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. However, the serial Infomap algorithm does not scale well for large graphs and due to the inherent sequential nature, parallelizing Infomap method is non-trivial. In this paper, we develop a hybrid parallel approach for community detection in graphs using Information Theory. We perform extensive microbenchmarking and analyze hardware parameters to identify and address performance bottlenecks. We also use cache optimized data structures to improve cache locality. All these optimizations lead to a more efficient and scalable community detection algorithm, HyPC-Map, which demonstrates a 16-fold speedup (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; Hybrid Parallel; Map Equation; MDL;**

1. Introduction

Discovering communities in graphs (e.g., social, biological, and communication networks) is an important problem in many scientific domains [1], [2], [3], [4], [5]. Due to the huge growth of network size, scalable algorithms and tools are needed [6], [7], [8]. Thus, the problem of community discovery has gained considerable attention among High-Performance Computing (HPC) researchers and practitioners [9], [10]. A wide variety of applications extensively use community discovery as a computing kernel. For example, in genomics application, finding protein groups with similar structural features [8], [11] uses community detection kernel. Other applications [12], [13] include detecting anomalous behavior in cyber-security domain, finding critical point/entity in rumour propagation or infectious disease spreading, classifying groups in social and business networks based on their activities [4], etc.

While the problem of discovering community has a rich literature [1], [14], [2], [3], [4], [5], [15], [6], it has attracted high attention lately because of the 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 sheer volume of the data that needs to be processed necessitate the development of parallel computational strategies both in homogeneous and heterogeneous computing platforms. Lancichinetti et al. presented a survey of prominent community detection algorithms in [12]. An information-theoretic approach, *Infomap* was identified as one of the better algorithms in terms of solution quality. In recent years, there have been efforts for parallelizing several community detection algorithms. Among those algorithms, Louvain method [3] has gained perhaps the highest amount of attention despite this method's *resolution limit* problem [16]. Markov clustering technique [11] is another algorithm that has been parallelized [8]. The high-quality solution given by serial Infomap [12] motivates us to develop scalable parallel method based on information-theoretic approach. Our contributions in this paper are as follows.

- We design a scalable parallel algorithm for community detection in large graphs based on information-theoretic approach. We tackle the inherent sequential nature of Infomap approach by developing heuristics, which enable us to achieve the similar high quality solution as serial algorithm.
- We combined both distributed-memory and shared-memory based parallelism to design our hybrid parallel algorithm. Our hybrid parallel algorithm demonstrates better scalability than related methods (e.g., [7], [9], [17]) in the literature.
- We performed extensive microbenchmarking and analyzed memory subsystems to identify and address performance bottlenecks. Such analyses allow us to optimize the algorithm design. We use cache-optimized data structures to improve cache locality.
- Our algorithm scales to large graphs. We achieve better speedups than the state-of-the-art techniques based on information theory without sacrificing the solution quality. We experimented on a range of social and scientific graph datasets. Our algorithm demonstrates up to 16x speedup comparing to its' sequential-self and up to 25x speedup comparing to the sequential implementation by Rosvall [18].

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 [19] to compress the information generated by a dynamic process (random walk) on a network. Lancichinetti et al. [12] in their comparative study, referred to Rosvall's approach of community detection as *Infomap*.
Rosvall devised an optimization function that he referred to as the *map equation* (1).

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

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 other part $\sum_{m \in M} p^i_{\circlearrowright} H(\rho^i)$ 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 $H(Q)$ is the entropy of the module, i.e., the average code length of the movements between the modules where Q stands for the probability distribution of the module entering rate. In the other part of the equation, the term $p^i_{\circlearrowright}$ stands for the stay probability of the random walk within module i. The parameter $p^i_{\circlearrowright}$ can be calculated by summing the vertex visit probability (PageRank) and the exit probability of the random walk for that module. The term $H(\rho^i)$ is the entropy within the module, i.e., the average code length of the random walk within module i, which is named as module code length. The term ρ^i is the probability distribution of the code of module i. For any vertex p, the vertex visit rate, i.e., the PageRank p_α can be computed taking teleportation τ into account.

2.1. Sequential Infomap Algorithm

Algorithm 1 shows the working procedure of Infomap. Lines $1-5$ present the notation used inside the algorithm. Line $6-9$ discusses the procedure of computing the vertex visit rate (i.e. PageRank) using the power iteration method. The algorithm starts with an initial number of communities equals to the number of vertices $N \leftarrow |V|$. M represents the set of the modules, m_i represents a single community. Initially, m_i has only one member vertex, but as the algorithm progresses, m_i may get more than one vertices to many. Line 10 tells us the initial number of modules M consists of all the modules m_i, and a vertex v belongs to a single module at a time. Line 12 computes the initial exit probability for a module q_i. Lines $15-28$ are the core part of the algorithm where the community optimization phase takes place in multiple iterations. In line 16, the current code length is preserved. Every vertex in the vertex set V is picked in a random order for community optimization (lines $17-19$). Considering all of the neighborhood modules of a vertex, the one that minimizes the code length most (line 20) is selected (or it stays in its current module if none of the neighborhood modules decreases the current code length). Line 22 discusses about creating the super node. Similar to the individual vertex module optimization phase, the super node level optimization phase is conducted in lines $23-27$. This continues until the change in code length falls below a certain user-defined threshold parameter γ. The return value of the algorithm is the discovered communities (line 29).

Algorithm 1 Sequential Infomap

Require: A graph $G(V, E)$, V total vertices, E total edges, $N \leftarrow |V|$
Ensure: $M : M \leqslant N$, M is the total number of communities, $M \ll N$
1: m_i, i[th] module
2: q_i, exit probability of module m_i
3: γ, minimum threshold for codelength improvement
4: L_{old}, codelength of previous iteration
5: L, codelength of current iteration
6: **for** $i = 1$ to N **do**
7: Calculate initial vertex visit rate $p_{v_i} \leftarrow 1/N$
8: Compute vertex visit rate p_{v_i} by power iteration
9: **end for**
10: Declare initial total module $M \leftarrow \{\forall m_i | (v_\alpha \in m_i) \& (v_\alpha \notin m_j), V = \sum_1^N v\}$
11: **for** $m_i = 1$ to M **do**
12: Calculate exit probability q_i
13: **end for**
14: Calculate initial codelength $L \leftarrow L(M)$
15: **do**
16: $L_{old} \leftarrow L$
17: **for** $j = 1$ to N **do**
18: Pick every vertex v_j in a random order
19: $m_{new} \leftarrow findNewBestModule(v_j)$
20: Compute L cumulatively
21: **end for**
22: Update $M \leftarrow createSuperNodes()$
23: **for** $j = 1$ to M **do**
24: Pick every super node m_j in a random order
25: $m_{new} \leftarrow findNewBestModule(m_j)$
26: Compute L cumulatively
27: **end for**
28: **while** $(L_{old} - L) > \gamma$
29: **return** M

3. Parallelizing Map-based Method for Community Detection

The sequential Infomap algorithm is known as one of the better algorithms for community detection [12], [13]. However, the serial algorithm demonstrates limited scalability. Parallelizing Infomap algorithm is non-trivial due to a number of challenges.

3.1. 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 other vertex. (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. (iii) *Inactive vertices:* The community assignment process for the vertices continues for multiple iterations. In the initial few iterations, most of the vertices change communities and then find their stable place (community). After a vertex moves to some community and stays in that communities for a few subsequent iterations, it is more likely to stay in that community until the program finishes. In later iterations, fewer to fewer community updates take place. This observation leads us to the conclusion that in every iteration, considering all of the vertices in the network for community update incurs redundant activities that waste computational resources. We need to distinguish the vertices and pick the ones that are more likely to change their communities in subsequent iterations. All these issues are discussed in detail in [17].

3.2. Solution Strategies: Our Heuristics

We developed the following heuristics to overcome the challenges mentioned above.

3.2.1. Solution to Vertex Bouncing Problem. To prevent the issue arising from vertex bouncing problem, we adopted *numeric ordering* during the synchronization step. To understand how it works, suppose, $u \rightarrow v$ in process P_1 and $v \rightarrow u$ in process P_2 is accepted and the other one is discarded. The way we take the decision of accepting and discarding a move is to first check the numeric value of the ids of the current communities of the vertices u and v. Then, select the move from lower id community to higher id community. For instance, between the two communities C_u and C_v, if the numeric value of the id for community C_u is less than the numeric value of the id for community C_v, we allow the move of u to C_v, but the move of v to C_u is ignored.

3.2.2. Solution to Inconsistent Update Ordering. For maintaining uniform community assignment for vertices across all of the processes, we have taken the heuristic of *priority-based* community assignment. In this scheme, the decision of community assignment for a specific vertex is taken by the owner process of a vertex. Every process will honor the community assignment information received for vertices processed by other MPI processes. This is a simple yet effective approach.

3.2.3. Solution to Inactive Vertices Problem. To prevent recomputing the community assignment for the vertices that are unlikely to change their current communities, we need to distinguish those vertices from the other vertices. We name those vertices as *Inactive Vertices* that are unlikely to change their communities in the current iteration. The vertices that may move to different communities in the next iteration, we call those *Active Vertices*. It is important to note that, 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. Therefore, we need to have a prediction list of the vertices that may become active before an iteration starts. The prediction list can be made from the outcome of the community assignment of the immediate previous iteration. The prediction list contains the vertices changing their communities in the previous iteration as well as their immediate neighbors.

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

We design our hybrid parallel algorithm, *HyPC-Map*, based on the map equation discussed in Section 2 and the heuristics in Section 3. We then optimize our algorithm based on microbenchmarking and hardware profiling.

4.1. Overview of the Algorithm

HyPC-Map can be divided into the following majors steps. The sequence of the steps is maintained during actual computation.

1) Computing the ergodic node visit frequency (PageRank) for the network vertices inside each MPI process with combination of OpenMP based parallelism.
2) Community optimization for each of the subgraphs inside each MPI process in parallel. Each MPI process spawns t-number of OpenMP threads to compute community for t-number of vertices concurrently. This continues for multiple iterations.
3) Each MPI process exchanges the information of the discovered communities of its subgraph vertices with the rest of the MPI processes. This phase is called the synchronization step.
4) Every MPI process creates super node of the modules it has with combination of OpenMP parallelism. The edges across distinct vertices of any two supernodes are replaced by a single edge of accumulated edge-weights.
5) Each MPI process spawns t-number of threads to find communities of t-number of super node concurrently. This continues for multiple iterations until no more improvement of the code length happens.
6) Each MPI process synchronizes the community membership for the supernode(s) it processed with other MPI processes for maintaining uniform community information across the MPI processes.

Algorithm 2 shows the design of the hybrid memory parallel *Infomap*. In lines $6-7$, P and t represent the number of distributed memory entity *MPI process* and shared memory

Algorithm 2 Hybrid Infomap

Require: A graph $G(V, E)$, $N \leftarrow |V|$, V total vertices set, E total edges set
Ensure: $M : M \leqslant N$, M is the total number of communities, $M \ll N$
1: m_i, i[th] module
2: q_i, exit probability of module m_i
3: γ, minimum threshold for codelength improvement
4: L_{old}, codelength of previous iteration
5: L, codelength of current iteration
6: P, number of MPI processes spawned
7: t, number of OpenMP threads spawned

8: **for** $i = 1$ to N in $t - way$ parallel **do**
9: Calculate initial vertex visit rate $p_{v_i} \leftarrow 1/N$
10: Compute vertex visit rate p_{v_i} by power iteration
11: **end for**
12: Declare initial total module $M \leftarrow \{\forall m_i | (v_\alpha \in m_i) \& (v_\alpha \notin m_j), V = \sum_1^N v\}$

13: **for** $m_i = 1$ to M in $t - way$ parallel **do**
14: Calculate exit probability q_i
15: **end for**
16: Calculate initial codelength $L \leftarrow L(M)$

17: **do**
18: $L_{old} \leftarrow L$
19: **for** process p = 1 to P **do**
20: Compute vertex indices range $[v_{start}, v_{end}]$
21: **end for**
22: **for** $j = v_{start}$ to v_{end} in $t - way$ parallel **do**
23: Pick every active vertex v_j in a random order
24: $m_{new} \leftarrow findNewBestModule(v_j)$
25: Compute L cumulatively
26: **end for**
27: Synchronize $m_{new} \in \{M | m_{new}, m_{old} \in M\}$ across P
28: Update $M \leftarrow createSuperNode()$ in $t - way$ parallel
29: **for** process p = 1 to P **do**
30: Compute super node indices $[m_{start}, m_{end}]$
31: **end for**
32: **for** $j = m_{start}$ to m_{end} in $t - way$ parallel **do**
33: Pick every active super node m_j in a random order
34: $m_{new} \leftarrow findNewBestModule(m_j)$
35: $m_j \leftarrow \{m_{new} | \forall v_j \in m_j\}$
36: Compute L cumulatively
37: **end for**
38: Synchronize $m_{new} \in \{M | m_{new}, m_{old} \in M\}$ across P
39: **while** $(L_{old} - L) > \gamma$
40: **return** M

entity *OpenMP thread* respectively. Lines $8 - 16$ are similar to lines $6 - 14$ in algorithm 1 except here we have used t number of threads to compute the vertex visit rate (line $9-10$) and exit probability (line 14) in each compute node. This is step 1 of the high-level overview. Lines $17 - 39$ in algorithm 2 do the computation of the community optimization. For each process p, the corresponding part of the original graph for community discovery is computed (line 20). The number of partition of the original graph is equal to the number of process P. For workload balance across the processes, metis [20] edge-cut partitioner is used. For the subgraph received by each process p, t-number of OpenMP threads are spawned to find the best community in an iteration for t-number of vertices in parallel. This is step 2. The synchronization of the community for each vertex takes place in line 27 and 38 in algorithm 2 (step 3). Line 28 describes the super node creation phase which is step 4 of the high-level overview. Lines $29 - 31$ describe the partitioning of the super node across multiple processes P. Lines $32 - 37$ are similar to the community optimization phase in individual vertex level except the entity processed here is super node instead of an individual vertex. This is step 5. At the end of each iteration, the synchronization (line 38) of the supernode communities occurs (step 6). Steps 5 and 6 continues until the change in code length falls below a certain threshold τ. At the end of the algorithm, the resultant number of communities M is returned (line 40).

4.2. Optimizing Computational Kernels

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 implementation of *HyPC-Map* can be broken down into five major kernels. Those kernels are illustrated in figure 1 as stack graph according to the percentage of their run time for different networks. The *PageRank* kernel computes the ergodic node visit frequency using PageRank [21] algorithm by power iteration method. The *Convert2SuperNode* kernel creates the super nodes from a group of vertices with the same module id with edge between super nodes are the combined edges with a sum of edge-weights. The *UpdateMembers* kernel updates the member vertices for each of the modules/communities after each iteration. The *CommunityOptimization* kernel comprises of finding the community in the individual vertex phase and in the super node phase. From figure 1, it is evident that *CommunityOptimization* kernel is the most time-consuming part of the algorithm. It takes as much as 89% of the execution time (Orkut network) to as much as 74% of the execution time (Youtube network). The performance of the data structure being used inside the kernel is a major contributor to the performance. To efficiently store, search, and process the community memberships and the corresponding flows of the neighboring adjacency list of a vertex, key-value (map) is used instead of the array (e.g., vector) data structure. One vertex may have the neighborhood vertices that belong to distinct communities or more than one neighboring vertices may collapse into (as the algorithm progresses) one community. That makes

key-value data structure a better choice for memory storage and efficient search. The map data structure built in C++ STL internally uses *RB-tree* that maintains the ordering of the key. On the contrary, the unordered_map data structure uses array and hashing for storing the data. From our micro-benchmark analysis listed in table 1, we saw a significant difference in insertion performance between map and unordered_map. The time taken for insertion of the entry (each entry is a key-value pair of integer and double value) is almost half for unordered_map compared to map for different numbre of entries. This observation led us to performance improvement for the *CommunityOptimization* kernel from 1095 seconds to 1030 seconds for the Orkut network, and from 55 seconds to 37 seconds for the Youtube network. However, the *CommunityOptimization* kernel still dominates. In an attempt to further optimize this kernel, we choose OpenMP multithreading with discretion on critical zones inside the kernel. This leads us to a massive performance gain as evident from figure 2. The execution time of *CommunityOptimization* kernel reduces to 240 seconds from the 1030 seconds for the Orkut network. Similar kinds of performance gain are observed for other networks as illustrated in figure 2.

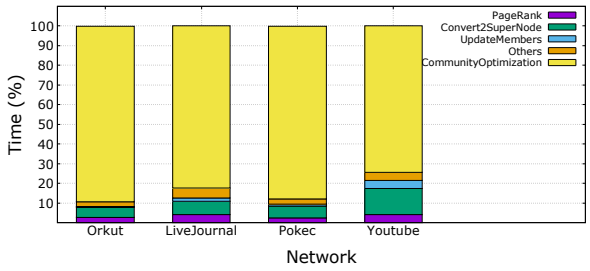


Figure 1: The operational kernels of our initial implementation of the distributed (MPI) *Infomap* algorithm. The percentage (%) breakdown of runtime for 4 different networks is illustrated. It is evident that the *CommunityOptimization* kernel is the major portion of the execution time.

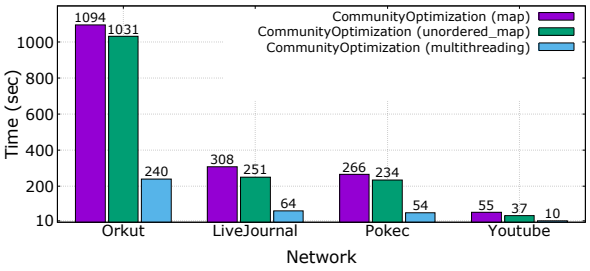


Figure 2: Runtime improvement of the operational kernel of Infomap because of cache-optimized kernel and multithreading

5. Experimental Setting

We implement our algorithm using C++ programming language, MPI, and OpenMP frameworks. We used g++ compiler for building the code. The program supports network in compressed sparse row (CSR) format.

5.1. Computational Infrastructure

We used large compute clusters to perform experimentation on our algorithm. We used both the Cori supercomputers owned by National Energy Research Scientific Computing Center (NERSC) and LONI QB2 [22] system to experiment on our algorithms.

5.2. Datasets Used in Experiments

We have used networks of different sizes ranging from about a million vertices and edges to over a hundred million edges. These are real-world networks with the power-law degree distribution, a common characteristic of the social networks. Such power-law distribution also allows us to test our algorithm in the worst-case scenario. All of the networks in our datasets are collected from SNAP [23].

6. Performance Evaluation

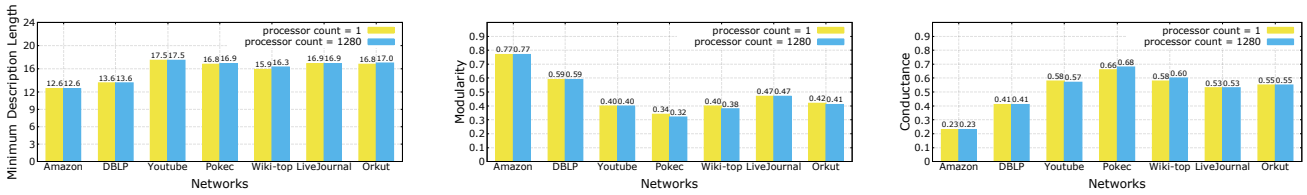
We evaluate our algorithm based on both solution quality and parallel scalability, and compare to other information-theoretic approaches as well as Markov Clustering algorithm (MCL).

6.1. Quality Assessment of Discovered Communities

Infomap delivers better quality of communities among state-of-the-art techniques as observed by several benchmark studies [12], [24]. For quality comparison of the detected communities, we used Modularity, Conductance, and convergence MDL value. We compare our result with sequential Infomap.

6.1.1. Convergence of the Objective Function. Our objective function of Infomap minimizes the MDL. It is challenging to improve the MDL in distributed implementation. In case of distributed implementation, there is a possibility of premature convergence resulting in an outcome of less minimization of the MDL as observed by [7]. The outcome we achieved by optimizing the objective function 1 is very close to the MDL improvement found in [25]. In figure 3a, we have shown the final converged value of the MDL. The difference in MDL is very insignificant in all the cases with a difference of as low as 0.08% (Amazon) to a maximum of 3% (Wiki-topcats). It indicates the detected communities after convergence are similar to that of *RelaxMap* [25].

6.1.2. Modularity. We can see from figure 3b the values of Modularity vary insignificantly for the different networks shown in the figure. We see no change in modularity for a different number of processors for the DBLP network, no change for the LiveJournal network, and less than 2% of the difference for 1280 processors for the Orkut network.



(a) (b) (c)

Figure 3: Illustration of the quality of discovered communities in terms of MDL, modularity, and conductance. The numeric value on top of each histogram bar in each figure demonstrates the values of the quality metrics in pair for 1 vs 1280 processors.

6.1.3. Conductance. From figure 3c we see there is no difference between 1 processor vs 1280 processors executions for the DBLP network and the LiveJournal network. For the Orkut network, the Conductance value is different by less than 2% for 1280 processors. Therefore, we can conclude that the quality of the detected communities of our *Infomap* implementation is scalable to a varying number of processors for different networks.

6.1.4. Normalized Mutual Information. Similar to Modularity and Conductance, NMI can be used to measure the scalability in terms of quality for a different number of processors. In tables 2 and 3, we report the consistency of the quality for different processors using both real-world networks and synthetic networks with known truth partition. Since NMI requires known truth partition, we used static graphs from *MIT GraphChallenge network data sets* [26] as shown in table 3.

6.2. Parallel Performance

6.2.1. Speedup Gain. In table 4, we show the performance gain in terms of speedup. To the best of our knowledge, our parallel implementation obtained better speedup than state-of-the-art information-theoretic approaches. For smaller network such as *Amazon*, *DBLP*, and *Youtube*, the speedup gain is 2.78, 3.66 and 4.58 respectively. For larger networks such as *Wiki-topcats*, *Soc-pokec* and *Orkut*, the speedup gain is 10.52, 12.52, and 16.16 respectively. All of these speedups are significantly higher than state-of-the-art implementations [27], [9]. Additionally, we compare the speedup with the original sequential implementation of *Infomap* [18] by Rosvall et al. [4]. We observe even higher speedup, reaching as high as $\sim$ 25X for the *LiveJournal* network and $\sim$ 21.4X for the *Orkut* network. Both of those are large networks. This demonstrates the benefit of using cache-optimized data structure and efficient community optimization kernel that lower the sequential computation time. The experiments are conducted with a different number of MPI processes ranging from 1 to 128 while each process spawns 10 OpenMP threads. This enables us to use all of the processors available in a single QB2 [28] compute node. The highest speedup comes from using 1280 (128×10) processors.

6.2.2. Scalability Analysis. In figure 4, we illustrate the runtime comparison of our implementation for 3 different networks. For *Orkut* network in figure 4a, the runtime in a single processor of 2836 seconds reduces to 176 seconds for 1280 processors. For *LiveJournal* network in figure 4b, the runtime reduces to 104.7 seconds with 1280 processors from a single processor runtime of 840 seconds. For the *Pokec* network in figure 4c, the runtime is as less as 63 seconds using 1280 processors than the runtime of single processor performance of 787 seconds.

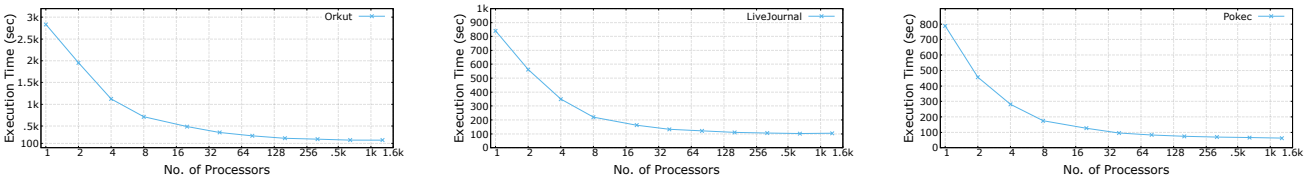
6.2.3. Comparison with state-of-the-art techniques. In table 5, we compared *HyPC-Map* with state-of-the-art techniques. We listed the original *Infomap* and the parallel algorithms developed until the time of this study. Additionally, the strength and the weakness of those works are listed. *GossipMap* and *Distributed Infomap* are good candidates for comparison with *HyPC-Map* as both of them use distributed-memory kernels. Despite using 4096 processing units, the maximum speedup reported for *Distributed Infomap* [27], [9] is $6.02X$, the implementation is not publicly available for comparison. Therefore, we choose to compare with the *GossipMap*. GossipMap reformulates the map equation as an incremental computation that can be updated based on a single vertex movement allowing it to be evaluated locally. It estimates the global MDL using a purely local computation and adopts an asynchronous gossiping protocol to approximate the needed information for each vertex. The reported scalability for this work is up to 128 parallel units. We ran the experiments in our local computing server due to a large number of dependencies for *GossipMap*. We compared *GossipMap* with *HyPC-Map* in its' single-threaded distributed (MPI) form and multi-threaded distributed form. Figure 5 shows runtime and scalability comparison for 3 different networks between *GossipMap* and single-threaded distributed *HyPC-Map* and multi-threaded distributed *HyPC-Map*. From figure 5, we observe *HyPC-Map* uses a highly efficient algorithm. The single-processor run times for *HyPC-Map* are 730, 660, and 600 seconds whereas the single processor run times for *GossipMap* are 6734, 4316, 5800 seconds respectively for the *LiveJournal* network, the *Pokec* network, and the *Wiki-topcat* network. There are differences in runtime using 16 or 32 processing units between distributed form and hybrid form of *HyPC-Map* as reported in table 6. The hybrid form attains better runtime and parallel efficiency

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

Network	Modularity								Conduct.							
	1	20	40	80	160	320	640	1280	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 processors

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



(a) (b) (c)

Figure 4: Illustrating the scalability and reduction in execution time with 3 different networks. For instance, for *Orkut* network (4a), the execution time reduces from 2836 sec to 176 seconds using 1280 processors. Similarly, for *Pokec* network (4c), the execution time reduces from 787 sec to 63 seconds using 1280 processors.

TABLE 4: Speedup comparison with original Infomap [18] by Rosvall et al. [4] (column 3), and with sequential HyPC-Map (column 2) on various social and information networks.

Network	Speedup (vs sequential self)	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

than its' distributed self thanks to the less communication overhead during the synchronization step among a fewer number of distributed processes while using the same number of processing units. Finally, the hybrid form of *HyPC-Map* outperforms *GossipMap* in terms of both execution time and relative parallel efficiency $\varepsilon_r = \frac{p_1 T(p_1)}{p_2 T(p_2)}$. The relative parallel efficiency drops for *GossipMap* in higher percentage with an increasing number of processing units. Here, $T(p_1)$ and $T(p_2)$ are the execution times for p_1 and p_2 parallel units respectively. Table 6 shows the comparison between *GossipMap* and *HyPC-Map* in terms of relative efficiency ε_r for different scenarios. It is important to note that, the target of *HyPC-Map* is to achieve better performance and scalability for fast community discovery. Using a hybrid form enables that while bridging the gap of high communication cost due to synchronization and less quality due to asynchronous community optimization. The comparison of the quality between *GossipMap* and *HyPC-Map* is illustrated in terms of MDL in figure 6 displaying similar values.

6.2.3.1. **Comparison with other community discovery strategies**. HipMCL [8] is a parallel community discovery algorithm that uses the Markov Clustering algorithm (MCL) [11] as its' core. Lancichinetti et al. [12] demonstrated different comparisons among Infomap, Louvain, and MCL in their work based on different benchmarks (e.g., GN, LFR, random graphs). HipMCL, one of the state-of-the-art techniques for clustering, overcomes the performance and memory limitation of MCL as demonstrated by Azad et al. [8]. HipMCL uses SpGEMM (sparse matrix-matrix multiplication) as its' community optimization kernel. The SpGEMM kernel implements a sparse form of SUMMA [29] for distributed computation of the graph in adjacency matrix form with the requirement of the number of MPI processes a perfect square number so that they can be divided into a square grid (e.g., 1×1, 2×2, 4×4). When we compared HipMCL with *HyPC-Map*, we observed *HyPC-Map* outperforms in memory requirement and runtime performance. HipMCL maintains 3 matrices in the SpGEMM kernel which is much higher than what *HyPC-Map* requires. Consequently, bigger networks of our used data sets in this study could not be processed by HipMCL using all available 128 GB memory of the NERSC Cori haswell node due to memory limit exceed (MLE) as listed in table 7. using a single compute node (e.g., *Youtube*, *Orkut*) and 4 compute nodes (e.g., *Orkut*). We observe that HipMCl takes a substantially large amount of time to process real-world social

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 [25]	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 [17]	Synchronous distributed-memory parallelism	Scales to 512 processors	Speedup up to $\sim$ 5X
Distributed Infomap [27]	Synchronous distributed-memory parallelism	Scales to $\sim$ 4k processors	Speedup up to $\sim$ 6X
HyPC-Map	Synchronous hybrid memory parallelism	High accuracy & speedup	

networks listed in table 7 that follows power-law degree distribution. Despite using all of 32 processors of the Haswell compute node and using multiple compute nodes (e.g., 4, 16), HipMCL still falls behind *HyPC-Map's* one compute node runtime. Fortunato et al. [16] discussed the resolution limit problem present in *modularity*-based community detection strategies (e.g., Louvain). A parallel implementation of the *modularity*-based algorithm inherits such a problem along with the additional challenges emerging from parallelization. Lancichinetti et al. [12] demonstrated detailed comparisons between sequential Louvain and Infomap in their work.

7. Related Work

Several parallel implementations exist for the modularity-based approach of the Louvain method. An OpenMP implementation is given by Bhowmick et al. [30]. Hiroaki et al. [31] and Zhang et al. [32] demonstrated a fast modularity-based community detection by avoiding searching all the vertices in each iteration. GPU-based parallel Louvain is presented in the study of Cheong et al. [33] and Naim et al. [34]. A combination of the Louvain algorithm and the breadth-first search (BFS) is used by Staudt et al. [35], [36] for distributed-memory parallelization. Zeng et al. [37] designed parallel Louvain with workload balancing. The works by Sattar et al. [38] and Sayan et al. [39] demonstrated a distributed+shared memory (MPI + OpenMP) based work on the Louvain algorithm. The work by Peixoto et al. [40] and [10] are shared-memory based parallel implementation of statistical inference method. Distributed memory-based parallel works are done by Uppal et al. [41], [42]. There is less effort in developing parallel algorithms for *Information-theoretic* approach. Bae et al. [25] developed an OpenMP-based algorithm and a distributed memory algorithm [7] using the graphlab framework [43]. The distributed memory parallel work by Faysal et al. [17] shows scalability of up to 512 MPI processes. The works by Zeng et al. on distributed memory parallel implementation [27], [9] shows 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 algorithm.

8. Conclusions

We design *HyPC-Map*, a hybrid memory parallel algorithm for community detection using information theory based on *Infomap* method. *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 [18], [7]. *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 of solution–the modularity, conductance, and MDL scores demonstrate high quality of detected communities, which are desirably similar to sequential *Infomap*. *HyPC-Map* may prove useful in analyzing emerging large-scale social and scientific networks.

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

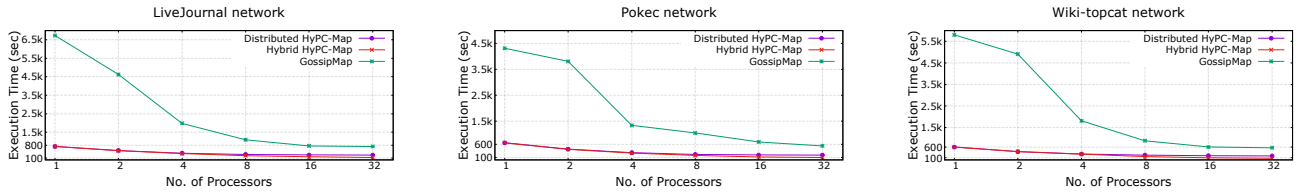
[5] M. E. J. Newman, "Spectral methods for community detection and graph partitioning," *Physical Review E*, vol. 88, no. 4, Oct 2013. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.88.042822

[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] 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.



(a) (b) (c)

Figure 5: Runtime comparison for 3 different networks between GossipMap and HyPC-Map (single-thread distributed and multi-thread distributed)

TABLE 6: Relative efficiency ε_r between *GossipMap* and *HyPC-Map*

Network	p_1	p_2	GossipMap $T(p_1)$	GossipMap $T(p_2)$	GossipMap ε_r	Dist. HyPC-Map $T(p_1)$	Dist. HyPC-Map $T(p_2)$	Dist. HyPC-Map ε_r	HyPC-Map $T(p_1)$	HyPC-Map $T(p_2)$	HyPC-Map ε_r
LiveJournal	16	32	760	727	**0.52**	279	267	**0.52**	177	135	**0.66**
Wiki-topcat	16	32	606	564	**0.54**	200	188	**0.53**	103	83	**0.62**
soc-Pokec	16	32	697	544	**0.64**	198	189	**0.52**	119	96	**0.62**

TABLE 7: 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	HipMCL (sec) 4 Compute Nodes	HipMCL (sec) 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

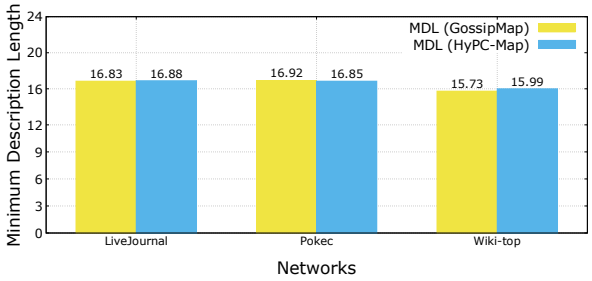


Figure 6: Similar MDL for 3 different networks after the convergence of both the *GossipMap* and *HyPC-Map*.

[10] 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.

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

[12] 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

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

[14] 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

[15] 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

[16] 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

[17] 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.

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

[19] 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

[20] 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

[21] 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

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

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[24] 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

[25] 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.

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

[27] 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

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

[29] R. A. van de Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," USA, Tech. Rep., 1995.

[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] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] ——, "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.

[37] 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.

[38] 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

[39] 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.

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

[41] 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.

[42] 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.

[43] 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