MG-αGCD: Accelerating Graph Community Detection on Multi-GPU Platforms

Shuai Yang
University of Chinese Academy of Sciences
Beijing, China
Institute of Software, Chinese Academy of Sciences
Beijing, China
yangshuai2022@iscas.ac.cn

Changyou Zhang
University of Chinese Academy of Sciences
Beijing, China
Institute of Software, Chinese Academy of Sciences
Beijing, China
changyou@iscas.ac.cn

Abstract

Graph community detection is widely applied in fields such as genetic engineering and social network analysis. As the scale of input graphs continues to grow and multi-GPU platforms become increasingly prevalent, utilizing the storage capacity and computational power of these platforms to scale graph community detection algorithms has become more feasible. However, existing multi-GPU graph community detection methods are constrained by the traditional CPU-dominated communication model, and fail to simultaneously account for the irregular sparse memory access patterns and the latency disparities between local and remote communication. Consequently, they do not fully exploit modern high-speed GPU interconnect technologies. Furthermore, while current solutions propose various strategies to mitigate the decline in clustering quality during parallelization, these approaches are often inefficient or compromise clustering quality. Finally, to address the high memory overhead of the algorithms, existing Multi-GPU solutions extend the graph size limit at the cost of reduced performance.

To address these challenges, we propose MG-αGCD, a novel graph community detection algorithm designed for multi-GPU platforms. First, MG-αGCD introduces a loading balancing and latency-aware computation-communication pipeline that effectively mitigates the overhead of high-latency remote communication. Second, MG-αGCD incorporates a bidirectional probing heuristic to enhance execution efficiency while outperforming existing methods in clustering quality. Lastly, MG-αGCD employs a two-phase graph coarsening algorithm consisting of a symbolic phase and a numeric phase, which significantly reduces GPU peak memory usage and minimizes data transfers between CPU and GPU. Extensive experiments demonstrate that MG-αGCD achieves average speedups of 34.60x, 30.75x, and 11.31x compared to the state-of-the-art solutions Grappolo, nido, and cuGraph, respectively.

CCS Concepts

• **Computing methodologies** → **Parallel algorithms**; • **Theory of computation** → *Graph algorithms analysis*; • **Software and its engineering** → *Massively parallel systems.*

Keywords

Graph Community Detection, Multiple GPUs, Scalability

ACM Reference Format:
Shuai Yang and Changyou Zhang. 2025. MG-αGCD: Accelerating Graph Community Detection on Multi-GPU Platforms. 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/10.1145/3721145.3725753

1 Introduction

Communities in networks identify entities with strong interactions. For example, in social networks, communities can be groups of closely connected users. Early research introduced many community detection (or graph clustering) methods, which primarily included approaches based on graph partitioning[21], spectral analysis[22], hierarchical structures[18], and density[1]. Later studies leveraged the rich information embedded in real-world graphs, such as edge weights, to develop more advanced methods. Among these, the modularity-based heuristic approach, known as the Louvain method[3], uses modularity[23] to measure the cohesion within communities. Due to its efficiency and its ability to reveal hierarchical community structures, the Louvain method has been widely adopted. Since real-world graphs are often large in scale, developing parallel community detection algorithms[9, 19, 20, 28, 33, 34] capable of efficiently processing large-scale graphs holds significant research and practical value.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725753

Although some efforts[6, 25] have been made to leverage the computational, storage, and communication capabilities of multi-GPU platforms (e.g., NVIDIA DGX) to scale up the performance of the Louvain method, the runtime performance of existing solutions has not scaled proportionally with the capabilities of these platforms. Several factors contribute to this issue. Firstly, existing solutions are limited by traditional communication paradigms and fail to adequately account for the architecture of multi-GPU platforms and the communication patterns of the algorithms. Despite the advantages of modern high-speed GPU interconnect technologies (e.g., NVLink), their scalability on real-world graphs remains below expectations. Secondly, the Louvain algorithm often relies on carefully designed heuristics to mitigate the decline in clustering quality caused by parallelization. However, these heuristics[6, 19] frequently introduce additional computational overhead or compromise solution quality to enhance execution efficiency, highlighting the trade-off between execution efficiency and solution quality[6]. Thirdly, the Louvain algorithm's high peak memory cost presents significant challenges in developing memory-efficient approaches. Existing solutions often neglect this issue or offload the high memory-consuming processes to the CPU, resulting in performance degradation.

To address these challenges, we propose a novel algorithm design for accelerating Louvain graph community detection on multi-GPU platforms, called MG-αGCD. Firstly, MG-αGCD introduces a one-sided communication paradigm and designs a well-structured computation-communication pipeline that effectively balances the workload across GPU logical processing units while accounting for inter-GPU communication latency differences. Secondly, MG-αGCD incorporates a bidirectional probing heuristic, introducing novel rules for more flexible control of movement directions. This addresses parallelization challenges and significantly enhances execution efficiency. Lastly, MG-αGCD proposes a fully multi-GPU two-phase graph coarsening algorithm that separates memory requirement estimation from graph construction.

We summarize our main contributions as follows:

- We design a well-structured computation-communication pipeline that effectively hide the costly inter-GPU communication.
- We incorporate a bidirectional probing heuristic, which enhances both execution efficiency and clustering solution quality.
- We propose a two-phase graph coarsening algorithm that significantly reduces peak memory cost and minimizes CPU-GPU data transfers.
- Comprehensive experiments demonstrate that MG-αGCD outperforms Grappolo, nido, and cuGraph, achieving average speedups of 34.60x, 30.75x, and 11.31x, respectively. Furthermore, MG-αGCD exhibits better scalability, delivers higher clustering quality, and significantly reduces storage costs.

2 Background

Given an undirected weighted graph $G(V, E, \omega)$ where V is the set of vertices, E is the set of edges, and ω represents the set of edge weights, graph community detection aims to partition the graph into a set of k communities, denoted as $P = \{C_1, C_2, \ldots, C_k\}$. Modularity[23] (denoted as Q) is the metric used to evaluate the quality of the partition P, and its definition is as follows:

$$Q = \sum_{i \in V} \frac{e_{i \to C(i)}}{2m} - \sum_{C_j \in P} \left(\frac{a_{C_j}}{2m} \right)^2 \tag{1}$$

where $e_{i \to C(i)}$ is the sum of the edge weights between vertex i and other vertices within its community $C(i)$, m denotes the total sum of all edge weights in graph G. The quantity $a_{(C_j)}$ represents the sum of the weights of all edges connected to the vertices in community C_j, defined as $a_{C_j} = \sum_{i \in C_j} k_i$, where k_i is the sum of the edge weights in the adjacency list of vertex i.

The Louvain method[3], a modularity-based derivative algorithm, is a multi-phase iterative heuristic algorithm. During each iteration, communities are progressively built by moving a vertex to a adjacent community to maximize the modularity gain. For a vertex i, originally belonging to community $C(i)$, the modularity gain (denoted as ΔQ) from moving it to a adjacent community C_j is defined as:

$$\Delta Q_{i \to C_j} = \frac{e_{i \to C_j} - e_{i \to C(i) \setminus \{i\}}}{m} + k_i \frac{a_{C(i) \setminus \{i\}} - a_{C_j}}{m^2} \tag{2}$$

where $e_{i \to C_j}$ represents the sum of the edge weights between vertex i and all vertices in the adjacent community C_j. The sum of the edge weights between vertex i and other vertices within its community, excluding itself, denoted as $e_{i \to C(i) \setminus \{i\}}$. Finally, $a_{C(i) \setminus \{i\}}$ represents the sum of the weights of all edges connected to the vertices in community $C(i)$, excluding vertex i. The Louvain method consists of two main steps: modularity optimization and graph coarsening:

Modularity Optimization. This step involves three parts: 1) *Vertex Movement*. For each vertex i in the graph, the algorithm iterates over all adjacent vertices of vertex i and calculates the modularity gain from moving vertex i to the community of its neighbor j, denoted as C_j, using Eq.(2). Vertex i is then moved to the adjacent community that yields the maximum modularity gain. 2) *Community Information Update*. After vertex movements are completed, the information

of all affected communities is updated. 3) *Modularity Calculation.* After all vertex movements and community updates are completed, the modularity of the current community assignments is recalculated using Eq.(1). The modularity optimization process is repeated until the modularity converges within the inner optimization loop.

Graph Coarsening. In this step, vertices within the same community are collapsed into a new vertex, with the weight of the new vertex being the sum of the weights of all vertices in the community. Multiple edges between communities are collapsed into a single new edge, with the weight of this new edge being the sum of all original edge weights. The coarsened graph then undergoes the modularity optimization process again, and the process is repeated until the modularity converges in the outer optimization loop.

3 Motivation

We propose various levels of optimization for the multi-GPU Louvain method to leverage the inherent communication, computation, and storage advantages of multi-GPU platforms. These optimizations are primarily derived from our three key observations.

Multi-GPU Communication Support: Our first observation highlights recent advancements in inter-GPU communication technologies and their programming support. NVLink interconnect technology, known for its lower latency and higher bandwidth[4, 8, 16, 17], is widely utilized in fields such as deep learning[13, 14, 32] and high-performance simulations[10]. However, replacing PCIe with NVLink does not directly enhance application performance in these fields[16]. This limitation arises from the CPU-dominated communication paradigm[4, 12, 16]. The NVSHMEM communication library[26] provides one-sided and collective communication interfaces across different granularities, initiated by various GPU logical units (e.g., threads, warps, blocks). This approach reduces CPU interference in communication pathways, thereby increasing GPU utilization and offering greater programming flexibility.

Despite the potential of NVLink and NVSHMEM technologies to improve the scalability of general multi-GPU applications, significant challenges remain for practical implementations, particularly in graph clustering application focused on in this paper. The absence of communication optimization strategies tailored to the memory access pattern and latency, communication granularity, and the structural characteristics of real-world graphs hinders achieving optimal scalability.

Parallelization Challenge: Our second observation is that parallelizing the Louvain method may delay convergence and reduce clustering solution quality[19]. Concurrent community movements can lead to swap and local maxima issues, which are exacerbated in GPU environments. To enhance parallelism, GPU-based Louvain methods introduce grid-wide synchronization between vertex movement and community information update. The movement results of each vertex are first written to a cache, meaning that all threads make decisions using outdated information without knowledge of the community assignments of other threads. Existing solutions have sought to address the parallelization challenges of Louvain by employing methods such as graph coloring[19], minimum labeling[19], and batching[6]. However, they still exhibit a trade-off between performance and solution quality[6], providing an opportunity for algorithmic optimization to enhance execution efficiency while maintaining solution quality.

High Storage Requirement: The third observation is that GPU-based Louvain methods exhibit significant peak memory overhead. This peak memory requirement arises from the graph coarsening step, which necessitates intermediate temporary space to simultaneously store both the original and coarsened graphs. Additionally, because the size of the coarsened graph cannot be determined before its construction, over-allocating temporary space equal to the size of the original graph exacerbates storage pressure. Inspired by SpGEMM[24, 27, 35], we propose a two-phase graph coarsening algorithm, comprising a symbolic phase and a numeric phase, which separately calculate memory requirements and construct the graph, distributing the storage requirements evenly across multiple GPUs.

These observations and insights have driven us to design MG-αGCD, a high-performance multi-GPU Louvain method. In the subsequent sections, we will describe the optimizations applied to the Louvain method at three different levels and evaluate the superiority of our approach through comprehensive experiments in comparison to previous work.

4 Load Balancing & Latency-Aware Pipeline

The design of our pipeline primarily faces two challenges: 1) *How to lay out graph data* so that the input graph data of Louvain and the memory of multi-GPU platforms can be carefully matched to effectively address the trade-off between storage resource utilization and communication volume, which in turn influences the execution of the pipeline. 2) *How to efficiently extract pipeline workloads* from unpredictable and irregular remote communications and local computations, and design the mapping of the pipeline onto GPU processing units to maximize its execution efficiency. MG-αGCD addresses these challenges through Graph Data Management and Load Balancing & Latency-aware Pipeline Construction and Mapping.

4.1 Graph Data Management

The NVSHMEM remote data access requires the allocation of a symmetric address space of the same size[26], restricting remote access to only those symmetric data objects located within this space. The placement of data in different memory spaces (*shared* or *private*) determines its remote accessibility, and the performance characteristics of these memory spaces can vary significantly. Therefore, our placement strategy encompasses two aspects: how to partition the graph data across GPUs and how to determine the graph data placement within different GPU memory spaces.

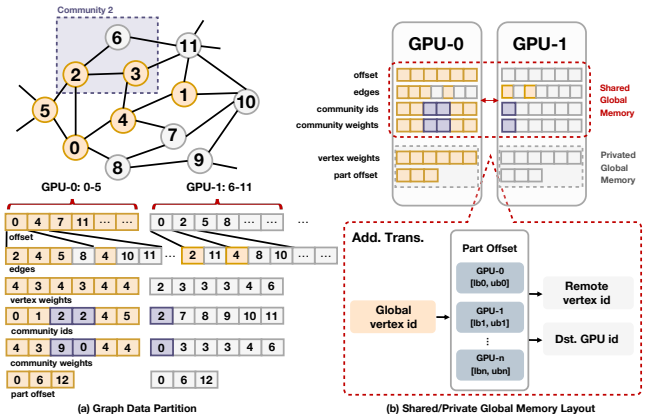

Figure 1: (a) Graph data Partition. The vertices within the purple dashed box belong to Community 2, while the yellow vertices and gray vertices are allocated to two different GPUs. (b) Shared/Private global memory layout. "GPUi-i" is the vertex id range [lowerbound, uppperbound] on the i-th GPU.

First, to fully utilize GPU memory and reduce peak memory costs, MG-αGCD stores the graph structure in Compressed Sparse Row (CSR) format and partitions it across multiple GPUs while balancing the number of edges, as shown in Figure 1(a). The *offset* array in the CSR format is renumbered starting from zero on each GPU. The storage structure of the *edge weights* array is identical to that of *edges* and is therefore omitted from the figure. In addition to the graph structure data, which constitutes the dominant portion of storage consumption, MG-αGCD further stores clustering-related data, including *vertex weights*, *community IDs*, and *community weights* across the memory of each GPU. *Vertex weights* is initialized as the sum of the weights of their adjacent edges, *community IDs* is initialized to be the same as *vertex IDs*, and *community weights* is initialized to be the same as *vertex weights*. Furthermore, each GPU stores a *vertex partition offset* array of length *n+1* (where *n* represents the number of GPUs), which is used to obtain the *GPU ID* where a vertex is located and the *local vertex ID* on that GPU.

To further optimize memory management, we place data in distinct GPU memory spaces (Figure 1(b)). Specifically, both *community IDs* and *community weights* are allocated in *shared* global memory, which ensures consistent access latencies across all GPUs. This uniform latency is essential for achieving a balanced distribution of remote access workloads among the GPUs. Additionally, to meet the access demands of graph structure data on remote GPUs during the graph coarsening process, the graph structure data is stored in *shared* global memory space. To retrieve the addresses of target vertices/communities on remote GPUs, we utilize the *vertex partition offset* array for address translation. The *vertex partition offset* and *vertex weights* arrays are placed in *private* global memory, which is only accessible to the local GPU and offers low-latency access. The data layout necessitates remote communication, while the disparity in memory access latency across different processors poses a significant challenge in designing strategies to hide communication.

4.2 Pipeline Construction & Mapping

4.2.1 Problem Description and Baseline. Consider m community weight update tasks distributed across P processors. For each processor, the total cost of the community information update process can be approximated as:

$$T_c = \frac{m}{P}\beta t_A^r + \frac{m}{P}(1-\beta)t_A^l \tag{3}$$

where β denotes the uncertain proportion of remote atomic operations (typically high), t_A^r denotes the cost of remote atomic operations for updating community weights on remote GPUs, and t_A^l denotes the cost of local atomic operations, with t_A^r being significantly greater than t_A^l.

Figure 1(a) illustrates a case of community information updating, where vertices $v3$ and $v6$ move to the community of $v2$ ($C2$). Because the information for $C2$ and $v6$ resides in different GPUs, $v6$ must perform a costly remote atomicAdd operation to update the weight of $C2$.

The irregularity, sparsity, and heterogeneity of the communication pattern make updating community information the most expensive communication operation, while also posing significant challenges in balancing the workload across multiple GPUs and hiding communication latency.

4.2.2 Method Workflow. To address these challenges, we construct and optimize a computation-communication pipeline within a GPU kernel, The workflow of our pipeline design can be summarized as follows:

Step 1: Irregularity-to-Regularity Transformation. This step aims to convert irregular and costly communication into regular, lower-latency communication. To achieve this,

we designed a local community weight increment cache that consolidates the irregular remote/local atomic operations on community weights into local atomic operations on the cache. The GPU logical processing units then initiate NVSHMEM remote/local access to a contiguous address space on the community weight increment cache, followed by accumulating the increments into the local community weight. At this point, the total cost of community information updates is given by:

$$T_c = \frac{m}{P} t_A^l + \frac{n}{P}\left(\frac{P-1}{P} t_l^r + \frac{1}{P} t_l^l + t_a\right) \tag{4}$$

where n denotes the number of communities, $\frac{P-1}{P}$ denotes the proportion of *remote accesses*, $\frac{1}{P}$ denotes the proportion of *local accesses*, and t_l^r, t_l^l, and t_a denote the costs of *remote access*, *local access*, and *local accumulation*, respectively.

Compared to Eq.(3), this step eliminates the costly remote atomic operations and identifies the proportions of remote and local accesses. These features serve as the robust foundation for designing a well-structured pipeline.

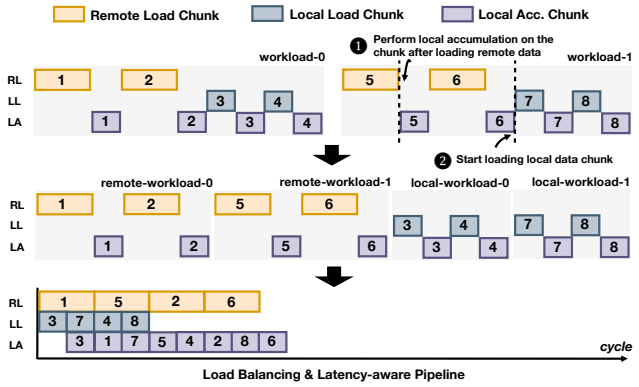

Figure 2: Pipeline construction and mapping. Each gray box represents a workload handled by a GPU logical processing unit, and is composed of the same number of chunks. The workload and pipeline here are located on one single GPU within a multi-GPU system.

Step 2: Load Balancing & Latency-Aware Pipeline Construction. To further reduce communication overhead, our insight is that NVSHMEM remote/local access can overlap with local accumulation processes. Thus, we propose a three-stage pipeline design. The three stages include *Remote Load* (**RL**), *Local Load* (**LL**), and *Local Accumulate* (**LA**). Each community weight aggregation requires two stages. The key to improving pipeline efficiency is balancing the workload among the massive processing units of the GPUs (e.g., SMs). To achieve this, we divide the communities within the local range into chunks, ensuring a balanced number of communities in each chunk. Subsequently, we mix remote and local community aggregations into a single workload, where workloads consist of an equal number of RL-LA or LL-LA chunk pairs (Figure 2). Each workload is processed by one GPU processing unit. Note that in the software pipeline, chunks from different workloads can overlap since they are processed by different GPU units, whereas chunks within the same workload must maintain their original relative execution order. Moreover, according to [16], the latency of remote access is approximately twice that of local access. To further improve pipeline efficiency, we categorize the workloads into two types: local workloads with lower latency, and remote workloads with higher latency. The potential execution pipeline illustrates that this construction effectively overlaps three stages, thereby reducing the overall pipeline execution time.

Step 3: Warp-Based Pipeline Mapping. Another key factor influencing the maximization of pipeline execution efficiency is how to map the constructed pipeline to the logical processing units (i.e., thread/warp/block). We choose warps as the basic unit of work and set the number of communities in each chunk to the number of threads in a warp (i.e., 32). This decision is based on two main observations. First, assigning a single thread to handle different workloads can lead to warp divergence, negatively impacting GPU execution efficiency. Additionally, NVSHMEM remote/local memory accesses initiated by threads within the same warp can be coalesced into a single memory transaction, thereby reducing access latency. For NVSHMEM remote access, temporary storage of remote data is required in local scratchpad memory. To reduce the frequent global memory access, we allocate a high-speed shared memory space of size $sizeof(double) \times 32$ for each warp.

5 The Bi-Probing Heuristic

In this section, we present the design of our Bi-Probing Modularity Optimization strategy, as well as the efficient implementation and optimization tailored for the scale-free characteristics of real-world graphs. Moreover, we extend this approach to a multi-GPU platform utilizing NVSHMEM as the communication backend.

5.1 Main idea

Recall that, community detection algorithms face significant challenges with swap and local maxima issues during parallelization[19]. *Graph coloring heuristic* introduces substantial computational overhead, while *batching heuristic* incur frequent synchronization overheads and is only moderately effective in mitigating parallelization challenges. The *minimum label heuristic* restricts vertices to communities with ids smaller than their current community, effectively

resolving the swap problem. However, this unidirectional constraint may prevent vertices from moving to communities with the highest modularity gain and reduces the frequency of vertex movements. In the worst-case scenario, if the community id yielding the highest modularity gain consistently exceeds the current community id, vertices may fail to move or face delays in doing so. Thus, it would be beneficial if we allowing vertices to move to any adjacent community. However, this introduces another challenge: *How can we address the parallelization issue under this condition?*

5.2 Global & Local Bi-Probing

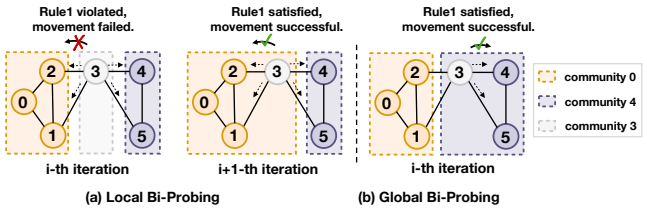

Figure 3: Local and global Bi-Probing. The vertices within the dashed boxes of different colors belong to different communities.

We now present a heuristic algorithm, denoted as Bi-Probing, that dynamically probes the potential for vertex movement in both directions: increasing and decreasing community ids, while addressing parallelization challenges. Let $\ell(v)$ denote the id of the community to which vertex v belongs, we select the probing direction and the destination community according to the following rules:

Rule1(v, i, C, ℓ)**:** During iteration i, the movement direction of vertex v is restricted as follows: 1) *If i is odd, v can only move to an adjacent community $C(v')$ where $\ell(v') > \ell(v)$*; 2) *If i is even, v can only move to an adjacent community $C(v')$ where $\ell(v') < \ell(v)$.*

Rule2(v, i, C, ℓ)**:** During iteration i, when vertex v encounters multiple adjacent communities $C(v')$ with identical modularity gain, the following priority rules apply: 1) *If i is odd, v move to the community $C(v')$ having the largest id*; 2) *If i is even, v move to the community $C(v')$ having the smallest id.*

Based on the differences in the scope of Rule1(v, i, C, ℓ), we further propose two variants: Bi-Probing(Lo) and Bi-Probing(Gl).

5.2.1 Bi-Probing(Lo). For each vertex v, we first calculate the modularity gain associated with moving to all adjacent communities and select the direction with the highest gain. Rule1(v, i, C, ℓ) is then applied, which states that a vertex can only move in this direction if the condition is satisfied. If this condition is not satisfied in the current iteration, the vertex remains stationary, and the direction of the constraint is alternated in the next iteration. In this variant, *vertices consistently move toward the direction of maximum modularity gain within two iterations.* For example, in Figure 3(a), during iteration i, $v3$ computes its modularity gains for moving to $C0$ and $C4$, identifying $C0$ as offering the highest gain. However, since the constraint requires vertices to move only toward communities with larger labels in that iteration, $v3$ does not move. In the following iteration $(i + 1)$, after the constraint direction is alternated, $v3$ successfully moves to $C0$.

Algorithm 1: Bi-Probing Heuristic

input : Bin (*bin*), edge weight (*eWeight*), vertex weight (*vtxWeight*), community weight (*comWeight*), community id of vertices (*comId*), total edge weight (m), constraint direction (*max_min*).
output : New community id array (*newComId*)

```
1  foreach vertex in bin.vertices do
2      ΔQ ← 0;
3      srcComId ← comId[vertex.id];
4      dstComId ← 0;
5      k_i ← vtxWeight[vertex.id];
6      ac_i ← nvshmem_g(comWeight[vertex.id]) - k_i;
7      foreach neighbor in vertex.neighbors do
8          w_ij ← eWeight[neighbor.id];
9          dstComId ← nvshmem_g(comId[neighbor.id]);
10         hash ← (key * SCAL) % tableSize;
11         /* Rule1 of Bi-Probing(Gl).                    */
12         if (dstComId > srcComId) = max_min then
13             while true do
14                 Try to insert into table, and accumulate
                    w_ij to obtain e_i c_j.
15         else if dstComId = srcComId then
16             e_i c_i ← e_i c_i + w_ij;
17     foreach Nonzero entry in hash table do
18         ac_j ← nvshmem_g(comWeight[entry.key]);
19         ΔQ_tmp ← Eq.(2)(e_i c_j, e_i c_i, ac_i, ac_j, k_i, m);
20         if ΔQ_tmp > ΔQ then
21             dstComId ← srcComId;
22             ΔQ ← ΔQ_tmp;
23         /* Rule2.                                      */
24         if ΔQ_tmp = ΔQ and (dstComId > srcComId =
             max_min) then
25             dstComId ← srcComId;
26             ΔQ ← ΔQ_tmp;
27     /* Rule1 of Bi-Probing(Lo).                        */
28     if (dstComId > srcComId) = max_min then
29         newComId[vertex.id] ← dstComId;
```

5.2.2 Bi-Probing(Gl). For each vertex v, we first apply Rule1 to all adjacent directions. Only directions that satisfy Rule1 are considered for potential movement (i.e., those that contribute to the calculation of modularity gain). In this variant, *vertices always move during each iteration*, even if the chosen direction does not result in the highest modularity gain, while also reducing nearly half of the modularity gain computations. As shown in Figure 3(b), during iteration i, $v3$ is restricted to selecting communities with labels greater than its own, which allows it to move to $C4$.

Overall, Bi-Probing(Lo) benefits from flexible bidirectional probing, addressing parallelization issues while ensuring that vertices move toward the direction of maximum modularity gain. Compared to other heuristic methods, it offers superior clustering results without the need for additional costly optimizations. Bi-Probing(Gl), on the other hand, benefits from a more proactive movement process, accelerating the evolution of the community structure, thereby significantly improving the algorithm's convergence efficiency. Additionally, due to the heuristic nature of the algorithm, the loss in clustering quality is minimal. We demonstrate this behavior in our experimental results (c.f. Section 7.6).

5.3 Scale-Free Aware Implementation

The scale-free characteristics of real-world graphs significantly affect the execution efficiency of Bi-Probing Heuristic on multi-GPU platforms. To address this challenge, we manage and allocate computational units and GPU scratchpad memory based on the number of vertex neighbors, aiming to balance the workload among computational units and enhance GPU resource utilization. Specifically, we categorize vertices into ten intervals (*bins*) based on their number of neighbors, launching a separate CUDA kernel for each bin and using stream technology to further increase the occupancy of streaming multiprocessors (SMs). Furthermore, we allocate between 2 to 1024 threads for vertex community movement tasks across different bins using NVIDIA cooperative group techniques[11], with each thread processing one neighbor of the vertex. We also use hashmaps as accumulators to calculate the sum of edge weights $e_{i \to C_j}$ between vertex i and all vertices in its adjacent community C_j. To maximize the utilization of the high-speed scratchpad memory, the size of hashmaps are set to the upper bound of the number of neighbors for vertices within each bin.

In Algorithm 1, we detail the specific process of modularity optimization. Rule2(v, i, C, ℓ) is applied when calculating the modularity gains for each movement direction (line 24). For clarity, we present the constraints of both variants simultaneously in the algorithm. Rule1(v, i, C, ℓ) for Bi-Probing(Gl) is applied before attempting to insert into hashmaps (line 12), while Rule1(v, i, C, ℓ) for Bi-Probing(Lo) is applied at the final stage of the movement operation (line 28).

6 Memory-Efficient Graph Coarsening

The peak memory overhead during the graph coarsening phase of the Louvain method is a critical factor limiting its ability to handle large-scale graphs. The challenge in designing a low-memory-overhead multi-GPU graph coarsening algorithm lies in reducing unnecessary storage consumption of the coarsened graph while balancing the size of the coarsened graph across multiple GPUs. To address this, MG-αGCD proposes a two-phase graph coarsening approach, consisting of symbolic and numeric phases.

Algorithm 2: Symbolic Graph Coarsening

input : Bin (*bin*), graph offset array (*offset*), graph edge array (*edge*), renumbered community id of vertices (*comIdRenum*).
output : Community Degree (*comDegree*)

```
1  foreach community in bin.comms do
2      foreach vertex in community.vertices do
3          lb ← nvshmem_g(offset[vertex.id]);
4          rb ← nvshmem_g(offset[vertex.id + 1]);
5          for e ← lb to rb do
6              neiId ← nvshmem_g(edge[e]);
7              key ← comIdRenum[neiId];
8              hash ← (key * SCAL) % tableSize;
9              while true do
10                 old_key ← atomicCAS(table + hash, -1,
                     key);
11                 if old_key = -1 or old_key = key then
12                     if old_key = -1 then
13                         degree ← degree + 1;
14                     break;
15     comDegree[community.id] ← degree;
```

Symbolic Graph Coarsening. In this phase, we calculate the number of vertices and edges in the coarsened graph without calculating the edge values. First, we gather the community ID information from all GPUs, allocating space equal to the number of vertices in the original graph to store the global community IDs. Subsequently, based on this information, we calculate the number of non-empty communities and renumber the community IDs, where non-empty communities will serve as the new vertices in the coarsened graph. Following this, we compute the sum of the degrees of the vertices within non-empty communities and use this information to assess the workload on processing units and the upper limit of scratchpad memory requirements when

calculating the number of adjacent communities for the non-empty communities in subsequent steps. Finally, we employ a hashing strategy to compute the number of adjacent communities for the non-empty communities (Algorithm 2).

While hashing performs efficiently in shared memory, its performance degrades significantly in the slower global memory. Real-world graphs often follow power-law distributions, posing additional challenges for load balancing and shared memory utilization during graph coarsening. To address these issues, we introduce a binning strategy similar to the Bi-Probing heuristic. Once the number of adjacent communities is determined, we compute the offset array in CSR format using a prefix sum.

Numeric Graph Coarsening. In this phase, we collapse the edges between vertices in adjacent non-empty communities to form new edges between the vertices of the coarsened graph. As the exact number of edges and memory requirements are known, the load can be balanced more precisely for the numeric phase. Based on the edge count information output from the previous phase, we partition the coarsened graph across the GPUs while balancing the number of edges within each GPU. Next, we employ a method similar to that in Algorithm 2 to calculate the *edge* and *edge weight* arrays. Note that the scale of the coarsened graph is typically much smaller than that of the original graph, thus, compared to allocating temporary space equal to the original graph size for storing the coarsened graph, this strategy significantly reduces peak storage overhead.

We highlights that, due to the proposed two-phase graph coarsening approach, MG-αGCD requires graph partitioning only during the first outer modularity optimization iteration. In subsequent iterations, partitioning is seamlessly integrated into the graph coarsening process. This significantly reduces the volume of CPU-GPU data transfers.

7 Evaluation

7.1 Platforms & Tools

Our major platform consists of four NVIDIA "Volta" V100-SXM2-32G GPUs (with 80 Streaming Multiprocessors), dual Intel(R) Xeon(R) E5-2698 v4 CPUs (with 40 cores, 2.20 GHz), and 512 GB of DDR4 memory. The GPUs are interconnected via NVLinks. We compile and link MG-αGCD using CUDA (v11.8), OpenMPI (v4.1.5), NVSHMEM (v2.9.0), CMake (v3.26.3), and GNU Make (v4.4.1) with the -O3 compilation option. The code is available at https://github.com/darius513/MG-alphaGCD.

7.2 Dataset

For real world graphs, we selected 12 graphs from the SuiteSparse Matrix Collection[7] as listed in Table 1. These graphs provide a reasonable variation in the graph characteristics with vertices ranging from 0.54M to 118.14M, edges ranging from 28.51M to 3.61B, max degree ranging from 9 to 3.00M, standard deviation of degreee (SD) ranging from 0.93 to 59.66, and 6 categories. Additionally, we also include graphs that are considered hard-to-cluster, as they yield low modularity in other studies.

Graphs	num.V	num.E	Max	SD
Road networks:				
road_usa	23.95M	57.71M	9	0.93
Citation networks:				
coPapersDBLP	0.54M	30.49M	3299	59.66
Wikipedia networks:				
wiki-topcats	1.79M	28.51M	3907	30.43
Social networks:				
ljournal-2008	5.36M	79.02M	2469	15.24
com-LiveJournal	4.00M	69.36M	14815	17.41
twitter7	41.65M	1.47B	2997469	35.45
com-Friendster	65.61M	3.61B	5214	8.98
Web networks:				
uk-2002	18.52M	298.11M	2450	13.03
webbase-2001	118.14M	1.02B	3841	9.49
it-2004	41.29M	1.15B	9964	28.19
sk-2005	50.64M	1.95B	12870	38.40
random networks:				
rgg_n_2_24_s0	16.78M	265.11M	40	15.50

Table 1: Real world graphs used in our evaluation."Max" represents the maximum degree of vertices, while"SD" represents the standard deviation of degrees.

7.3 Baselines

We compare MG-αGCD with state-of-the-art multi-GPU Louvain methods and the CPU-based Louvain method Grappolo. In all benchmarks, we set the modularity convergence threshold to 1.0E-06.

(1) **cuGraph**[25] is a widely-used graph analytics library by NVIDIA, which includes a state-of-the-art multi-GPU Louvain implementation. As part of the RAPIDS GPU-accelerated platform, it leverages Dask[30] to enable multi-GPU operations and extensively utilizes parallel libraries like Thrust[2] to accelerate the algorithm.

(2) **nido**[6] is a batched multi-GPU Louvain algorithm that transfers graph data in batches from host to device, enabling the processing of large graphs that exceed the total GPU memory capacity. nido performs graph coarsening entirely on the CPU. In our experiments, we set the batch size to 8.

(3) **Grappolo**[19] is a well-known CPU-based parallel Louvain method that introduces the minimum label, coloring, and the vertex following heuristic to address Louvain parallelization challenges.

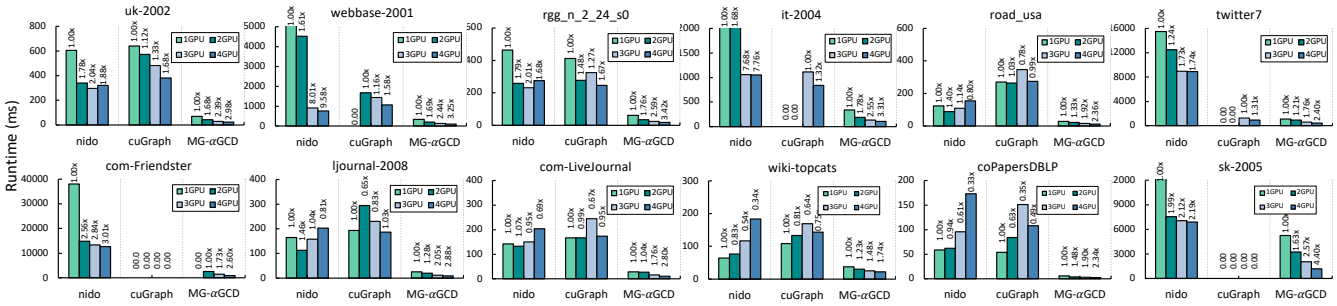

Figure 4: The runtime per iteration and scalability of MG-αGCD and the compared multi-GPU Louvain methods. The y-axis shows the runtime, while the x-axis shows different methods. Different colors represent GPU configurations ranging from 1 to 4 GPUs. The number on each bar indicates the speedup of the method using i GPUs compared to its performance with 1 GPU. The '0.00' indicates that the method fails to run at this GPU configuration due to out-of-memory.

7.4 Performance Comparison over Existing Work

7.4.1 Total Runtime. We first study the differences in total runtime among MG-αGCD, Grappolo, nido, and cuGraph. The results in Table 2 reveal that MG-αGCD achieves significant overall performance advantages over the other methods, with an average speedup (geometric mean) of 34.60x, 30.75x, and 11.31x compared to Grappolo, nido, and cuGraph, respectively. The maximum speedups reach 137.82x, 818.82x, and 44.15x, respectively.

Notably, MG-αGCD demonstrates substantial acceleration on 'road_usa' and 'coPapersDBLP'. The 'road_usa' graph exhibits the lowest *SD* among all tested graphs, while 'coPapersDBLP' has the highest *SD*, underscoring MG-αGCD's consistent performance across graphs with varying power-law degree distribution. Additionally, MG-αGCD achieves superior acceleration on datasets with low modularity, such as 'ljournal-2008', 'com-LiveJournal', 'wiki-topcats', and 'coPapersDBLP'.

Furthermore, MG-αGCD shows considerably acceleration on large-scale graphs. As for 'sk-2005' and 'com-Friendster', MG-αGCD completes the computation in the shortest time, while cuGraph fails to run due to `out-of-memory`. We do note that for 'twitter7', the speedup is considerably smaller compared to other large-scale graphs, primarily due to the exceptionally high maximum degree, which leads to severe load imbalance and GPU shared memory pressure. Addressing this bottleneck through new programming paradigms, such as edge-centric approaches, represents a promising direction for future work.

7.4.2 Modularity. This subsection evaluates the clustering quality of MG-αGCD. Table 2 summarizes the results, with *modularity* serving as the evaluation metric. MG-αGCD demonstrates superior performance compared to the other three methods across the majority of datasets. Notably, for graphs that are challenging to cluster, MG-αGCD outperforms the second-best method, the CPU-based Grappolo. Even for datasets where MG-αGCD's clustering quality is lower than that of Grappolo, such as 'com-LiveJournal' and 'it-2004', the modularity achieved by MG-αGCD remain comparable.

To the best of our knowledge, MG-αGCD is the only GPU-based Louvain method that surpasses Grappolo in clustering quality. In contrast, other approaches often trade off clustering quality for higher execution efficiency. MG-αGCD incorporates only the Bi-Probing heuristic described in Section 5 and does not include other optimizations used in previous studies, such as vertex pruning or modifying initial tolerance.

7.4.3 Scalability. To further compare the execution efficiency and scalability of the algorithms, we evaluated the average iteration time during the first phase for each method under 1 to 4 GPU configurations, as well as the relative speedup compared to their respective single-GPU configurations.

As shown in Figure 4, MG-αGCD demonstrates significantly lower average iteration times across all GPU configurations compared to other methods. Specifically, in the 4-GPU configuration, MG-αGCD achieved an average speedup of 14.43x and 12.58x over nido and cuGraph, respectively, with maximum speedups of 68.47x and 42.70x.

Scalability is significantly influenced by the power-law degree distribution of graphs. However, as illustrated in Figure 4, MG-αGCD consistently demonstrates excellent scalability across graphs of varying sizes, power-law distributions, vertex degrees, and complexities. It also exhibits a promising scalability trend, with relative speedup increasing as the number of GPUs increases, suggesting the potential for even

Graphs	Grappolo		nido		cuGraph		MG-αGCD (ours)		MG-αGCD vs cuGraph Speedup
	Q	T	Q	T	Q	T	Q	T	
uk-2002	0.991968	109.02	0.916482	248.95	0.989517	19.16	**0.992201**	**1.30**	14.74
webbase-2001	0.986583	213.31	0.821354	1692.72	0.982278	69.47	**0.987285**	**14.15**	4.91
rgg_n_2_24_s0	0.992610	82.95	0.841726	96.75	0.992967	14.77	**0.993041**	**0.87**	16.98
it-2004	**0.993613**	250.06	0.86308	259.68	0.975552	42.47	0.991942	**7.24**	5.87
road_usa	0.997919	45.48	0.592551	270.21	0.997978	14.57	**0.997996**	**0.33**	44.15
twitter7	0.510858	2131.31	0.389706	265.05	0.486518	86.56	**0.526612**	**57.22**	1.51
ljournal-2008	0.761307	304.62	0.721337	51.39	0.754148	38.83	**0.767071**	**3.23**	12.02
com-LiveJournal	**0.747204**	100.82	0.746983	21.75	0.746374	33.78	0.746273	**2.84**	11.89
wiki-topcats	0.638614	14.79	0.606305	10.96	0.638289	21.13	**0.642356**	**1.76**	12.01
coPapersDBLP	0.857962	10.55	0.851131	4.74	0.85923	4.55	**0.860105**	**0.11**	41.36
sk-2005	0.983659	216.56	0.904583	1137.29	*NA*	*NA*	**0.984680**	**66.27**	*NA*
com-Friendster	0.618186	5426.54	0.638332	1276.53	*NA*	*NA*	**0.641119**	**346.96**	*NA*

Table 2: Performance comparison of total runtime and modularity. Bold text indicates the lowest runtime and the highest modularity. In this comparison, MG-αGCD employs the local Bi-Probing heuristic, and all multi-GPU methods are run on a 4-GPU configuration. The 'NA' indicates that the method fails to run due to out-of-memory. Here, Q = modularity, T = time(s).

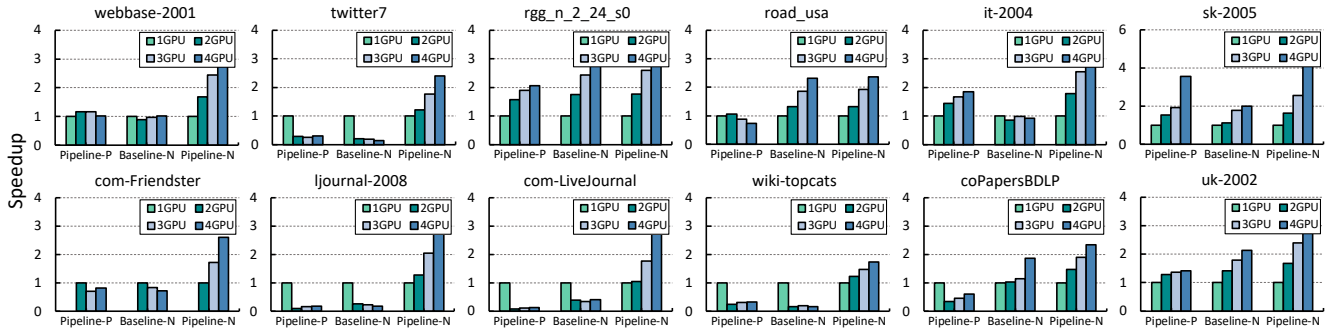

Figure 5: The impact of pipeline and GPU Interconnect on scalability.

greater speedup with additional GPUs. In contrast, both nido and cuGraph exhibit a phenomenon where increasing the number of GPUs results in longer runtimes, particularly in cases where communication dominates execution time, such as 'ljournal-2008', 'com-LiveJournal', 'wiki-topcats', 'coPapersDBLP', 'rgg_n_2_24_s0', and 'road_usa'.

We also observe that nido achieved significant relative speedup on 'webbase-2001', 'it-2004', and 'com-Friendster'. This is primarily due to its exceptionally low efficiency on a single GPU. Additionally, in these cases, nido's performance no longer scales with the increase in GPU count after reaching a certain threshold, which aligns with the observations reported in its original paper. The batch strategy allows nido to process 'com-Friendster' on a single GPU, but it also constrains its performance on a single GPU.

7.5 Effect of Pipeline and GPU Interconnect

In this subsection, we analyze the impact of the pipeline and GPU interconnect on the scalability of MG-αGCD. The *baseline* refers to the irregular atomic implementation described in Section 4.2.1, the *pipeline* denotes the pipeline optimization detailed in Section 4.2.2. The suffix *-P* indicates the PCIe interconnect, whereas *-N* indicates the NVLink interconnect. The results are shown in Figure 5.

Our observations indicate that *Pipeline-N* demonstrates superior scalability compared to *Baseline-N* across all test cases, with its advantages being particularly significant for graphs with high degrees and skewness, such as 'com-LiveJournal', 'twitter7', and 'sk-2005'. The *Pipeline-N* achieves robust scalability on these graphs, whereas the *Baseline-N* suffers from performance degradation as the number of GPUs increases.

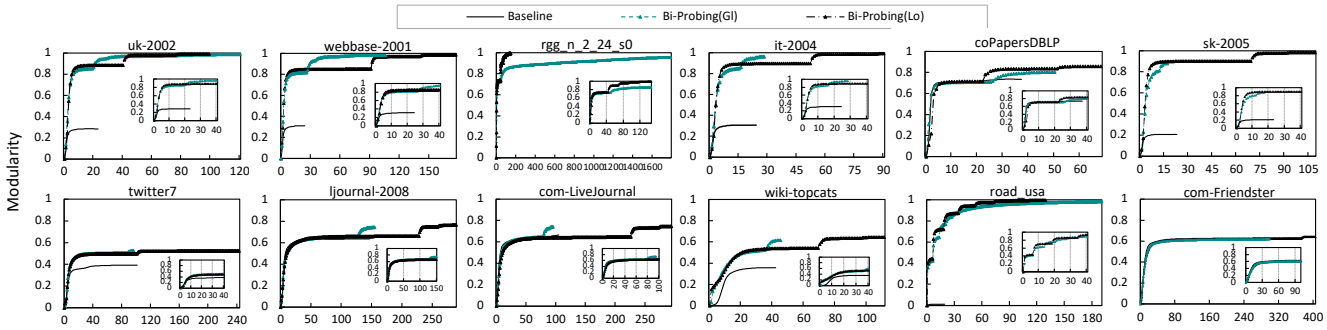

Figure 6: Comparison of modularity evolution. The abrupt increase in modularity corresponds to transitions between phases.

This suggests that the pipeline design effectively mitigates inter-GPU communication overheads, thereby enhancing MG-αGCD's scalability. These findings underscore the importance of algorithm-level communication-hiding strategies in achieving scalable performance.

To further analyze the impact of GPU interconnect on scalability, we conducted experiments on a platform comprising four V100-32G GPUs interconnected via PCIe. We observed that *Pipeline-N* exhibits better scalability than *Pipeline-P*, primarily because NVLink offers higher communication bandwidth and lower latency compared to PCIe. Moreover, NVLink achieves relatively better bandwidth efficiency with smaller payload sizes compared to PCIe. Given its higher bandwidth efficiency, low latency, and high bandwidth, NVLink is more suitable for applications with small access patterns.

In summary, both algorithm-level communication optimization and the effective use of high-speed GPU interconnect technologies are essential for achieving excellent scalability in MG-αGCD.

7.6 Impact of Bi-Probing Heuristic

Next, we explore the impact of Bi-Probing heuristic on algorithm convergence and clustering quality. Here: (1) *Baseline* refers to the approach without any optimizations addressing the challenge of Louvain parallelization. (2) *Bi-Probing(Lo)* refers to the local bidirectional probing variant outlined in Section 5.2.1. (3) *Bi-Probing(Gl)* refers to the global bidirectional probing variant described in Section 5.2.2.

Figure 6 illustrates the modularity evolution for the three methods. While the *baseline* converges more quickly, the clustering quality it achieves is significantly inferior to that of *Bi-Probing* proposed in this study.

We further compare the two variants. In terms of modularity, *Bi-Probing(Gl)* consistently produces results that are, on average, 1.5% lower than those of *Bi-Probing(Lo)*. Additionally, we observe that the choice of variant significantly impacts modularity convergence efficiency. The difference in the timing of the abrupt modularity increase indicates that *Bi-Probing(Gl)* evolves through phases more rapidly compared to *Bi-Probing(Lo)*, and reaches the final phase faster. For hard-to-cluster graphs, such as 'twitter7,' as well as in most other cases, *Bi-Probing(Gl)* converges faster than *Bi-Probing(Lo)*. This implies that on most datasets, employing Bi-Probing(Gl) allows MG-αGCD to achieve even greater speedups compared to the results shown in Table 2. A different behavior is observed on a few datasets, such as 'rgg_n_2_24_s0' and 'road_usa', where *Bi-Probing(Gl)* requires more iterations. The road networks are known to behave diffrently for a large variety of problems, due to the lack of small world characteristic[29]. Similarly, random networks, being synthetic, also lack this characteristic.

7.7 Comparison of Peak Space Cost

Figure 7 shows the comparison of the peak memory consumption between MG-αGCD and two other multi-GPU Louvain methods, nido, and cuGraph, under a 4-GPU configuration. The maximum peak memory cost of MG-αGCD is reduced by 41.7% compared to cuGraph and by 36.1% compared to nido. Besides, MG-αGCD consistently demonstrates lower peak memory consumption than cuGraph across all datasets. Specifically, for 'uk-2002', 'webbase-2001', 'twitter7', and 'rgg_n_2_24_s0', MG-αGCD reduces peak memory cost by 69.86%, 58.37%, 56.90%, and 50.19%, respectively, compared to cuGraph. Additionally, since nido does not perform graph coarsening on the GPU and introduces a batch strategy to reduce the storage overhead of graph structure information, its peak memory consumption is lower than both cuGraph and MG-αGCD on relatively smaller graphs, such as 'road_usa'. However, for larger graphs, MG-αGCD achieves significantly lower peak memory consumption than nido, due to precise GPU memory management.

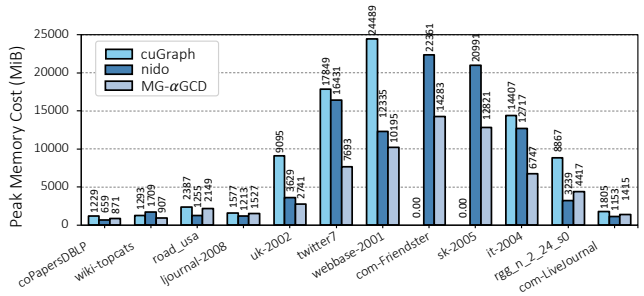

Figure 7: Peak memory cost of MG-αGCD and the compared multi-GPU Louvain methods.

7.8 Runtime Breakdown

Figure 8 shows the runtime breakdown of MG-αGCD. It includes the first step (*vertex movement*), the second step (*community information update*), the third step (*modularity calculation*) of modularity optimization, as well as the symbolic and numeric phases of graph coarsening. As shown, modularity optimization accounts for an average (arithmetic mean) of 89.2% of the total runtime. In contrast, graph coarsening only takes up an average of 10.8% of the total runtime. Specifically, the first, second, and third steps of modularity optimization and the symbolic and numeric phases of graph coarsening contribute to the total runtime by an average of 50.5%, 8.0%, 30.7%, 3.4%, and 7.4%, respectively.

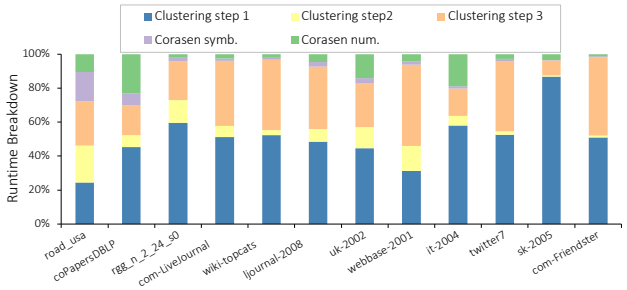

Figure 8: The runtime breakdown of MG-αGCD.

8 Discussion

Graph Community Detection and Graph Partitioning. Although we have primarily focused on the Louvain method, our communication-hiding and memory-efficiency optimizations/designs could also be extended to other graph community detection algorithms, such as Leiden, as well as multilevel graph partitioning algorithms, including the multilevel recursive-bisection and multilevel k-way algorithms used in METIS[15]. These design optimizations offer crucial insights for adapting these applications to multi-GPU platforms.

9 Related Work

In previous research, significant efforts have been made to parallelize the serial Louvain method. Grappolo[19] employs heuristic techniques such as graph coloring, minimum labeling, and vertex following. In a single GPU context, Naim et al.[20] improved load balancing across processing units by enabling parallel access to individual edges. Cheong et al.[5] proposed a hierarchical Louvain method, in which the original graph is partitioned into several disjoint subgraphs, and the edges between the subgraphs are temporarily removed.

With the increasing computational power of multi-GPU platforms, multi-GPU Louvain methods have also emerged. Chou et al.[6] introduced a batch-based multi-GPU Louvain method, nido, which divides vertex and edge information into batches and processes them sequentially using a bulk synchronization parallel model. NVIDIA's cuGraph library[25], one of the most advanced GPU-based graph analysis libraries, extends multi-GPU capabilities through Dask and offers a highly efficient Louvain method. cuGraph adopts a tiling-like technique to coarsen the graph. However, cuGraph converts the CSR format into a vertex-pair format, which increases memory requirement. Sattar et al.[31] extended the Louvain algorithm to a distributed multi-GPU environment. Although they employed CUDA-Aware MPI to reduce the latency of data transfers between GPUs, their scalability is still limited by the following factors: (1) bidirectional and aggregation communication initiated by the CPU; (2) the graph coarsening process is performed by the CPU.

10 Conclusion

This paper presents MG-αGCD, a novel algorithm design and implementation that accelerates the Louvain method on multi-GPU platforms. MG-αGCD incorporates a computation-communication pipeline that effectively hides inter-GPU communication. Additionally, MG-αGCD includes a bidirectional probing heuristic that enhances efficiency while providing better clustering solutions. Finally, MG-αGCD design a two-phase graph coarsening method that significantly reduces memory usage. Experimental results demonstrate that MG-αGCD achieves substantial speedups in efficiency and scalability compared to state-of-the-art methods, while also conserving considerable amounts of memory.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and suggestions to improve this work. Changyou Zhang is the corresponding author of this paper. This research was supported by National Key Research and Development Program of China (2023YFB3611303).

References

[1] Xueying Bai, Peilin Yang, and Xiaohu Shi. 2017. An overlapping community detection algorithm based on density peaks. *Neurocomputing* 226 (2017), 7–15.

[2] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[4] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluç, Katherine Yelick, and John D Owens. 2022. Scalable irregular parallelism with GPUs: Getting CPUs out of the way. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE, 1–16.

[5] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. 2013. Hierarchical parallel algorithm for modularity-based community detection using GPUs. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19 (Euro-Par '13)*. Springer, 775–787.

[6] Han-Yi Chou and Sayan Ghosh. 2022. Batched Graph Community Detection on GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '22)*. 172–184.

[7] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[8] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17.

[9] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *2018 IEEE international parallel and distributed processing symposium (IPDPS '18)*. IEEE, 885–895.

[10] Bagus Hanindhito, Dimitrios Gourounas, Arash Fathi, Dimitar Trenev, Andreas Gerstlauer, and Lizy K. John. 2022. GAPS: GPU-acceleration of PDE solvers for wave simulation. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 30.

[11] M. Harris and K. Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups. Accessed: 2025-01-08.

[12] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 192–202.

[13] Mohammad Kefah Taha Issa, Muhammad Aditya Sasongko, Ilyas Turimbetov, Javid Baydamirli, Doğan Sağbili, and Didem Unat. 2024. Snoopie: A Multi-GPU Communication Profiler and Visualizer. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 525–536.

[14] Xuewu Jiao, Xinsheng Luo, Miao Li, Jiang Bian, Junchao Yang, Wei Hu, Mingqing Hu, Weipeng Lu, Shikun Feng, Danlei Feng, Dongxu Yang, Haoyi Xiong, Shuanglong Li, and Lin Liu. 2024. A Multi-Node Multi-GPU Distributed GNN Training Framework for Large-Scale Online Advertising. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) *(CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 4595–4602.

[15] George Karypis and Vipin Kumar. 1998. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* 38 (1998), 7–1.

[16] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.

[17] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: evaluating modern GPU interconnect via a multi-GPU benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC '18)*. IEEE, 191–202.

[18] Chun-Cheng Lin, Jia-Rong Kang, and Jyun-Yu Chen. 2015. An integer programming approach and visual analysis for detecting hierarchical community structures in social networks. *Information Sciences* 299 (2015), 296–311.

[19] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel Comput.* 47 (2015), 19–37.

[20] Md Naim, Fredrik Manne, Mahantesh Halappanavar, and Antonino Tumeo. 2017. Community detection on the GPU. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, 625–634.

[21] Mark EJ Newman. 2013. Community detection and graph partitioning. *Europhysics Letters* 103, 2 (2013), 28003.

[22] Mark EJ Newman. 2013. Spectral methods for community detection and graph partitioning. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 88, 4 (2013), 042822.

[23] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.

[24] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 90–106.

[25] NVIDIA. 2024. NVIDIA RAPIDS cuGraph. https://github.com/rapidsai/cugraph Accessed: 2025-01-08.

[26] NVIDIA. 2024. NVSHMEM Communication Library. https://developer.nvidia.com/nvshmem. Accessed: 2025-01-08.

[27] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. 362–375.

[28] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. 2015. Scalable community detection with the louvain algorithm. In *2015 IEEE international parallel and distributed processing symposium (IPDPS '15)*. IEEE, 28–37.

[29] Luis Remis, Maria Jesus Garzaran, Rafael Asenjo, and Angeles Navarro. 2018. Exploiting social network graph characteristics for efficient BFS on heterogeneous chips. *J. Parallel and Distrib. Comput.* 120 (2018), 282–294.

[30] Matthew Rocklin et al. 2015. Dask: Parallel computation with blocked algorithms and task scheduling.. In *SciPy*. 126–132.

[31] Naw Safrin Sattar, Hao Lu, Feiyi Wang, and Mahantesh Halappanavar. 2024. Distributed Multi-GPU Community Detection on Exascale Computing Platforms. In *2024 IEEE International Parallel and Distributed*

Processing Symposium Workshops (IPDPSW). 815–824.

[32] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. 2023. UGACHE: A Unified GPU Cache for Embedding-based Deep Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 627–641.

[33] Christian L Staudt and Henning Meyerhenke. 2015. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 171–184.

[34] Jesmin Jahan Tithi, Andrzej Stasiak, Sriram Aananthakrishnan, and Fabrizio Petrini. 2020. Prune the Unnecessary: Parallel Pull-Push Louvain Algorithms with Automatic Edge Pruning. In *Proceedings of the 49th International Conference on Parallel Processing* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 44.

[35] Shuai Yang, Changyou Zhang, and Ji Ma. 2023. DeltaSPARSE: High-Performance Sparse General Matrix-Matrix Multiplication on Multi-GPU Systems . In *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 194–202.