Distributed-Memory Fast Maximal Independent Set

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

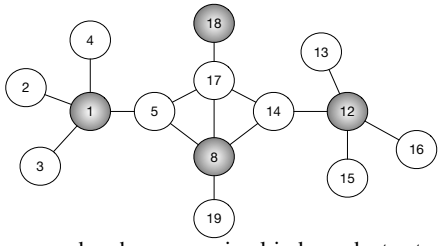

Fig. 1: The gray nodes show a maximal independent set of this graph.

***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 MIS problems suggests the use of distributed-memory hardware as a cost-effective approach to providing necessary compute and memory resources. Luby proposed four randomized algorithms to solve the MIS problem. All those algorithms are designed focusing on shared-memory machines and are analyzed using the PRAM model. These algorithms do not have direct efficient distributed-memory implementations. In this paper, we extend two of Luby's seminal MIS algorithms, "Luby(A)" and "Luby(B)," to distributed-memory execution, and we evaluate their performance. We compare our results with the "Filtered MIS" implementation in the Combinatorial BLAS library for two types of synthetic graph inputs.**

I. Introduction

Let $G = (V, E)$ be a graph where V represents the set of vertices and E represents the set of edges in the graph. An *independent set* in G is a set of vertices in a graph such that no two vertices in the set are adjacent. The largest independent sets (there may be more than one) are called the *maximum independent sets*. Since finding a maximum independent set is NP-hard, most applications settle for finding a *maximal* independent set. A maximal independent set (MIS) of a graph is an independent set that is not a subset of any other independent set (see Figure 1). Finding a MIS is an important graph problem used in many applications, including computer vision, coding theory, molecular biology and process scheduling. Although efficient MIS algorithms are well-known [1], the increasing scale of data-intensive applications suggests the use of distributed-memory hardware (clusters), which in turn requires distributed-memory algorithms.

Luby's Monte Carlo[2] MIS algorithms are often used for parallel MIS implementations. Luby MIS algorithms are designed focusing on shared memory machines and analyzed using the *Parallel Random Access Machine* (PRAM) model. Luby's algorithms do not immediately lend itself to efficient distributed memory parallel algorithms due to overhead incurred by synchronization and distributed subgraph computations. In this paper, we present distributed versions of Luby's Monte Carlo algorithms (Algorithm A and Algorithm B) that minimize these overheads. Furthermore, we derive a variation of Luby(A) that avoids computing random numbers in every iteration. All presented algorithms are implemented in the AM++ runtime [3] and their performance is evaluated. Our results show that the proposed algorithms scale well in distributed settings. We also compare our results with the *FilteredMIS* implementation in the Combinatorial BLAS library [4], and we show that our implementations are several times faster compared to FilteredMIS algorithm.

II. Related Work

Most parallel MIS algorithms focus on shared memory, using the PRAM model for parallel complexity analysis (e.g., [5], [6], [7], [8], [9], [10]). In this work, we specifically focus on Luby's [2] randomized algorithms (see Section III for details). Luby provided a detailed analysis of his algorithms using the PRAM machine model. Later, Luby's algorithm concepts were used to implement distributed versions. Lynch et al. [11] discuss a distributed version of Luby's algorithm for synchronous distributed networks. Métivier et al. [12] present an improved version of Lynch's algorithm, improving communication message complexity. Kuhn et al. [13] provide a deterministic distributed MIS algorithm. However, they assume a synchronous communication model and provide no experimental results.

The Combinatorial BLAS library[4] implements a distributed version of Luby's algorithm. Their implementation uses linear algebra primitives in implementing Luby's algorithm and also the algorithm works on filtered graphs[14]. Salihoglu et al. [15] implemented a distributed version of Luby's algorithm for Pregel-like systems. They used Luby's MIS algorithm to solve the graph coloring problem. Garimella et al. [16] compare the performance of Luby's implementation in Pregel with a parallel algorithm designed by Blelloch et al. [10]. Blelloch et al., parallelize the sequential greedy lexicographically-first MIS algorithm. This algorithms uses a *priority DAG* constructed 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.

The problem of parallel MIS has been the focus of much theoretical research. Part of the related work discussed above

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Algorithm 1: General Iterative Scheme in Luby MIS

Input: Graph $G = (V, E)$
Output: Maximal Independent Set S_{mis}

1: $S_{mis} \leftarrow \emptyset$
2: $S_{is} \leftarrow \emptyset$ ▷ initializing the independent set
3: $G_s(V_s \leftarrow V, E_s \leftarrow E)$
4: **while** $G_s \neq \emptyset$ **do**
5: $S_{is} \leftarrow$ Select an independent set from G_s
6: $S_{mis} \leftarrow S_{mis} \cup S_{is}$
7: $V_r \leftarrow S_{is} \cup \{\text{neighbors of vertices in } S_{is}\}$
8: $E_r \leftarrow \{\text{edges incident on vertices in} V_r\}$
9: $G_s \leftarrow G_s(V_s \leftarrow (V_s - V_r), E_s \leftarrow (E_s - E_r))$

involves analyzing parallel time complexity or bit complexity of the algorithm discussed. Existing implementations of parallel MIS mostly adopt Luby's algorithms or they use an algorithm based on Luby's MIS. However, Luby's MIS does not immediately extend to efficient distributed memory parallel algorithm due to reasons we discuss in Section III.

A distributed implementation of Luby's MIS is openly available in CombBLAS library. However, CombBLAS algorithm is designed to work on "filtered" graphs. Further, it performs several pre-processing steps including removing self-edges and load balancing. The distributed Luby algorithms we present in this paper are designed specifically for large-scale static graph processing under a distributed memory setting and capable of processing unstructured graphs without preprocessing.

III. Luby's Algorithms

Luby's algorithms are the most widely used parallel algorithms for finding a MIS in shared memory. In his original publication, Luby discussed a *general iterative scheme* and four particular variations based on it. The general iterative scheme is listed in Algorithm 1. In every iteration, the general iterative scheme selects a non-empty independent set (Line 5) and merges it to the output (S_{mis}). Then, the selected independent set and its neighbors are removed from the input graph, and the resulting subgraph is fed into the scheme for the next iteration (Lines 7– 9). This process is repeated until the resulting subgraph is empty. In every iteration, the general iterative scheme generates a new independent set. Luby proved that the union of all those independent sets is a maximal independent set.

To select an independent set from a subgraph in an iteration, Luby proposed two Monte Carlo algorithms: *Select A* and *Select B*. Select B is further enhanced to create two more variations, *Select C* and *Select D*. All four of those variations use randomization to calculate an independent set. Select algorithms A, B, and C are non-deterministic, while Select D is deterministic. In this paper, we focus on Select algorithms A and B (since C and D are variations of B). Select A and Select B algorithms are summarized in Table I.

Select A is the simplest of the algorithms. It considers all the vertices (V_s) in the subgraph to be in an independent set. Then, it assigns a random number, r ($1 \leq r \leq |V_s|^4$) to each vertex in the subgraph. Then, for every edge in the subgraph (edges in E_s), Select A removes the vertex in the edge that has the greater random value.

Select A	**Select B**
1. Assume all the vertices in the subgraph are independent	1. Assume vertices that satisfy the $coin$ random variable test are independent
2. Assign random values to vertices in V_s	2. Calculate the independent set based on the degree distribution of the subgraph
3. Calculate the independent set based on assigned random values	

TABLE I: Independent set selection criteria for Select A and Select B algorithms

Unlike Select A, Select B does not consider all vertices in the subgraph to be in the independent set in an iteration. Instead, Select B uses a random variable (*coin*) to decide whether a vertex in the subgraph should be selected to be in an independent set. The value of the coin is determined based on a probability distribution defined using degree distribution of the subgraph. More precisely, if $d(v)$ is the degree associated with a vertex $v \in V'$, then $coin(v) = 1$ with probability $1/2d(v)$. If $d(v) = 0$, then $coin(v)$ is always 1. For more details about Algorithm B, we refer the reader to Luby's original publication in [2].

A. Luby's Algorithms in Distributed Memory

The Luby algorithms do not lend themselves directly to efficient distributed-memory parallel implementations. Luby's algorithms are designed focusing on shared memory machines and are analyzed using the PRAM model. In the PRAM model, all processors need to synchronize after reading from the shared memory and also before writing to the shared memory. A natural way to extend a shared-memory Luby algorithm to distributed memory is to use the *Bulk Synchronous Parallel* (BSP) approach. In BSP [17], shared memory operations can be converted to compute, communication and barrier synchronization phases. However, this approach results in many barrier synchronization phases.

Another issue is that the "general iterative scheme" (Algorithm 1) depends on subgraph computations. That is, in every iteration, the algorithm constructs a new subgraph the by removing vertices and edges of the independent set calculated in the current iteration from the graph. Constructing a subgraph in every iteration is inefficient in distributed memory as it involves communication and synchronization even if the subgraph is maintained implicitly through vertex masking.

In addition, the Select A algorithm requires a new choice of random numbers in every iteration. The range of the numbers depends on the number of vertices remaining in the subgraph. Therefore, the random number generation requires a reduction over the number of vertices in the subgraph and a barrier in every iteration. Furthermore, random number generation incurs a significant computational overhead.

In the next section, we discuss how we extend Luby's algorithms to distributed execution while avoiding the drawbacks discussed above. In the proposed algorithm, the overhead of barrier synchronization phases are minimized by overlapping computation and communication. The subgraph computation is achieved through vertex filtering. However, vertex filtering cripples the ability to iterate over the graph data structure in

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

parallel. Therefore, in our implementation, we use a parallel data structure. We also present a variation of Select A algorithm that avoids random number generation in each iteration and uses random numbers generated initially.

IV. Distributed Memory Parallel Luby Algorithms

The proposed distributed-memory parallel Luby algorithms use a 1D distribution to distribute the graph vertices among participating ranks. Every rank gets a subset of vertices and an edge subset relevant to the vertices. Within a rank, a vertex subset and its associated edge subset is represented using a local graph representation ($G^{local} = (V^l, E^l)$). A vertex is "owned" by a rank and vertices owned by different ranks communicate by passing messages. Message passing communication between ranks is designed based on BSP processing, but with overlapped communication and computation for improved efficiency.

A. Distributed General Iterative Scheme

Algorithm 2: Distributed General Iterative Scheme

```
1:  procedure LubyIterate(G^local, select^fn)
2:    buffer ← {}
3:    delete ← {}
4:    while there are NIL vertices in G do
5:      select^fn(&buffer, &delete)
6:      epoch {
7:        for each Vertex v in buffer in parallel do
8:          if v is not in delete then
9:            mis[v] ← IN
10:           for each u in adjacencies(v, G^local) do
11:             Send(u, OUT)
12:     }
13:
14: procedure Receive(v : Vertex, s : State)
15:   mis[v] ← s
```

The distributed general iterative scheme (Algorithm 2) requires subgraph computation. Though explicit subgraph computation may be practical in sequential and shared-memory parallel environments, it is inefficient in distributed memory due to overheads of creating and distributing a new subgraph at every iteration. Equivalent distributed subgraph computation functionality can be achieved with vertex filtering (i.e., apply a filtering predicate to indicate whether a vertex is to be considered in the current computation). Although with vertex filtering more edges then strictly necessary are traversed in every iteration, the increased parallel efficiency outweighs the cost of the unnecessary traversals.

To alleviate the overhead of subgraph computations in distributed memory, we use two data structures:

1) An *append buffer* (*buffer*) – for efficient parallel access; and
2) A *set structure* (*delete* set).

These two data structures are created by the general iterative scheme and are passed into a specific Select (A or B) algorithm. The Select algorithm is responsible for populating the append buffer and delete set. When the Select algorithm decides a vertex is a candidate to be in the MIS, it adds the vertex to the buffer. Next, after buffer contains the initial MIS candidate vertex set, all vertices that have a neighbor with a lesser random value assigned to it, are placed in the delete set. The general iterative scheme traverses the append buffer in parallel and checks whether a vertex is in the delete set; if the vertex is *not* present in the delete set, then the vertex is added to the result MIS. To reduce the contention when operating on the delete set we implement delete set as a collection of sets, where each thread maintains a set local to the thread. Insertion to a delete set is local to the invoking thread. When querying an element from the delete set, we first check whether the element resides in the thread-local set, and then we search for the element in sets belonging to other threads. During the search phase the sets are only read, so they can be safely shared between threads. The two-step design (buffer and then delete set) limits contention between threads to the low-overhead insert contention on the append buffer.

During computation, a vertex can be in one of three states (stored in a property map)

1) *IN*– vertex is in MIS;
2) *OUT*– vertex is not in MIS; and
3) *NIL*– vertex is not yet processed.

Initially, all the vertices are in state *NIL*. When the algorithm terminates, all vertex states are changed either to *IN* or *OUT*.

Whether a vertex state should be changed from *NIL* to *IN* or *NIL* to *OUT* is decided within the general iterative scheme (*LubyIterate* in Algorithm 2). First, the general iterative scheme invokes the appropriate "Select" algorithm *select*^{fn} (*Select*^{A} or *Select*^{B}). The specific select algorithm is responsible for populating the append buffer and the delete set (Line 5). The general iterative scheme iterates through the append buffer in parallel and checks whether a vertex is present in the delete set (Lines 7–11). If a vertex is *not* in the delete set then that vertex state is updated to *IN* (Line 9). When a vertex state is changed to *IN* state, all its neighbors' states are to changed to *OUT* state (Line 11). The *Send* operation determines to which rank the message should be sent based on the destination vertex and the graph distribution. Messages sent through *Send* are received in the *Receive* (Lines 14–15) function. Traversing through vertices in the append buffer and updating vertex states takes place within a single super-step (i.e., within a single *epoch*).

B. Distributed Select A

Select A (Luby(A)) algorithm takes a local graph representation, an append buffer, and a delete set. Select A algorithm is listed in Algorithm 3. G^{local} is the local graph representation, *abuffer* represents the append buffer and *deleteset* represents the delete set. Select A algorithm first calculates the number of vertices in *NIL* state using a *global reduction* (Lines 4–7). Then, for each vertex in *NIL* state, a random value, k ($1 < k <$ *globalcount*^{4}), is assigned (Lines 10–13). When generating random values, a combination of rank id and thread id is used to generate a unique random value seed for every thread ($Seed()$ on Line 8). Random values are stored in a property map; a rank only stores random values for vertices in its local graph. While assigning a random value to each local

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Algorithm 3: Distributed Select^A

```
1: procedure Select^A(G^local, ref abuffer, ref deleteset)
2:   localcount ← 0
3:   globalcount ← 0
4:   for each Vertex v in G^local in parallel do
5:     if mis[v] == NIL then
6:       localcount ← (localcount + 1)
7:   globalcount ← Reduce(localcount,SUM)
8:   random ← RandomDist(1, globalcount^4, Seed())
9: /*Assign random values to vertices in NIL state*/
10:  for each Vertex v in G^local in parallel do
11:    if mis[v] == NIL then
12:      π[v] ← random.Value()
13:      abuffer.add(v)
14: /*Use random values to remove conflicting
    vertices*/
15:  epoch {
16:    for each Vertex i in abuffer in parallel do
17:      for each j in adjacencies(v, G^local) do
18:        if u belongs to G^local then
19:          if π[i] >= π[j] then
20:            deleteset.add(i)
21:          else
22:            deleteset.add(j)
23:        else
24:          Send(j, i, π[i], COMPARE)
25:  }
26: /*Every Send call invokes a Receive*/
27: procedure Receive(j:Vertex, i:Vertex, irnd:Real, act:Action)
28:  if act == COMPARE then
29:    if irnd >= π[j] then
30:      Send(i, j, π[j], REMOVE)
31:    else
32:      deleteset.add(j)
33:  else
34:    if act == REMOVE then
35:      deleteset.add(j)
```

vertex in a *NIL* state, Select A algorithm inserts those vertices to the append buffer (Line 13).

In the next phase, the algorithm traverses through vertices in the append buffer in parallel and inserts adjacencies with higher random values that are in *NIL* state to the delete set (Lines 16–24). If the adjacent vertex does not belong to G^{local}, then a message is sent to the appropriate locality (Line 24), including the source vertex id i and its random value. The rank that owns the destination vertex j checks whether the received random value is less than the random value of j. If so, j is added to the delete set, otherwise a message is sent back to i to add i to the delete set. The first message is denoted using the action *COMPARE* and the second action is represented using the action *REMOVE*. Every *Send* call corresponds to a *Receive* function (Lines 27–35) invocation. At the end of the execution of epoch in Lines 15–25, vertices in the append buffer but not in the delete set represent an independent set.

C. Select AV (A variation of Select A)

The Select A algorithm generates random numbers in every iteration. Random number generation is computationally expensive, also, to generate random numbers we need to calculate the total subgraph vertex set size (across all distributed ranks) which incurs additional communication overhead due to distributed reduction and barrier synchronization.

Select AV (Luby(AV)) is almost same as Select A, except that, we do not generate random numbers to calculate an independent set. Instead, we use graph representation vertex IDs to break the symmetry and calculate an independent set. In other words, $\pi[i] = i \; \forall i \in V_s$, where V_s represents the vertex set of a subgraph. This method depends on the distribution of vertex identifiers in the graph data structure, but it works well with our representation which randomly permutes vertices before the algorithm begins.

D. Distributed Select B

Algorithm 4: Distributed Select^B

```
1: procedure Select^B(G^local, ref abuffer, ref deleteset)
2:   degree ← {} /*Calculate vertex degrees relative to
     the subgraph*/
3:   epoch {
4:     for each Vertex v in G^local in parallel do
5:       if mis[v] == NIL then
6:         for each u in adjacencies(v, G^local) do
7:           if u belongs to G^local then
8:             if mis[u] == NIL then
9:               degree[v] ← (degree[v] + 1)
10:          else
11:            Send^1(u, v, ISNIL)
12:  }
13: /*Build independent set based on coin value*/
14:  for each Vertex v in G^local in parallel do
15:    if coin(v, degree[v]) == 1 then
16:      abuffer.add(v)
     /*Remove conflicting vertices*/
17:  epoch {
18:    for each Vertex i in abuffer in parallel do
19:      for each j in adjacencies(v, G^local) do
20:        if j belongs to G^local then
21:          if j is in abuffer then
22:            if degree[i] <= degree[j] then
23:              deleteset.add(i)
24:            else
25:              deleteset.add(j)
26:        else
27:          Send(j, i, degree[i], COMPARE)
28:  }
29: /*Every Send^1 call invokes a Receive^1*/
30: procedure Receive^1(u:Vertex, v:Vertex, act:Action)
31:  if act == ISNIL then
32:    if mis[u] == NIL then
33:      Send^1(v, u, NILTRUE)
34:  else
35:    if act == NILTRUE then
36:      degree[u] ← (degree[u] + 1)
37: /*Every Send call invokes a Receive*/
38: procedure Receive(j:Vertex, i:Vertex, irnd:Real, act:Action)
    ▷ /*Same as the Receive procedure in Algorithm 3*/.
```

Select B (or Luby(B)) algorithm does not add all the vertices in *NIL* state to the append buffer, instead it only adds a subset of vertices in *NIL* state. The subset is calculated based on the random variable *coin*. The random variable coin has two values 0 and 1 that are assigned based degree distribution of vertices. Therefore, Select B algorithm first calculates the degree of each vertex relative to the subgraph. Then, the algorithm selects

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

subset vertices in *NIL* state for an independent set based on the coin value. Afterwards, the algorithm checks in parallel if any adjacent vertices were selected. If so, the algorithm uses degrees of vertices to resolve the conflict and remove any non-independent vertices. Algorithm pseudocode for Select B algorithm is presented in Algorithm 4.

Calculating vertex degrees in the current subgraph requires communicating with remote ranks. Lines 4–11 show the degree calculation. If an adjacent vertex is not in current locality, the algorithm sends a message to the remote locality to check the status of the adjacent vertex (Line 11), using the *ISNIL* action to query the status of the adjacent vertex. These messages invoke the *Receive*[1] procedure on the remote locality (Lines 30–36). If the adjacent vertex is in the *NIL* state, the remote rank sends back a reply *NILTRUE*.

Lines 14–16 show how Select B algorithm invokes *coin*, a random variable, to select a subset of vertices as a candidate for an independent set. The coin function takes a vertex and its degree to decide whether the random variable value is 1 or 0. If the coin function returns 1, the vertex is added to the append buffer.

After adding a subset of vertices in the subgraph to the append buffer, Select B algorithm iterates through the content in the append buffer in parallel. If a vertex in the append buffer has an adjacent vertex that is also in the append buffer, then the vertex with the smaller degree is removed. The vertex that has a lower degree is added to the delete set (Lines 18–27). If the adjacent vertex is in a remote locality a message is sent (Line 27). The receive code to handle messages sent (Line 27) is similar to the *Receive* procedure in Select A (Lines 27–35, in Algorithm 3). Like in Select A, when Select B finishes executing the epoch in Lines 17–28, vertices in the append buffer but not in the delete set represents an independent set.

V. Experiments & Results

A. Implementation

The proposed algorithms are implemented on top of an active messaging framework AM++ [3], using *pthreads* for in-node threading.

Graph vertices are equally distributed among participating nodes (1D block distribution). The local graph is represented using compressed sparse row (CSR) format. Every undirected edge is represented using two directed edges.

Algorithm implementations do not require pre-processing of inputs and can deal with parallel edges and self-loops (common artifacts in synthetic inputs). 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:       ...
```

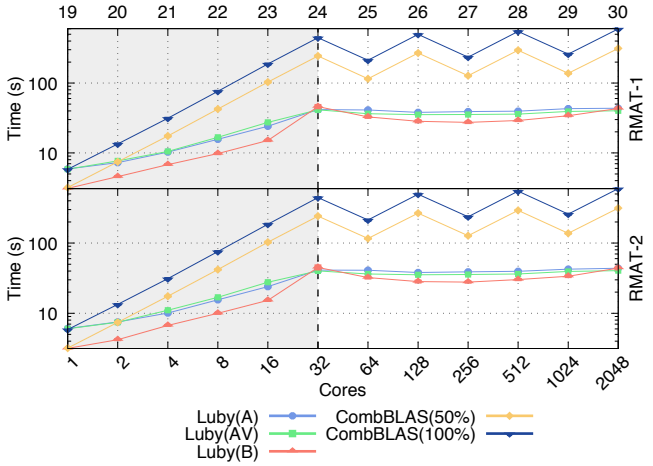

Fig. 2: Weak scaling results of MIS algorithms for RMAT graphs, including FilteredMIS. The shaded area shows the shared memory execution.

B. Experimental Setup

We ran our experiments on a Cray XC system with 2 Broadwell 22-core Intel Xeon processors and 128 GB DDR4-2400 memory per node. For scaling results, we used only up to 32 cores per node to provide uniform scaling. We used Cray MPICH MPI (version 7.4.4) and GCC 6.3.0. We used two processes per node (one per NUMA domain), and we used MPI in thread-multiple mode.

C. Graph Input

We evaluate the MIS algorithms in-terms of *weak scaling* and *strong scaling*. We use *RMAT* [18] synthetic graphs. Two types of RMAT synthetic graphs are used. They are:

- RMAT-1: Graphs based on the current Graph500 [19] Breadth First Search benchmark specification with RMAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$.
- RMAT-2: Graphs generated based on the proposed Graph500 [20] SSSP benchmark specification with RMAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

Both types of graphs have 16 undirected edges per vertex. Strong scaling experiments were carried on RMAT-1 and RMAT-2 scale 25 graphs (largest possible before memory exhaustion).

D. Weak Scaling Results

For weak scaling we compare our implementation to FilteredMIS[14] implementation in the CombBLAS [4] library. The FilteredMIS algorithm runs Luby(A) with edge filtering. However, implementation presented in this paper does 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.

Figure 2 shows weak-scaling results for of Luby(A), Luby(AV), Luby(B), and FilteredMIS for RMAT-1 and RMAT-2 graph inputs. For both graph inputs distributed Luby algorithms

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

presented in this paper outperform CombBLAS,FilteredMIS (for both 50% and 100% edge filtering).

Results from distributed execution of FilteredMIS 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. We enabled multi-threading in CombBLAS so the tasks can take advantage of multiple cores. We observed that CombBLAS performed the worst with one task per core (no multi-threading).

As per Figure 2, Luby(B) performs better in shared memory (when, cores < 32) for both graph inputs. Unlike Luby(A), Luby(B) does not consider all vertices in the subgraph to be in the initial approximation to the independent set. Luby(B) has a choice step, where it calculates a subset from the subgraph vertices based on the probability distribution (See Section III, "coin" function). Since Luby(B) selects subset from the subgraph vertices, Luby(B) is able to calculate an independent set faster than Luby(A) in an iteration. On the other hand, Luby(A) does not have a choice step and it considers all the vertices in the subgraph as a candidate for an independent set. The data statistics we collected shows that Luby(A) spends most of its time in calculating an independent set in the first iteration, especially in shared-memory execution. For example, the statistics in Table II are collected for scale 20 RMAT-1 graph on 2 cores. Other scales behave in the same way.

	Luby(A)	Luby(B)
Exec. Time (sec.)	7.13	4.17
Time for 0th iteration (sec.)	6	0.01
No.of Iterations	5	20
Vertices in 0th iteration	1048576	517394
Deleted Set Sz.	917623	2043

TABLE II: Runtime statistics for Luby(A) and Luby(B) on RMAT-1, Scale 20 graph on 2 cores.

Luby(A), however, converges much faster than Luby(B). As shown in Table II, Luby(A) takes 6 iterations to terminate while Luby(B) takes 20 iterations. When the number of iterations are higher, the overhead of global synchronization also increases. At scale 24 (when cores = 32) we see a sudden increase in Luby(B)'s runtime. This is because at scale 24 execution runs in 2 processes and the overhead of communication become significant. The performance of Luby(B) is more affected at scale 24 than the performance of Luby(A). Since the number of iterations are higher in Luby(B), synchronization overhead is more important in Luby(B) than in Luby(A).

The difference between Luby(A) and Luby(AV) are not prominent. Luby(AV) is able to achieve a slight improvement over Luby(A) in distributed execution. This is mainly because Luby(AV) avoid the need to calculate vertex set size using a global reduction and also avoids the need to generate random numbers.

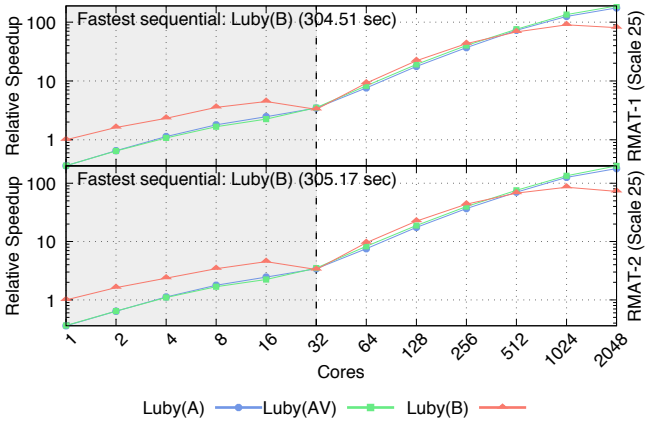

Fig. 3: Strong scaling results of MIS algorithms for RMAT-1 and RMAT-2, Scale 25 graph inputs. Shaded region shows the shared memory execution.

All the Luby algorithms (A, AV, B) presented in this paper do not show perfect weak scaling performance in shared memory due to the contention created on data structures with the increasing number of threads. However, we see good weak scaling in distributed memory for all Luby (A, AV, B) algorithms.

E. Strong Scaling Results

For strong scaling experiments, we ran MIS algorithms on RMAT-1 and RMAT-2 scale 25 graphs. To have better understanding about how algorithms scale relative to each other, 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.

Figure 3 shows the strong scaling results of MIS algorithms presented in this paper for the graph inputs discussed above. Due to synchronization overhead discussed in the context of weak-scaling results, Luby(B) shows better speedup in shared memory, but in distributed memory Luby(B) speedup drops at higher scales due to higher synchronization overhead.

VI. Conclusions

Most of the existing research on MIS focuses on theoretical analysis than practical implementation. Further, the few practical implementations mostly implement Luby's MIS algorithms. Luby's MIS does not immediately extend as efficient distributed-memory parallel algorithm due to synchronization overheads and subgraph computation overheads.

In this paper we presented distributed versions of parallel Luby's algorithms. The algorithms we propose minimize the synchronization overhead by overlapping communication and computation and minimizes the subgraph computation overhead using vertex filtering and by maintaining parallel data structures. Our results show that the algorithms we present are several times faster than existing MIS algorithms.

VII. Acknowledgments

The research is in part supported by the Defense Advanced Research Projects Agencys (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] 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.

[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] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

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

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

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