A Multi-GPU Algorithm for Computing Maximal Independent Sets in Large Graphs

Anju Mongandampulath Akathoott
Texas State University
San Marcos, USA
anju.m.a@txstate.edu

Benila Virgin Jerald Xavier
Texas State University
San Marcos, USA
benila.jerald@txstate.edu

Martin Burtscher
Texas State University
San Marcos, USA
burtscher@txstate.edu

Abstract

Computing a maximal independent set (MIS) of a graph is an important problem in many scientific applications. Several parallel algorithms exist to perform this computation quickly. Though the state-of-the-art GPU implementation is very efficient, it cannot process graphs that do not fit in the global memory of a single GPU. We propose MG-MIS, a multi-GPU algorithm that addresses this problem. It distributes the computation across the GPUs in a compute node and uses novel techniques to minimize inter-GPU communication. Our results show that, for graphs that require more than 32 GB memory, MG-MIS outperforms the state-of-the-art single-GPU code with UVM by a geometric mean of 17.73× on a system with 4 V100 GPUs, each with 32 GB global memory. For another set of graphs that require more than 12 GB memory, MG-MIS outperforms the same single-GPU code by 22.88× on a system with 2 RTX 3080 GPUs, each with a global memory of 12 GB. On average, the size of the MIS computed by MG-MIS is 2.6% smaller than that produced by the state-of-the-art single-GPU code.

CCS Concepts

• **Computing methodologies** → **Massively parallel algorithms**.

Keywords

Maximal independent sets, Multi-GPU computing, Parallelism, Communication optimization

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730416

ACM Reference Format:
Anju Mongandampulath Akathoott, Benila Virgin Jerald Xavier, and Martin Burtscher. 2025. A Multi-GPU Algorithm for Computing Maximal Independent Sets in Large Graphs. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3730416

1 Introduction

A subset of vertices in an undirected graph is considered an independent set if no two vertices within the subset are connected by an edge. An independent set is *maximal* if no other vertex can be added without violating this independence. Such a set is a *maximum* independent set if it has the largest cardinality among all possible maximal independent sets. A graph can have multiple maximal as well as maximum independent sets. Finding maximum independent sets is computationally very expensive due to the exponential number of possibilities that need to be examined [12]. A Maximal Independent Set (MIS) serves as a useful alternative, since a high-quality MIS can be quite close in size to the maximum independent set and can be computed much faster, based on some heuristics [5].

Computation of a MIS is a key step in many scientific problems such as job scheduling [8], structural analysis of proteins [11], VLSI design [27], etc. The parallel computing domain itself benefits from MIS, since a MIS in this context corresponds to a large set of jobs that can be run in parallel. For the results to be useful in this domain, the MIS implementation must be fast. Several parallel algorithms [5, 14, 18, 19] have been proposed to perform the MIS computation quickly. Among the publicly available parallel implementations for multi-core CPU [3, 29] and GPU [5–7] systems, ECL-MIS [5], a single-GPU algorithm, is the fastest we could find.

Although ECL-MIS is highly efficient and produces larger maximal independent sets than other MIS algorithms in the literature, it has a major drawback: it cannot process graphs that do not fit in the global memory of a single GPU. By changing ECL-MIS to use Unified Virtual Memory (UVM) [10, 24], we enabled it to process larger graphs. However, its efficiency is significantly reduced due to the high frequency

of data transfers between the CPU and the GPU. In this paper, we describe MG-MIS, a multi-GPU algorithm that efficiently computes maximal independent sets of large graphs that do not fit in the global memory of a single GPU.

Multi-GPU algorithms typically divide the data among the available devices (where *host* refers to the CPU and *devices* refers to the GPUs in the system). In this context, memory accesses can be categorized as follows: (i) *local*, that is, a device accesses data in its own global memory, and (ii) *remote*, that is, a device accesses data that is physically located in another device. GPUs typically have high-bandwidth memory, making local accesses very efficient. In contrast, remote memory accesses are much slower because they must go through an interconnect such as PCIe or NVLink and might conflict with the requests from other GPUs.

Support for Peer-to-Peer (P2P) memory accesses makes inter-device communications faster since the data transfers are not host-mediated; otherwise, the data transfer between two devices must go through the host memory, making it slower. Though NVLink can provide higher bandwidths than PCIe, it is still slower than local memory accesses. Using unified memory is another option, but it is even slower since it uses on-demand paging to migrate data between the host and the GPUs or between the GPUs.

Even in the best-case scenario — where NVLink is available and the system supports P2P access — inter-GPU communication can still be a performance bottleneck if not optimized properly. Hence, the key challenge in designing an efficient multi-GPU algorithm lies in identifying ways to reduce both the volume and the frequency of inter-GPU data transfers. MG-MIS addresses these challenges by minimizing the frequency of communications between GPUs and by combining a large number of remote accesses into more efficient bulk data transfers between devices. It efficiently processes large graphs whose data does not fit in the global memory of a single GPU but fits in the total global memory of all the GPUs in the system.

MG-MIS divides the vertices of the graph into equally sized chunks of vertices with consecutive IDs. Each GPU in the system owns a chunk of the vertices, all edges incident to them, and the corresponding auxiliary data structure that stores the MIS-inclusion status of those vertices. We use the term *status* of a vertex to indicate if it is included in the MIS. The possible values of status are *in*, *out*, and *undecided*.

MG-MIS employs the elegant priority-based approach proposed by Luby in his seminal paper on parallel MIS computation [19] in which vertices are assigned random priority values. If a vertex v has either the highest priority among its neighbors, or all of its higher-priority neighbors have already been excluded from the MIS, v is added to the MIS.

The algorithm proceeds in rounds. In each round, all *undecided* vertices are processed to see if their status can be finalized. Once each vertex's status is decided, the algorithm terminates. Note that the neighbor vertices whose inclusion status needs to be checked will require remote memory accesses if those neighbors reside on another device.

Handling the remote memory accesses efficiently is non-trivial and the most challenging part of adopting Luby's approach in our multi-GPU MIS algorithm. To minimize inter-GPU memory accesses, MG-MIS employs the following strategies. (i) Whenever possible, we finalize the status of a vertex without checking any remote neighbors. (ii) If a vertex v is decided to be *in* the MIS, all its neighbors in each remote device are informed together so that they all can mark themselves *out*, avoiding the need for these neighbors to later individually query the status of v over multiple remote memory accesses. (iii) If each vertex in a set of vertices S, owned by a device, requires to check the status of some neighbor on a remote device, we combine these requests into a buffer and send it to the target device in a single transaction. The target device prepares the response for all vertices in S and sends it back to the requesting device via another single transaction. (iv) We assign priorities to vertices in such a way that priority values depend only on the vertex IDs and a fixed hash function, avoiding inter-GPU communication altogether when determining priorities of remote neighbors.

These strategies help minimize remote accesses and, in some cases, avoid them entirely. Whenever remote accesses are performed, we combine them into a single transfer to reduce the frequency of inter-GPU communication.

Our results from a system with four V100 GPUs (each with 32 GB of global memory) demonstrate that for inputs with a memory requirement exceeding 32 GB, MG-MIS achieves a geometric-mean performance improvement of 17.73× over ECL-MIS, the state-of-the-art single-GPU algorithm augmented with UVM. On a system with two RTX 3080 GPUs (each with 12 GB of global memory), our results show that, for inputs with a memory requirement exceeding 12 GB, MG-MIS is 22.88× faster than ECL-MIS with UVM.

This paper makes the following main contributions.

- It presents MG-MIS, the first multi-GPU algorithm to compute a maximal independent set in large graphs that do not fit in the global memory of a single GPU.
- It describes algorithmic optimizations that help reduce inter-GPU communication.
- It shows how dividing the computation into local and remote memory access phases that are mutually exclusive leads to fast processing.
- It discusses how the communication of priority values can be avoided altogether.
- It demonstrates that our implementation of MG-MIS significantly outperforms the state-of-the-art single-GPU code with UVM on large graphs.

Our CUDA implementation of MG-MIS is available in open source on the web [21] and on github [22].

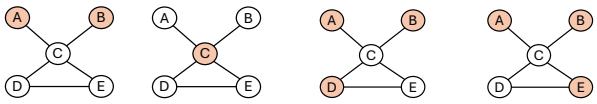
The rest of the paper is organized as follows. Section 2 presents background information and related work. Section 3 explains our technique and optimizations in detail. Section 4 describes the experimental methodology. Section 5 presents and analyzes the results. Section 6 concludes the paper.

2 Computation of MIS

This section provides background information on MIS computation and reviews Luby's approach to set the stage for describing MG-MIS in detail.

2.1 Background

An independent set of an undirected graph $G = (V, E)$ is a subset S of V such that, for any pair of vertices (u, v) where $u \in S$ and $v \in S$, u and v are non-adjacent in G, that is, $(u, v) \notin E$. Figure 1(a) highlights $\{A, B\}$, an independent set in the graph shown. However, it is not a *maximal* independent set (MIS) since either vertex D or vertex E can be added to it without violating the definition of an independent set. Figure 1(b) shows an independent set, $\{C\}$, that is maximal since no other vertex can be added to it. Several MISs are possible for the given graph. A MIS is called a *maximum* independent set if no other MIS with a larger cardinality exists for the same graph. By adding either D or E to the set $\{A, B\}$, we get two independent sets that are both maximal and maximum, shown in Figure 1(c).



(a) A set **(b) Maximal** **(c) Two maximum sets**

Figure 1: Examples of independent sets

Computing a maximum independent set is an NP-hard problem in general, whereas computing a MIS can be done much faster based on heuristics. Computing MISs of large graphs in parallel is the focus of this paper. To choose which vertices to include in the MIS, we use the priority-based approach proposed by Luby [19], which works as follows.

In the initialization phase, each vertex is assigned a random priority value as shown in Figure 2. This graph will be used as a running example throughout the paper. Each vertex is labeled with an <ID, Priority> pair. The MIS-inclusion *status* of a vertex can have three possible values – *undecided*, *in*, or *out*. Initially, all vertices are *undecided*. One thread is assigned to each vertex. The algorithm proceeds in rounds. In a round, each thread processing an undecided vertex v checks if any of the neighbors of v are in the MIS; If so, v is

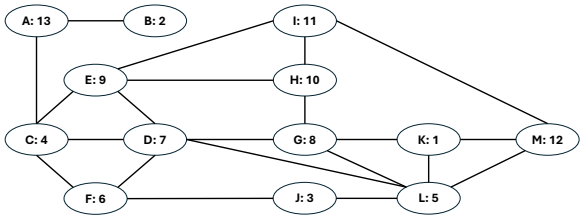


Figure 2: Input graph with random vertex priorities

decided to be *out* of the MIS. Otherwise, the thread checks if v has a higher priority than all of its neighbors that are yet to be decided. If it does, the thread marks v to be *in* the MIS. If v has at least one higher-priority neighbor whose status is undecided, v's status cannot be decided immediately; it has to wait for that neighbor's status to be decided first. In each round, at least one vertex will be added to the MIS, and several vertices may be excluded from the MIS. At the end of a round, if there is at least one vertex whose status is undecided, the algorithm performs another round of the same computation. When the status of all vertices has been decided, the algorithm terminates – The MIS consists of the vertices whose status is marked as *in*. Note that the status of each vertex changes from *undecided* to either *in* or *out* exactly once. The monotonic nature of this update along with the guarantee that at least one vertex's status will be decided in each round ensures termination of the algorithm.

In the example shown in Figure 2, in the first round, vertices A and M are included in the MIS, and vertices B, C, I, K and L are marked as *out*. In the second round, only H is added to the MIS, and E and G are marked as *out*. The third round marks D as *in* and F as *out*. With the fourth round marking J to be *in*, the algorithm terminates and the final MIS contains the vertices $\{A, D, H, J, M\}$. The discussion here assumes that the rounds are executed synchronously, but this is not a requirement. Section 3.3 provides more detail on why certain vertices are included and others are not.

2.2 Related work

There are several parallel algorithms that compute maximal independent sets. There are a few multi-GPU implementations of other graph algorithms, but none for computing MISs. In this section, we discuss the prior work that is relevant to MG-MIS.

The first parallel algorithm for computing MISs was proposed by Karp and Wigderson [14]. It identifies a suitable subgraph of the given graph G on which independent sets can be computed in parallel and then shortens G by removing the vertices in the computed set and their neighbors. The process is repeated on the new graph until the graph is empty. Based on this algorithm, Luby proposed a number

of more efficient parallel algorithms [18, 19], which vary in the way they choose the vertices to include in the MIS. We discuss one of them in detail in Section 2.1.

Similar to MG-MIS, ECL-MIS [5], the state-of-the-art single-GPU algorithm for computing MISs, is also based on Luby's approach. The priority values in it take the degree of the vertices into consideration and, hence, are not totally random. ECL-MIS is able to compute larger MISs compared to other algorithms that use entirely random priorities, but to compute a neighbor's priority, it requires the neighbor's degree information. Whereas this is not a problem when dealing with small graphs that fit in the global memory of a GPU, it can become a bottleneck in a multi-GPU processing setup – In order to know the priority of each remote neighbor of a vertex v, inter-GPU communication is required. Hence, we use random but fixed priority values in MG-MIS so that the vertex ID alone is sufficient to compute a vertex's priority.

Since ECL-MIS cannot directly process graphs that do not fit in the global memory of a single GPU, we extended it to support large graphs through the use of unified virtual memory. The modifications we made to the original ECL-MIS code are: (i) we changed the *int* types to *long* since the range of vertex IDs on our input graphs exceeds *INT_MAX*, (ii) we changed the allocation of all arrays from global memory to unified memory, and (iii) we added prefetching of data to the device before the processing starts so that as much data as possible is available in the global memory when the processing starts. Note that the prefetch time is not included in the runtimes we report. Importantly, due to the prefetching, the unified memory version of ECL-MIS delivers the same performance as the original code on graphs that fit completely in the global memory of a single GPU. We also note that the sets produced by the original and the unified memory versions of ECL-MIS are exactly the same. We compare MG-MIS to the unified memory version of ECL-MIS.

Even though there exist more single-GPU implementations for MIS in suites such as CUSP [7], Pannotia [6], and IrGL [26], they are slower than ECL-MIS [5]. The parallel CPU codes for computing MISs from suites such as PBBS [3] and Ligra [29] are also much slower than ECL-MIS. We tried running PBBS on a system with 256 GB of main memory after changing the data type from *int* to *long*, but the code does not terminate on most of our large inputs.

Recent work by Ma et al. [20] discusses computation of MISs of k-mers with an edit distance, which has applications in computational biology. It proposes three serial algorithms, two of which use greedy strategies, and the third one uses a BFS-based approach. Work by Kelley and Rajamanickam [15] discusses a variant of MIS, known as *Distance-2 MIS*, which computes a maximal subset of vertices of a given graph such that no two vertices in the subset are connected by a path of length less than or equal to 2 in the input graph.

Several recent works [12, 13, 30, 31] exist on *maximum* independent set computation. Though they target the problem of maximum (and not maximal) independent set computation, two of them by Imanaga et al. [12, 13] perform repeated computation of maximal sets on multiple instances of the same graph that vary in the vertex priorities and finally output the largest cardinality set found among these instances as the maximum independent set. They use ECL-MIS (which they refer to as DP-MIS in their paper to indicate the use of degrees in deciding priorities) for the MIS computations. They also run their algorithm on a multi-GPU system. However, they assign one full graph instance to each GPU and run several graph instances in parallel using the multiple GPUs. They do *not* use multiple GPUs to compute a MIS on a single large graph, which is the target of our work.

3 Technique

We propose a new multi-GPU algorithm named MG-MIS to compute a maximal independent set (MIS) of a large graph that does not fit in the global memory of a single GPU. The graph data is divided among the devices for processing. The devices communicate with each other as needed. In each round, MG-MIS decides the status of as many vertices as possible, based on their priorities. The algorithm is designed in such a way that inter-GPU communication, when unavoidable, is performed as bulk transfers. In this section, we discuss MG-MIS, detailing its key algorithmic optimizations to reduce the volume as well as the frequency of inter-GPU data movement. It can process graphs whose data fits in the global memory of all devices in the system combined.

3.1 Outline of MG-MIS

Initially, the graph vertices V are partitioned into equal-sized sets based on consecutive vertex IDs. Each set S is assigned to a device together with a list of all edges incident to the vertices in S. We refer to the neighbors of a vertex v as *local* neighbors if they are on the same device as v. Otherwise, they are *remote* neighbors. Each device also maintains an array to store the status of the vertices in S and another array to record which remote neighbor each vertex in S is waiting for, if any. Importantly, we compute the priority of each vertex using a deterministic hash function that only takes the ID of the vertex as input (and uses vertex IDs to break ties). This is essential because it means that the priority of any neighbor, including remote neighbors, can be computed locally and does not require a remote access. However, a remote access is required to obtain the status of a remote neighbor. The status of a vertex can be decided only after the statuses of all its higher-priority neighbors have been decided. We note that, unlike many other graph algorithms, Luby's MIS algorithm tends to visit only a couple of neighbors per vertex as it

Algorithm 1 ComputeStatus

```
1:  repeat
2:      done[gpuId] ← true;
3:      runLocalRound(&done[gpuId]);
4:      for each remote device do
5:          sendNbrData();                          ▷ barrier
6:      for each remote device do
7:          setVerticesToOut();                     ▷ barrier
8:      if !done[gpuId] then
9:          for each remote device do
10:             prepareRemoteNbrQueries();
11:             sendQueries();                      ▷ barrier
12:     for each remote device do
13:         prepareResponse();
14:     for each remote device do
15:         sendResponse();                         ▷ barrier
16:     for each remote device do
17:         processResponse();                      ▷ barrier
18:     done ← done[0] && ··· && done[GPUs - 1];
19: until done
```

Algorithm 2 runLocalRound(done)

```
1:  v ← vertex assigned to the current thread;
2:  if status[v] is undecided then
3:      for each neighbor u in edge-list[i] of v do
4:          if isLocal(u) && status[u] = out then
5:              continue;
6:          if priority(u) > priority(v) then
7:              if isRemote(u) then
8:                  if i < remoteNbrIndex[v] then
9:                      continue;      ▷ u is known to be out
10:                 else
11:                     remoteNbrIndex[v] ← i;
12:             break;
13:     if i points to a neighbor of v then
14:         done ← false;                           ▷ v waits
15:     else
16:         status[v] ← in;
17:         processAllNeighbors(v);
18: else if status[v] = in then
19:     if remoteNbrIndex[v] points to a neighbor of v then
20:         addRemainingNbrsToBuffer();
```

stops when reaching the first higher-priority neighbor. This property alleviates the need for load balancing [5], rendering warp- or block-based processing unnecessary.

In the initialization phase, the status of all vertices is set to *undecided*, and the *remote-neighbor-index* value of each vertex is set to point to its first neighbor in the edge list. Then, the main iterative computation phase starts. Algorithm 1 outlines the steps of this phase. The main phase ends when the status of every vertex has been decided. The following subsection explains each step in this phase in detail.

3.2 Detailed description

Each iteration of the main computation phase shown in Algorithm 1 is run by the threads on all devices in parallel. It works as follows. For each device, a single thread sets the *done* flag to *true* (Line 2). Once the flag is set, all threads in the device proceed to the local computation step, which is outlined in Algorithm 2.

Computation - runLocalRound: This kernel is launched with one thread per vertex (Line 1). The goal of the thread is to see if v has a higher priority than all of its neighbors so that its status can be marked as *in*. If the status of v is currently *undecided*, the thread scans the neighbors (Line 3). If the neighbor u is local, its status is read (Line 4). If it is *out*, the status of v does not depend on it, irrespective of the priority of u. Hence, the thread proceeds to the next neighbor of v (Line 5). Otherwise, the priorities of v and u are compared. If u has a higher priority than v, the status of v can be *in* only if the status of u is *out*. If u is a local neighbor, it cannot be *out* due to the *continue* statement on Line 5. Hence, v must wait for u and there is no point in scanning the remaining neighbors, so the thread breaks out of the loop (Line 12).

If u is a remote neighbor (Line 7), we need to know if its status is *out*. The value stored in *remoteNbrIndex[v]* provides the edge-list index of the first remote neighbor whose status is still unknown to v. If the edge-list index i pointing to u is less than *remoteNbrIndex[v]*, it is guaranteed that u is *out* or has a lower priority than v. In either case, the thread can skip u and proceeds to the next neighbor (Line 9). Otherwise, *remoteNbrIndex[v]* is set to i to indicate that the next remote neighbor whose status v needs to know is at index i, and the thread breaks out of the *for* loop (Line 12).

Once the thread comes out of the neighbor-scanning loop, it checks if it has processed all of v's neighbors (Line 13). If not, v either has an undecided local neighbor with a higher priority or a higher-priority remote neighbor whose status is unknown. Hence, the thread sets the *done* flag to *false* (Line 14) to indicate that another iteration of Algorithm 1 is needed. If, instead, the thread has successfully scanned all neighbors, the status of v can be marked as *in* (Line 16). Since v is included in the MIS, we can mark all of v's neighbors as *out*. To accomplish this, the thread invokes the device function outlined in Algorithm 3.

Any thread that invokes Algorithm 3 scans all neighbors of the argument v (Line 3). If a neighbor u is local, its status is updated to *out* via a local memory access (Line 5). If u is a remote neighbor, MG-MIS applies the following optimization

Algorithm 3 processAllNeighbors(v)

```
1:  myId ← ID of the GPU owning v;
2:  minIndex ← (index of the last neighbor of v) + 1; ▷ end
3:  for each neighbor u in edge-list[i] of v do
4:      if isLocal(u) then
5:          status[u] ← out;
6:      else
7:          target ← getOwningGpuId(u);
8:          buffer ← send_buffer[myId][target];
9:          index ← next slot in buffer;          ▷ atomic op
10:         if index < MAX_BUF_SIZE then
11:             buffer[index] ← u;
12:         else
13:             done ← false;
14:             minIndex ← min(minIndex, i);
15: remoteNbrIndex[v] ← minIndex;
```

to minimize inter-GPU data transfers – In each device, it collects the IDs of the vertices whose status needs to be set to *out* on a remote device and records them in a per-target-device buffer. To do so, we identify the ID of the GPU that owns u (Line 7), pick the dedicated buffer for sending data from the owner of v to the owner of u (Line 8), and then atomically insert u into the next available slot in the buffer (Line 11). After all threads in a device have finished their local round of processing, another kernel is launched on the device to transfer these buffers to the respective target devices (Line 5 in Algorithm 1).

MG-MIS uses fixed-size buffers for these bulk data transfers. If there are d devices in the system, each device is allotted $d-1$ such buffers (i.e., one per remote device). Similarly, each device has another set of $d-1$ buffers for receiving data from the remote devices. Hence, each device has $2 \times (d-1)$ dedicated buffers for sending and receiving data.

In Algorithm 3, the case may arise where a thread is unable to insert all remote neighbors of vertex v it added to the MIS due to buffer-size limitations. In that case, it will store the edge-list index of the first such remote neighbor in *remoteNbrIndex[v]* and set the *done* flag to *false* to force another round of processing. In this next iteration, if the status of a vertex v is *in* and its *remoteNbrIndex* points to a valid neighbor, the thread recognizes that there are more remote neighbors whose status needs to be set to *out* and tries again to add them to the corresponding buffers (Line 20 in Algorithm 2).

Communication - sendNbrData: Once the *sending buffers* have been filled, their data needs to be transferred to the respective *receive buffers* in the target devices. The communication phase invoked from Line 5 of Algorithm 1 accomplishes this task. If the underlying system supports Peer-To-Peer (P2P) access between devices, the data is copied directly between devices; otherwise, it must go through the host.

Computation - setVerticesToOut: As soon as the data transfers have completed, the next kernel is invoked (Line 7 in Algorithm 1) once per remote device. This kernel is implemented as illustrated in Algorithm 4. The receive buffer is passed to the kernel as input. Each thread uses its ID as an index into the buffer. If it is valid, the thread reads the vertex ID from the indexed buffer element and sets the corresponding status to *out* via a local memory access (Lines 2-4).

Algorithm 4 setVerticesToOut

```
1: index ← ID of thread
2: if index < buffersize then
3:     v ← buffer[index];
4:     status[v] ← out;
```

After all devices have completed this process, they check if their *done* flag is set to *false* (Line 8 in Algorithm 1). If so, the status of some vertices owned by the device are not yet decided due to a higher-priority neighbor. If those neighbors are local, they will be checked again in the next iteration of Algorithm 2. However, if some of those neighbors are remote, we must check if their status is *out* so that the respective local vertices need not wait for them any longer. The next kernel prepares buffers that combine all such requests from each device.

Algorithm 5 prepareRemoteNbrQueries

```
1:  v ← ID of the vertex assigned to the thread;
2:  stop ← (index to the last neighbor of v) + 1;      ▷ end
3:  if status[v] = undecided then
4:      r ← remoteNbrIndex[v];
5:      if r < stop then
6:          u ← neighbor at index r of v's edge list;
7:          if isRemote(u) then
8:              target ← getOwningGpuId(u);
9:              buffer ← send_buffers[myId][target];
10:             index ← atomicAdd(curBufferSize, 2);
11:             if index < MAX_BUF_SIZE then
12:                 buffer[index] ← v;
13:                 buffer[index + 1] ← u;
```

Computation - prepareRemoteNbrQueries: Each thread performs the steps outlined in Algorithm 5 if the status of the vertex v assigned to it is still *undecided*. It retrieves the ID of the next remote neighbor, if any, that v is waiting for by looking up the *remoteNbrIndex* array (Lines 4-6). The check on Line 7 ensures that u is a remote neighbor. If so, the ID of the device that owns u (Line 8) and the buffer to be used are identified (Line 9). The next available index in the buffer is

computed via an atomicAdd operation (Line 10). Each thread reserves two consecutive slots in the buffer. It inserts v into the first slot and u into the second slot. In effect, the thread sends a <*requesterVertex*, *remoteVertex*> pair to the remote device. The ID of the requester vertex is needed in the next kernel, which prepares the response data for these queries.

Communication - sendQueries: At this point, the next data transfer phase is invoked (Line 11 in Algorithm 1). The vertex-ID pairs in the *send buffers* of each device are copied to the respective *receive buffers* in the target devices using P2P accesses if available. Once the data transfers have completed, the next kernel is launched on all devices to prepare the response for these queries (Line 13 in Algorithm 1).

Algorithm 6 prepareResponse

```
1: index ← ID of thread;
2: if index < (requestBufferSize / 2) then
3:     requester ← reqBuffer[2 * index];
4:     u ← reqBuffer[2 * index + 1];
5:     if status[u] = out then
6:         resp_idx ← atomicAdd(responseBufferSize, 1);
7:         if resp_idx < MAX_BUF_SIZE then
8:             responseBuffer[resp_idx] ← requester;
```

Computation - prepareResponse: In this kernel, outlined in Algorithm 6, each thread uses its ID as an index to access the request buffer (Line 1). Since each thread must read two consecutive slots, the number of threads in this kernel will be equal to half of the request-buffer size (Line 2). Each thread reads the indexed pair of data (Lines 3-4). It then checks the status of the requested vertex u via a local memory access. If the status is *out*, the thread inserts the ID of the requester into the response buffer, which is again done by atomically reserving a slot.

Communication - sendResponse: Once the response data has been prepared, all devices send the buffers to the respective receive buffers in the target devices (Line 15 in Algorithm 1). Upon completion, all devices launch the last kernel of the current iteration (Line 17 in Algorithm 1).

Computation - processResponse: The kernel shown in Algorithm 7 scans the response buffer. Each thread uses its ID, if it represents a valid index, to retrieve a vertex ID v from the buffer (Line 3). The mere presence of v in the response buffer indicates that the status of the remote neighbor that v queried is *out*. Hence, v does not have to wait for that neighbor anymore. To skip it, *remoteNbrIndex[v]* is updated to point to the next remote neighbor in v's edge list (Line 4) or to the end of the list if there are no more remote neighbors.

Termination Condition: At the end of each iteration of the main computation phase, MG-MIS combines the *done* flags from all devices using a logical AND operation and

Algorithm 7 processResponse

```
1: index ← ID of thread;
2: if index < buffersize then
3:     v ← buffer[index];
4:     remoteNbrIndex[v] ← index of the next remote
   neighbor in edge list of v;
```

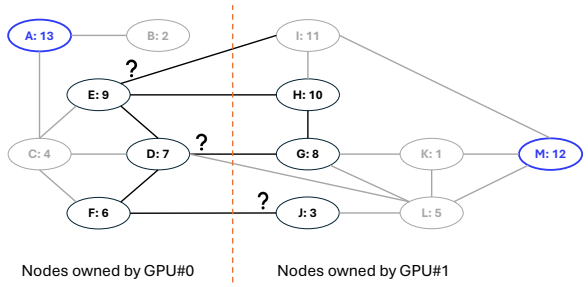


Figure 3: Iteration 1: asking for remote neighbor status

copies the result to the host. If the result is *false*, at least one device has a vertex whose status has not yet been decided. Hence, MG-MIS performs another iteration of the outer loop in Algorithm 1. If, instead, the result is *true*, the algorithm terminates since the status of all vertices has been decided.

3.3 Working example

This subsection illustrates the operation of MG-MIS on the example graph from Figure 2. Let us assume that our system has two devices. The first half of the vertices ($\{A, B, \cdots, F\}$) is assigned to GPU_0 and the second half ($\{G, H, \cdots, M\}$) to GPU_1. Initially, the status of all vertices is *undecided*.

In the first iteration of the repeat-until loop in Algorithm 1, the status of vertices A and M are decided to be *in* since they both have a higher priority than all of their neighbors. As a consequence, their neighbors, which all happen to be local, are set to *out*. Since the two *in* vertices have no remote neighbors, the data transfer and setting of remote neighbors to *out* steps (Lines 5-7) are irrelevant in this iteration.

In the next step, vertices D, E, and J prepare to query the status of their first higher-priority remote neighbors G, I, and F, respectively. The devices transfer these queries to each other (Line 11). Figure 3 shows the state at this point in the algorithm. Vertices with thick blue borders are included in the MIS, grayed out vertices are *out*, grayed out edges are no longer relevant, and a question mark on an edge next to a vertex indicates that the vertex is querying the status of the corresponding remote neighbor.

Among the queried vertices, only I is set to *out*. Hence, only I sends a response back, allowing E to move its remote-neighbor-index forward to point to H.

In the second iteration, H is added to the MIS because its only higher-priority neighbor I is *out*, H's local neighbor G is set to *out* immediately, and its remote neighbor E is set to out after the communication round. Then, D and J query the status of their higher-priority remote neighbors G and F, respectively. The response from G allows D to update its remote-neighbor index. Since D has no more higher-priority remote neighbors, this index is set to the end of D's edge list.

In the third iteration, D is set to *in* since it has no undecided or higher-priority neighbors, and its neighbor F is set to *out*. Finally, J queries the status of F and, based on the response, sets itself to *in*. At this point, all vertices are decided and the algorithm terminates with the MIS $\{A, D, H, J, M\}$.

4 Experimental Methodology

We compare the performance of MG-MIS with ECL-MIS [5], the state-of-the-art single-GPU code, since there is no prior multi-GPU MIS implementation. We obtained the code for ECL-MIS from the official website [4]. The original version of ECL-MIS can only process graphs that fit in the global memory of a GPU. Hence, we wrote a second version in which we enabled the use of Unified Virtual Memory (UVM) so that it can process larger graphs. In the rest of the paper, we use "ECL-MIS-UVM" to refer to this extended version. Note that, for all inputs that fit in the global memory of a single GPU, the complete data is prefetched into the global memory before the processing starts. Hence, ECL-MIS-UVM performs nearly identical to the original version of ECL-MIS on all such inputs. *We verified experimentally that the runtimes of the two versions are the same on these graphs.*

We evaluate the codes on two systems. **System 1** is a compute node with 2 NVIDIA RTX 3080 GPUs, each with 8704 CUDA cores and 12 GB global memory. It uses the PCIe bus for data transfers between the GPUs and the CPU. In addition, it does not support P2P accesses between the GPUs. We use nvcc version 12.6 with the "-O3 -arch=sm_86" flags to compile the codes. Due to the lower memory capacity, we only use the smaller inputs on this system.

System 2 has 4 NVIDIA V100 SMX2 GPUs, each with 5120 CUDA cores and 32 GB global memory. It uses NVLink, a high-speed interconnect that enables fast data transfer between the GPUs as well as between the GPUs and the CPU. It supports Peer-To-Peer (P2P) access, which enables direct transfers between the GPUs, without the need for the host to intervene. We use nvcc version 12.2 with the "-O3 -arch=sm_70" flags to compile the codes. On this system, we use inputs that result in a single-GPU memory requirement in the range of 20 GB to 52 GB. System 2 is a compute node of the Expanse system [28] at the San Diego Supercomputer Center, accessed via ACCESS [1].

We repeated each experiment 9 times and report the results corresponding to the execution yielding the median runtime. We list the results as throughput (in edges per second), which is the total number of edges in the graph divided by the measured runtime. Throughput is a higher-is-better metric and normalizes the execution times with respect to the graph size. We verified the correctness of all codes by checking the computed set of vertices for maximality and ensuring that adjacent vertices are not included in the set.

4.1 Input graphs

We use the 21 graphs listed in Table 1 to evaluate the codes. We need large graphs that do not fit in the global memory of a single GPU to demonstrate the use of MG-MIS. We created the graphs using generators from Graph500 [9, 23] and Indigo3 [16, 17]. We employed two generators to not only cover a good range of graph sizes but also a wide range of vertex degrees. Whereas the graphs generated by Graph500 have a large maximum degree, the ones generated by Indigo3 have a much smaller maximum degree. The number of vertices in these graphs ranges from a few million to over a billion, and the number of edges range from more than a hundred million to over 6 billion. The last column in Table 1 lists the memory requirement of each graph in ECL-MIS-UVM. Due to the large difference in GPU memory capacity between our two systems, we use the 10 smallest graphs as inputs in System 1 and the remaining graphs as inputs in System 2. We included a few graphs that fit in the global memory of a single GPU to also study MG-MIS's behavior on such small inputs. Note that the total amount of memory required for MG-MIS is a little higher than that of ECL-MIS-UVM (listed in Table 1) since MG-MIS uses buffers and an additional array to store the remote-neighbor-index values.

5 Results

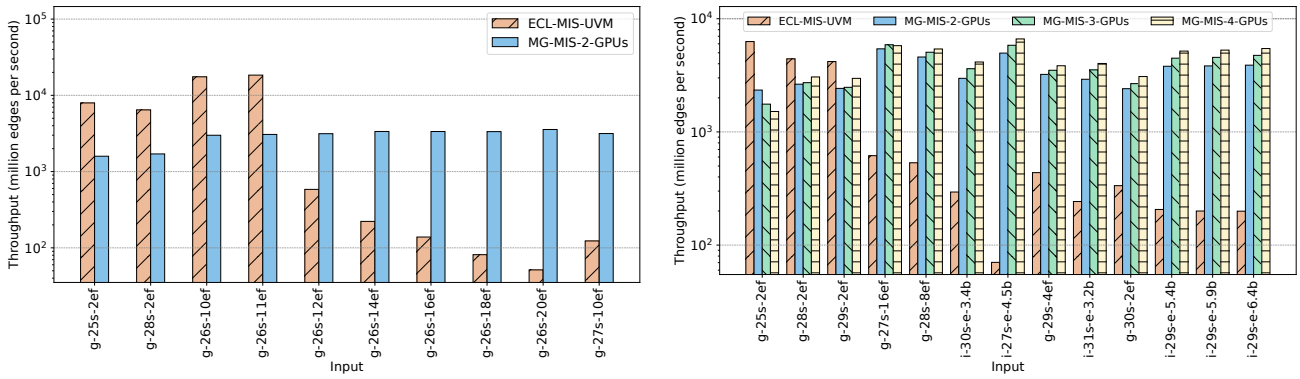
In this section, we first compare the performance of MG-MIS to the state-of-the-art single-GPU code [4, 5] that we extended with UVM support (ECL-MIS-UVM) on systems with and without NVLink and P2P support. Then, we analyze the effect of varying the buffer size in MG-MIS and study the actual buffer-size requirement for two large inputs. Next, we investigate the impact of storing the buffers in UVM as well as the effect of turning off P2P access in a system with NVLink on MG-MIS. We also evaluate the effect of prefetching data into the GPU's global memory in ECL-MIS-UVM. Lastly, we compare the sizes of the maximal independent sets computed by MG-MIS and ECL-MIS-UVM.

5.1 Overall performance

We compare the throughput of MG-MIS running on multiple GPUs to ECL-MIS-UVM running on one GPU in Figure 4.

Table 1: Information about the input graphs

input	Generator	|V|	|E|	d_{avg}	d_{max}	Memory Usage (GB)
g-25s-2ef	Graph500	33,554,432	133,516,554	3.98	115,060	1.28
g-28s-2ef	Graph500	268,435,456	1,070,786,720	3.99	419,426	10.23
g-26s-10ef	Graph500	67,108,864	1,322,545,710	19.71	694,301	10.42
g-26s-11ef	Graph500	67,108,864	1,453,274,642	21.66	748,870	11.39
g-26s-12ef	Graph500	67,108,864	1,583,792,718	23.60	802,382	12.36
g-26s-14ef	Graph500	67,108,864	1,844,210,488	27.48	905,020	14.30
g-26s-16ef	Graph500	67,108,864	2,103,844,848	31.35	1,003,347	16.24
g-26s-18ef	Graph500	67,108,864	2,362,730,378	35.21	1,097,929	18.17
g-26s-20ef	Graph500	67,108,864	2,620,933,926	39.05	1,188,533	20.09
g-27s-10ef	Graph500	134,217,728	2,652,246,640	19.76	1,081,828	20.89
g-29s-2ef	Graph500	536,870,912	2,142,721,430	3.99	643,804	20.46
g-27s-16ef	Graph500	134,217,728	4,223,282,274	31.47	1,572,126	32.59
g-28s-8ef	Graph500	268,435,456	4,259,434,956	15.87	1,399,843	33.99
g-29s-4ef	Graph500	536,870,912	4,278,119,462	7.97	1,196,042	36.37
g-30s-2ef	Graph500	1,073,741,824	4,287,290,382	3.99	987,912	40.94
i-30s-e-3.4b	Indigo3	1,073,741,824	3,393,687,134	3.16	4	34.28
i-27s-e-4.5b	Indigo3	134,217,728	4,478,382,004	33.37	36	34.49
i-31s-e-3.2b	Indigo3	1,600,000,000	3,200,000,000	2.00	3	37.25
i-29s-e-5.4b	Indigo3	536,870,912	5,368,709,070	10.00	34	44.50
i-29s-e-5.9b	Indigo3	536,870,912	5,904,900,130	11.00	35	48.49
i-29s-e-6.4b	Indigo3	536,870,912	6,442,450,866	12.00	38	52.50



(a) 2-GPU system with neither NVLink nor P2P support

(b) 4-GPU system with NVLink and P2P support

Figure 4: Overall throughput comparison of MG-MIS against ECL-MIS

The x-axis lists the inputs. The y-axis shows the throughput in million edges per second on a log scale. The two panels display results from two different systems for different inputs. The corresponding absolute runtimes are listed in Tables 3 and 4 at the end of the paper. Note that MG-MIS determines the amount of global memory left after loading the graph partition and its supporting data structures onto each GPU. Then, it divides this amount by the number of buffers needed to compute the size of each buffer and allocates them.

Figure 4(a) shows the results from a system with 2 GPUs, no NVLink, and no P2P support. Both GPUs have 12 GB of global memory. We observe that, on the first 4 inputs, ECL-MIS-UVM performs much better than MG-MIS because these inputs fit fully in the global memory of a single GPU. In such scenarios, using a multi-GPU solution leads to low performance due to the communication cost. On the remaining inputs, where the data size exceeds the single-GPU memory capacity of 12 GB, MG-MIS substantially outperforms ECL-MIS-UVM. On these six larger inputs, MG-MIS delivers a geometric-mean speedup of 22.88× over ECL-MIS-UVM.

We analyzed the UVM page transfers in ECL-MIS-UVM using NVIDIA Nsight Systems [25]. Column 2 of Table 2 shows the total number of GPU page faults that led to UVM page transfers in ECL-MIS-UVM. Column 3 shows the percentage

of the total data that was transferred due to UVM. Note that a UVM page fault typically moves 64 KB [2]. Column 4 shows the percentage of the total data that was transferred between devices in MG-MIS, and column 5 shows the total number of bulk data transfers. Since the 4 smallest inputs fit in a single GPU's global memory, they do not incur any UVM page faults in ECL-MIS-UVM (after prefetching); but since MG-MIS divides the total data among the 2 GPUs, data transfers are involved, leading to lower performance. On the remaining inputs, the amount of data transferred in ECL-MIS-UVM is not high compared to the amount of data transferred in MG-MIS. Still, MG-MIS performs much better since the number of times it performs data transfers is much lower, resulting in more efficient transfers. This shows that the frequency of data transfers plays an important role in the communication cost. We also note that the total volume of data transferred among devices in MG-MIS grows much more slowly with the input sizes, making it a scalable solution.

Table 2: Comparison of data transfers in ECL-MIS-UVM and MG-MIS on System 1

	ECL-MIS-UVM		MG-MIS	
input	# Page faults	Data moved (%)	Data moved (%)	# Transfers
graph-25s-2ef	0	0.00%	13.57%	213
graph-28s-2ef	0	0.00%	12.50%	262
graph-26s-10ef	0	0.00%	9.76%	452
graph-26s-11ef	0	0.00%	9.50%	464
graph-26s-12ef	6818	3.37%	9.27%	490
graph-26s-14ef	11472	4.89%	8.85%	530
graph-26s-16ef	18353	6.89%	8.49%	594
graph-26s-18ef	28543	9.58%	8.19%	656
graph-26s-20ef	51806	15.74%	7.92%	666
graph-27s-10ef	24533	7.17%	9.40%	468

Figure 4(b) shows the result from a system with 4 GPUs, NVLink, and P2P support. Each GPU has 32 GB of global memory, which is why we use larger inputs on this system. We separately show the performance of MG-MIS when using 2, 3, and 4 GPUs. The results follow a similar trend as with the previous system. Except for the first three graphs, which fit in 32 GB, MG-MIS is significantly faster. The geometric-mean speedups over the ten larger graphs show that MG-MIS is faster than ECL-MIS-UVM by 13.72×, 15.86×, and 17.73× when run with 2, 3, and 4 GPUs, respectively.

The performance of MG-MIS with 4 GPUs is consistently higher than with 2 GPUs. However, doubling the number of GPUs does not double the performance because more inter-GPU communication is needed with 4 GPUs. On the input *i-27s-e-4.5b*, we observe a huge drop in the performance of ECL-MIS-UVM. This is due to this graph having an average degree that is almost equal to the maximum degree. To determine the status of the neighbors, ECL-MIS-UVM frequently moves pages back and forth in the UVM, making the code slow.

For conciseness, the following subsections only present and discuss results for the system with 4 GPUs.

5.2 Effect of varying the buffer size

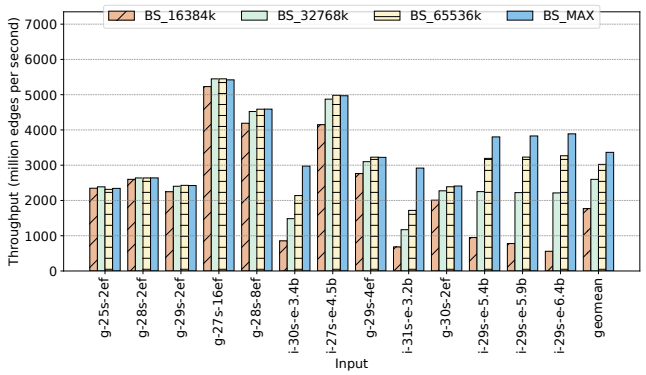
In this subsection, we show the impact of varying the buffer size in MG-MIS. Recall that, in a system with d GPUs, each GPU has $d-1$ *send buffers* and $d-1$ *receive buffers*. Hence, the total number of buffers in the system is $2 * d * (d-1)$. The reported buffer size indicates the size of one such buffer in number of words.

Figures 5(a) and 5(b) compare the throughput when the buffer size is set to 16384000, 32768000, 65536000, and the maximum possible words on 2 and 4 GPUs, respectively. Note that the y-axes use a linear scale. For the larger graphs, as we increase the buffer size, the performance improves. The improvement is more significant when MG-MIS is run with 2 devices. This is expected – The load per device is higher when there are only 2 devices and, hence, each device will need to transfer more data.

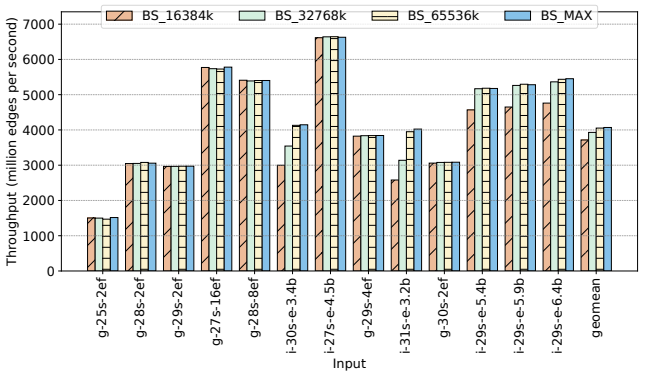
To study the performance behavior in more detail, we profiled MG-MIS on two large graphs and plotted the maximum buffer size required per iteration. Figure 6 shows the results. On the *i-29s-e-6.4b* input with 2 devices, the first four iterations require buffer sizes above 16384000 words (Figure 6(a)). In this case, increasing the buffer size leads to large performance gains. With 4 GPUs, only the first two iterations require buffer sizes above 16384000 words. Moreover, the buffer-size requirements are substantially lower than with 2 GPUs, explaining why the performance benefit of larger buffers is smaller in Figure 5(b) than in Figure 5(a). On the second input, *g-30s-2ef*, shown in Figure 6(b), we see that fewer iterations require large buffer sizes. Since the requirement is lower, increasing the buffer size has less impact on the performance with this graph. In fact, in the 4-device case, increasing the buffer size almost makes no difference at all.

5.3 Impact of unified virtual memory

This subsection investigates the impact of employing unified memory in MG-MIS. We used the system with NVLink for performing these experiments. Figure 7(a) shows how the throughputs change when the buffers are stored in UVM. In this case, the contents of one GPU's buffer is not explicitly copied over to another GPU. Instead, the pages containing the buffer data of the sender GPU are moved to the receiver GPU by UVM when the receiver accesses the data. Note that both code versions in this experiment use a fixed buffer size of 65536000 words since the notion of *MAX_BUFFER_SIZE* does not apply to the UVM version. Also, in the original version of MG-MIS, using this buffer size yields performance close to that of the *MAX_BUFFER_SIZE* version.

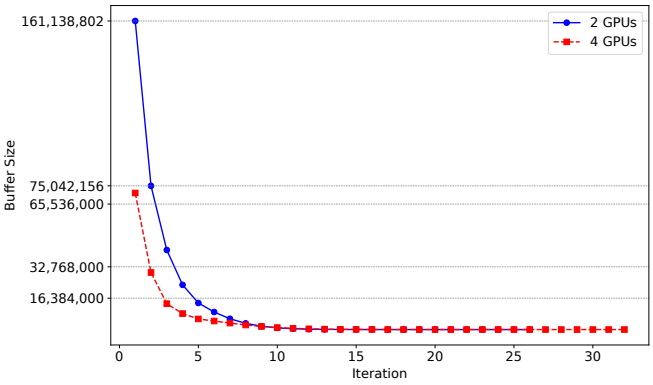


(a) MG-MIS on 2 GPUs

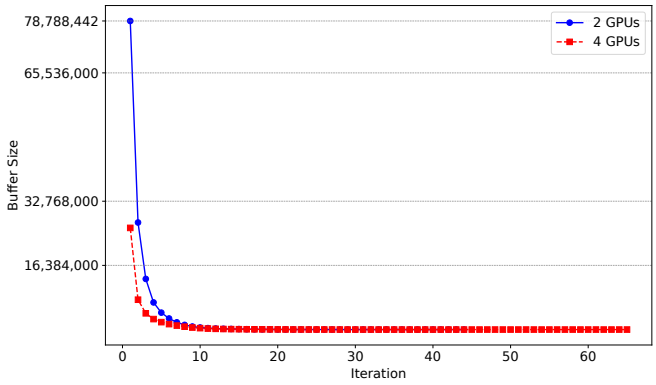


(b) MG-MIS on 4 GPUs

Figure 5: Effect of varying buffer-size in MG-MIS

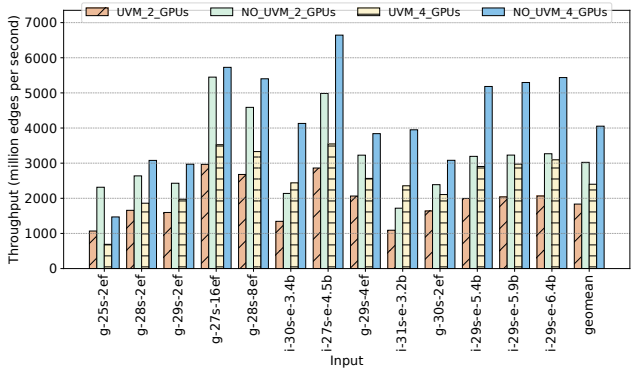


(a) On input *i-29s-e-6.4b*

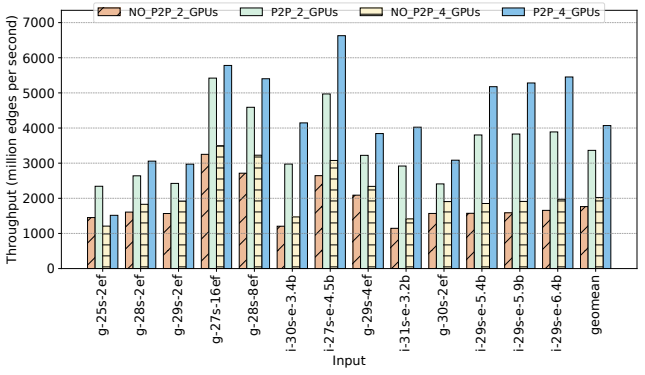


(b) On input *g-30s-2ef*

Figure 6: Maximum buffer size requirement per iteration in MG-MIS



(a) Effect of keeping buffers in UVM



(b) Performance with and without P2P access

Figure 7: Effect of using P2P access and Unified Virtual Memory (UVM) in MG-MIS on a system with NVLink

Clearly, the movement of unified memory pages across GPUs incurs significant overhead. The performance of the base version of MG-MIS, which explicitly controls the buffer transfers, is much higher than the version using UVM.

The UVM code, just like the no-UVM code, yields higher performance with 4 GPUs than with 2 GPUs, except on the smallest input, which has such a short runtime that the extra overhead incurred by using 2 more GPUs is not worth it. More surprisingly, even the no-UVM version with just 2 GPUs outperforms the UVM version with 4 GPU on all but two inputs. This clearly shows that using UVM is not beneficial in this setting.

5.4 Effect of peer-to-peer access

We also studied the impact of disabling Peer-To-Peer (P2P) accesses in MG-MIS. If there is no P2P access, the data transfers between GPUs have to go via the host's memory. Of course, this is much slower than direct transfers between GPUs, as we can see in Figure 7(b). With and without P2P access, the throughput of MG-MIS is higher with 4 than with 2 GPUs on all inputs except the smallest. Without P2P, using 4 instead of 2 GPUs does not increase performance much, whereas with P2P, using 4 GPUs significantly improves performance over using 2 GPUs. Moreover, MG-MIS with 2 GPUs on a P2P-enabled system yields much higher performance than it does with 4 GPUs on a system without P2P accesses.

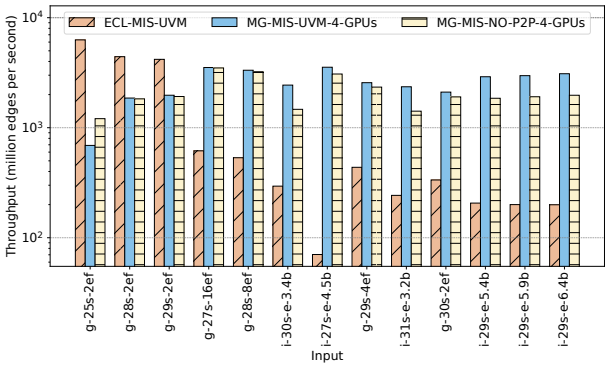


Figure 8: Comparison between ECL-MIS-UVM and MG-MIS running on 4 GPUs when (a) buffers are in UVM and (b) P2P access is disabled

In Figure 8, we contrast the performance of ECL-MIS-UVM with MG-MIS running on 4 GPUs with (a) the buffers allocated in the UVM and (b) P2P access disabled. Except for the three smallest inputs, both versions of MG-MIS are much faster than ECL-MIS-UVM. This demonstrates that the performance benefit of MG-MIS is not primarily due to using P2P accesses or not using UVM, but a result of our implementation that optimizes when and at what granularity

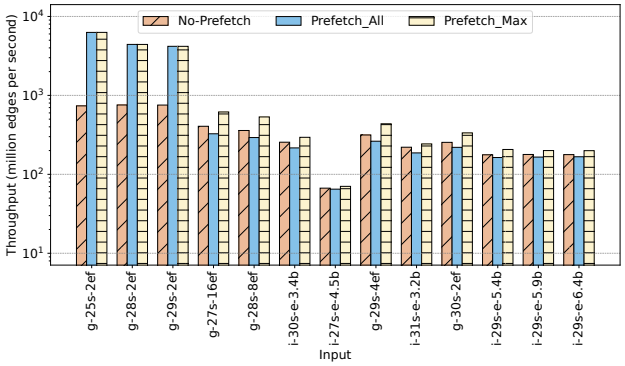


Figure 9: Effect of varying the amount of data prefetched into the GPU in ECL-MIS-UVM

to transfer data between the devices. It also shows that using P2P accesses is more important than avoiding UVM.

5.5 Effect of prefetching in ECL-MIS-UVM

Since our augmented version of ECL-MIS-UVM uses unified virtual memory to enable the processing of large graphs, it is important to minimize page movement. To this end, we studied the impact of prefetching graph data into the (single) GPU's global memory before the processing starts. Figure 9 shows the throughput comparison of ECL-MIS-UVM when the amount of data prefetched is varied. Note that the y-axis uses a log scale. For the first three inputs that fit in the global memory of a single GPU, the performance is significantly lower if no data is prefetched. The version *Prefetch_All* tries to load all graph data into the GPU. If the data size exceeds the global memory size, the earlier pages brought into the global memory will get replaced by later pages. This can lead to page faults during processing of the initial vertices. As a consequence, on all large graphs, this version performs worse than the *No_Prefetch* version. In the *Prefetch_Max* version, we compute how much data can fit in the global memory and prefetch only that amount. This ensures that the initial vertices' data will be present in the global memory for the threads to begin their processing. As expected, this version performs better than the other two versions on all large graphs. We used this best-performing version of ECL-MIS-UVM for comparison with MG-MIS in Section 5.1.

5.6 Size of the computed MIS

Figure 10 compares the sizes of the MISs computed by MG-MIS and ECL-MIS-UVM on each input. Note that all implementations of Luby's algorithm with the same priority assignment compute the same solution and, hence, yield the same result quality. Since ECL-MIS-UVM assigns priorities to vertices based on their degrees to boost the set sizes [5],

it produces somewhat larger sets than MG-MIS, which employs Luby's random priorities. However, the gap between the MIS sizes is small. It should be noted that both ECL-MIS-UVM and MG-MIS are deterministic and always compute the same set for a given input, independent of the hardware.

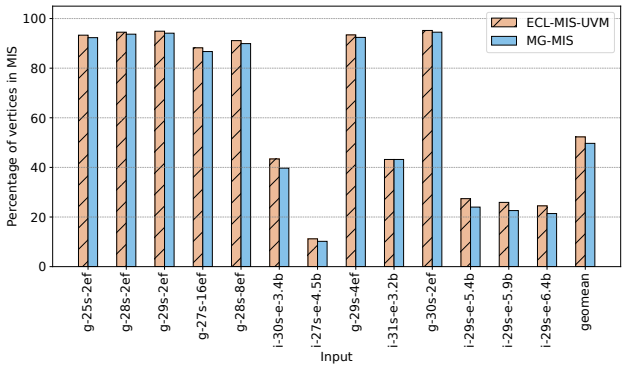


Figure 10: Comparison of the sizes of the MISs computed by ECL-MIS-UVM and MG-MIS

Table 3: Absolute runtimes in seconds on a 2-GPU system with neither NVLink nor P2P support

input	ECL-MIS	MG-MIS-2-GPUs
g-25s-2ef	0.017	0.084
g-28s-2ef	0.166	0.628
g-26s-10ef	0.076	0.441
g-26s-11ef	0.079	0.473
g-26s-12ef	2.710	0.504
g-26s-14ef	8.297	0.549
g-26s-16ef	15.174	0.627
g-26s-18ef	29.059	0.707
g-26s-20ef	51.013	0.735
g-27s-10ef	21.538	0.841

Table 4: Absolute runtimes in seconds on a 4-GPU system with both NVLink and P2P support

input	ECL-MIS	MG-MIS-2-GPUs	MG-MIS-4-GPUs
g-25s-2ef	0.021	0.057	0.088
g-28s-2ef	0.242	0.406	0.350
g-29s-2ef	0.512	0.884	0.721
g-27s-16ef	6.850	0.779	0.731
g-28s-8ef	7.985	0.928	0.788
i-30s-e-3.4b	11.527	1.141	0.819
i-27s-e-4.5b	63.610	0.901	0.676
g-29s-4ef	9.797	1.328	1.113
i-31s-e-3.2b	13.183	1.096	0.795
g-30s-2ef	12.799	1.779	1.390
i-29s-e-5.4b	25.996	1.412	1.037
i-29s-e-5.9b	29.518	1.542	1.118
i-29s-e-6.4b	32.309	1.657	1.182

6 Conclusions

Computing a maximal independent set (MIS) of a graph is a frequently needed step in many computations. With the ever-increasing data sizes, many applications need to compute MISs in large graphs. Among the several parallel algorithms for MIS computation, ECL-MIS, a single-GPU algorithm, is currently the fastest. Though ECL-MIS is very efficient in processing graphs that fit in the global memory of a single GPU, it is not designed to process larger graphs. We enabled it to process larger graphs by exploiting unified virtual memory, but this approach is relatively slow.

As a remedy, we propose MG-MIS, the first multi-GPU algorithm for computing MISs. It divides the computation among the available GPUs and incorporates several algorithmic optimizations to reduce the frequency as well as the volume of inter-GPU communication. Moreover, it separates computation and communication into mutually exclusive phases and performs inter-GPU communications only in bulk to improve performance. By using a deterministic hash function that requires only the ID of a vertex to compute its priority, it is able to completely avoid inter-GPU communication when it needs the priority of a remote vertex.

Our evaluation of MG-MIS on a system with 2 RTX 3080 GPUs (each with 12 GB of global memory, no NVLink, and no P2P support) shows a geometric-mean performance improvement of 22.88× over ECL-MIS-UVM on graphs that require more than 12 GB memory. On a system with 4 V100 GPUs (each with 32 GB of global memory, NVLink, and P2P support), MG-MIS delivers a geometric-mean performance improvement of 13.72× and 17.73× when run on 2 and 4 GPUs, respectively, over ECL-MIS-UVM on graphs that require more than 32 GB of memory. In all these cases, the set sizes of MG-MIS are close to those of ECL-MIS-UVM.

Based on our results, we conclude that whenever an input fits completely in the global memory of a single GPU, it is best to use the single-GPU ECL-MIS code to compute a MIS. Using a multi-GPU algorithm in such scenarios leads to low performance since it involves slow inter-GPU communication. However, when the graph data does not fit in the memory of a single GPU, it is beneficial to use our multi-GPU algorithm, MG-MIS, that minimizes inter-GPU communication. Extending MG-MIS to support multi-node distributed systems and incorporating support for dynamic graph processing are exciting directions for future work.

Acknowledgments

We thank the reviewers for their valuable feedback. This work has been supported by the National Science Foundation under Award #1955367. We acknowledge the use of computational resources from Expanse at the San Diego Supercomputer Center through ACCESS allocation CIS240129.

References

[1] ACCESS. 2025. Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS). https://access-ci.org. National Science Foundation.

[2] Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory. *ACM Trans. Archit. Code Optim.* 21, 1, Article 14 (Jan. 2024), 24 pages. https://doi.org/10.1145/3632953

[3] Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The Problem-Based Benchmark Suite (PBBS), v2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 445–447. https://doi.org/10.1145/3503221.3508422

[4] Martin Burtscher and Sindhu Devale. 2018. ECL-MIS. https://userweb.cs.txstate.edu/~burtscher/research/ECL-MIS/. Accessed: 2025-02-23.

[5] Martin Burtscher, Sindhu Devale, Sahar Azimi, Jayadharini Jaiganesh, and Evan Powers. 2018. A high-quality and fast maximal independent set implementation for GPUs. *ACM Transactions on Parallel Computing (TOPC)* 5, 2 (2018), 1–27.

[6] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, IEEE, USA, 185–195.

[7] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. CUSP: Generic parallel algorithms for sparse matrix and graph computations.

[8] Ilya Gertsbakh and Helman I Stern. 1978. Minimal resources for fixed and variable job schedules. *Operations Research* 26, 1 (1978), 68–85.

[9] Graph500. 2017. Graph500 Benchmark Version 3.0.0. https://github.com/graph500/graph500. Accessed: 2025-02-23.

[10] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. NVIDIA Developer Blog.

[11] Uwe Hobohm, Michael Scharf, Reinhard Schneider, and Chris Sander. 1992. Selection of representative protein data sets. *Protein Science* 1, 3 (1992), 409–417.

[12] Tomohiro Imanaga, Koji Nakano, Masaki Tao, Ryota Yasudo, Yasuaki Ito, Yuya Kawamata, Ryota Katsuki, Yusuke Tabata, Takashi Yazane, and Kenichiro Hamano. 2020. Efficient GPU implementation for solving the maximum independent set problem. In *2020 Eighth International Symposium on Computing and Networking (CANDAR)*. IEEE, USA, 29–38.

[13] Tomohiro Imanaga, Koji Nakano, Ryota Yasudo, Yasuaki Ito, Yuya Kawamata, Ryota Katsuki, Yusuke Tabata, Takashi Yazane, and Kenichiro Hamano. 2023. Simple iterative trial search for the maximum independent set problem optimized for the GPUs. *Concurrency and Computation: Practice and Experience* 35, 14 (2023), e6681.

[14] Richard M Karp and Avi Wigderson. 1985. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM (JACM)* 32, 4 (1985), 762–773.

[15] Brian Kelley and Sivasankaran Rajamanickam. 2022. Parallel, portable algorithms for distance-2 maximal independent set and graph coarsening. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, USA, 280–290.

[16] Yiqian Liu, Noushin Azami, Avery Vanausdal, and Martin Burtscher. 2024. Indigo3. https://github.com/burtscher/Indigo3Suite. Accessed: 2025-02-23.

[17] Yiqian Liu, Noushin Azami, Avery Vanausdal, and Martin Burtscher. 2024. Indigo3: A parallel graph analytics benchmark suite for exploring implementation styles and common bugs. *ACM Transactions on Parallel Computing* 11, 3 (2024), 1–29.

[18] Michael Luby. 1985. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. Association for Computing Machinery, New York, NY, USA, 1–10.

[19] Michael Luby. 1986. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.* 15, 4 (1986), 1036–1053. https://doi.org/10.1137/0215074

[20] Leran Ma, Ke Chen, and Mingfu Shao. 2023. On the Maximal Independent Sets of k-mers with the Edit Distance. In *Proceedings of the 14th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, New York, NY, USA, 1–6.

[21] Anju Mongandampulath Akathoott, Benila Virgin Jerald Xavier, and Martin Burtscher. 2025. MG-MIS. https://userweb.cs.txstate.edu/~burtscher/research/MG-MIS/. Accessed: 2025-04-30.

[22] Anju Mongandampulath Akathoott, Benila Virgin Jerald Xavier, and Martin Burtscher. 2025. MG-MIS. https://github.com/burtscher/MG-MIS/. Accessed: 2025-04-30.

[23] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* 19, 45-74 (2010), 22.

[24] NVIDIA Corporation. 2024. *CUDA C++ Programming Guide*. NVIDIA. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[25] NVIDIA Corporation. 2025. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2025-04-29.

[26] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, USA, 1–19.

[27] Richard L Rudell. 1989. *Logic synthesis for VLSI design*. University of California, Berkeley, California, USA.

[28] San Diego Supercomputer Center. 2025. Expanse. https://doi.org/10.1145/3437359.3465588. University of California San Diego. Service.

[29] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, New York, NY, USA, 135–146.

[30] Moisés Silva-Muñoz, Carlos Contreras-Bolton, Carlos Rey, and Victor Parada. 2023. Automatic generation of a hybrid algorithm for the maximum independent set problem using genetic programming. *Applied Soft Computing* 144 (2023), 110474.

[31] Xubo Wang, Dong Wen, Wenjie Zhang, Ying Zhang, and Lu Qin. 2023. Distributed Near-Maximum Independent Set Maintenance over Large-scale Dynamic Graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, USA, 2538–2550.