Parallel Asynchronous Distributed-Memory Maximal Independent Set Algorithm with Work Ordering

Thejaka Kanewala, Marcin Zalewski, Andrew Lumsdaine
Pacific Northwest National Laboratory & University of Washington
Seattle, WA, USA
Email: {thejaka.kanewala, marcin.zalewski, andrew.lumsdaine}@pnnl.gov

***Abstract*—The maximal independent set (MIS) graph problem arises in many applications such as computer vision, information theory, molecular biology, and process scheduling. The growing scale of graph data suggests the use of distributed memory hardware as a cost-effective approach to providing necessary compute and memory resources. Existing distributed memory parallel MIS algorithms rely on synchronous communication and use techniques such as subgraph computations. In this paper, we present an asynchronous distributed-memory parallel graph algorithm that relies on a virtual directed acyclic graph (DAG) that is created during the algorithm execution. We introduce two additional algorithms that save computations by ordering generated work. The first algorithm applies ordering globally to reduce computations, and the second algorithm applies ordering locally at the level of threads to minimize the synchronization overhead. We use two different implementations of Luby's algorithm variants as baseline to compare the performance of the presented algorithms: (1) vertex-centric Luby A and Luby B implementations, and (2) the CombBLAS linear-algebra Luby A implementation. Results show that proposed algorithms outperform both implementations of Luby algorithms, especially in distributed execution. Furthermore, we show that for low-diameter graphs the algorithm that applies global ordering scales better than other algorithms and for high diameter graphs the original asynchronous algorithm and thread-level ordering algorithm show better performance.**

***Keywords*-maximal independent set (MIS); distributed-memory graph algorithms**

I. Introduction

The Maximal Independent Set (MIS) problem arises in many application domains including computer vision, information theory, molecular biology, and process scheduling. For a given graph $G = (V, E)$ where V represents the set of vertices and E represents the set of edges in the graph, an *independent set* or a *stable set* of G is a set of vertices in the graph where no two vertices are adjacent. The largest possible independent set is called the *maximum independent set*. Since finding the maximum independent set is an NP-hard problem, most applications settle for finding a *maximal independent set* (MIS). Maximal independent set is an independent that is not a subset of any other independent set(e.g., Figure 1). Although efficient MIS algorithms are well-known [1], the increasing scale of data-intensive applications requires the use of distributed-memory hardware (clusters), which in turn requires distributed-memory algorithms.

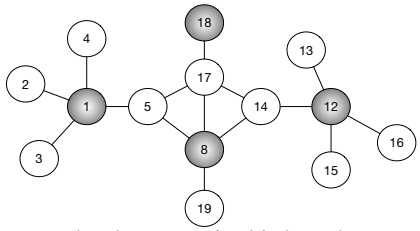

Fig. 1: The gray nodes show a maximal independent set of this graph.

Existing research on parallel algorithms for the MIS problem focuses primarily on theoretical analyses (most often in shared memory settings), and this provides incomplete insight into real-world performance. The available distributed-memory parallel algorithms for MIS are based on Luby's randomized MIS[2] algorithms. Luby's algorithms were developed primarily focusing on shared memory; they use techniques that do not scale to distributed-memory. In this paper we propose an asynchronous distributed-memory parallel MIS algorithm based on a Directed Acyclic Graph (DAG) induced on the input graph, using unique randomly assigned vertex identifiers. *FIX*, the algorithm we propose, can further reduce the number of computations by arranging the computations in a particular order. We present two different orderings of FIX, FIX-Bucket and FIX-PQ, and we experimentally evaluate their performance against two different implementations of distributed-memory parallel Luby's algorithms.

The FIX algorithm we present in this paper is an asynchronous distributed-memory parallel algorithm. It maintains a state for every vertex, indicating whether the vertex is in MIS (FIX1) or not in MIS (FIX0). FIX creates a virtual DAG based on randomly assigned vertex identifiers. Processing starts from the sources of this DAG, and state changes are propagated towards the sinks of the DAG. We show that the algorithm terminates, and that at termination vertices that have the state FIX1 form an MIS.

The correctness of the FIX algorithm does not depend on the order of changes to vertex states, and the execution can proceed in any of the correct orders (the algorithm execution is not deterministic, but the result is). We derive two algorithms from the basic FIX algorithm by applying the following orderings:

1) Order work based on where it originates – First process

0-7695-6326-0/17/$31.00 ©2017 IEEE
DOI 10.1109/HiPC.2017.00016

work originating from vertices in FIX1 and then process work originating from vertices in FIX0 (*FIX-Bucket* algorithm);

2) Order work based on state and then order work based on the monotonic distance from the vertex that started the work (source of the DAG). In other words, the work that is in FIX1 is processed immediately and the work that is not in FIX1 is processed based on the distance from the source. Vertices that have smaller distance from the source gets priority over vertices that have higher distance from the source when the work is in FIX0 state (*FIX-PQ* algorithm).

We show that above two orderings reduce the number of computations relative to the original FIX algorithm.

The performance of the proposed FIX algorithm (including the two ordering variations) is evaluated and compared with Luby's algorithms for both weak scaling and strong scaling. Direct implementations of the original Luby algorithms are inefficient in distributed-memory runtimes, mainly because of subgraph computation, random number generation, and synchronization. [3] presented an efficient implementation of two of Luby's seminal algorithms with overlapping computation and communication. In addition, CombBLAS[4] has an implementation of Luby's algorithm that is implemented using linear algebra primitives. We use these two implementations as our baselines to compare the performance of the FIX algorithms.

Our results include two types of synthetic graph inputs to evaluate the weak scaling performance and two synthetic graphs and two natural graphs (one with a small diameter and one with a higher diameter) to evaluate the strong scaling performance. Results show that the three FIX algorithms outperform Luby's algorithms implemented in [3] and CombBLAS [4]. Furthermore, we show that for low-diameter connected graphs, the FIX-Bucket algorithm has better performance than FIX and FIX-PQ, and for higher diameter graphs, the FIX and FIX-PQ algorithms have better performance than FIX-Bucket algorithm.

In summary, the main contributions of this paper are:

1) Development and characterization of three distributed-memory parallel MIS algorithms;
2) Comparison of weak scaling results to two different implementations of distributed Luby algorithms;
3) Analysis of results showing that for low-diameter graphs FIX-Bucket outperforms other algorithms and that for high-diameter graphs the FIX or the FIX-PQ algorithm outperforms other algorithms.

II. FIX Algorithm

The FIX algorithm begins by generating a directed acyclic graph (DAG) on the input graph G. First, every vertex is assigned a unique random identifier. Every undirected edge (v, u) is given a direction based on the identifiers of v and u, where the vertex with the greater identifier becomes a *successor*, and the vertex with the lesser identifier becomes a *predecessor* (see Fig. 2). The vertices without predecessors are the *sources* of the induced DAG, and the vertices without successors are the *sinks*.

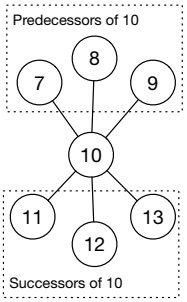

Fig. 2: Successors and predecessors of a vertex.

The FIX algorithm is shown in Algorithm 1. FIX has three main subroutines: *Initialize*, *Begin*, and *Receive*. Every *Send* call in the algorithm invokes the *Receive* subroutine. The FIX algorithm maintains a state per each vertex. The state is represented using the maps *mis* and *count*, where the *mis* map represents the membership in MIS (FIX0 indicates not in MIS, and FIX1 indicates in MIS), and the *count* map represents the number of predecessors with state FIX1. In *Initialize*, the *mis* state of every vertex is initialized to *UNFIX*, indicating that the membership of the vertex in the MIS is not decided yet, and the *count* state is initialized to 0, indicating that no predecessors are in MIS. The FIX algorithm first adds source vertices to the MIS (Lines 1–5) by changing their state to FIX1. Upon changing the state of a source, a message is sent to its successors to notify them of the change.

The initial messages sent from *Begin* are handled by *Receive*. The *Receive* routine is a *message handler* that is implicitly invoked per every *Send* call. If a message comes from a vertex with a state of FIX1, the state of the receiving vertex u is immediately changed to FIX0, and all successors of v are notified of this change (Lines 1 and 4). If a message comes from a vertex in the FIX0 state, the count of predecessors with FIX0 state is incremented. If all predecessors have the FIX0 state (Line 7), then the vertex joins the MIS. Its state is set to FIX1, and all of its neighbors are notified. The algorithm traverses the DAG induced on the input graph G by following the predecessor and successor links, and it terminates when the states of all sinks are set to FIX0 or FIX1, or, equivalently, when there are no vertices left in the state UNFIX.

At termination, vertices with state FIX1 form an MIS. To prove this, we first need to show that at termination, all vertices must either be in FIX1 or FIX0 state. Suppose there is a vertex, v, in the UNFIX state at termination. The vertex v can be in any one of the following 4 situations within the induced DAG: (1) v has neither predecessors nor successors; (2) v has predecessors but not successors; (3) v has no predecessors but it has successors; or (4) v has both predecessors and successors. In cases (1) and (3) the *Begin* routine promotes the state of v to FIX1 (Line 3 in Algorithm 1). In cases (2) and (4), if v is in UNFIX state, then, there is no predecessor with state FIX1 because of the message sent on Line 10 in Algorithm 1. A message sent in Line 10 calls the *Receive* procedure and it assures successor of a FIX1 vertex is in FIX0 (Lines 1 and 2). Then, either all the predecessors must be in the FIX0 state, or there is at least one predecessor that is in the UNFIX state. If all the predecessors are in FIX0, v must be in FIX1 (Lines 7 and 8 in Algorithm 1). Since we assume v is in the UNFIX state, there must be at least one predecessor in the UNFIX state. Let this predecessor of v be

Algorithm 1: FIX Algorithm

Initialize $G = (V, E)$:

```
1: for each v in V do
2:   mis[v] ← UNFIX, count[v] ← 0
```

Begin $G = (V, E)$:

```
1: for each v in V do
2:   if v is a source then
3:     mis[v] ← FIX1
4:     for each u in Successors of v do
5:       Send(u, FIX1)
```

Receive u, vstate:

```
1: if vstate == FIX1 then
2:   mis[u] ← FIX0
3:   for each u_s in Successors of u do
4:     Send(u_s, FIX0)
5: else
6:   count[u] ← count[u] + 1
7:   if count[u] == |Predecessors| then
8:     mis[u] ← FIX1
9:     for each u_s in Successors of u do
10:      Send(u_s, FIX1)
```

u. We can make a similar argument for u and can conclude that u has a predecessor in the UNFIX state. We continue the argument until we reach a vertex without predecessors (since the DAG is finite and acyclic, we will reach a source level vertex). However, by cases (1) and (3), we know that a vertex without predecessors will be promoted to FIX1. Therefore, the sources of the DAG (they are also predecessors) cannot be in the UNFIX state. Therefore, our assumption that there is a vertex in UNFIX state after termination causes a contradiction, and we can conclude that every vertex is in either FIX0 or FIX1 after algorithm termination.

Furthermore, none of the vertices that are in the state FIX0 can be changed to FIX1 because a vertex's state transitions to FIX0 if, and only if, it has a neighbor that is in FIX1. Therefore, the set of vertices in the FIX1 state cannot be expanded any further. Hence, a set of vertices that is in the FIX1 state is an MIS.

Compared to Luby's algorithms, the FIX algorithm avoids the overhead of random number generation in every iteration by assigning a random permutation of identifiers to vertices. In addition, FIX also avoids the need to maintain a subgraph for every iteration. Maintaining a subgraph requires $O(n)$ space; hence, compared to Luby's algorithms, FIX is space efficient.

A. Distributed FIX

The distributed implementation of FIX divides vertices equally among participating processes and stores those vertices and their adjacencies locally. G^{local} represents the local subgraph. Every rank runs some number of threads, and distributed messages are exchanged between ranks within *epochs*.

An epoch is a code region where distributed programs can exchange messages. A global barrier is executed at the start of an epoch and at the end of an epoch. Computation and communication can be overlapped within epochs.

Our distributed implementation is listed in Algorithm 3. The implementation follows the single program, multiple data (SPMD) paradigm, in the sense that every rank runs the same program. Every rank maintains two per-vertex counter maps of counters (the counters are indexed by the vertex identifier). The first counter, *predecessor count* (*predcount*), counts the number of predecessors per each vertex and this is calculated at the initialization of the algorithm (Line 15). The second counter, *predecessor completed count* (*predcc*) keeps track of the number of predecessors that were moved to FIX0 state. This counter is updated during algorithm execution. These two counters are used to identify whether all the predecessors of a vertex have been moved to FIX0 state. For a given vertex, v, *predcount*$[v]$ = *predcc*$[v]$ means that all the predecessors of the vertex v are in FIX0, and v can be set to FIX1.

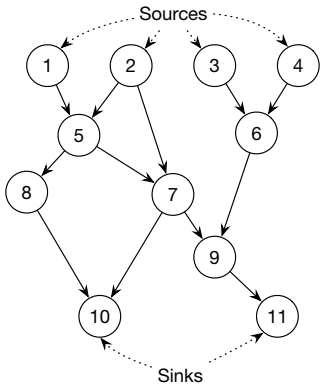

Fig. 3: The virtual DAG created based on predecessors and successors.

The algorithm execution starts with the *FIXMIS* (Lines 12–19) procedure. Inside an epoch, each rank goes through vertices in the local graph in a parallel thread and changes the state of vertices that have a zero *predcount*. When a vertex's state is changed, the vertex notifies all of its successors (Line 18). The notification implicitly invokes the *Receive* procedure, as in Algorithm 1. The *Receive* function first checks whether the destination vertex's state is already changed to either FIX1 or FIX0 (Line 22). If not, it checks whether the source vertex's state has been changed from UNFIX to FIX1. If so, the receiving vertex's state is changed to FIX0, and the receiving vertex notifies, in turn, all of its successors (Lines 24–28). If the source vertex's state is changed from UNFIX to FIX0, the receiving vertex increments its *predcc* counter. Then, if all of the receiving vertex's neighbors with higher priorities are transferred to FIX0 state (i.e., if predcc = predcount), the receiving vertex is promoted to FIX1 state (Lines 21–36).

In our particular implementation of the FIX algorithm, we did not use a thread-level distribution of vertices. Therefore, counters and MIS states can be updated by two threads at the same time. To avoid race conditions we use atomic operations.

As is, the Algorithm 3 suffers from load imbalance within threads in shared memory execution. The algorithm processes each vertex in parallel (Line 14), and some of the vertices, that threads process are not sources (i.e., they do not satisfy the condition *predcount*$[v] == 0$, Line 15). Because of that, those threads do little work compared to threads that process sources.

To balance the load between threads, we pre-calculate the source vertices and insert them into an append buffer, and *FIXMIS* procedure iterates over the sources instead of all

Algorithm 2: Modified FIX for better load balance between threads

```
1: ...
2: procedure Initialize(G^local)
3:   for each Vertex v in G^local in parallel do
4:     ...
5:     for each u in adjacencies(v, G^local) do
6:       if (u < v) then
7:         predcount[v] ← predcount[v] + 1
8:     if predcount[v] == 0 then
9:       appendbuffer.insert(v)
10:
11: procedure FIXMIS(G^local)
12:   ...
13:   for each Vertex v in appendbuffer in parallel thread do
14:     ...
15:   ...
16: ...
```

vertices in the graph. This way we make sure that every thread processes about the same number of source vertices. The modified distributed MIS with better thread level load balance is given in Algorithm 2 (Lines 5–7 and Line 13).

Algorithm 3: Distributed Memory Parallel FIX Algorithm

```
1: predcount ← {0...0}
2: predcc ← {0...0}
3: procedure Initialize(G^local)
4:   for each Vertex v in G^local in parallel do
5:     mis[v] ← UNFIX
6:     predcount[v] ← 0
7:     predcc[v] ← 0
8:     for each u in adjacencies(v, G^local) do
9:       if (u < v) then
10:        predcount[v] ← predcount[v] + 1
11:
12: procedure FIXMIS(G^local)
13:   epoch {
14:     for each Vertex v in G^local in parallel thread do
15:       if (predcount[v] == 0) then
16:         mis[v] ← FIX1
17:         for each u in adjacencies(v, G^local) do
18:           Send(u, v, mis[v])
19:   }
20:
21: procedure Receive(destv, srcv, srcstate)
22:   if mis[destv]! = UNFIX then
23:     return
24:   if srcstate == FIX1 then
25:     mis[destv] ← FIX0
26:     for each u in adjacencies(destv, G^local) do
27:       if (u > destv) then
28:         Send(u, destv, mis[destv])
29:   else                                   ▷ srcstate=FIX0
30:     if (srcv < destv) then
31:       predcc[destv] ← (predcc[destv] + 1)
32:     if predcc[destv] == predcount[destv] then
33:       mis[destv] ← FIX1
34:       for each u in adjacencies(destv, G^local) do
35:         if (u > destv) then
36:           Send(u, destv, mis[destv])
```

III. Ordering in FIX

The FIX algorithm discussed above is an unordered algorithm. Therefore, the order in which the states are updated does not affect the correctness of the algorithm, but with certain ordering schemes we can reduce the amount of computations in FIX.

The FIX algorithm decides a vertex is in the MIS (FIX1) if all of its predecessors are not in MIS. Therefore, the sooner a vertex finds out that it is in MIS, the more computations it can avoid. For example, in Figure 4, the vertex v has four predecessors; out of those four, one predecessor is in state FIX1. If v gets a state change from the vertex in FIX1 first, v will not have to execute the logic relevant to increase predcc count and the "if condition" that compares predcc and predcount. If the thread level vertex distribution is random, the predcc count will be a atomic variable and processing FIX1 predecessors first will considerably reduce the contention when processing a large graph, especially in shared memory execution. However, if we process FIX0 predecessors before FIX1, the computation we did for FIX0 state changes have no effect on the final outcome.

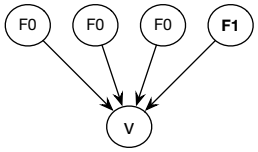

Fig. 4: DAGs created by FIX algorithm execution for the graph input in 1.

We came up with two forms of orderings that reduce computations as described above. The first approach orders the execution on the state. That is, messages generated are separated into two *buckets*. The first bucket contains the work originated from vertices whose states transferred to FIX1. The second bucket contains the work originated from vertices who transferred their states to FIX0. How execution proceeds in this ordering is depicted in Figure 5. Implicitly, the bucket ordering traverse the DAG in a level synchronous fashion. We call this algorithm FIX-Bucket and it is discussed in detail in Section III-A.

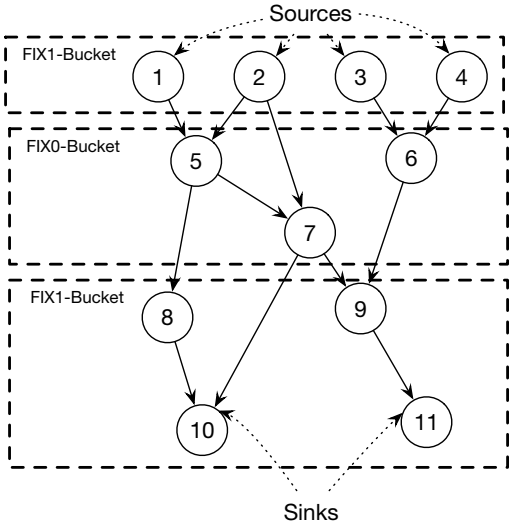

Fig. 5: How DAG is executed in FIX-Bucket ordering.

	FIX	FIX-PQ	FIX-Bucket
Skipped	84208408 (65.10%)	119592110 (92.46%)	112771147 (87.16%)
Called	45130908 (34.89%)	9747206 (7.53%)	16613655 (12.84%)
Total	129339316	129339316	129339316

TABLE I: Saved computations for FIX-Bucket and FIX-PQ algorithms, relative to FIX. Results are for scale 23, RMAT-1 (refer Section IV-C for details) graph with 4 cores running in parallel.

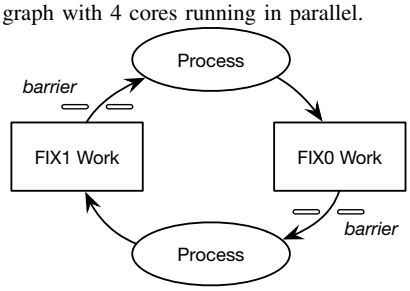

Fig. 6: An overview of the FIX-Bucket algorithm.

The above ordering involves a global barrier to be executed in every level. The second ordering we propose (FIX-PQ), skips global synchronization and performs local ordering on work. The work generated is ordered based on the state of the source vertex as well as on the distance from the originating source (in the DAG). The work with FIX1 state is immediately processed and work that has FIX0 state is ordered by distance from the relevant source. When the distance is higher the priority is also high. The purpose of distance ordering is to propagate state changes deeper into the DAG.

Both the FIX-Bucket and FIX-PQ algorithms require fewer computations than the original FIX algorithm. Table I shows the number of computations saved by each algorithm relative to the original FIX algorithm. The "Skipped" denotes the number of times the *Receive* function is invoked even though there is no state change or increase in predcc count. The "Called" denotes the number of times the *Receive* function was invoked and there was a state change or predcc counter was increased.

As per the statistics in Table I, the FIX-PQ algorithm saves the most amount of work. This is primarily because FIX-PQ is ordered using both distance and state and so able to propagate FIX1 state changes to successors faster than FIX-Bucket is able to.

In the following section we discuss the FIX-Bucket and the FIX-PQ algorithms in detail.

A. FIX-Bucket Algorithm

The *FIX-Bucket* algorithm maintains two distributed containers, called *buckets* (Figure 6). Locally, a bucket is implemented as an append buffer, but before processing the append buffer all ranks must globally synchronize. In other words an append buffer is processed within an epoch. The first container, which we call *fix0bucket*, stores all the vertices where state is transferred to FIX0. The second container, *fix1bucket*, stores all the vertices where state is changed to FIX1.

The FIX-Bucket algorithm is listed in Algorithm 4. The initialization code is the same as the initialization procedure in Algorithm 3. The FIX-Bucket algorithm starts in the same way as the FIX algorithm but at the end of the *FIXBucketMIS* (Lines 7–15) procedure, *FIXBucketMIS* calls the *HandleBuckets* procedure. In every rank, the *HandleBuckets*(Lines 17–30) procedure iterates through each bucket in parallel threads and sends state changes to successors. However, at a given time, all the ranks iterate through only one bucket. Therefore, unlike in the FIX algorithm, in the FIX-Bucket algorithm there are no messages originating from FIX0 vertices when processing *fix1buckets*. Also there are no messages originating from FIX1 vertices when processing *fix0buckets*. The *Receive* function handles incoming messages and populates them to appropriate buckets.

B. FIX-PQ Algorithm

The execution time of the FIX algorithm depends on the maximum height of the virtual DAG (e.g., Figure 3). The longest path of the DAG is important, especially when deciding whether a vertex should be transferred to FIX1 state, because a vertex's state can only be updated to FIX1 if all predecessors of the vertex are in FIX0 state. Processing work generated by FIX1 vertices is straightforward since neighbors of FIX1 must be transferred to FIX0 irrespective of the number of predecessors. Furthermore, notifying successors about a state change of a vertex to FIX1 helps to save computations. Therefore, the FIX-PQ algorithm processes FIX1 work immediately and orders work generated by FIX0 vertices based on the distance from a source. The ordering is applied at the thread level to avoid the overhead of synchronization.

The FIX-PQ algorithm is listed in Algorithm 5. The algorithm keeps an array of priority queues and the size of the array is equal to the number of threads (Line 3). The function *getnumthreads* returns the number of threads the algorithm is executing. The *Initialize* procedure is the same as the *Initialize* procedure in Algorithm 3. The *FIXPQMIS* procedure (Lines 6–14) adds source vertices to the MIS and notifies state changes. In the same epoch the algorithm calls the function *HandlePQs* (Line 13). Also, algorithm uses *workitem* structure to encapsulate destination vertex, source vertex, source state and distance.

For each shared memory thread, the *HandlePQs*(Lines 16–27) procedure pop work from the priority queue and process it. The priority queue only contains work related to FIX0 predecessors, therefore, the processing logic updates predcc count and checks whether the destination vertex can be promoted to FIX1. The *HandlePQs* procedure is executed until there is work available in the system. The function *terminate()* returns *True* when the termination detection detects that there is no more work to be processed.

The *Receive* function (Lines 29–36) processes work items originating from FIX1 vertices and work items originating from FIX0 vertices are added to the priority queue for the current thread.

Note that *HandlePQs*' procedure is invoked within the epoch of the *FIXPQMIS* procedure. Therefore, Algorithm 5 executes asynchronously, but work is ordered at the thread level.

Algorithm 4: Distributed Memory Parallel FIX-Bucket Algorithm

```
1: predcount ← {0...0}
2: predcc ← {0...0}
3: fix0bucket ← {}
4: fix1bucket ← {}
5: procedure Initialize(G^local)
   ▷ /*Same as the Initialization procedure in Algorithm 3*/.
6:
7: procedure FIXBucketMIS(G^local)
8:   epoch {
9:     for each Vertex v in G^local in parallel do
10:      if (predcount[v] == 0) then
11:        mis[v] ← FIX1
12:        for each u in adjacencies(v, G^local) do
13:          Send(u, v, mis[v])
14:  }
15:  HandleBuckets()
16:
17: procedure HandleBuckets(void)
18:   while fix0bucket not empty and fix1bucket not empty do
      ▷ /*Handle FIX0 bucket.*/
19:     epoch {
20:       for each Vertex v in fix0bucket in parallel do
21:         for each u in adjacencies(v, G^local) do
22:           if (u > v) then
23:             Send(u, v, mis[v])
24:     }
        ▷ /*Handle FIX1 bucket.*/
25:     epoch {
26:       for each Vertex v in fix1bucket in parallel thread do
27:         for each u in adjacencies(v, G^local) do
28:           if (u > v) then
29:             Send(u, v, mis[v])
30:     }
31:
32: procedure Receive(destv, srcv, srcstate)
33:   if mis[destv]! = UNFIX then
34:     return
35:   if srcstate == FIX1 then
36:     mis[destv] ← FIX0
37:     fix0bucket → push(destv)
38:   else                                   ▷ srcstate=FIX0
39:     if (srcv < destv) then
40:       predcc[destv] ← (predcc[destv] + 1)
41:     if predcc[destv] == predcount[destv] then
42:       mis[destv] ← FIX1
43:       fix1bucket → push(destv)
```

Algorithm 5: Distributed Memory Parallel FIX-PQ Algorithm

```
1: predcount ← {0...0}
2: predcc ← {0...0}
3: pqs[getnumthreads()]
4: procedure Initialize(G^local)
   ▷ /*Same as the Initialization procedure in Algorithm 3*/.
5:
6: procedure FIXPQMIS(G^local)
7:   epoch {
8:     for each Vertex v in G^local in parallel do
9:       if (predcount[v] == 0) then
10:        mis[v] ← FIX1
11:        for each u in adjacencies(v, G^local) do
12:          Send(u, v, mis[v])
13:    HandlePQs()
14:  }
15:
16: procedure HandlePQs(void)
17:   while terminate() is False do
18:     while pqs[getthreadid()] not empty do
19:       workitem wi ← pqs[getthreadid()].pop()
20:       if (wi.srcv < wi.destv) then
21:         predcc[wi.destv] ← (predcc[wi.destv] + 1)
22:       if predcc[wi.destv] == predcount[wi.destv] then
23:         mis[wi.destv] ← FIX1
24:         for each u in adjacencies(wi.destv, G^local) do
25:           if (u > wi.destv) then
26:             workitem wn(u, wi.destv, mis[wi.destv], (wi.dist+1))
27:             Send(wn)
28:
29: procedure Receive(wi : workitem)
30:   if wi.srcstate == FIX1 then
31:     mis[wi.destv] ← FIX0
32:     for each u in adjacencies(wi.destv, G^local) do
33:       if (u > wi.destv) then
34:         Send(u, wi.destv, mis[wi.destv])
35:   else                                   ▷ wi.srcstate=FIX0
36:     pqs[getthreadid()] → push(wi)
```

IV. Implementation & Experiments

A. Implementation

The proposed algorithms are implemented on top of an MPI-wrapped, lightweight, active messaging framework called AM++ [5]. Graph vertices are equally distributed among participating nodes (*1D block distribution*). The local graph is represented using a *compressed sparse row* format. In the local graph each undirected edge is represented using two directed edges.

Algorithm implementations are resilient to parallel edges and self-loops. Both parallel edges and self-loops are handled within algorithms. Whenever there is code that iterates through adjacencies of a vertex, the algorithm inserts adjacent vertices to a local set. The body of the loop is executed only if the adjacent vertex is not present in the set (See the code below).

```
1: ...
2: adjacentvertices ← {}
3: for each u in adjacencies(v, G^local) do
4:   if (u! = v) then                    ▷ /*Exclude self-loops*/
5:     if u not in adjacentvertices then ▷ /*Exclude parallel edges*/
6:       adjacentvertices.insert(u)
7:       ...
```

B. Experiment Setup

We ran our experiments on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly double the problem size and to double the number of processors in weak scaling. Each node consists of 128 GB DDR4-2400 memory. The MPI implementation we used is Cray MPICH (version 7.4.4).

Graph	Vertices	Edges
Friendster	65608366	1806067135
US Road	23947347	58333344
RMAT-1(26)(rmat1)	67108864	1073741824
RMAT-2(26)(rmat2)	67108864	1073741824

TABLE II: Graph inputs and their attributes used in strong scaling experiments

Preliminary results show that to get the best performance results for all the algorithms, we need to run two processes per node (because there are two sockets per node) in MPI thread multiple mode.

C. Graph Input

We evaluate the MIS algorithms in terms of *strong scaling* and *weak scaling*. For weak scaling experiments, we use *R-MAT* [6] synthetic graphs. Two types of RMAT synthetic graphs are used. They are:

- RMAT-1: Graphs based on the current Graph500 [7] Breadth First Search benchmark specification with R-MAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$.
- RMAT-2: Graphs generated based on the proposed Graph500 [8] SSSP benchmark specification with R-MAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

Strong scaling experiments were carried out on the graphs listed in Table II.

V. Results

The weak scaling results of the proposed algorithms are compared against two different implementations of Luby algorithms : 1. Luby(A) and Luby(B) discussed in [3] and; 2. Luby(A) in CombBLAS library. Section V-A discusses these results. In the results, when we use *FIX** we refer to FIX, FIX-Bucket and FIX-PQ algorithms collectively.

A. Weak Scaling Results

1) Comparison with Vertex-Centric Luby Algorithms: Weak scaling results of FIX* algorithms are presented in Figure 7 on RMAT-1 and RMAT-2 graph inputs. Both in shared memory and in distributed memory FIX* algorithms outperform Luby(A). Luby(B) shows better performance for few initial scales in shared memory and as the execution moves to distributed memory, FIX* algorithms supersede the performance of Luby(B). We were unable to collect Luby(A) results for scale 32 graphs at 64 nodes due to an "out of memory" error.

Ordering helps to improve the performance of FIX* algorithms by reducing the required number of computations (See Section III). In shared memory, we see that FIX-PQ shows better performance than FIX. In shared memory FIX-PQ is able to avoid more computations than FIX algorithm, but, when the execution becomes distributed, performance improvement in FIX-PQ is not prominent compared to FIX. As the execution becomes distributed, the compute / communication ratio decreases and more time is spent on communication. Therefore, the performance improvement gained by reducing computation is small relative to the much higher overhead cost of distributing execution.

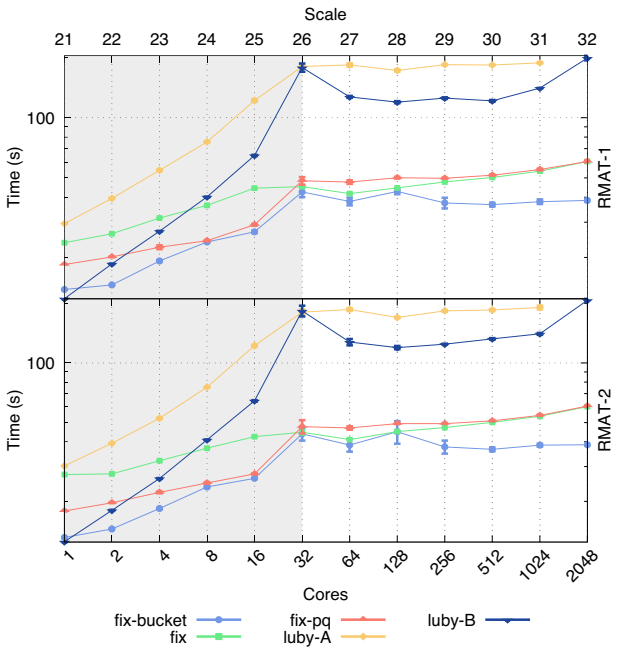

Fig. 7: FIX* & Luby algorithms weak scaling results for RMAT-1 and RMAT-2 graphs. Shaded region shows the shared memory execution.

The FIX-Bucket algorithm shows better weak scaling results in distributed execution. Starting from the sources of the DAG, The FIX-Bucket algorithm progress by processing vertices in each level. This way, algorithm assures that predecessors are always processed and hence it is able to avoid most of the redundant computations. However, the FIX-Bucket algorithm requires a barrier synchronization after processing each level in the graph. Overhead of this barrier synchronization is not as significant as RMAT graphs generally have fewer synchronization levels. Also, RMAT graphs are well-connected; therefore, there is enough work to keep all processors busy.

2) Comparison with CombBLAS Luby Algorithms: The FilteredMIS [9] algorithm runs Luby(A) with edge filtering. However, implementations presented in this paper do not perform any edge filtering. We show FilteredMIS results with 0% and 50% edge filtering where no edges and half of the edges are ignored, respectively. The more edges are ignored, the better FilteredMIS performs.

Distributed execution of FilteredMIS results show a zig-zag pattern (when cores > 32). The CombBLAS version we use only supports a square number of tasks; therefore, when executing on a non-square number of nodes (2, 8, 32) we used two *tasks* per node to make the execution on a square number of processes. When the number of tasks per node is two, FilteredMIS execution time decreases and when the number of tasks per node is 1, the execution time increases.

B. Strong Scaling Results

For strong scaling experiments, we ran MIS algorithms on graphs listed in Table II over 1–1024 cores. To have better understanding about how algorithms scale relative to each other,

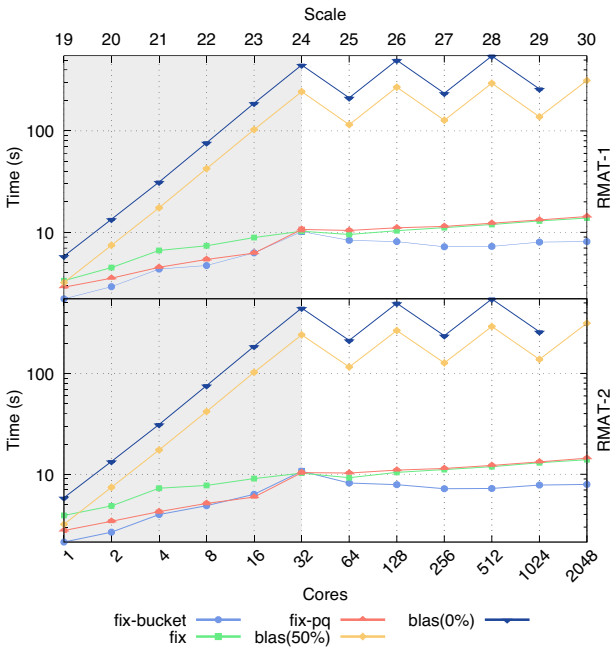

Fig. 8: FIX* & CombBLAS FilteredMIS algorithms, weak scaling results for RMAT-1 and RMAT-2 graphs. Shaded region shows the shared memory execution.

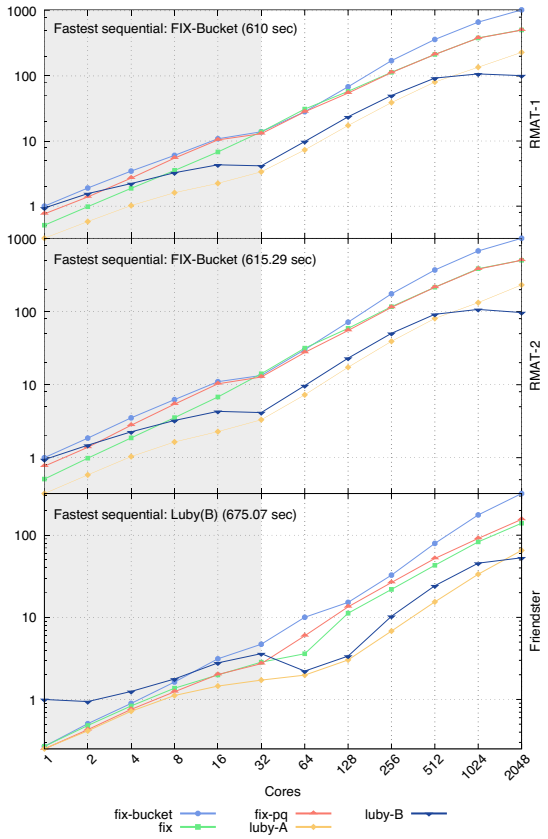

Fig. 9: Strong scaling results of FIX* algorithms and vertex centric Luby's algorithms

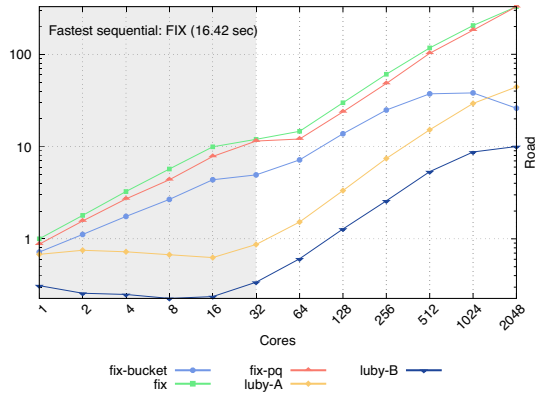

Fig. 10: Strong scaling results of FIX* algorithms and vertex centric Luby's algorithms.

we measured Relative Speedup, $= \frac{T_{ref\_1}}{T_n}$ i.e., the ratio of the execution time of the fastest sequential algorithm, T_{ref_1} and the parallel execution time on n processing elements, T_n.

1) Low diameter graphs: Strong scaling results of FIX* algorithms and Luby's algorithms on RMAT graphs are shown in Figure 9. For both RMAT-1 and RMAT-2 graphs we see better speedup in FIX* algorithms than Luby's algorithms. We see some drop in speedup when algorithm execution moves from shared memory to distributed memory. When execution reaches 2048 cores, the Luby B algorithm speedup decrease due to overhead of barrier synchronization. Further, we see an unusual scaling behavior of Luby(B) on the Friendstr network both in shared memory and for some cores in distributed memory. However, we validated our results. FIX* algorithms show better scaling behavior compared to both Luby algorithms.

2) Higher diameter graphs: In general, both RMAT-1 and RMAT-2 generate low diameter graphs (diameter 10-50, the diameter is estimated using the approximate diameter algorithm) relative to road networks. Further, Friendster graph's diameter is 32 [10]. Compared to the diameters of those graphs, road networks have larger diameters. Figure 10 shows strong scaling results for US road networks. The approximate diameter of a US road network is 850 [10].

As per Figure 10, both FIX and FIX-PQ show sound, strong scaling results for US road networks. Due to large diameter in road networks, the FIX-Bucket algorithm executes many synchronization phases (≈850 barriers). Because of the overhead of barrier synchronization FIX and FIX-PQ outperform FIX-Bucket. We see the FIX algorithm achieving slightly better speedup compared to FIX-PQ. Unlike the RMAT and the Friendstr graphs, road networks is not a well- connected graph and it has a higher-diameter compared to RMAT and Friendstr. Therefore, the number of saved computations in FIX-PQ in road network is less and it is unable to gain performance over the FIX algorithm.

VI. Related Work

Many parallel algorithms have been proposed to solve the MIS problem. Most of the work is focused on shared memory, using the PRAM model for parallel complexity analysis (e.g.,

[11], [12], [13], [14], [15], [16]). Some of these algorithms use randomization to break symmetry. Four related randomized algorithms were introduced by Luby [2]. Luby's algorithms select an arbitrary nonempty independent set from the original graph and remove all the selected vertices and their neighboring vertices from the original graph. This process is repeated in iterations until all the graph vertices are removed from the original graph. Luby presented four different ways to select an independent set in an iteration (based on randomization). Luby's algorithms steps are executed in a synchronized fashion, i.e., each step (examining neighbor states and updating current state) is carried out in a single lockstep. Furthermore, the algorithm needs to synchronize after each iteration.

Luby provides a detailed analysis of his algorithm using the PRAM machine model. Later Luby's algorithm's concepts were used to implement distributed versions. Lynch et al.discuss a distributed version of Luby's algorithm for synchronous distributed networks in [17]. [18] presented a version of [17] with improved communication message complexity. [19] provided a deterministic distributed MIS algorithm. However, they assume a synchronous communication model and provide no experimental results.

Luby also showed that the MIS problem can be generalized to a *Maximal Coloring Problem*. Some of the overheads of Luby's approach are discussed in [20], in the context of graph coloring. Further, [20] presented an asynchronous graph coloring heuristic and a distributed implementation of this algorithm is presented in [21]. However, deriving an MIS from the output of coloring requires additional post processing. Furthermore, those implementations do not discuss the notion of ordering to reduce computations.

The Combinatorial BLAS library [4] implements a distributed version of Luby's algorithm (Luby A). Their implementation uses linear algebra primitives in implementing Luby's algorithm and also the algorithm works on filtered graphs [9]. An efficient distributed implementation of Luby's algorithm is presented in [3]. The implementation provided in [3] overlaps computation and communication to reduce synchronization overhead and uses parallel data structures to avoid subgraph computations. [22] implemented a distributed version of Luby's algorithm for Pregel graph processing system. They used Luby's MIS algorithm to solve the graph coloring problem. [23] compares the performance of Luby's implementation in Pregel with a parallel algorithm designed in [16].

The parallel MIS algorithm in [16] shows that a trivial parallelization of the sequential greedy algorithm, also called lexicographically first MIS, is in fact highly parallel (polylogarithmic time) when the order of vertices is random. In particular, simply by processing each vertex as in the sequential algorithm but as soon as it has no earlier neighbors, gives a parallel linear work algorithm. The algorithms discussed in[16] use a *priority DAG* over the vertices of the input graph, where the DAG edges connect higher-priority to lower-priority endpoints based on random values assigned to vertices. Each step adds the nodes with no incoming edges in the DAG to the MIS and removes them and their neighbors from the input graph. This process continues until no vertices remain.However, they assumed shared memory execution and the algorithm relies on computing a subset of vertices based on a predefined function to execute greedy MIS in parallel. Further, each round and step in the algorithm needs to be synchronized, which may not be efficient in distributed execution.

Most of the MIS algorithms are designed for PRAM architectures with synchronous communication, and so most of the algorithms assume a synchronized step (for example reading neighboring vertex states and updating the current vertex state done in a single lockstep). Another important fact about the above algorithms is that they are iterative; at the end of each iteration all processes need to be synchronized. This synchronization causes significant overhead in large scale graph processing that is proportional to the size of the distributed machine. This synchronization wastes computation resources in distributed settings due to *straggler effects* in a distributed setting. Further, most of the distributed MIS algorithms are focused on process communication, not on large scale data graphs.

The algorithms we present in this paper are designed specifically for large-scale static graph processing in a distributed memory setting and they focus on deriving MIS without using any form of reduction. Further, exploration of different orderings to derive efficient algorithms is not common in the existing literature.

VII. Conclusions

Maximal Independent Set (MIS) is a well-studied graph problem, and there are parallel algorithms to solve it. However, most of those algorithms show poor performance in distributed settings due to synchronous communication, subgraph construction, and random number generation.

In this paper, we presented three MIS algorithms suitable for distributed memory parallel execution. The FIX algorithm is an asynchronous algorithm designed for distributed execution and FIX-Bucket and FIX-PQ algorithms make use of ordering to reduce computations. While FIX-Bucket performs ordering at a global level, the FIX-PQ algorithm performs ordering at the thread level to avoid global synchronization.

Weak scaling results on RMAT graphs show that FIX* algorithms outperform both vertex-centric Luby implementations as well as CombBLAS FilteredMIS algorithm. Weak scaling results and strong scaling results (except road network) show that the FIX-Bucket algorithm performs well for low-diameter graphs. For higher-diameter graphs FIX and FIX-PQ outperform other algorithms.

VIII. Acknowledgments

The research is in part supported by the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit Program at the DOE Pacific Northwest National Laboratory (PNNL) and National Science Foundation grant 1716828. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. Access to computational resources was supported in part by Lilly Endowment, Inc.,

through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU was also supported in part by Lilly Endowment, Inc.

REFERENCES

[1] S. A. Cook, "A taxonomy of problems with fast parallel algorithms," *Information and control*, vol. 64, no. 1, pp. 2–22, 1985.

[2] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[3] T. Kanewala, M. Zalewski, and A. Lumsdaine, "Distributed-memory fast maximal independent set," in *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017, accepted for publication.

[4] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, p. 1094342011403516, 2011.

[5] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A Generalized Active Message Framework," in *Proc. 19th Internat. Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 401–410.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

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

[8] Graph500Contributors. (2016) Graph 500 benchmark 1 ("search"). [Online]. Available: http://www.cc.gatech.edu/ jriedy/tmp/graph500/

[9] A. Buluc, E. Duriakova, A. Fox, J. R. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams, "High-productivity and high-performance analysis of filtered semantic graphs," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 237–248.

[10] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[11] M. Goldberg and T. Spencer, "Constructing a maximal independent set in parallel," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 3, pp. 322–328, 1989.

[12] A. Goldberg, S. Plotkin, and G. Shannon, "Parallel symmetry-breaking in sparse graphs," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 315–324.

[13] M. K. Goldberg, "Parallel algorithms for three graph problems," *Congressus Numerantium*, vol. 54, no. 111-121, pp. 4–1, 1986.

[14] N. Alon, L. Babai, and A. Itai, "A fast and simple randomized parallel algorithm for the maximal independent set problem," *Journal of algorithms*, vol. 7, no. 4, pp. 567–583, 1986.

[15] R. M. Karp and A. Wigderson, "A fast parallel algorithm for the maximal independent set problem," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 762–773, 1985.

[16] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy Sequential Maximal Independent Set and Matching Are Parallel on Average," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2012, pp. 308–317.

[17] N. A. Lynch, *Distributed algorithms*. Morgan Kaufmann, 1996.

[18] Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari, "An optimal bit complexity randomized distributed mis algorithm," *Distributed Computing*, vol. 23, no. 5-6, pp. 331–340, 2011.

[19] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer, "Fast deterministic distributed maximal independent set computation on growth-bounded graphs," in *International Symposium on Distributed Computing*. Springer, 2005, pp. 273–287.

[20] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[21] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce, "Graph colouring as a challenge problem for dynamic graph processing on distributed systems," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 347–358.

[22] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 577–588, 2014.

[23] K. Garimella, G. De Francisci Morales, A. Gionis, and M. Sozio, "Scalable facility location for massive graphs on pregel-like systems," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 2015, pp. 273–282.