High-Performance Implementation of Louvain Algorithm with Representational Optimizations

Subhajit Sahu[1], Kishore Kothapalli[1], and Dip Sankar Banerjee[2]

[1] IIIT Hyderabad, Professor CR Rao Rd, Gachibowli, Hyderabad, Telangana, India,
{subhajit.sahu@research., kkishore@}iiit.ac.in,
[2] IIT Jodhpur, NH 62, Surpura Bypass Rd, Karwar, Rajasthan, India,
dipsankarb@iitj.ac.in

Abstract. Community detection is the problem of identifying natural divisions in networks. Efficient parallel algorithms for identifying such divisions is critical in a number of applications, where the size of datasets have reached significant scales. This paper presents one of the most efficient multicore implementations of the Louvain algorithm, a high quality community detection method. On a server equipped with dual 16-core Intel Xeon Gold 6226R processors, our Louvain, which we term as GVE-Louvain, outperforms Vite, Grappolo, NetworKit Louvain, and cuGraph Louvain (running on NVIDIA A100 GPU) by $50\times$, $22\times$, $20\times$, and $5.8\times$ faster respectively - achieving a processing rate of $560M$ edges/s on a $3.8B$ edge graph. In addition, GVE-Louvain improves performance at an average rate of $1.6\times$ for every doubling of threads.

Keywords: Community detection, Parallel Louvain algorithm

1 Introduction

Community detection is the problem of uncovering the underlying structure of complex networks, i.e., identifying groups of vertices that exhibit dense internal connections but sparse connections with the rest of the network, in an unsupervised manner. It has numerous applications in domains such as drug discovery, protein annotation, topic discovery, anomaly detection, and criminal identification. Communities identified are intrinsic when based on network topology alone, and are disjoint when each vertex belongs to only one community [11]. The *Louvain* method [4] is a popular heuristic-based approach for community detection, with the modularity metric [18] being used to measure the quality of communities identified.

In recent years, the collection of data and the relationships among them, represented as graphs, have reached unmatched levels. This has necessitated the design of efficient parallel algorithms for community detection on large networks. Existing studies on Louvain propose several optimizations [10, 12, 17, 26] and parallelization techniques [2, 3, 6, 7, 10, 12, 17, 22, 27]. Further, significant research effort has been focused on developing efficient parallel implementations of Louvain algorithm for multicore CPUs [7, 12, 23, 24], GPUs [17], CPU-GPU hybrids [3], multi-GPUs [6, 8], and multi-node systems — CPU only [10] / CPU-GPU hybrids [2].

However, many of the aforementioned works concentrate on optimizing the local-moving phase of the Louvain algorithm — these optimization techniques are scattered over a number of papers, making it difficult for a reader to get a grip over them — but do not address optimization for the aggregation phase of the algorithm, which emerges as a bottleneck after the local-moving phase has been optimized. Some implementations also fail to adequately parallelize the algorithm. Moreover, much attention has been directed towards GPU-based solutions. However, developing algorithms that efficiently utilize GPUs can be challenging both in terms of initial implementation and ongoing maintenance. Further, the soaring prices of GPUs present hurdles. The multicore/shared memory environment holds significance for community detection, owing to its energy efficiency and the prevalence of hardware with ample DRAM capacities. Through our implementation of the Louvain algorithm[3], we aim to underscore that CPUs remain adept at irregular computation , especially for algorithms where workload diminishes progressively with each iteration. Additionally, we show that achieving optimal performance necessitates a focus on the data structures.

2 Related work

The *Louvain* method is a greedy modularity-optimization based community detection algorithm, and is introduced by Blondel et al. from the University of Louvain [4]. It identifies communities with resulting high modularity, and is thus widely favored [15]. Algorithmic improvements proposed for the original algorithm include early pruning of non-promising candidates (leaf vertices) [12], attempting local move only on likely vertices [22], ordering of vertices based on node importance [1], moving nodes to a random neighbor community [25], threshold scaling [12, 17], threshold cycling [10], subnetwork refinement [26], multilevel refinement [22], and early termination [10].

To parallelize the Louvain algorithm, a number of strategies have been attempted. These include using heuristics to break the sequential barrier [16], ordering vertices via graph coloring [12], performing iterations asynchronously [22], using adaptive parallel thread assignment [17], parallelizing the costly first iteration [27], using vector based hashtables [12], and using sort-reduce instead of hashing [6].

We now discuss about a number of state-of-the-art implementation of Parallel Louvain. Ghosh et al. [9] propose Vite, a distributed memory parallel implementation of the Louvain method that incorporates several heuristics to enhance performance while maintaining solution quality, while Grappolo, by Halappanavar et al. [12], is a shared memory parallel implementation. Qie et al. [20] present a graph partitioning algorithm that divides the graph into sets of partitions, aiming to minimize inter-partition communication delay and avoid community swaps, akin to the graph coloring approach proposed by Halappanavar et al. [12]. We do not observe the community swap issue on multicore CPUs (it likely resolves itself), but do observe it on GPUs (likely due to lockstep execution). NetworKit [24] is a software package designed for analyzing the structural aspects of graph data sets with billions of connections. It is implemented as a hybrid with C++ kernels and a Python frontend, and includes a parallel implementation

[3] https://github.com/puzzlef/louvain-communities-openmp

of the Louvain algorithm. Finally, cuGraph [13] is a GPU-accelerated graph analytics library that is part of the RAPIDS suite of data science and machine learning tools. It harnesses the power of NVIDIA GPUs to significantly speed up graph analytics compared to traditional CPU-based methods. cuGraph's core is written in C++ with CUDA and is accessed through a Python interface.

However, most existing works only focus on optimizing the local-moving phase of the Louvain algorithm, and lack effective parallelization. For instance, the implementation of NetworKit Louvain exhibits several shortcomings. It employs plain OpenMP parallelization for certain operations, utilizing a static schedule with a chunk size of 1, which may not be optimal when threads are writing to adjacent memory addresses. Additionally, NetworKit Louvain employs guided scheduling for the local-moving phase, whereas we utilize dynamic scheduling for better performance. NetworKit Louvain also generates a new graph for each coarsening step, leading to repeated memory allocation and preprocessing due to recursive calls. The coarsening involves several sequential operations, and adding each edge to the coarsened graph requires $O(D)$ operations, where D represents the average degree of a vertex, which is suboptimal for parallelism. Lastly, NetworKit Louvain lacks parallelization for flattening the dendrogram, potentially hindering its performance. Our parallel implementation of Louvain addresses these issues.

3 Preliminaries

Consider an undirected graph $G(V, E, w)$ with V representing the set of vertices, E the set of edges, and $w_{ij} = w_{ji}$ denoting the weight associated with each edge. In the case of an unweighted graph, we assume unit weight for each edge ($w_{ij} = 1$). The neighbors of a vertex i are denoted as $J_i = \{j \mid (i, j) \in E\}$, the weighted degree of each vertex as $K_i = \sum_{j \in J_i} w_{ij}$, the total number of vertices as $N = |V|$, the total number of edges as $M = |E|$, and the sum of edge weights in the undirected graph as $m = \sum_{i,j \in V} w_{ij}/2$.

3.1 Community detection

Disjoint community detection is the process of identifying a community membership mapping, $C : V \to \Gamma$, where each vertex $i \in V$ is assigned a community-id $c \in \Gamma$, where Γ is the set of community-ids. We denote the vertices of a community $c \in \Gamma$ as V_c, and the community that a vertex i belongs to as C_i. Further, we denote the neighbors of vertex i belonging to a community c as $J_{i \to c} = \{j \mid j \in J_i \ and \ C_j = c\}$, the sum of those edge weights as $K_{i \to c} = \sum_{j \in J_{i \to c}} w_{ij}$, the sum of weights of edges within a community c as $\sigma_c = \sum_{(i,j) \in E \ and \ C_i = C_j = c} w_{ij}$, and the total edge weight of a community c as $\Sigma_c = \sum_{(i,j) \in E \ and \ C_i = c} w_{ij}$.

3.2 Modularity

Modularity serves as a metric for assessing the quality of communities identified by heuristic-based community detection algorithms. It is computed as the difference between the fraction of edges within communities and the expected fraction if edges were randomly distributed, yielding a range of $[-0.5, 1]$ where higher values indicate better results [5]. The modularity Q of identified communities is determined using Equation 1, where δ represents the Kronecker delta function ($\delta(x, y) = 1$ if $x = y$, 0 otherwise). The *delta modularity* of moving a vertex i from community d to community c, denoted as $\Delta Q_{i:d \to c}$, can be computed using Equation 2.

$$Q = \frac{1}{2m} \sum_{(i,j) \in E} \left[w_{ij} - \frac{K_i K_j}{2m} \right] \delta(C_i, C_j) = \sum_{c \in \Gamma} \left[\frac{\sigma_c}{2m} - \left(\frac{\Sigma_c}{2m} \right)^2 \right] \tag{1}$$

$$\Delta Q_{i:d \to c} = \frac{1}{m}(K_{i \to c} - K_{i \to d}) - \frac{K_i}{2m^2}(K_i + \Sigma_c - \Sigma_d) \tag{2}$$

3.3 Louvain algorithm

The Louvain method [4] is a modularity optimization based agglomerative algorithm for identifying high quality disjoint communities in large networks. It has a time complexity of $O(L|E|)$ (with L being the total number of iterations performed), and a space complexity of $O(|V| + |E|)$ [15]. The algorithm consists of two phases: the *local-moving phase*, where each vertex i greedily decides to move to the community of one of its neighbors $j \in J_i$ that gives the greatest increase in modularity $\Delta Q_{i:C_i \to C_j}$ (using Equation 2), and the *aggregation phase*, where all the vertices in a community are collapsed into a single super-vertex. These two phases make up one pass, which repeats until there is no further increase in modularity [4].

4 Approach

4.1 Optimizations for Louvain algorithm

We use a parallel implementation of the Louvain method to determine suitable parameter settings and optimize the original algorithm through experimentation with a variety of techniques. We use *asynchronous* version of Louvain, where threads work independently on different parts of the graph. This allows for faster convergence but can also lead to more variability in the final result [4, 12].

For each optimization, we test a number of relevant alternatives, and show the relative time and the relative modularity of communities obtained by each alternative in Figure 1. This result is obtained by running the tests on each graph in the dataset (see Table 1), 5 times on each graph to reduce the impact of noise, taking their geometric mean and arithmetic mean for the runtime and modularity respectively, and representing them as a ratio within each optimization category.

Adjusting OpenMP loop schedule We attempt *static*, *dynamic*, *guided*, and *auto* loop scheduling approaches of OpenMP (each with a chunk size of 2048) to parallelize the local-moving and aggregation phases of the Louvain algorithm. Results indicate that the scheduling behavior can have small impact on the quality of obtained communities. We consider OpenMP's `dynamic` loop schedule to be the best choice, as it helps achieve better load balancing when the degree distribution of vertices is non-uniform, and offers a 7% reduction in runtime with respect to OpenMP's *auto* loop schedule, with only a 0.4% reduction in the modularity of communities obtained (likely to be just noise).

Limiting the number of iterations per pass Restricting the number of iterations of the local-moving phase ensures its termination within a reasonable number of iterations, which helps minimize runtime. This can be important since the local-moving phase performed in the first pass is the most expensive step of the algorithm. However, choosing too small a limit may worsen convergence rate. Our results indicate that limiting the maximum number of iterations to 20 allows for 13% faster convergence, when compared to a maximum iterations of 100.

Adjusting tolerance drop rate (threshold scaling) Tolerance is used to detect convergence in the local-moving phase of the Louvain algorithm, i.e., when the total delta-modularity in a given iteration is below or equal to the specified tolerance, the local-moving phase is considered to have converged. Instead of using a fixed tolerance across all passes of the Louvain algorithm, we can start with an initial high tolerance and then gradually reduce it. This is known as threshold scaling [12, 17], and it helps minimize runtime of the first pass of the algorithm (which is usually the most expensive). Based on our findings, a tolerance drop rate of 10 yields 4% faster convergence, with respect to a tolerance drop rate of 1 (threshold scaling disabled), with no reduction in quality.

Adjusting initial tolerance Starting with a smaller initial tolerance allows the algorithm to explore broader possibilities for community assignments in the early stage, but comes at the cost of increased runtime. We find an initial tolerance of 0.01 provides a 14% reduction in runtime of the algorithm with no reduction in the quality of identified communities, when compared to an initial tolerance of 10^{-6}.

Adjusting aggregation tolerance The aggregation tolerance determines the point at which communities are considered to have converged based on the number of community merges. In other words, if too few communities merged this pass we should stop here, i.e., if $|V_{aggregated}|/|V| \geq$ aggregation tolerance, we consider the algorithm to have converged. Adjusting aggregation tolerance allows the algorithm to stop earlier when further merges have minimal impact on the final result. According to our observations, an aggregation tolerance of 0.8 appears to be the best choice, as it presents a 14% reduction in runtime, when compared to the aggregation tolerance being disabled (1), while identifying final communities of equivalent quality.

Vertex pruning Vertex pruning is used to minimize unnecessary computation [22]. Here, when a vertex changes its community, its marks its neighbors to be processed. Once a vertex has been processed, it is marked as not to be processed. However, it comes with the added overhead of marking/unmarking of vertices. Based on our results, vertex pruning justifies this overhead, and should be enabled for 11% performance gain.

Finding community vertices for aggregation phase In the aggregation phase of the Louvain algorithm, the communities obtained in the previous local-moving phase of the algorithm are combined into super-vertices in the aggregated graph, with the edges between two super-vertices being equal to the total weight of edges between the respective communities. This requires one to obtain the list of vertices belonging to each community, instead of the mapping of community membership of each vertex that we have after the local-moving phase ends. A straight-forward implementation of this would make use of two-dimensional arrays for storing vertices belonging to each community, with the index in the first dimension representing the community id c, and the index in the second dimension pointing to the n^{th} vertex in the given community c. However, this requires memory allocation during the algorithm, which is expensive. We employ a parallel prefix sum technique along with a preallocated Compressed Sparse Row (CSR) data structure, eliminating repeated memory allocation and deallocation, and enhancing performance. Indeed, our findings indicate that using parallel prefix sum along with a preallocated CSR is $2.2\times$ faster than using 2D arrays for aggregating vertices.

Storing aggregated communities (super-vertex graph) After the list of vertices belonging to each community have been obtained, the communities need to be aggregated (or compressed) into super-vertices, such that edges between two super-vertices being equal to the total weight of edges between the respective communities. This is generally called the super-vertex graph, or the compressed graph. It is then used as an input to the local-moving phase of the next pass of the Louvain algorithm. A simple data structure to store the super-vertex graph in the adjacency list format would be a two-dimensional array. Again, this requires memory allocation during the algorithm, which is bad for performance. Utilizing two preallocated CSRs, one for the source graph and the other for the target graph (except the first pass, where the dynamic graph may be stored in any desired format suitable for dynamic batch updates), along with parallel prefix sum can help here. We observe that using parallel prefix sum along with preallocated CSRs for maintaining the super-vertex graph is again $2.2\times$ faster than using 2D arrays.

Hashtable design for local-moving/aggregation phases One can use C++'s inbuilt maps as per-thread (independent) hashtables for the Louvain algorithm — but this has poor performance. We use a key-list and a full-size values array (collision-free) to dramatically improve performance. However, if memory addresses of the hashtables are nearby (*Close-KV*) — as with NetworKit Louvain [24], performance is not as high, even if each thread uses its own hashtable exclusively. This is possibly due to false cache-sharing. Alternatively, if we ensure that the memory address of each hashtable are farther away (*Far-KV*), the performance improves. Our results indicate that *Far-KV* has the best performance and is $4.4\times$ faster than *Map*, and $1.3\times$ faster than *Close-KV*.

4.2 Our optimized Louvain implementation

A flow diagram illustrating the first pass of GVE-Louvain for a Weighted 2D-vector based or a Weighted CSR with degree based input graph, is shown in Figure 2. In the local-moving phase, vertex community memberships are updated until the total change

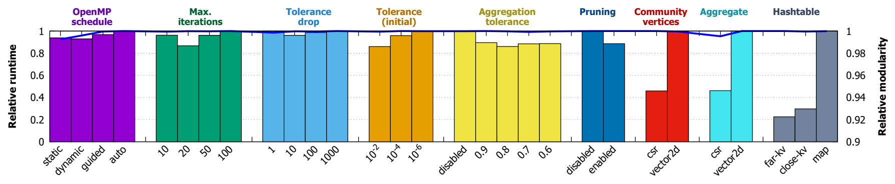


Fig. 1: Impact of various parameter controls and optimizations on the runtime and modularity of the *Louvain* algorithm. The impact upon relative runtime is shown as colored bars on the left y-axis, and upon relative modularity as a blue line on the right y-axis.

in delta-modularity across all vertices reaches a specified threshold. Community memberships are then counted and renumbered. In the aggregation phase, community vertices in a CSR are first obtained. This is used to create the super-vertex graph stored in a Weighted Holey CSR with degree. In subsequent passes, the input is a Weighted Holey CSR with degree and initial membership for super-vertices from the previous pass. The detailed algorithm of GVE-Louvain is included in our extended report [21].

We aim to incorporate GVE-Louvain into our upcoming command-line graph processing tool named "GVE", derived from "Graph(Vertices, Edges)". GVE-Louvain exhibits a time complexity of $O(KM)$, where K signifies the total iterations conducted. Its space complexity is $O(TN + M)$, where T denotes the number of threads utilized, and TN represents the collision-free hash tables H_t employed per thread.

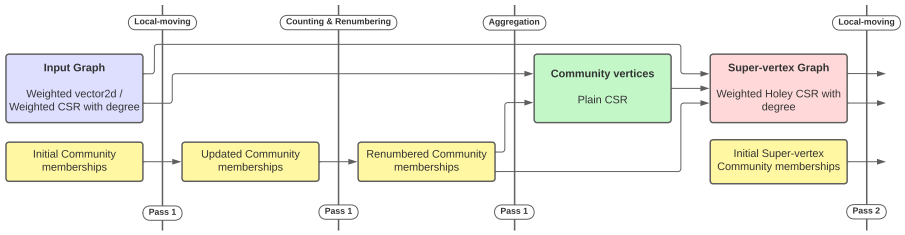

Local-moving
Counting & Renumbering
Aggregation
Local-moving
Input Graph
Weighted vector2d / Weighted CSR with degree
Community vertices
Plain CSR
Super-vertex Graph
Weighted Holey CSR with degree
Initial Community memberships
Updated Community memberships
Renumbered Community memberships
Initial Super-vertex Community memberships
Pass 1
Pass 1
Pass 1
Pass 2

Fig. 2: A flow diagram depicting the first pass of GVE-Louvain.

5 Evaluation

5.1 Experimental Setup

System used We use a server that has two 16-core x86-based Intel Xeon Gold 6226R processors running at 2.90 GHz. Each core has an L1 cache of 1 MB, an L2 cache of 16 MB, and a shared L3 cache of 22 MB. The machine has 93.4 GB of system memory and runs on CentOS Stream 8. For our GPU experiments, we employ a system featuring an NVIDIA A100 GPU (108 SMs, 64 CUDA cores per SM, 80 GB global memory) paired with an AMD EPYC-7742 processor (64 cores, 2.25 GHz). This server is equipped with 512 GB of DDR4 RAM and operates on Ubuntu 20.04.

Configuration We employ 32-bit integers for vertex IDs and 32-bit floats for edge weights, but use 64-bit floats for both computations and hashtable values. Our implementation leverages 64 threads to align with the number of cores on the system, unless otherwise specified. For compilation, we use GCC 8.5 and OpenMP 4.5 on the CPU system, while on the GPU system, we use GCC 9.4, OpenMP 5.0, and CUDA 11.4.

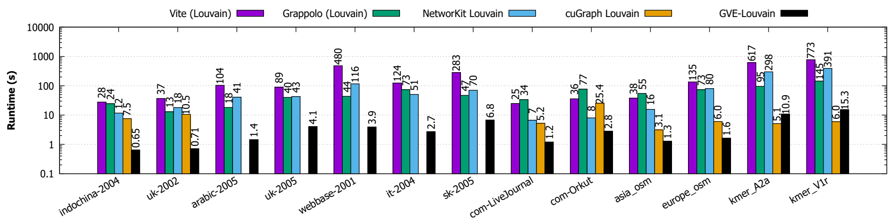
Dataset The graphs used in our experiments are given in Table 1. These are sourced from the SuiteSparse Matrix Collection [14]. The graphs have vertex counts ranging from 3.07 to 214 million and edge counts ranging from 25.4 million to 3.80 billion. We ensure that the edges are undirected and weighted, with a default weight of 1. We avoid using SNAP datasets with ground-truth communities because they are non-disjoint, whereas our work focuses on disjoint communities. Importantly, ground truth communities may represent different or unrelated aspects of the network structure — relying solely on this correlation could overlook other meaningful structures [19].

Table 1: List of 13 graphs from the SuiteSparse Matrix Collection [14] ($* \Rightarrow$ directed). Here, $|V|$ is the vertex count, $|E|$ the edge count (including reverse edges), D_{avg} the average degree, and $|\Gamma|$ the number of communities obtained using GVE-Louvain.

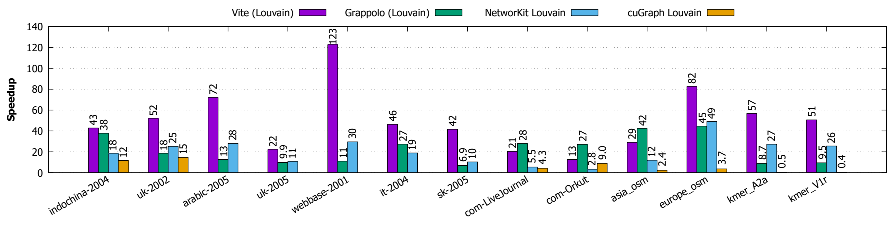
Graph	$\|V\|$	$\|E\|$	D_{avg}	$\|\Gamma\|$	**Graph**	$\|V\|$	$\|E\|$	D_{avg}	$\|\Gamma\|$
Web Graphs (LAW)					**Social Networks (SNAP)**				
indochina-2004*	7.41M	341M	41.0	4.24K	com-LiveJournal	4.00M	69.4M	17.4	2.54K
uk-2002*	18.5M	567M	16.1	42.8K	com-Orkut	3.07M	234M	76.2	29
arabic-2005*	22.7M	1.21B	28.2	3.66K	**Road Networks (DIMACS10)**				
uk-2005*	39.5M	1.73B	23.7	20.8K	asia_osm	12.0M	25.4M	2.1	2.38K
webbase-2001*	118M	1.89B	8.6	2.76M	europe_osm	50.9M	108M	2.1	3.05K
it-2004*	41.3M	2.19B	27.9	5.28K	**Protein k-mer Graphs (GenBank)**				
sk-2005*	50.6M	3.80B	38.5	3.47K	kmer_A2a	171M	361M	2.1	21.2K
					kmer_V1r	214M	465M	2.2	6.17K

5.2 Comparing Performance of GVE-Louvain

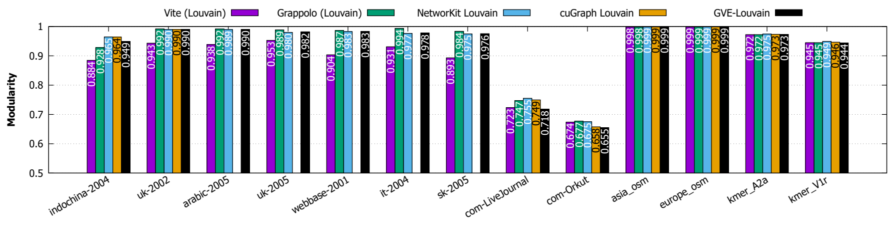
We now compare the performance of GVE-Louvain with Vite (Louvain) [9], Grappolo (Louvain) [12], NetworKit Louvain [24], and cuGraph Louvain [13]. For Vite, we convert the graph datasets to Vite's binary graph format, run it on a single node with threshold cycling/scaling optimization, and measure the reported average total time. For Grappolo, we measure the run it on the same system, and measure the reported total time. For NetworKit Louvain, we use a Python script to invoke `PLM` (Parallel Louvain Method), and measure the total time reported with `getTiming()`. To test cuGraph's Louvain algorithm, we write a Python script that configures the Rapids Memory Manager (RMM) to use a pool allocator for fast memory allocations. We then execute `cugraph.louvain()` on the loaded graph. For each graph, we measure the runtime and the modularity of the obtained communities (as reported by each implementation), performing five runs to calculate an average. When using cuGraph, we discard the runtime of the first run to ensure that subsequent measurements accurately reflect RMM's pool usage without the overhead of initial CUDA memory allocation.



(a) Runtime in seconds (logarithmic scale) with *Vite (Louvain)*, *Grappolo (Louvain)*, *NetworKit Louvain*, *cuGraph Louvain*, and *GVE-Louvain*



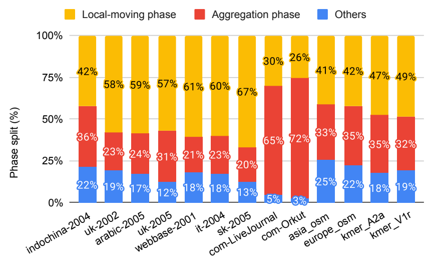
(b) Speedup of *GVE-Louvain* with respect to *Vite (Louvain)*, *Grappolo (Louvain)*, *NetworKit Louvain*, and *cuGraph Louvain*.



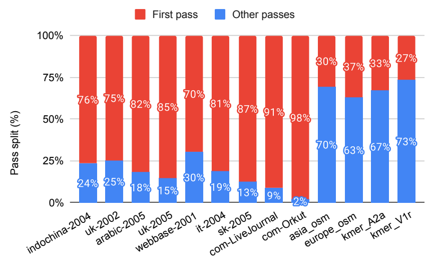
(c) Modularity of communities obtained with *Vite (Louvain)*, *Grappolo (Louvain)*, *NetworKit Louvain*, *cuGraph Louvain*, and *GVE-Louvain*.

Fig. 3: Runtime in seconds (logarithmic scale), speedup, and modularity of communities obtained with *Vite (Louvain)*, *Grappolo (Louvain)*, *NetworKit Louvain*, *cuGraph Louvain*, and *GVE-Louvain* for each graph in the dataset.

Figure 3(a) shows the runtimes of Vite (Louvain), Grappolo (Louvain), NetworKit Louvain, cuGraph Louvain, and GVE-Louvain on each graph in the dataset. cuGraph's Louvain algorithm fails to run on *arabic-2005*, *uk-2005*, *webbase-2001*, *it-2004*, and *sk-2005* graphs because of out-of-memory issues. On the *sk-2005* graph, GVE-Louvain finds communities in 6.8 seconds, and thus achieve a processing rate of 560 million edges/s. Figure 3(b) shows the speedup of GVE-Louvain with respect to each implementation mentioned above. GVE-Louvain is on average $50\times$, $22\times$, $20\times$, and $5.8\times$ faster than Vite, Grappolo, NetworKit Louvain, and cuGraph Louvain respectively. Figure 3(c) shows the modularity of communities obtained with each implementation. GVE-Louvain on average obtains 3.1% higher modularity than Vite (especially on web graphs), and 0.6% lower modularity than Grappolo and NetworKit (especially on social networks with poor clustering), and 2.6% higher modularity than cuGraph Louvain (primarily because cuGraph Louvain failed to run on graphs that are well-clusterable).

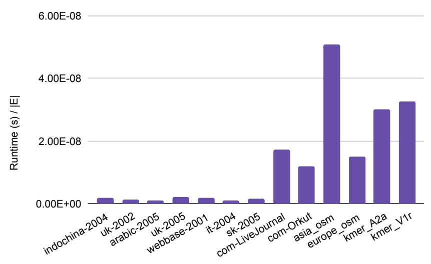


(a) Phase split

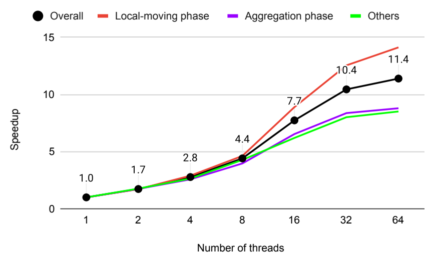


(b) Pass split

Fig. 4: Phase split of *GVE-Louvain* shown on the left, and pass split shown on the right for each graph in the dataset.



(a)



(b)

Fig. 5: (a) Runtime $/|E|$ factor with *GVE-Louvain* for each graph in the dataset. (b) Overall speedup of *GVE-Louvain*, and its various phases (local-moving, aggregation, others), with increasing number of threads (in multiples of 2).

5.3 Analyzing Performance of GVE-Louvain

The phase-wise and pass-wise split of GVE-Louvain is shown in Figures 4(a) and 4(b) respectively. Figure 4(a) indicates that GVE-Louvain spends most of the runtime in the local-moving phase on *web graphs*, *road networks*, and *protein k-mer graphs*, while it devotes majority of the runtime in the aggregation phase on *social networks*. The pass-wise split (Figure 4(b)) indicates that the first pass dominates runtime on high-degree graphs (*web graphs* and *social networks*), while subsequent passes prevail in execution time on low-degree graphs (*road networks* and *protein k-mer graphs*).

On average, 49% of GVE-Louvain's runtime is spent in the local-moving phase, 35% is spent in the aggregation phase, and 16% is spent in other steps (initialization, renumbering communities, looking up dendrogram, and resetting communities) of the algorithm. Further, 67% of the runtime is spent in the first pass of the algorithm, which is the most expensive pass due to the size of the original graph (later passes work on super-vertex graphs) [27].

We also observe that graphs with lower average degree (*road networks* and *protein k-mer graphs*) and graphs with poor community structure (such as `com-LiveJournal` and `com-Orkut`) have a larger runtime$/|E|$ factor, as shown in Figure 5(a).

5.4 Strong Scaling of GVE-Louvain

Finally, we measure the strong scaling performance of GVE-Louvain. To this end, we adjust the number of threads from 1 to 64 in multiples of 2 for each input graph, and measure the overall time taken for finding communities with GVE-Louvain, as well as its phase splits (local-moving, aggregation, others), five times for averaging. The results are shown in Figure 5(b). With 32 threads, GVE-Louvain obtains an average speedup of $10.4\times$ compared to running with a single thread, i.e., its performance increases by $1.6\times$ for every doubling of threads. Scaling is limited due to the various sequential steps/phases in the algorithm. At 64 threads, GVE-Louvain is impacted by NUMA effects, and offers speedup of only $11.4\times$.

6 Conclusion

This paper presented our parallel multicore implementation of the Louvain algorithm — a high quality community detection method, which, as far as we are aware, stands as the most efficient implementation of the algorithm on multicore CPUs. Here, we explored 9 different optimizations, including 4 novel ones aimed at enhancing the aggregation phase, which collectively, significantly improve the performance of both the local-moving and the aggregation phases of the algorithm. Future work could focus of designing fast community detection algorithms that enable interactive updation of community memberships of vertices in large dynamic graphs.

References

1. A. Aldabobi, A. Sharieh, and R. Jabri. An improved louvain algorithm based on node importance for community detection. *JATIT*, 100(23):1–14, 2022.
2. A. Bhowmick, S. Vadhiyar, and V. PV. Scalable multi-node multi-gpu louvain community detection algorithm for heterogeneous architectures. *CCPE*, 34(17):1–18, 2022.
3. A. Bhowmik and S. Vadhiyar. HyDetect: A Hybrid CPU-GPU Algorithm for Community Detection. In *IEEE HiPC*, pages 2–11, Goa, India, Dec 2019.
4. V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech.: Theory Exp.*, 2008(10):P10008, Oct 2008.
5. U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity clustering. *IEEE TKDE*, 20(2):172–188, 2007.
6. C. Cheong, H. Huynh, D. Lo, and R. Goh. Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs. In *Proceedings of the 19th Euro-Par*, pages 775–787, Berlin, Heidelberg, 2013. Springer-Verlag.

7. M. Fazlali, E. Moradi, and H. Malazi. Adaptive parallel Louvain community detection on a multicore platform. *Microprocessors and microsystems*, 54:26–34, Oct 2017.
8. N. Gawande, S. Ghosh, M. Halappanavar, A. Tumeo, and A. Kalyanaraman. Towards scaling community detection on distributed-memory heterogeneous systems. *Parallel Computing*, 111:102898, 2022.
9. S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, and A.H. Gebremedhin. Scalable distributed memory community detection using vite. In *2018 IEEE HPEC*, pages 1–7, 2018.
10. S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin. Distributed louvain algorithm for graph community detection. In *IEEE IPDPS*, pages 885–895, Vancouver, British Columbia, Canada, 2018.
11. S. Gregory. Finding overlapping communities in networks by label propagation. *New Journal of Physics*, 12:103018, 10 2010.
12. M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo. Scalable static and dynamic community detection using Grappolo. In *IEEE HPEC*, pages 1–6, Waltham, MA USA, Sep 2017.
13. S. Kang, C. Hastings, J. Eaton, and B. Rees. cuGraph C++ primitives: vertex/edge-centric building blocks for parallel graph computing. In *IEEE IPDPS Workshops*, pages 226–229, 2023.
14. S. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The SuiteSparse matrix collection website interface. *The Journal of Open Source Software*, 4(35):1244, Mar 2019.
15. A. Lancichinetti and S. Fortunato. Community detection algorithms: a comparative analysis. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, 80(5 Pt 2):056117, Nov 2009.
16. H. Lu, M. Halappanavar, and A. Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel computing*, 47:19–37, Aug 2015.
17. M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. Community detection on the GPU. In *IEEE IPDPS*, pages 625–634, Orlando, Florida, USA, May 2017.
18. M. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.
19. L. Peel, D. B. Larremore, and A. Clauset. The ground truth about metadata and community detection in networks. *Science advances*, 3(5):e1602548, 2017.
20. H. Qie, S. Li, Y. Dou, J. Xu, Y. Xiong, and Z. Gao. Isolate sets partition benefits community detection of parallel louvain method. *Scientific Reports*, 12(1):8248, 2022.
21. S. Sahu. GVE-Louvain: Fast Louvain Algorithm for Community Detection in Shared Memory Setting. *arXiv preprint arXiv:2312.04876*, 2023.
22. J. Shi, L. Dhulipala, D. Eisenstat, J. Lacki, and V. Mirrokni. Scalable community detection via parallel correlation clustering, 2021.
23. C.L. Staudt and H. Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE TPDS*, 27(1):171–184, 2015.
24. C.L. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.
25. V. Traag. Faster unfolding of communities: Speeding up the Louvain algorithm. *Physical Review E*, 92(3):032801, 2015.
26. L. Waltman and N. Eck. A smart local moving algorithm for large-scale modularity-based community detection. *The European physical journal B*, 86(11):1–14, 2013.
27. C. Wickramaarachchi, M. Frincu, P. Small, and V. Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *IEEE HPEC*, pages 1–6, Waltham, MA USA, 2014.