HyPC-Map: A Hybrid Parallel Community Detection Algorithm Using Information-Theoretic Approach

Md Abdul M Faysal*, Shaikh Arifuzzaman*, Cy Chan†, Maximilian Bremer†, Doru Popovici†, John Shalf†
*University of New Orleans, †Lawrence Berkeley National Laboratory,
{mfaysal, smarifuz}@uno.edu, {cychan, mb2010, dtpopovici, jshalf}@lbl.gov

***Abstract*—Community detection has become an important graph analysis kernel due to the tremendous growth of social networks and genomics discoveries. Even though there exist a large number of algorithms in the literature, studies show that community detection based on an information-theoretic approach (known as *Infomap*) delivers better quality solutions than others. Being inherently sequential, the Infomap algorithm does not scale well for large networks. In this work, we develop a hybrid parallel approach for community detection in graphs using Information Theory. We perform extensive benchmarking and analyze hardware parameters to identify and address performance bottlenecks. Additionally, we use cache-optimized data structures to improve cache locality. All of these optimizations lead to an efficient and scalable community detection algorithm, HyPC-Map, which demonstrates a 25-fold speedup (much higher than the state-of-the-art map-based techniques) without sacrificing the quality of the solution.**

***Index Terms*—Community Detection; Parallel Algorithms; Information-Theory; Map Equation; MDL; Graphs;**

1. Introduction

Discovering community in graphs (e.g., biological and social networks) is an important problem in many scientific domains [1]–[5]. Due to the huge growth of network size, scalable algorithms are required [6]–[10]. A wide variety of applications extensively use community discovery–examples include finding similar proteins, detecting anomalous behavior in cyber-security domain, finding critical point/entity in rumour propagation or infectious disease spreading, and classifying groups in social and business networks based on their activities [4], [11], [12].

While the problem of community discovery has a rich literature [1]–[6], [8], [13]–[17], it has attracted high attention lately because of the tremendous growth of social (e.g., human contacts, friendship on social media, disease spreading), biological (e.g., protein interaction, genomics), and other graph-related applications. The huge volume of data that needs to be processed necessitates the development of parallel computational strategies both in homogeneous and heterogeneous computing platforms. In their studies, Lancichinetti et al. [11] and Aldecoa et al. [18] identified the information-theoretic approach of community discovery, i.e., *Infomap* delivers better quality than many other community detection algorithms. Among those algorithms, Louvain method [3] has gained a greater amount of attention for parallelization despite its *resolution limit* problem [19]. The high-quality solution given by serial Infomap [11] has motivated us to develop a highly parallel and scalable implementation of Infomap. Our contributions in this paper are as follows: (i) We have designed a parallel information-theoretic community discovery with clever heuristics to tackle the inherent sequential nature and low scalability of the algorithm. (ii) Our approach is hybrid, i.e., we combined both distributed-memory and shared-memory parallelism. It demonstrates better speedup than relevant literature (e.g., [7], [16], [20]). (iii) We have performed extensive benchmarking and analyzed memory subsystems to use cache-optimized data structures resulting in efficient compute kernels. (iv) We achieve better speedup than state-of-the-art techniques without sacrificing the solution quality (less than 2% impact on modularity and conductance). Our algorithm demonstrates $25\times$ speedup compared to the sequential Infomap by Rosvall [21].

2. Information-Theoretic Approach for Community Detection

Rosvall et al. [4] first proposed the approach of discovering community by using Shanon's minimum entropy theorem [22] to compress the information generated by a dynamic process (random walk) on a network. Lancichinetti et al. [11] named Rosvall's approach of community discovery as *Infomap*. Rosvall called his devised optimization function as *the map equation* (eq. 1).

$$L(M) = q_{\curvearrowright} H(Q) + \sum_{m_i \in M} p^i_{\circlearrowright} H(\rho^i) \quad (1)$$

Equation 1 has two parts, the first part $q_{\curvearrowright} H(Q)$ of the right side of the equation represents the movement of the random walk between the modules whereas the second part $\sum_{m_i \in M} p^i_{\circlearrowright} H(\rho^i)$ represents the movement of the random walk within a module. The term $q_{\curvearrowright}$ is the probability of the random walk exiting a module. The term $H(Q)$ represents the average code length of the movements between the modules. In the second part of the equation, the term $p^i_{\circlearrowright}$ stands for the stay probability (vertex visit probability + exit probability) of the random walk within module m_i. The term $H(\rho^i)$ is the average code length within a module. The term ρ^i is the

probability distribution of the code of module m_i. For any vertex p, the vertex visit rate, i.e., the PageRank [23] p_α can be computed taking teleportation τ into account.

2.1. Sequential Infomap Algorithm

Algorithm 1: Sequential Infomap

Data: A graph $G(V, E)$, $N \leftarrow |V|$
Result: Set of communities M, where $|M| \ll N$
1 m_i, i[th] module
2 q_i, exit probability of module m_i
3 γ, minimum threshold for code length improvement
4 L_{old}, code length of previous iteration
5 L, code length of current iteration
6 Initialize vertex visit rate, $p_{v_i} \leftarrow 1/N$
7 Compute ergodic vertex visit rate, p_{v_i} by PageRank
8 $M \leftarrow \{\forall m_i | (v_\alpha \in m_i) \& (v_\alpha \notin m_j), \forall v_\alpha \in V\}$
9 **for** $i = 1$ *to* $|M|$ **do**
10 | Calculate exit probability, q_i
11 **end**
12 Initial code length, $L \leftarrow L(M)$
13 **do**
14 | $L_{old} \leftarrow L$
15 | **for** $j = 1$ *to* N **do**
16 | | Select randomly each vertex, v_j
17 | | $m_{new} \leftarrow FindBestModule(v_j)$
18 | | Compute L cumulatively
19 | **end**
20 | $CreateSuperNode(M)$
21 | **for** $j = 1$ *to* $|M|$ **do**
22 | | Select randomly each SuperNode, m_j
23 | | $m_{new} \leftarrow FindBestModule(m_j)$
24 | | $m_j \leftarrow \{m_{new} | \forall v_j \in m_j\}$
25 | | Compute L cumulatively
26 | **end**
27 **while** $(L_{old} - L) > \gamma$
28 **return** $|M|$

Algorithm 1 describes the working procedure of Infomap. Lines $1-5$ list the notation used inside the algorithm. Line $6-7$ compute the vertex visit rate (i.e., PageRank) using the power iteration method. The algorithm begins with the number of communities equals to the number of vertices $N \leftarrow |V|$. M represents the set of the modules, m_i represents a single module/community. Initially, m_i has only one member vertex, but as the algorithm progresses, m_i may get zero or more than one vertex. M consists of all the modules (line 8). A vertex v can have a single community membership at a time. Lines $9-11$ compute the initial exit probability q_i for the module m_i. Line 12 calculates initial code length following equation 1. Lines $13-27$ describe community discovery procedure continuing for multiple iterations. Line 14 preserves the current code length at the beginning of an iteration. Every vertex in the vertex set V is picked up in a random order in an iteration (lines $15-16$). Function *FindBestModule* returns the neighboring module of a vertex that minimizes the code length most of all of the neighborhood modules (line $17-18$). Function *CreateSuperNode* takes the set of modules M as argument to create supernode objects consisting of one or more vertices (line 20). Finding the best module in supernode level is conducted in lines $21-25$. This continues until the change in code length falls below a certain user-defined threshold γ (line 27). The return value of the algorithm is the number of discovered communities (line 28).

3. HyPC-Map: Hybrid Parallel Community Discovery using Infomap

3.1. Overview of the Algorithm

HyPC-Map can be divided into the following majors steps as per the actual computation sequence: (i) Calculating the ergodic node visit frequency (PageRank) by OpenMP parallelism. (ii) Finding best community of its subgraph in parallel by each MPI process using OpenMP threads. (iii) Synchronization of the discovered communities for the subgraph belonging to each MPI process. (iv) Creating supernodes of its modules by each MPI process using OpenMP parallelism. (v) Finding best community of its supernodes in parallel by each MPI process using OpenMP threads. (vi) Synchronizing the community membership of the vertices of the supernodes belonging to each MPI process.

Algorithm 2 presents the design of the hybrid memory parallel *Infomap*. Lines $1-7$ list the notations used in the algorithm. Lines $8-9$ describe calculating *PageRank* by power iteration in parallel using t OpenMP threads. Line 10 initializes the set of modules M, where each vertex of V has its own module at the very beginning. Lines $11-12$ calculate exit probability q_i in parallel using t OpenMP threads. Line 14 initializes the code length. Lines $15-37$ do the community discovery in multiple iterations until the change in code length falls below a certain threshold γ. For each process p, the corresponding range of vertices is computed (line $17-18$) in parallel from the list of *Active* vertices (details in section 3.3). Metis [24] edge-cut partitioner is used for workload balance across the processes. Community discovery for vertices takes place in lines $19-22$ and community discovery for supernodes takes place in lines $29-33$ using t number of OpenMP threads inside each MPI process by the help of the function *FindBestModule*. This function takes input of a vertex or a supernode and returns the module that minimizes the code length most. Function *CreateSuperNode* in line 26 describes creating supernode objects in parallel consisting of one or more vertices from the set of modules M. The synchronization of the community memberships of the vertices takes place in line 25 and 36. Line 38 returns the number of communities $|M|$ after algorithm converges.

3.2. Challenges in Distributing Computation/Data

While distributing computation and data among processing units, our map-based approach demonstrates the following challenges and problems – (i) *Vertex bouncing problem:* The notion behind this problem is when two vertices having strong affinity are distributed to two different processes, each of the vertices tries to move to the community of the

Algorithm 2: Hybrid Infomap

Data: A graph $G(V, E)$, $N \leftarrow |V|$
Result: Set of communities M, where $|M| \ll N$

```
1  m_i, i^th module
2  q_i, exit probability of module m_i
3  γ, minimum threshold for code length improvement
4  L_old, code length of previous iteration
5  L, code length of current iteration
6  P, total MPI processes spawned
7  t, total OpenMP threads spawned
8  Initialize vertex visit rate, p_{v_i} ← 1/N
9  Compute ergodic vertex visit rate, p_{v_i} by PageRank
     in t − way parallel
10 M ← {∀m_i|(v_α ∈ m_i)&(v_α ∉ m_j), ∀v_α ∈ V}
11 for m_i = 1 to |M| in t − way parallel do
12   | Calculate exit probability, q_i
13 end
14 Initial code length, L ← L(M)
15 do
16   | L_old ← L
17   | for process p = 1 to P in parallel do
18   |   | Compute vertex indices range [v_start, v_end]
     |   |   from Active vertices list
19   |   | for j = [v_start, v_end], t − way parallel do
20   |   |   | Select randomly each vertex, v_j
21   |   |   | m_new ← FindBestModule(v_j)
22   |   |   | Compute L cumulatively
23   |   | end
24   | end
25   | Synchronize m_new ∈ M across P
26   | CreateSuperNode(M) in t − way parallel
27   | for process p = 1 to P in parallel do
28   |   | Compute SuperNode indices [m_start, m_end]
     |   |   from Active SuperNodes list
29   |   | for j = [m_start, m_end], t − way parallel do
30   |   |   | Select randomly each SuperNode, m_j
31   |   |   | m_new ← FindBestModule(m_j)
32   |   |   | m_j ← {m_new|∀v_j ∈ m_j}
33   |   |   | Compute L cumulatively
34   |   | end
35   | end
36   | Synchronize m_new ∈ M across P
37 while (L_old − L) > γ
38 return |M|
```

other vertex. This causes a non-converging oscillation of the vertices. (ii) *Inconsistent update ordering:* We consider a synchronous parallel approach. Maintaining uniformity of community assignment during synchronization is challenging. This happens because of different synchronization orders by different processes. Consequently, a vertex may have inconsistent modular state across different MPI processes affecting the solution quality. (iii) *Inactive vertices:* In the initial few iterations, most of the vertices change communities and find their stable place (community). Those vertices become *inactive* for the later iterations. Fewer and fewer vertices remain *active* to continue changing communities in subsequent iterations until convergence. This observation leads us to the conclusion that in every iteration, considering all of the vertices in the network for community update incurs redundant computation and wastes computational resources. All these issues are discussed in detail in [20].

3.3. Solution Strategies: Our Heuristics

Solution to *Vertex Bouncing* Problem: To prevent the issue arising from vertex bouncing problem, we adopted *numeric ordering* of the community/module id of each vertex during the synchronization step (lines 25 and 36 of algorithm 2). To understand how it works, suppose, vertex u moves to the community of vertex v, i.e., $u \rightarrow v$ in process P_1 and the opposite move, i.e., $v \rightarrow u$ happens in process P_2. The way we take the final decision of accepting and discarding a move is to first check the numeric values of the current community ids of the vertices u and v. Then, select the move from lower id community to higher id community.

Solution to *Inconsistent Update Ordering*: For maintaining uniform community assignment for vertices distributed across all of the processes, we have taken the heuristic of *priority-based* community assignment during synchronization phase (lines 25 and 36 of algorithm 2). In this scheme, the decision to community assignment for a specific vertex is taken and broadcast by the owner process of a vertex. All other processes will honor the community assignment information received irrespective of their own information regarding that vertex.

Solution to *Inactive Vertices* Problem: There is no deterministic way to decide which vertices will be active or which vertices will be inactive in the immediate next iteration. It is empirically observed that the vertices that change their communities in an iteration will likely change their communities in the immediate next iteration. Additionally, the neighbors of those vertices may become active too. A list is maintained for those *active* vertices (lines 18 and 28 of algorithm 2).

4. Experimental Settings & Evaluations

We have implemented our algorithm using C++ programming language, MPI, and OpenMP frameworks. We have evaluated our algorithm based on the quality of the discovered community and the scalability of our parallel implementation. We compare the results to other information-theoretic approaches as well as HipMCL [8] (a parallel implementation of Markov Clustering algorithm).

Computational Infrastructure: We have run our experiments on the Cori supercomputer owned by National Energy Research Scientific Computing Center (NERSC) and LONI QB2 [25] compute cluster.

4.1. Optimizing Computational Kernels

The major compute kernels of *HyPC-Map* without multi-threading and cache-optimization are illustrated in figure 1 with the percentage of their run time for different networks. It is evident that *FindBestModule* kernel is the most time-consuming part of the algorithm. It takes as much as 89% of

TABLE 1: Performance micro-benchmark of insertion and read operations between c++ map vs unordered_map

Number of entries	Insertion map (µs)	Insertion unordered_ map (µs)	Read map (µs)	Read unordered_ map (µs)
2048	1904	1284	70	58
4096	3991	2586	110	123
8192	8499	5139	230	239
16384	16927	9764	462	465
32768	34916	19197	887	902
65536	75827	37914	1689	1810
131072	166398	76855	3936	3608

the execution time (Orkut network). This kernel having other necessary computations, extensively uses C++ STL map (key-value) which is a major contributor to the execution time. From our micro-benchmark analysis listed in table 1, we see a significant difference in insertion performance between the RB tree-based STL map and the hash table (cache-friendly if hash function is good) based STL unordered_map that avoids expensive tree-traversal during insertion operation. This change resulted in a performance improvement for the *FindBestModule* kernel from 1095 seconds to 1030 seconds (Orkut network). OpenMP multithreading leads to significant optimization of this kernel as evident from figure 2. The execution time further reduces to 240 seconds by using 10 OpenMP threads per MPI process.

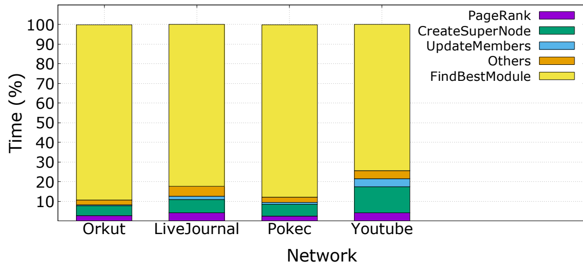


Figure 1: The percentage (%) breakdown of run time for the unoptimized operational kernels of the distributed-memory *HyPC-Map*. *FindBestModule* kernel consumes major portion of the execution time.

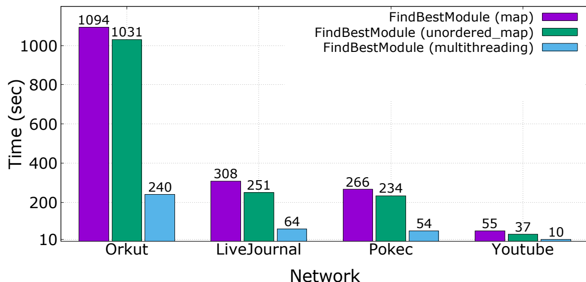


Figure 2: Runtime improvement of *FindBestModule* by using cache-friendly data structures (STL unordered_map) and OpenMP multi-threading (10 threads per MPI process)

4.2. Quality Assessment of the Community

We have used the metrics *Modularity*, *Conductance*, and *convergence MDL* for comparing the quality of the detected communities against [26].

Convergence of the Objective Function: The objective function (eq. 1) of Infomap minimizes the MDL. In distributed implementation, there is a possibility of premature convergence as observed by [7]. The outcome we achieved by optimizing the objective function is very close to the MDL found in [26]. In figure 3a, we have shown the final MDL value after convergence. The difference in MDL is very insignificant in all cases with a minimum difference of 0.08% (Amazon) to a maximum of 3% (Wiki-topcats).

Modularity: We observe from figure 3b, the values of modularity vary insignificantly for the network datasets of our experiments. We see no change in modularity for a different number of cores for the DBLP network and the LiveJournal network, and less than 2% of the difference for 1280 cores for the Orkut network.

Conductance: We observe no difference in *conductance* between 1 core and 1280 cores executions for the DBLP network and the LiveJournal network from figure 3c. For the Orkut network, the *conductance* value is different by less than 2% for 1280 cores.

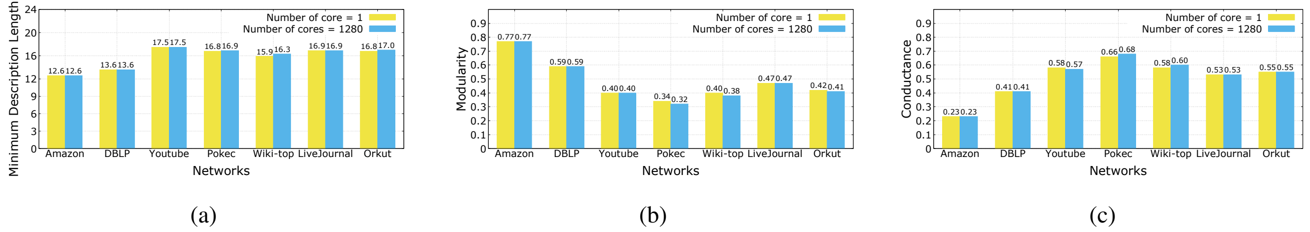
Normalized Mutual Information (NMI): NMI can be used to measure the scalability in terms of quality for a different number of processing cores. In table 3, we report the consistency of the quality for different number of cores using synthetic networks with a known truth partition. Since NMI requires a known truth partition, we used static synthetic graphs from *MIT GraphChallenge network datasets* [27].

4.3. Parallel Performance

Speedup Gain: In table 4, we show the speedup of our implementation against the original Infomap [4] in column 3 and against the sequential *HyPC-Map* in column 2. The parallel experiments are conducted using up to 64 compute nodes of the QB2 [28] server, each node running 2 MPI processes and each MPI process spawning 10 OpenMP threads (1 thread per core). To the best of our knowledge, our parallel implementation obtained significantly better speedup than state-of-the-art information-theoretic approaches [16], [29]. Moreover, we have higher speedup for larger networks than the smaller ones. For larger networks such as *Wiki-topcats*, *Soc-pokec* and *Orkut*, the speedup gain is 10.52, 12.52, and 16.16 respectively against the sequential *HyPC-Map* (2nd column). We observe even higher speedup, reaching as high as $\sim 25\times$ for the *LiveJournal* network and $\sim 21.4\times$ for the *Orkut* network against the original Infomap shown in column 3. This demonstrates the benefit of using optimized data structures and efficient compute kernels.

Scalability Analysis: Figure 4 illustrates the run time comparison of our implementation for 3 large networks. For the *Orkut* network, the run time of 2836 seconds for a single core reduces to 176 seconds for 1280 cores. For the *LiveJournal* network, the run time reduces to 104.7 seconds from 840 seconds.

Comparison with State-of-the-Art Techniques: We



(a) (b) (c)

Figure 3: Illustration of the quality of the discovered communities in terms of (a) MDL, (b) Modularity, and (c) Conductance. The value of the quality metrics are displayed on top of the histogram bars for the respective number of processing core(s) and network configurations

TABLE 2: Scalability of HyPC-Map in terms of the quality metrics: Modularity and Conductance

Network	Modularity 1	20	40	80	160	320	640	1280	Conduct. 1	20	40	80	160	320	640	1280
Amazon	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.23	0.23	0.23	0.23	0.23	0.23	0.23	0.23
DBLP	0.59	0.59	0.59	0.59	0.59	0.59	0.59	0.59	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41
LiveJournal	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.53	0.53	0.53	0.53	0.53	0.53	0.53	0.53
Orkut	0.42	0.38	0.40	0.40	0.42	0.42	0.41	0.41	0.55	0.51	0.55	0.55	0.54	0.56	0.54	0.56

TABLE 3: Scalability of HyPC-Map in terms of Normalized Mutual Information (NMI) for a different number of cores

Network	# Vertices	# Edges	NMI 1	20	40	80	160	320	640	1280
SG_50000	50000	1011755	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90
SG_500000	500000	10160671	0.85	0.85	0.86	0.86	0.87	0.87	0.87	0.88
SG_2000000	2000000	40670978	0.84	0.84	0.84	0.85	0.86	0.85	0.87	0.87

TABLE 4: Speedup comparison with original Infomap [21] by Rosvall et al. [4] (column 3), and with sequential HyPC-Map (column 2) on various social and information networks.

Network	Speedup (vs sequential HyPC-Map)	Speedup (vs original Infomap)
Amazon	2.78	8.79
DBLP	3.66	7.00
Youtube	4.58	9.43
LiveJournal	8.19	25.11
Wiki-topcats	10.52	16.06
Soc-pokec	12.52	20.67
Orkut	16.16	21.42

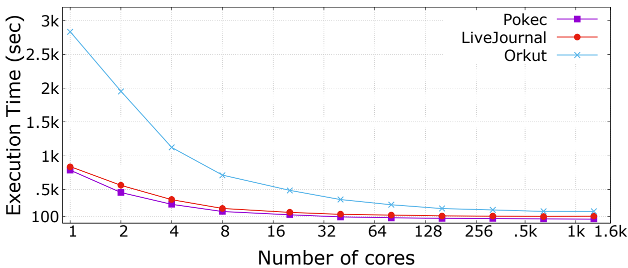


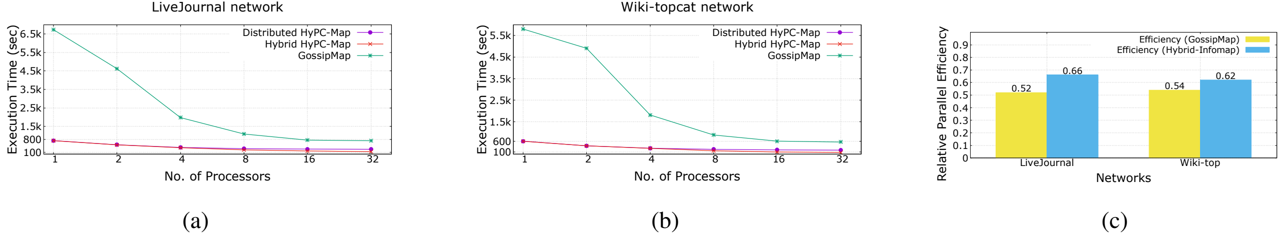
Figure 4: Illustrating the scalability and reduction in execution time with 3 different networks. For instance, the execution time reduces for *Orkut* network from 2836 seconds to 176 seconds using 1280 cores.

compared *HyPC-Map* with state-of-the-art techniques in table 5 and listed the strengths and the weaknesses of the state-of-the-arts. Despite using 4096 processing units, the maximum speedup reported for *Distributed Infomap* [16], [29] is 6.02×. The implementation is not publicly available. Therefore, we compared *HyPC-Map* with *GossipMap*. We ran the experiments in our local computing server due to a large number of dependencies for *GossipMap*. Figures 5a and 5b show run time comparison for 2 sample networks among *GossipMap*, single-threaded distributed *HyPC-Map* and multi-threaded distributed *HyPC-Map*. The single core run times are 730 seconds and 6734 seconds respectively for *HyPC-Map* and *GossipMap* for the *LiveJournal* network. *HyPC-Map* demonstrates higher relative parallel efficiency (ε_r) than *GossipMap* as shown in figure 5c. Here, $\varepsilon_r = \frac{p_1 T(p_1)}{p_2 T(p_2)}$, where $T(p_1)$ and $T(p_2)$ are the execution times for p_1 and p_2 parallel units respectively.

Comparison with another Community Discovery Strategy: *HipMCL* [8] is a parallel community discovery algorithm based on the Markov Clustering technique (MCL) [15]. HipMCL does not perform well for scale-free networks. As observed in table 6, *HyPC-Map* outperforms *HipMCL* in memory requirement and run time performance for scale-free networks following power-law degree distribution. The large networks of our experiments could not be processed by HipMCL using 128 GB memory of the NERSC Cori haswell node as memory limit exceeded (MLE) due to the storage of intermediate results generated during the computation.

TABLE 5: Comparison of *HyPC-Map* with state-of-the-art techniques

Work Name	**Type**	**Strength**	**Weakness**
Infomap [4]	Sequential	High accuracy	Computationally expensive
RelaxMap [26]	Shared-memory parallelism	High accuracy	Scalability limited to single node
Gossipmap [7]	Asynchronous distributed-memory parallelism	Asynchronous	Scalability up to 128 parallel units
Distributed Infomap [20]	Synchronous distributed-memory parallelism	Scales to 512 processors	Speedup up to $\sim$ 5X
Distributed Infomap [29]	Synchronous distributed-memory parallelism	Scales to $\sim$ 4k processors	Speedup up to $\sim$ 6X
HyPC-Map	Synchronous hybrid memory parallelism	High accuracy & speedup	



(a) (b) (c)

Figure 5: Run time and relative efficiency ε_r comparison for different networks between *GossipMap* and *HyPC-Map*

TABLE 6: Execution performance comparison between *HipMCL* [8] and *HyPC-Map*. MLE: Memory Limit Exceeded

Network	**HyPC-Map (sec)** 1 Compute Node	**HipMCL (sec)** 1 Compute Node	4 Compute Nodes	16 Compute Nodes
Amazon	3.50	85.18	50.61	20.24
DBLP	3.90	278.64	166.42	57.35
Youtube	21.14	MLE	9251.05	2545.89
soc-Pokec	82.05	MLE	37014.52	10792.05
Orkut	235.0	MLE	MLE	35715.63

Additionally, HipMCl takes a substantially large amount of time to process real-world scale-free networks.

5. Related Work

Several parallel implementations exist for the modularity-based community detection. OpenMP implementations such as [30] by Bhowmick et al. and [31] by Bader et al. are available. Hiroaki et al. [32] and Zhang et al. [33] demonstrated a fast modularity-based community detection by avoiding searching all the vertices in each iteration. GPU-based parallel Louvain are presented in the studies of Cheong et al. [34] and Naim et al. [35]. A combination of the Louvain algorithm and the breadth-first search (BFS) is used by Staudt et al. [36], [37] for distributed-memory parallelization. Zeng et al. [38] designed parallel Louvain with workload balancing. The works by Sattar et al. [39] and Sayan et al. [40] demonstrated a distributed+shared memory (MPI + OpenMP) based work on the Louvain algorithm. The work by Peixoto et al. [41] and [17] are shared-memory based parallel implementation of statistical inference method. Distributed memory-based parallel works are done by Uppal et al. [42], [43]. There is less effort in developing parallel algorithms for *Information-theoretic* approach. Bae et al. [26] developed an OpenMP-based algorithm and a distributed memory algorithm [7] using the graphlab framework [44]. The distributed memory parallel work by Faysal et al. [20] shows scalability of up to 512 MPI processes. The works by Zeng et al. [16], [29] on distributed memory parallel implementation show limited speedup despite using thousands of processors. The work we present in this paper addresses the parallelization scheme for high scalability while maintaining accuracy as good as the sequential Infomap algorithm.

6. Conclusions

HyPC-Map integrates the benefits of both distributed- and shared-memory parallelism to achieve higher scalability performance than state-of-the-art techniques. Additionally, our algorithm is more efficient while using a single processing unit than other prominent map-based algorithms [7], [21]. *HyPC-Map* achieves significantly higher parallel performance than other map-based parallel algorithms in literature. While achieving such speedup, HyPC-Map does not fall short in maintaining the quality. The modularity, conductance, and MDL scores demonstrate high quality of detected communities, which are desirably similar to sequential *Infomap*. We believe *HyPC-Map* may prove useful in analyzing emerging large-scale social, information and scientific networks.

Acknowledgment

This work has been partially supported by US Department of Energy/Berkeley Lab/University of California Subcontract Award # 7551418 (Prime Award # DE-AC02-05CH11231) and Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RDA-25.

References

[1] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002. [Online]. Available: https://www.pnas.org/content/99/12/7821

[2] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices," *Physical Review E*, vol. 74, no. 3, Sep 2006. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.74.036104

[3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct 2008. [Online]. Available: http://dx.doi.org/10.1088/1742-5468/2008/10/P10008

[4] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *Proceedings of the National Academy of Sciences*, vol. 105, no. 4, pp. 1118–1123, 2008. [Online]. Available: https://www.pnas.org/content/105/4/1118

[5] M. E. J. Newman, "Spectral methods for community detection and graph partitioning," *Physical Review E*, vol. 88, no. 4, Oct 2013. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.88.042822

[6] T. P. Peixoto, "Efficient monte carlo and greedy heuristic for the inference of stochastic block models," *Phys. Rev. E*, vol. 89, p. 012804, Jan 2014. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.89.012804

[7] S.-H. Bae and B. Howe, "Gossipmap: a distributed community detection algorithm for billion-edge directed graphs," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[8] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic Acids Research*, vol. 46, no. 6, pp. e33–e33, 01 2018. [Online]. Available: https://doi.org/10.1093/nar/gkx1313

[9] S. Zhou, K. Lakhotia, S. G. Singapura, H. Zeng, R. Kannan, V. K. Prasanna, J. Fox, E. Kim, O. Green, and D. A. Bader, "Design and implementation of parallel pagerank on multicore platforms," in *The 21st Annual IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*. IEEE Computer Society, 2017, pp. 1–6, graph Challenge Student Innovation Award. [Online]. Available: https://doi.org/10.1109/HPEC.2017.8091048

[10] S. Arifuzzaman, M. Khan, and M. Marathe, "Fast parallel algorithms for counting and listing triangles in big graphs," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 1, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3365676

[11] A. Lancichinetti and S. Fortunato, "Community detection algorithms: A comparative analysis," *Phys. Rev. E*, vol. 80, p. 056117, Nov 2009. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.80.056117

[12] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining Social-Network Graphs*, 2nd ed. Cambridge University Press, 2014, p. 325–383.

[13] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.70.066111

[14] B. Karrer and M. E. J. Newman, "Stochastic blockmodels and community structure in networks," *Phys. Rev. E*, vol. 83, p. 016107, Jan 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.83.016107

[15] S. M. Van Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, University of Utrecht, 2000.

[16] J. Zeng and H. Yu, "Effectively unified optimization for large-scale graph community detection," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 475–482.

[17] M. A. M. Faysal and S. Arifuzzaman, "Fast stochastic block partitioning using a single commodity machine," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3632–3639.

[18] R. Aldecoa and I. Marìn, "Exploring the limits of community detection strategies in complex networks," *Scientific Reports*, vol. 3, p. 2216, Jul 2013. [Online]. Available: https://doi.org/10.1038/srep02216

[19] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007. [Online]. Available: https://www.pnas.org/content/104/1/36

[20] M. A. M. Faysal and S. Arifuzzaman, "Distributed community detection in large networks using an information-theoretic approach," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 4773–4782.

[21] M. Rosvall and C. T. Bergstrom, "Source code of the original infomap." [Online]. Available: https://www.mapequation.org/code_old.html

[22] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x

[23] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, APR 1998. [Online]. Available: http://dx.doi.org/10.1016/S0169-7552(98)00110-X

[24] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[25] "Louisiana optical network infrastructure." [Online]. Available: http://hpc.loni.org/resources/hpc/system.php?system=QB2

[26] S.-H. Bae, D. Halperin, J. West, M. Rosvall, and B. Howe, "Scalable flow-based community detection for large-scale network analysis," in *2013 IEEE 13th International Conference on Data Mining Workshops*, Dec 2013, pp. 303–310.

[27] . S. P. C. D. with Known Truth Partitions, "Mit graphchallenge data sets." [Online]. Available: https://graphchallenge.mit.edu/data-sets

[28] L. HPC, "Qb2 cluster." [Online]. Available: http://www.hpc.lsu.edu/docs/guides.php?system=QB2

[29] J. Zeng and H. Yu, "A distributed infomap algorithm for scalable and high-quality community detection," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 4:1–4:11. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225137

[30] S. Bhowmick and S. Srinivasan, *A Template for Parallelizing the Louvain Method for Modularity Maximization*. New York, NY: Springer New York, 2013, pp. 111–124. [Online]. Available: https://doi.org/10.1007/978-1-4614-6729-8_6

[31] D. A. Bader and K. Madduri, "Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE Computer Society, 2008, pp. 1–12. [Online]. Available: https://doi.org/10.1109/IPDPS.2008.4536261

[32] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "Fast algorithm for modularity-based graph clustering," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, ser. AAAI'13. AAAI Press, 2013, p. 1170–1176.

[33] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritizing iterative computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1884–1893, Sep. 2013.

[34] C. Y. Cheong, H. P. Huynh, D. Lo, and R. S. M. Goh, "Hierarchical parallel algorithm for modularity-based community detection using gpus," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 775–787.

[35] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo, "Community detection on the gpu," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 625–634.

[36] C. L. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 180–189.

[37] ——, "Engineering parallel algorithms for community detection in massive networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 171–184, Jan 2016.

[38] J. Zeng and H. Yu, "Parallel modularity-based community detection on large-scale graphs," in *2015 IEEE International Conference on Cluster Computing*, Sep. 2015, pp. 1–10.

[39] N. S. Sattar and S. Arifuzzaman, "Parallelizing louvain algorithm: Distributed memory challenges," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing (DASC 2018), Athens, Greece, August 12-15, 2018*, 2018, pp. 695–701. [Online]. Available: https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00122

[40] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 885–895.

[41] T. Peixoto, "graph-tool." [Online]. Available: https://graph-tool.skewed.de/

[42] A. J. Uppal and H. H. Huang, "Fast stochastic block partition for streaming graphs," *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–6, 2018.

[43] A. J. Uppal, G. Swope, and H. H. Huang, "Scalable stochastic block partition," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–5.

[44] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: https://doi.org/10.14778/2212351.2212354