.

VI. Related Work

Lumsdaine et al. [25] provide excellent overview of the challenges in implementing graph algorithms on HPC platforms. Gregor and Lumsdaine provide an overview of their experiences with the Parallel Boost Graph Library in [14]. Buluç and Gilbert provide an alternative means to implement graph algorithms using linear algebra kernels in [4].

Thorsen et al. propose a partitioned global address space (PGAS) implementation of maximum weight matching in [31]. We discussed a set of closely related work on half-approximate matching in §IV.

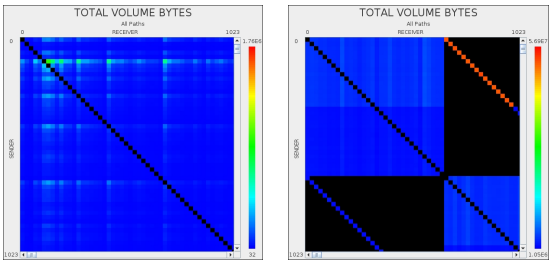
Besta et al. provides extensive analysis on the role of communication direction (*push* or *pull*) in graph analytics, and uses MPI RMA in implementing *push* or *pull* variants of graph algorithms [2]. Our distributed-memory half-approximate matching is based on the *push* model.

Kandalla *et al.* study the impact of nonblocking neighborhood collectives on a two-level breadth-first search (BFS) algorithm [22]. Communication patterns for the matching algorithm are not comparable with the communication patterns for BFS. Since the authors experiment only with synthetic graphs featuring small-world properties (average shortest path lengths are small), BFS converges in a few iterations and communication properties are conducive for collective operations. However, matching displays dynamic and unpredictable communication behavior compared to BFS, as shown in Fig. 11.

Dang et al. provide a lightweight communication runtime for supporting distributed-memory thread-based graph algorithms in Galois graph analytics system [7, 28]. They use MPI RMA (not passive target synchronization like us, but active target synchronization, which is more restrictive) and Send-Recv (particularly `MPI_Iprobe`, that we use as well), but not neighborhood collectives, in their communication runtime.

VII. Conclusions

We investigated the performance implications of designing a prototypical graph algorithm, half-approx matching,



(a) Matching. (b) Graph500 BFS.

Fig. 11: Communication volumes (in terms of bytes exchanged) of baseline implementation of half-approximate matching and Graph500 BFS, using R-MAT graph of 134.2M edges on 1024 processes.

with MPI-3 RMA and neighborhood collective models and compared them with a baseline Send-Recv implementation. We demonstrated speedups of $\mathbf{1.4-6}\times$ (using up to 16K processes) for the RMA and neighborhood collective implementations relative to the baseline version, using a variety of synthetic and real-world graphs.

We explored the concept of graph reordering by reducing bandwidth using the Reverse Cuthill-McKee algorithm. We demonstrated the impact of reordering on communication patterns and volume, especially for the neighborhood collective model. Although we did not observe expected benefits in our limited experiments, we believe that careful distribution of reordered graphs can lead to significant performance benefits, which we plan to explore in the near future.

We believe that the insight presented in this work will benefit other researchers in exploring the novel MPI-3 features for irregular applications such as graph algorithms, especially on the impending exascale architectures with massive concurrency coupled with restricted memory and power footprints.

Acknowledgments

We used resources of the NERSC facility, supported by U.S. DOE SC under Contract No. DE-AC02-05CH11231. The research is supported by NSF award IIS-1553528, NSF award 1815467, the U.S. DOE ExaGraph project at DOE PNNL, and the DOE award DE-SC-0006516 to WSU. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.

[2] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104. ACM, 2017.

[3] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[4] Aydin Buluc and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[5] Ümit V Çatalyürek, Florin Dobrian, Assefaw Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Distributed-memory parallel algorithms for matching and coloring. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1971–1980. IEEE, 2011.

[6] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In *Sparse Matrices and Their Applications*, pages 157–166. Springer, 1972.

[7] H. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, and M. Snir. A lightweight communication runtime for distributed graph analytics. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 980–989, May 2018.

[8] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer, 2008.

[9] Jack Dongarra. Compressed row storage. http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html.

[10] Doratha E Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[11] Message Passing Interface Forum. MPI: A message-passing interface standard version 3.0. http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[12] Harold N Gabow. An efficient implementation of edmonds' algorithm for maximum matching on graphs. *Journal of the ACM (JACM)*, 23(2):221–234, 1976.

[13] Steven Gold and Anand Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Transactions on pattern analysis and machine intelligence*, 18(4):377–388, 1996.

[14] Douglas P. Gregor and Andrew Lumsdaine. The parallel bgl : A generic library for distributed graph computations. 2005.

[15] Mahantesh Halappanavar. *Algorithms for Vertex-weighted Matching in Graphs*. PhD thesis, Norfolk, VA, USA, 2009. AAI3371496.

[16] T. Hoefler and T. Schneider. Optimization principles for collective neighborhood communications. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, Nov 2012.

[17] Torsten Hoefler and Jesper Larsson Traff. Sparse collective operations for MPI. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[18] Torsten Hoefler et al. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23(4):293–310, 2011.

[19] Torsten Hoefler et al. Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing*, 2(2):9, 2015.

[20] Jaap-Henk Hoepman. Simple distributed weighted matchings. *arXiv preprint cs/0410047*, 2004.

[21] John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[22] K. Kandalla, A. Bulu, H. Subramoni, K. Tomko, J. Vienne, L. Oliker, and D. K. Panda. Can network-offload based non-blocking neighborhood mpi collectives improve communication overheads of irregular graph algorithms? In *2012 IEEE International Conference on Cluster Computing Workshops*, pages 222–230, Sept 2012.

[23] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[24] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.

[25] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17:5–20, 2007.

[26] Fredrik Manne and Rob H Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *International Conference on Parallel Processing and Applied Mathematics*, pages 708–717. Springer, 2007.

[27] S. H. Mirsadeghi, J. L. Trff, P. Balaji, and A. Afsahi. Exploiting common neighborhoods to optimize mpi neighborhood collectives. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 348–357, Dec 2017.

[28] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[29] Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 259–269. Springer, 1999.

[30] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[31] Alicia Thorsen, Phillip Merkey, and Fredrik Manne. Maximum weighted matching using the partitioned global address space model. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 109. Society for Computer Simulation International, 2009.