December 2023

Scalable Algorithm Design and Performance Analysis for Graph Motifs Discovery

Md Abdul Motaleb Faysal
University of Nevada, Las Vegas

SCALABLE ALGORITHM DESIGN AND PERFORMANCE ANALYSIS FOR GRAPH MOTIFS DISCOVERY

By

Md Abdul Motaleb Faysal

Bachelor of Science – Computer Science and Engineering
Bangladesh University of Engineering & Technology
2014

Master of Science – Computer Science
University of New Orleans
2020

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2023

UNLV | GRADUATE COLLEGE

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

November 13, 2023

This dissertation prepared by

Md Abdul Motaleb Faysal

entitled

Scalable Algorithm Design and Performance Analysis for Graph Motifs Discovery

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy – Computer Science
Department of Computer Science

Shaikh Arifuzzaman, Ph.D.
Examination Committee Chair

Laxmi Gewali, Ph.D.
Examination Committee Member

Wolfgang Bein, Ph.D.
Examination Committee Member

Cy Chan, Ph.D.
Examination Committee Member

Brian Labus, Ph.D.
Graduate College Faculty Representative

Alyssa Crittenden, Ph.D.
Vice Provost for Graduate Education & Dean of the Graduate College

Abstract

Discovering motifs or structural patterns, such as communities, is a significant graph application utilized for classifying groups in social and business networks, identifying similar proteins, detecting anomalous behavior in the cybersecurity domain, and finding critical entities in rumor propagation or infectious disease spreading. Existing state-of-the-art techniques for community discovery face challenges related to scalability, performance limitations, and methodological inaccuracies. The objective of this doctoral dissertation is to introduce novel parallel algorithms and propose high-performance computing architecture designs to address the performance constraints of current community detection approaches when processing large-scale social and biological data. This research focuses on two main categories of community discovery problems: i) global community discovery and ii) local community discovery. We identify two notable sequential approaches, one from each category, and develop and implement parallel algorithm solutions for both. Our parallel algorithm design for global community discovery achieves a substantial speedup of up to 25× compared to the original sequential approach, leveraging hybrid memory parallelism. Furthermore, we conduct comprehensive benchmarking and performance analysis of software hash accumulation on two prominent community discovery approaches. Based on our findings, we propose a generalized accelerator design for hash accumulation and simulate the proposed architecture, achieving up to a 5.6× speedup. For the local/goal-oriented community discovery approach, we design an innovative parallel algorithm based on shared memory parallelism, demonstrating a remarkable speedup ranging from 20× to 55× for massive graphs with millions of vertices and billions of edges.

Acknowledgements

I express my deepest gratitude to my Ph.D. supervisor, Dr. Shaikh Arifuzzaman, whose brilliance and unwavering support have been instrumental throughout this doctoral journey. His guidance, endless suggestions, and tireless efforts, even during his busy schedule, have been invaluable. I appreciate his commitment to refining my writing and providing assistance during challenging times, not only in the academic realm but also for my overall well-being. Dr. Arifuzzaman's mentorship has played a pivotal role in my successful attainment of this doctoral degree.

I extend my thanks to the Computer Architecture Group (CAG) at Berkeley Lab for facilitating the logistics that led to significant research outcomes. I am particularly indebted to John Shalf, the department head of computer science at Berkeley Lab, for initiating our collaborative research and offering me the position of graduate affiliate. His mentorship and the opportunities provided, including three consecutive summer internships, have significantly shaped my understanding of algorithmic performance on computing architectures.

Dr. Cy Chan, a research scientist of the CAG group at Berkeley Lab and a brilliant mind to work with, shaped the trajectory of my research in a significant extent. I value his mentorship which helped me grow as an aspiring researcher. I am grateful for getting the opportunity for all of those weekly brainstorming sessions where I was awed by his scholarly excellence that set the stepping stones of this doctoral research

I acknowledge the significant contribution of Dr. Maximilian Bremer to my doctoral research. His in-depth technical knowledge never ceases to amaze me. During the tenure of our research collaboration with the CAG group at Berkeley Lab, Dr. Bremer was the person I interacted with mostly for research and technical guidance. He is a brilliant scholar and an amazing person. I feel fortunate to be able to work with and learn from him.

Dr. Doru Thom Popovici's contribution to my doctoral research, especially in shaping the narrative for my defense, has been significant. His valuable feedback and mentorship during my summer 2023 research internship at Berkeley Lab have enhanced both my technical and presentation skills.

I am very much thankful to Dr. Wolfgang Bein - Professor of Computer Science at UNLV, Dr. Laxmi Gewali - Professor of Computer Science at UNLV, and Dr. Brian Labus - Professor of the School of Public Health at UNLV for their approval to serve on my Ph.D. dissertation committee and for their constructive feedback, which greatly contributed to organizing and improving my doctoral work.

I would like to express my gratitude to Farzad Fatollahi-Fard, FPGA computer system engineer of the Computer Science Department at Berkeley Lab and Matthew Avery Toups, former IT Director of the Computer Science Department at the University of New Orleans for extending constant technical support during the tenure of my doctoral research.

Finally, I acknowledge the generous support from various research facilitating bodies. This dissertation project has received partial funding from the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-AC02-05CH11231, the National Science Foundation (NSF) under Award Number 2323533, the Nevada State of Higher Education (NSHE), the Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RDA-25, and the University of New Orleans (UNO) ORSP SCORE award 2019.

Dedication

To my family and friends

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1: Introduction

Graph (Network) is a powerful abstraction for representing underlying relations and structures in large complex systems. Finding structural patterns within a graph is required to analyze entities based on their relations/interactions in social, biological, and communication networks. Some examples include grouping people in social networks based on mutual interests or similar backgrounds, classifying cells or biological units that perform similar kinds of activities in biological networks, e.g., clustering brain cells based on their interconnection and activity to perform a specific operation of the body, classifying proteins that might be responsible for cancer or other diseases, detecting internet anomaly, e.g., detecting fraudulent websites, building efficient product recommendation system by grouping customers based on their purchase habits, connecting research community based on collaboration network and so on.

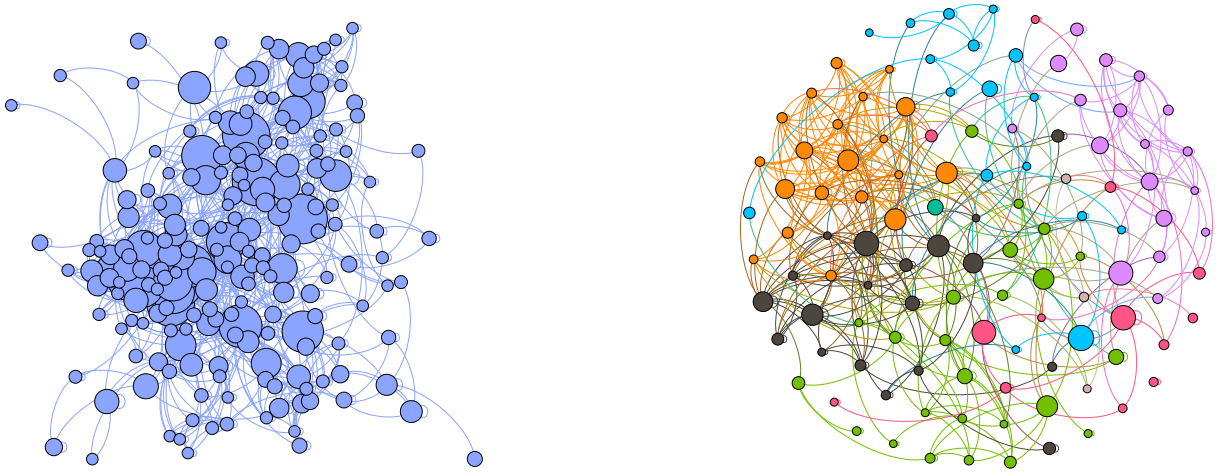
Community discovery is a fundamental technique for finding structural patterns in graphs (e.g., social, biological, and communication networks) and has been widely used in many scientific domains [59, 97, 26, 114, 98]. Scalable algorithms are required [106, 16, 13, 155, 8] to process the massive networks available nowadays. A wide variety of applications extensively use community discovery. Some applications are finding similar proteins, detecting anomalous behavior in the cyber-security domain, finding critical points/entities in rumor propagation or infectious disease spreading, and classifying groups in social and business networks based on their activities [81, 85, 114]. A few definitions pertaining to the problem of community discovery along with an illustrative example are presented below.

Definition 1 (Graph:) *A graph is defined as* $G = (V, E)$*, where* V *is a set of vertices* v *and* E *is a set of edges (links)* (u, v)*, with* $u, v \in V$.

Definition 2 (Community:) *A community in a network is a set of entities that share some closely correlated action/similarity with the other entities of that set. The problem of community detection pertains to seeking community assignment* C_u *for each vertex* $u \in V$ *in graph* G *where the vertices in the same community/group share the same community membership.*

In this dissertation, the terms *graphs* and *networks* will be used interchangeably. Generally, networks exhibit natural divisions into sets of vertices with dense connectivity (edges) within each set and sparse connectivity

across vertices of different sets [107, 55, 96]. The task of identifying these group structures is known as community discovery (see Figure 1.1). Various forms of the community discovery problem exist (see Figure 1.2). Many algorithms [59, 43, 34, 97, 26, 114, 106] focus on finding global disjoint communities, breaking down the entire network into distinct sets of vertices. In contrast, some algorithms [137, 2] explore local overlapping communities, where a vertex may simultaneously belong to multiple communities. In local community discovery, the objective is to identify other member vertices within the community or communities of a given query vertex.



(a) (b)

Figure 1.1: Demonstration of community discovery using a protein-protein interaction network. (a) Representation of a yeast protein-protein interaction network [19]. Each vertex (blue circles) represents distinct proteins, and the arcs (blue lines) denote interactions between proteins. (b) Proteins grouped by community detection based on functional similarities. Proteins within the same group share the same color and exhibit similar biological properties. Visualizations created using Gephi [18].

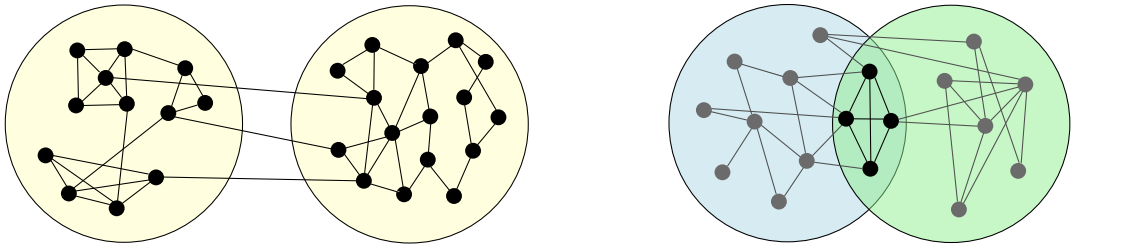


Figure 1.2: Demonstration of the primary categories (global, local) of community discovery. The sub-figure on the left illustrates distinct communities (oval shapes), with each vertex (dark circles) potentially belonging to only one community at a time. The sub-figure on the right illustrates overlapping communities, where certain vertices (highlighted in dark gray) belong to multiple communities simultaneously.

The concept of community discovery is not recent. Many community discovery approaches have been designed more than a decade ago. Since then, the size of the graphs has increased by several orders of magnitude. The computational capability of modern computers also increased manifolds. In recent years, we see supercomputers have emerged with exascale computing capability. We need novel parallel algorithms to utilize modern high-performance computing resources to process the high volume of data with high throughput. The goal of this doctoral dissertation is to design novel parallel algorithms and propose high-performance computing architecture for overcoming performance limitations in existing community detection approaches. The research delves into the two categories (global and local) of community detection problems, identify two outstanding sequential algorithms, one from each category, and design parallel algorithm solution for both of the sequential algorithms. We identify the scalability issues in current community discovery approaches, pose them as research questions, and then discuss the outcomes by answering those questions as the constituent parts of my doctoral dissertation. The research questions are the following.

1. What are the current limitations in existing approaches that perform global community discovery and how do we contribute to overcoming the limitations?
2. How do we address the issue of low speedup from a synchronous parallel community discovery algorithm that is inherently sequential?
3. Many graph kernels are memory-bound. Can we identify the instructions/operations limiting performance and use software-hardware co-design for certain graph operations to overcome low throughput in our community detection algorithm?
4. What are the limitations/challenges in existing local community discovery approaches? How can we overcome those challenges in our research?

1.1 Scope of Research

In this section, we illustrate the scope of my doctoral research. In the latter part of this section, we discuss our solution approaches to the research questions by the order of their listing.

In Figure 1.3, we show the three major components of this doctoral dissertation. Those components are designing a parallel algorithm for global disjoint community discovery, simulating accelerator design for speeding up hash accumulation in community discovery, and designing a parallel algorithm for local overlapping community discovery. There are two phases for global community discovery. First, we choose a community discovery approach known as Infomap [114, 81]. Studies [81, 3] show that Infomap detects community with higher accuracy than other approaches. For our global community detection approaches, we first design a distributed memory Infomap, and then design a hybrid (distributed + shared) memory parallel Infomap that delivers better scalability. Second, we perform extensive profiling and instrumentation

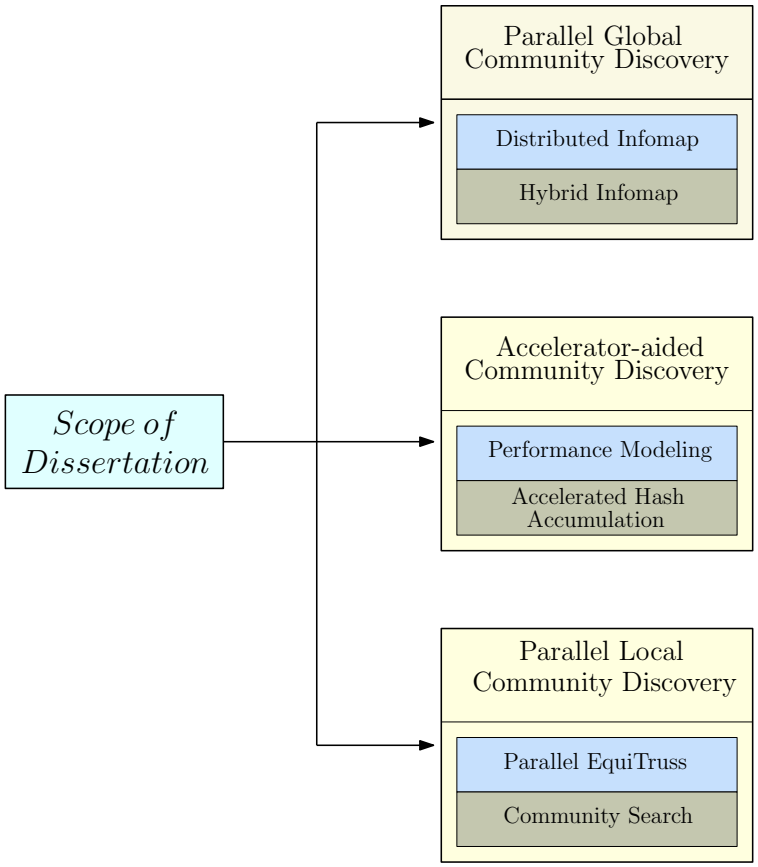


Figure 1.3: The scope of this doctoral dissertation. The research scope encompasses novel parallel algorithmic techniques for local overlapping community discovery, combining heuristics and parallel computing frameworks for global community discovery and software-hardware co-design for accelerator-aided global community discovery

on two parallel community discovery approaches, HipMCL [13], and HyPC-Map [49]. We perform roofline modeling and notice a significant performance gap between expected and true performance. We observe both of these approaches are dependent on a high volume of hash operations. Traditional software hash (key-value) accumulation suffers from low throughput due to branch misprediction and high CPI rate (cycles retired per instruction). Therefore, we propose a generalized accelerator design for fast hash accumulation. Finally, unlike global community discovery, there is not much research conducted on the problem of scalable algorithm design for local community discovery. However, the problem of local community discovery has its own application domain where a user may be interested in the community of a query vertex/entity. K-truss [35] oriented index construction facilitates the efficient formulation of local overlapping communities and query-based community search. Therefore, we design a novel parallel algorithm for k-truss-based index construction and use that index structure for parallel local community search.

Response to Q1 (work published [46]): The global community detection problem has received high attention recently among research communities for the need to process the massive volume of graph data. The challenges

in global community detection algorithms are as follows. i) Most of the approaches for discovering global communities [59, 43, 34, 97, 26, 114] were developed more than a decade ago and did not foresee the massive growth of information. ii) Several parallel algorithms in CPU [25, 14, 124, 154, 128, 129, 147, 118, 58] and GPU [33, 95] platforms have been developed for the modularity maximization-based Louvain [26] community detection approach. Fortunato et al. [57] demonstrated in their study that there is a *resolution-limit problem* in modularity-maximization-based community discovery strategies. There is a highly scalable parallel algorithm called HipMCL [13] for protein clustering based on Markov clustering [43]. HipMCL does not perform well in processing scale-free social networks [49]. iii) Community discovery using an information-theoretic approach [114], known as Infomap delivers a better quality solution in the LFR (Lancichinetti–Fortunato–Radicchi) benchmark [82] as observed by Lancichinetti et al. [81] and Aldecoa et al. [3]. However, the algorithm is highly sequential. There are existing parallel designs that have low scalability [15, 16] and poor speedup despite using up to 4 thousand processing cores [149]. We observe there are scopes of improvement in state_of_the_arts parallel *Infomap* and design a highly scalable distributed memory parallel Infomap [46] algorithm for the CPU platform. Our contributions are the following.

i) We design an MPI-based distributed-memory parallel algorithm for community detection for Infomap. Our design has a similar speedup to this work [149] while using a lower number of processing cores demonstrating better parallel efficiency.

ii) The algorithm follows a synchronous memory parallel approach. Therefore, despite the distributed computation, our algorithm can maintain the quality of the discovered community of the sequential approach. Detailed quality comparisons are presented in our work [46].

iii) We apply problem-specific heuristics to overcome the challenges in distributed graph processing.

iv) We use the Metis partitioner [77] for efficient load balancing resulting in high scalability of up to 512 MPI processes.

Response to Q2 (work published [49]): During our development of the study [46], we observe low scalability due to workload imbalance across MPI processes and high communication overhead during synchronization among the MPI processes. Therefore, we design a hybrid memory parallel algorithm [49] that combines shared-memory and distributed-memory parallelism while reducing synchronization overhead. Our contributions are the following.

i) Our approach is hybrid, i.e., we combine both distributed-memory and shared-memory parallelism. It demonstrates better speedup than relevant literature (e.g., [16, 150, 46]), and achieves 25× speedup compared to the sequential algorithm [114]

ii) We have performed extensive bench-marking and analyzed memory subsystems to use cache-optimized data structures resulting in efficient compute kernels.

iii) We achieve better speedup than state-of-the-art techniques without sacrificing the solution quality (less than 2% impact on modularity and conductance) while achieving scalability of up to 1280 processing cores.

Response to Q3 (work published [51]): In our performance analysis, we decompose the various computational kernels of the Infomap approach. We identify a critical and time-consuming phase in determining the community membership of a vertex, which involves computing and aggregating flow information from neighboring vertices. Both sequential [114] and parallel implementations [15, 16, 46, 49] of Infomap rely on software hash tables to store data about neighboring vertices. However, software hash accumulation is highly resource-intensive and a significant bottleneck in hardware resource utilization, leading to issues such as stalls due to branch misprediction [151]. We show that introducing an accelerator for hash accumulation with fast Content-Addressable Memory (CAM) can effectively address the challenges associated with software hash tables, bridging the gap between achievable and utilized hardware resources. Notably, none of the existing works [114, 15, 16, 149] on Infomap community detection has explored hardware acceleration for speeding up hash operations. Building on Chao et al.'s [151] Accelerator for Hash Accumulation (ASA) designed for SpGEMM computation, we extend and adapt the ASA architecture for use in the context of Infomap. The contributions of our software-hardware co-design are summarized below.

i) We extend the ASA interface introduced by Chao et al. [151] to accelerate SpGEMM computation and showcase its effectiveness in an application involving a substantial number of hash operations. To our knowledge, this represents the first instance where an accelerator is employed to enhance the speed of hash operations for Infomap community discovery.

ii) In the context of the Infomap application, ASA reduces branch misprediction by 59%, the CPI rate by 21%, and the total number of instructions by 24% through the elimination of resource-intensive software hash accumulation and collision handling operations.

iii) We illustrate that the limited capacity of on-chip Content-Addressable Memory (CAM) poses no hindrance in managing large social and biological networks. Our observations indicate that a core-local on-chip CAM size of 8KB can process over 99% of the vertices effectively.

Response to Q4 (work published [54]): In various real-world scenarios, there is often a greater interest in determining the communities to which an entity (a vertex in a graph) belongs, rather than identifying the independent disjoint communities of the entire graph. For example, a user within a social network may be more concerned about the social groups or communities in which they actively participate, rather than all communities present in the network. This entity-centered personalized search is more meaningful, as the communities a user engages with provide valuable social and behavioral context. While the former problem typically applies a global criterion or optimization function to uncover all eligible communities, the latter problem generally constructs and maintains an index-based structure with the aim of retrieving

community subgraphs containing the query vertex. We refer to the latter problem as local or goal-oriented community search. A notable distinction between these problems lies in the fact that in global community discovery, a vertex may belong to only one community at a time (resulting in disjoint communities), whereas in local community discovery, a vertex may belong to more than one community simultaneously (resulting in overlapping communities) (see Figure 1.2). Consequently, we propose a parallel k-truss-induced index construction for local community search. We identify a k-truss-induced community discovery technique [2] capable of detecting local communities in polynomial time. Previous studies primarily explored k-truss-induced local community formation [2, 69] in a serial setting, rendering them unsuitable for large graphs. To the best of our knowledge, our work represents the initial attempt to parallelize this algorithmic approach, accompanied by extensive performance analysis. Our choice of the k-truss-oriented index construction, *EquiTruss*, as the fundamental building block for local community search is motivated by the following considerations.

- The identification of graph motifs based on cliques is excessively restrictive for real-world scenarios, compounded by the fact that it poses a problem that is not polynomially tractable.
- The k-core problem involves finding the maximal subgraph where each vertex has at least k adjacent vertices. Despite being polynomially solvable, the k-core approach suffers from the drawback of lacking cohesion, which is a crucial property for community subgraphs [35].
- The k-truss, serving as a more relaxed variant of the clique, can be computed in polynomial time. Unlike primitive features such as vertex sets or edge sets, k-truss employs a higher-order graph motif based on triangle connectivity as the foundational element for formulating a community, thus enabling a comprehensive representation of multiple overlapping communities.
- Recent advancements in k-truss-based goal-oriented community search have been noted [69, 2]. Despite the seeming promise of k-truss-oriented community search formulations, they face challenges in harnessing the processing power of modern multi-core or many-core platforms for handling massive networks within a shorter time frame.
- Studies [126, 72, 140] have explored parallel k-truss decomposition in shared-memory systems. However, these studies specifically address the problem of k-truss decomposition, which is only a subproblem within the broader context of the k-truss-oriented formulation for local community search.

We developed a parallel algorithm for local community search based on k-truss in multiple phases. First, we implemented a sequential algorithm for truss-based local community search using C++ following the work [2]. We identified three key computational kernels: i) determining support, which involves calculating the number of triangles associated with an edge, ii) truss decomposition, which entails categorizing the edges of the original graph into various groups or subsets based on their trussness, and iii) constructing an index

substructure known as *EquiTruss* [2] from the distinct k-truss groups.

In contrast to the extensively studied parallelization of *k-truss* decomposition, the *EquiTruss* problem has been overlooked in the context of parallel algorithms, despite its demonstrated computational complexity as discussed in a subsequent chapter (Chapter 5). Therefore, this doctoral dissertation exclusively concentrates on the parallel algorithmic design for solving the *EquiTruss* problem. Our approach commences with the implementation of a parallel connected component (CC) algorithm [123] to construct *EquiTruss* in parallel, denoted as *Baseline EquiTruss*. Subsequently, we enhance the storage and retrieval of neighborhood information to optimize cache-locality, termed as *Optimal EquiTruss*. Finally, we employ a cutting-edge sampling-based parallel connected component algorithm [130] to construct supernodes in *EquiTruss*, referred to as *Afforest EquiTruss*. This version surpasses the performance of the initial two versions. Our contributions are summarized as follows:

i) We develop a parallel index construction method for k-triangle-induced structures (*EquiTruss*) using OpenMP. This innovative algorithm utilizes triangle connectivity and *k-trusses* as conditions for constructing the super graph. To the best of our knowledge, our approach is the first parallel algorithm designed for building such index structures, specifically tailored for facilitating local community search.

ii) Our parallel *EquiTruss* incorporates the Afforest algorithm [130], a state-of-the-art connected components approach. Additionally, we utilize the Shiloach-Vishkin (SV) algorithm [123] for parallel connected components, presenting a comparative analysis of the performance of these two approaches.

iii) The summary graph is constructed through a combination of parallel super node creation and parallel super edge generation, resulting in a speedup of up to 30× on the *NERSC Perlmutter* compute node compared to its sequential counterpart and up to 55× when compared to the *Baseline EquiTruss*.

Our experiments demonstrate a significant performance improvement, with speedups from $20x$ to $55x$ for graphs with hundreds of millions to billions of edges, using NERSC $Perlmutter$ compute nodes. Such a capability leads to a fast and efficient analysis of the underlying structure and behavior of large-scale graphs.

1.2 Summary of Research Findings

This dissertation write-up is organized as follows. In this chapter (Chapter 1), we discuss the research problem and our motivation in brief. Later, in Chapter 2, we present our novel distributed algorithm design for *Infomap* community discovery. In Chapter 3, we further improve the parallel performance of our distributed *Infomap* by combining both shared memory and distributed memory parallelism. In Chapter 4, we propose a generalized accelerator design for high-throughput hash accumulation. Finally, in Chapter 5, we present a novel parallel algorithm for local community discovery. In Table 1.1, we summarize the outcomes of our research on parallel algorithm designs.

Table 1.1: Listing the summary of the outcome of this doctoral research

Contribution	**Chapter**	**Methodology**	**Performance**
Distributed Infomap	2	MPI-based distributed parallelism	5.1×
Hybrid-Memory Infomap	3	(Distributed + shared)-memory parallelism	25×
Accelerator-aided Infomap	4	Accelerator for Fast Hash	5.6×
Parallel EquiTruss	5	Parallel Index Construction	$30\times - 55\times$

Chapter 2: A Distributed Memory Parallel Information-Theoretic Community Discovery

There exist various approaches for uncovering communities in a network (graph). Despite their approximating nature, community discovery grounded in the principles of *Information Theory* has demonstrated a standardized level of accuracy. The information-theoretic algorithm recognized as *Infomap*, conceived a decade ago for community detection, failed to anticipate the tremendous growth of social and biological networks, multimedia, and massive-scale datasets. For the identification of communities in extensive networks, we have developed a distributed-memory-parallel *Infomap* within the MPI framework. Our design achieves scalability of up to 512 MPI processes, enabling the processing of networks with millions of edges while maintaining a quality comparable to the sequential *Infomap*.

2.1 Introduction

Though community identification has emerged as a prominent method in network analysis, there lacks a precise definition of the term "community" within the context of network analysis. As defined by Porter et al. [107], Fortunato et al. [55], and Newman et al. [96], community detection, sometimes referred to as network clustering, involves dividing the vertices of an observed network into groups such that connections are dense within groups but sparser between different groups. This study focuses on global disjoint community discovery. The objective of global community discovery is to partition the vertices of a network into disjoint groups, where each vertex may belong to one community/group at a time. Several algorithms exist for discovering global communities. The global community detection algorithms that are computationally feasible for real-world applications are primarily approximation algorithms, as determining the exact number of communities based on optimization techniques poses an NP-hard problem [55, 38]. These approximation algorithms can be categorized based on the methodology being used. Henceforth in this chapter, when we mention community discovery, we specifically refer to global disjoint community discovery to keep the discussion concise.

The categorization of community detection methodologies, as outlined in Table 2.1, is centered around static networks. Arzum et al. [74] presented a concise overview of community detection methodology categories.

Table 2.1: Classification of the community detection approaches based on methodology

Category	**Methodology**	**Drawbacks**
Optimization Methods	Optimizing quality metrics -Modularity, Conductance	-Suffers from resolution limit
Statistical Inference	Network generative models -Stochastic Block Model	-Accuracy suffers -Computationally expensive
Spectral Methods	Based on spectral properties -Eigenvalue & eigenvector	-Computationally inefficient -Unreliable for sparse network
Information Theoretic Approach	Uses dynamic process -Random walk, MDL	-Complex logic -Computationally expensive

Our approach is tailored for static networks, where network attributes (e.g., vertices, links) remain unchanged over time. Table 2.1 enumerates the categories specifically addressing static networks. It is crucial to note that the applicability of these methods is not confined to static networks; there are existing works that integrate the capacity to handle dynamic networks in optimization methods, such as the work by Halappanavar et al. [63] and the study by Tiago et al. [106] based on stochastic block modeling (SBM). Additionally, the categories outlined in Table 2.1 face a common challenge of striking a balance between speed and accuracy. To sustain high accuracy, most of these categories must compromise speed. When processing extensive dynamic networks segmented into snapshots at different time frames, these categories demand a substantial amount of time to execute the same algorithm repeatedly on each snapshot.

In section 2.2 we discuss in brief the methodologies of the community detection strategies mentioned in table 2.1.

2.2 Descriptions of the Static Community Detection Approaches

In Fortunato et al.'s comparative study of community detection [56], there are descriptions of 12 algorithms, which can be classified into 4 major groups.

- Optimization methods hinge on optimizing a quality function for community discovery within a network. Modularity serves as a prevalent optimization function, aiming to maximize the difference between structural patterns within an actual network and another network with a random structural pattern. Due to its NP-hard nature, approximation algorithms, coupled with heuristics, are employed to optimize this quality function. Depending on how the optimization function is formulated, the optimization approaches can operate as either divisive or agglomerative, commonly known as the top-down or bottom-up approaches, respectively. Among the 12 different approaches for community detection studied in

the work by Fortunato et al. [56], 5 of them utilize the modularity maximization approach in one form or another to detect communities. The initial approach in this realm is modularity maximization based on edge betweenness, as outlined in the work of Girvan and Newman [59, 99]. Another well-known algorithm employing modularity maximization is the Louvain method, proposed by Blondel et al. [26]. Techniques based on modularity optimization encounter a challenge known as the *resolution limit*, as mentioned by Fortunato et al. [57], where the algorithm tends to overlook very small communities beside larger ones, often considering the small community as part of the larger one.

- Another category of community detection is based on statistical inference. Stochastic block modeling stands out as one such approach, elucidated in the works of Tiago et al. [106, 105, 104] and Newman et al. [75]. This approach stems from the concept that a network can be represented by a generative model, where the model parameters dictate the properties of the network. While it is impractical to find the exact parameters that generated a specific real-world network, statistical inference can be employed to determine these parameters. Various statistical processes such as Markov Chain Monte Carlo (MCMC) or Bayesian Inference can be applied to ascertain the partitioning of the network.

- Spectral methods leverage the spectral properties of the network. The underlying concept is that well-defined communities exhibit eigenvector components with similar values. The eigenvalue spectrum of the Laplacian matrix and the adjacency matrix is harnessed for community detection. A projection of vertices into a metric space is achieved by utilizing eigenvectors as coordinates. A restricted number of eigenvectors, denoted as n, is taken into consideration. Each network vertex is treated as a geometric point in an Euclidean n-dimensional space, with coordinates representing the eigenvector components for that vertex. These points are subsequently grouped using conventional clustering techniques such as K-means clustering. The works by Newman et al. [97, 98], and Donetti et al. [41] present community detection based on spectral methods.

- The information-theoretic approach to community detection involves considering the dynamics of a random walk to unveil the network's community structure. This strategy relies on principles from information theory and statistics. The minimum entropy theorem is applied to compress data generated by the dynamic process, leveraging the notion that higher regularity in information corresponds to more compressible data. Additionally, statistical concepts such as Minimum Description Length (MDL) are employed to represent the overall quality of the compressed information across the entire network. For a network having more prominent structural patterns (communities), there is an increased opportunity to compress the information for that network, ultimately revealing the communities during the compression process. Rosvall et al. [114] have presented a method for discovering communities using an information-theoretic approach.

Among the four categories mentioned above, the modularity optimization method, or more specifically, the *Louvain* method, is more popular than others because of its easily comprehensible nature. However, as

mentioned earlier, the modularity maximization strategy is not only NP-hard, but it also has the *resolution limit* problem that may affect the accuracy of the detected communities in a network. The study conducted by Fortunato et al. [56] reveals the information-theoretic algorithm of community detection by Rosvall et al. [114] to have the highest accuracy in the LFR [82, 80] benchmark. There is an MDL-based quality function named the *Map equation* by the authors in the study [114]. Fortunato et al. [56] named that algorithm as *Infomap*.

2.2.1 Motivation for Parallel Algorithm Design for Information-Theoretic Community Discovery

The various community detection approaches detailed in Section 2.2 share a common characteristic—they are all sequential in nature. These algorithms were developed over a decade ago when the size of networks rarely exceeded a million vertices. However, with the substantial growth of social networks, multimedia-capturing devices, and cost-effective storage, networks now encompass billions of vertices and edges. The sample networks utilized by Fortunato et al. [56] for the LFR [82] benchmark had thousands of vertices. During the comparison of these algorithms, execution time performance was not a significant consideration. In today's context, when evaluating algorithms for massive datasets, the efficiency of the algorithm becomes crucial, not just its accuracy. The sequential nature of the methods outlined in Section 2.2 significantly impacts computational efficiency. Modern computers are equipped with multiple processing cores that inherently support parallel computing. Recent state-of-the-art techniques focus on devising algorithms capable of leveraging shared-memory parallelism or distributed-memory parallelism. Sequential algorithms are now being reworked to process network data in parallel, utilizing numerous threads or processing cores.

In this undertaking, we propose the development of a parallel algorithm for the *Infomap* approach [114]. Several factors motivate our effort to craft a parallel algorithm for *Infomap*. Firstly, despite its approximate nature, this algorithm demonstrates remarkable accuracy. Additionally, it surpasses the *Louvain* method [26] in accuracy, as indicated in Fortunato et al.'s study [56], and avoids the *resolution limit* issue inherent in modularity optimization techniques like the *Louvain* method. Unfortunately, there is a limited body of research focused on devising an efficient parallel algorithm for *Infomap*. Contemporary methods are emerging for crafting parallel algorithms based on statistical inference, such as Stochastic Block Modeling (SBM). Further details on these parallel algorithms will be discussed in the literature review section. Our objective is to formulate an efficient and scalable parallel algorithm for *Infomap* to process extensive networks more rapidly and with greater accuracy in community discovery.

We devised a distributed-memory parallel *Infomap* algorithm exhibiting high scalability with 512 MPI processes. Achieving such scalability involved enhancements in computation strategy, communication, and workload balance across various phases. Initially, we adopted a straightforward partitioning approach,

distributing an equal number of vertices among processes for community computation in each iteration. However, the uneven degree distribution of real-world network vertices prompted us to adopt the Metis [78] graph partitioning strategy based on approximately even edge-cuts to ensure proper load balancing among working processes. To address challenges inherent in distributed graph processing algorithms, we developed several heuristics to handle issues arising in graph processing across distributed environments. To summarize, we made the following contribution

- We developed a distributed-memory parallel algorithm for community detection using the information-theoretic approach, implemented with the Message Passing Interface (MPI).
- We employed a graph partitioning strategy based on vertices to distribute the workload among processes, achieving scalability up to 256 processes. Subsequently, we enhanced our load-balancing approach by incorporating the Metis [78] graph partitioner, allowing for edge-cut-based partitioning across MPI processes. This refinement elevated the scalability to 512 processes, ensuring a higher execution speedup.
- We devised several heuristics to expedite the processing of extensive networks across distributed platforms, all the while preserving a level of accuracy akin to the sequential *Infomap*. These heuristics can be extended to address analogous graph computation challenges in a distributed environment. Further discussion on these heuristics will be presented in a subsequent section.

2.3 Problem Specification

We present the symbols and notations utilized in Table 2.2. Specific notations and definitions related to information theory will be elaborated upon in their respective sections.

Infomap employs a conventional data compression technique on a dynamic process, specifically a random walk. It leverages the dual nature between compressing a dataset and extracting significant patterns or structures within the data. This duality is explored in the field of statistics known as MDL or Minimum Description Length statistics [112, 60]. The data of interest in this context is the network flow. The trace of the flow can be represented as a binary codeword. If an optimal code can efficiently describe the locations traced by a path on a network, it simultaneously addresses the duality problem of identifying the structural features of that network. Therefore, *Infomap* seeks a method to assign codewords to vertices, considering the dynamics of the network. This leads us to the core of information theory, where Shannon's source coding theorems, also known as Shannon's minimum entropy theorem [120], can be employed to determine the limits on how effectively we can compress the information.

Shannon's minimum entropy theorem can be mathematically expressed as follows:

Table 2.2: Symbols and Their Descriptions Utilized in Our Parallel Infomap Work

Symbol	Description
$G(V, E)$	Graph G with vertex set V and edge set E
M	Total number of communities in a graph
L	Average length of the codeword for a move within the network
L_{old}	Average codeword length in the previous iteration of two consecutive iterations
$L(M)$	Average length of the computed codeword at a particular time with M communities
$q_{\curvearrowright}$	Sum of exit probability of the random walk for each module
$p^i \circlearrowright$	Stay probability of the random walk within a module i
Q	Probability distribution of the module entering rate (distinct from the quality function *modularity*, also denoted by the symbol Q)
$H(Q)$	Average code length for moves between modules (inter-module entropy)
p^i	Probability distribution of the random walk within module i
$H(p^i)$	Average code length of the random walk within the module
τ	Threshold value for the *Infomap* algorithm to halt execution
$u \rightarrow v$	Vertex u is moving to the community of vertex v
$v \rightarrow u$	Vertex v is moving to the community of vertex u
C_u	Current community membership C of vertex u
$Cp \leftarrow C_u$	Assignment of the community membership of vertex p to the community of vertex u
$C_v \leftarrow C_p (= C_u)$	Vertex p belongs to the community of vertex u. Assignment of the community of vertex p to vertex v
P_1, P_2	Two arbitrary MPI processes with rank 1 and rank 2
e_{ii}	Fraction of edges within the same community
a_i^2	Expected value of the metric e_{ii} for a random network showing no community structure
$\lvert {E_c}^{in} \rvert$	Number of edges within the same community in a network
$\lvert {E_c}^{out} \rvert$	Number of edges spreading from one community to another in a network

$$H = \sum_{i=1}^{n} p_i \times \log_2(X) \tag{2.1}$$

or,

$$H = \sum_{i=1}^{n} p_i \times \log_2(1/p_i) \tag{2.2}$$

or,

$$H = -\sum_{i=1}^{n} p_i \times \log_2(p_i) \tag{2.3}$$

To comprehend the functioning of Shannon's minimum entropy and its application for optimal information compression, we resort to the following example [1] on information entropy. Let's consider two machines generating information in the form of events. Machine 1 generates four events A, B, C, and D with the following probabilities:

$$P(A) = 0.25$$
$$P(B) = 0.25$$
$$P(C) = 0.25$$
$$P(D) = 0.25$$

Machine 2, on the other hand, generates the same four events with the following probabilities:

$$P(A) = 0.50$$
$$P(B) = 0.125$$
$$P(C) = 0.125$$
$$P(D) = 0.25$$

In order to determine which machine is producing more information between the two, we can formulate the problem using a decision tree, as depicted in Figure 2.1. This illustration helps us identify which machine produces more information and which produces less. Suppose both machines generated 100 events each within a timeframe, and we want to determine how many questions are needed to correctly guess all the 100 events. In Equation 2.1, there is a term X. If we express it in terms of probability, the number of possible outcomes of an event is equal to the inverse of the probability of that event, i.e., $X = 1/p$. This transformation leads to Equation 2.2 from Equation 2.1. From Equation 2.2, for machine 1, the average number of questions Q_n needed to determine a particular event is:

$$Q_n = p_A \times \log_2(1/p_A) + p_B \times \log_2(1/p_B) + p_C \times \log_2(1/p_C) + p_D \times \log_2(1/p_D)$$

$$Q_n = 0.25 \times 2 + 0.25 \times 2 + 0.25 \times 2 + 0.25 \times 2$$

$$Q_n = 2$$

For machine 2, the average number of questions required to determine the exact event that occurs can be expressed as:

$$Q_n = p_A \times \log_2(1/p_A) + p_B \times \log_2(1/p_B) + p_C \times \log_2(1/p_C) + p_D \times \log_2(1/p_D)$$

$$Q_n = 0.5 \times 1 + 0.125 \times 3 + 0.125 \times 3 + 0.25 \times 2$$

$$Q_n = 1.75$$

Since both machines generate 100 events each, for machine 1, we need to ask 200 questions to determine the outcomes of the 100 events, and for machine 2, we need to ask 175 questions to determine the outcomes of the 100 events. This example illustrates that machine 1 is producing more information than machine 2. The reason for machine 2 producing less information is the regularity of the information it generates. Based on the probability of event A for machine 2, it is more likely for machine 2 to be generating event A more than other

events. In other words, event A is more regular than other events in machine 2. As a result, the information produced by machine 2 is compressed, on average, to 1.75 questions compared to 2 questions in machine 1. This aspect highlights the significance of Shannon's minimum entropy theorem, where the regularity of information can be leveraged to compress that information. The theorem establishes a theoretical limit on how much information can be compressed without physically encoding the information and then compressing that code.

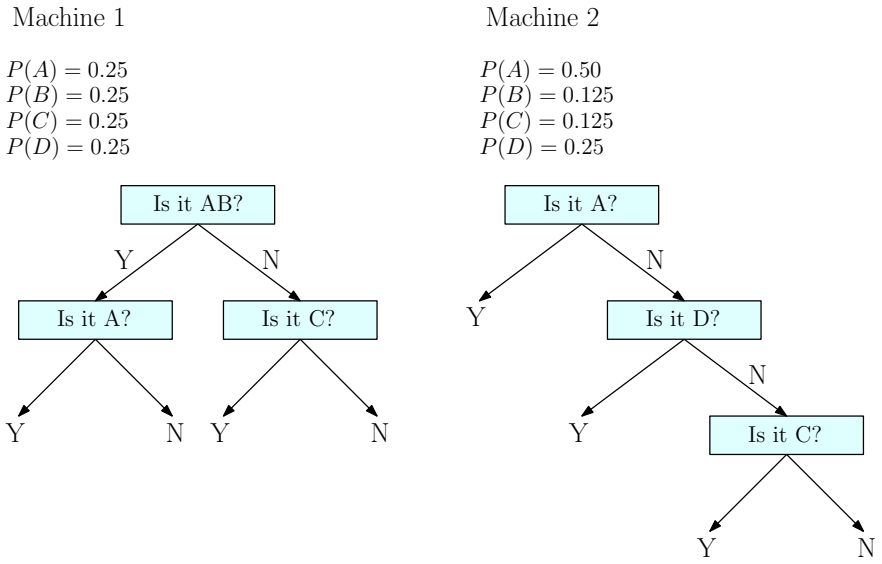


Figure 2.1: Describing the correlation between the regularity of information and the compression achieved by Shannon's Entropy theorem. On the left, we have the information generated by machine 1, and on the right, we have the information generated by machine 2. This information represents, on average, the number of questions needed to correctly guess the exact event produced by each machine.

As *Infomap* seeks efficient codewords, one direct approach for assigning codewords to vertices is through Huffman coding. This method provides shorter codewords for common events and longer codewords for rare ones. The resulting codewords for all vertices create a codebook. Within this codebook, each Huffman codeword uniquely represents a specific vertex, and the lengths of the codewords are determined by the ergodic node visit frequencies of a random walk. The average node visit frequencies for an infinite-length random walk can be computed using Google's PageRank algorithm [27].

Imagine there is a significant structural pattern in a network, and our objective is to unveil these structural patterns, often referred to as "modules." The movement of a random walker in the network can be described through two distinct types of moves. One involves the random walker moving within a structural pattern, traversing from one vertex to another within the same module. The other involves the random walker transitioning across different modules. A relatable analogy for understanding how *Infomap* operates based on the random walk is akin to traffic patterns within and between cities. Traffic within a city tends to stay longer within its

bounds, traveling rarely across cities. Similarly, the goal of discovering a city (structure or community within a network) is to identify the region where traffic (random walk) experiences maximum flow. Maximizing flow within a cluster and minimizing flow among clusters ensures the accuracy and quality of detected communities.

In alignment with this concept, the codebook of the vertices can be divided into two parts. The codewords representing moves across modules are designated as the index codebook. Conversely, the codewords representing successive moves within a module are labeled the module codebook. The lengths of codewords in the index codebook are derived from the relative rates at which a random walker enters each module, while the lengths for each module codebook are derived from the relative rates at which a random walker visits each node in the module or exits the module. By employing multiple codebooks, the task of minimizing the description length of paths taken is transformed into determining the optimal partitioning of the network concerning flow dynamics.

Although the Huffman coding process is explained to elucidate the coding structure, the ultimate goal of community detection is not to encode a specific path through the network. Instead, the objective is to uncover the modular structure of the network concerning flow and leverage the inference-compression duality to achieve this. There is no necessity to devise an optimal code for a given partition to estimate its efficiency. Detecting the optimal community structure of a network becomes the challenge of computing the theoretical limit for different partitions and greedily selecting the one that yields the shortest code length. The optimization function enabling us to compute this theoretical limit is known as the *Map Equation*.

2.3.1 The Map Equation

The optimization function representing the code length is known as the *Map Equation*. The objective of this optimization is to minimize the code length across all potential assignments of vertices into communities. Rooted in the concept of MDL (Minimum Description Length), which asserts [60] that any regularity in information can be exploited for compressing said information, the *Map Equation* is formulated as presented in Eq. 2.4 by Rosvall et al. [114].

$$L(M) = q_{\curvearrowright} H(Q) + \sum_{i=1}^{m} p^{i}_{\circlearrowright} H(\rho^{i}) \tag{2.4}$$

In this equation, the right side consists of two main parts. The first part, $q_{\curvearrowright} H(Q)$, can be further divided into two terms. The initial term, $q_{\curvearrowright}$, signifies the summation of the exit probability of the random walk for each module in the network. The term $H(Q)$ represents the average code length of movements between the modules, with Q denoting the probability distribution of the module entering rate. This average code length of movements is referred to as the index code length.

The second part on the right side of the *Map Equation* is $\sum_{m \in M} p^i \circlearrowleft H(\rho^i)$, where $p^i \circlearrowleft$ denotes the stay probability of the random walk within module m. This parameter can be computed by summing the visit probability of the random walk and the exit probability of the random walk for that module. The term $H(\rho^i)$ corresponds to the average code length of the random walk within the module, known as the module code length. The variable ρ^i represents the probability distribution of the code of module m. For a more detailed representation of the *Map Equation*, refer to Eq. 2.5.

$$L(M) = (\sum_{m \in M} q_m) \log (\sum_{m \in M} q_m) - 2 \sum_{m \in M} q_m \log q_m - \sum_{\alpha \in V} p_\alpha \log(p_\alpha) + \sum_{m \in M} (q_m + \sum_{\alpha \in m} p_\alpha) \log(q_m + \sum_{\alpha \in m} p_\alpha) \tag{2.5}$$

In this context, q_m represents the exit probability of module m and is determined by the relative weight of links exiting the module m. The summation $\sum_{m \in M} q_m$ corresponds to the total relative weight of links between modules. The term p_α denotes the visit probability of a vertex α during the random walk, where V stands for the set of all vertices in the network. Additionally, p_m signifies the visit probability of a module m and is computed as $\sum_{\alpha \in m} p_\alpha$. For a more in-depth understanding, interested readers are recommended to explore the appendix section of the original work on Infomap [114].

2.3.2 Sequential Infomap Algorithm

Algorithm 1 outlines the procedural steps of the sequential *Infomap*. Lines $1-5$ detail the notation employed within the algorithm. Lines $6-7$ calculate the vertex visit rate (i.e., PageRank) using the power iteration method. The algorithm initiates with the number of communities equal to the number of vertices, denoted as $N \leftarrow |V|$. Here, M signifies the set of modules, and m_i denotes an individual module/community. Initially, m_i has only one member vertex, but its composition may evolve as the algorithm progresses. The set M encompasses all the modules m_i (line 8). A vertex v is associated with a single module at a given time. Lines $9-11$ compute the initial exit probability q_i for each module m_i. Line 12 calculates the initial code length based on Equation 2.4. Subsequently, lines $13-27$ depict the community discovery procedure iterating through multiple cycles. Line 14 retains the current code length at the outset of an iteration. Each vertex in the vertex set V is randomly selected for community optimization (lines $15-16$). Among all neighboring modules of a vertex, the one minimizing the code length is chosen (lines $17-18$). Line 20 describes the creation of supernodes, comprising one or more vertices. Analogous to the individual vertex module optimization, the supernode-level optimization phase unfolds in lines $21-25$. This process continues until the change in code length falls below a user-defined threshold γ (line 27). The return value of the algorithm is the number of discovered communities (line 28).

The sequential *Infomap* algorithm has demonstrated superior accuracy compared to many algorithms in various community detection comparative studies [81, 3, 85]. However, its scalability is constrained in the

Algorithm 1: Sequential Infomap

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

m_i, i$^{\text{th}}$ module
q_i, exit probability of module m_i
γ, the minimum threshold for code length improvement
L_{old}, code length of previous iteration
L, code length of current iteration
Initialize vertex visit rate, $p_{v_i} \leftarrow 1/N$
Compute ergodic vertex visit rate p_{v_i} by PageRank
$M \leftarrow \{\forall m_i | (v_\alpha \in m_i) \& (v_\alpha \notin m_j), V = \sum_1^N v\}$
for $i = 1$ *to* $|M|$ **do**
    Calculate exit probability q_i
Initial code length $L \leftarrow L(M)$
do
    $L_{old} \leftarrow L$
    for $j = 1$ *to* N **do**
        Select randomly each vertex, v_j
        $m_{new} \leftarrow FindBestModule(v_j)$
        Compute L cumulatively
    $M \leftarrow CreateSuperNode()$
    for $j = 1$ *to* $|M|$ **do**
        Select randomly each SuperNode m_j
        $m_{new} \leftarrow FindBestModule(m_j)$
        $m_j \leftarrow \{m_{new} | \forall v_j \in m_j\}$
        Compute L cumulatively
while $(L_{old} - L) > \gamma$
return $|M|$

serial paradigm. The parallelization of the *Infomap* algorithm poses non-trivial challenges, which we will elaborate on in the following section, along with the heuristics employed to mitigate them.

2.4 Challenges in Distributing Computation/Data

While distributing computation and data among processing units, our map-based approach presents the following challenges and issues.

2.4.1 Vertex bouncing problem

The issue arises when two vertices with a strong affinity are assigned to two different processes. Each vertex attempts to move to the community of the other vertex. In sequential execution, the first chosen vertex would move to the community of the other, and when the second vertex is selected, it recognizes the shared

community, avoiding unnecessary moves. However, in distributed execution, both vertices lack awareness of the other's community assignment, leading to additional moves. This problem is illustrated in Figure 2.2, where vertices u and v are distributed across processes P_1 and P_2—resulting in unnecessary movements between their communities.

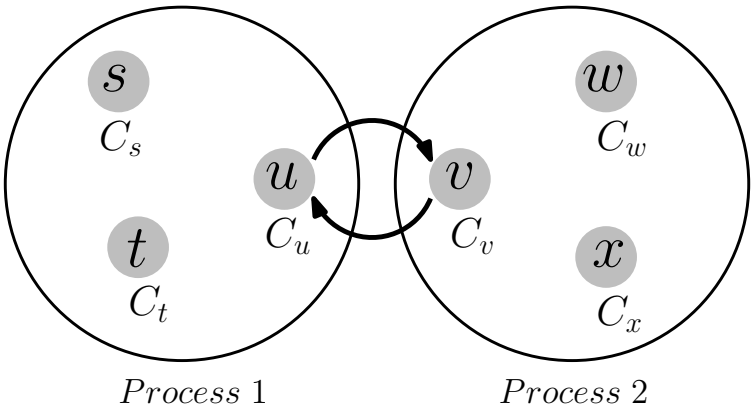


Figure 2.2: The distributed graph processing encounters the vertex bouncing problem when vertices u and v share a close affinity and belong to the same community. Despite this affinity, as they are partitioned into different processes (with vertex u in process 1 and vertex v in process 2), both processes independently compute the same community update.

2.4.2 Inconsistent update ordering

We adopt a synchronous parallel approach, and maintaining consistent community assignments during synchronization poses a challenge due to varying synchronization orders across processes. To illustrate, consider a scenario with 7 vertices (p, q, r, s, t, u, v) divided into two processes, P_1 and P_2, where vertices p, v, s are in process P_1, and vertices q, r, t, u are in process P_2 as depicted in Figure 2.3. The two large circles represent the two different processes, and the gray circles inside represent individual vertices. The arrows denote the direction of moves between communities. A vertex without an arrow (e.g., s) indicates that no suitable move was found in a particular iteration. Based on one possible processing order of the vertices, the following are the moves of the vertices from their current module to another module. $p \longrightarrow u$, $t \longrightarrow p$, $v \longrightarrow p$, $q \longrightarrow r$. In this context, the notation $p \longrightarrow u$ signifies that vertex p moves to the community of vertex u, denoted by C_u. If the described moves are executed in the exact order mentioned, the following community assignments will result from the execution of those moves in the sequential algorithm.

$$C_p \leftarrow C_u,\ C_t \leftarrow C_p(= C_u),\ C_v \leftarrow C_p(= C_u),\ C_q \leftarrow C_r \tag{2.6}$$

The community assignments in process P_1 are $C_p \leftarrow C_u$ and $C_v \leftarrow C_p\ (= C_u)$. The notation $C_p\ (= C_u)$ signifies that the current community of vertex p is the same as the community of vertex u. In process P_2, the community assignments are $C_t \leftarrow C_p$ and $C_q \leftarrow C_r$. After synchronization across processes P_1 and P_2, the

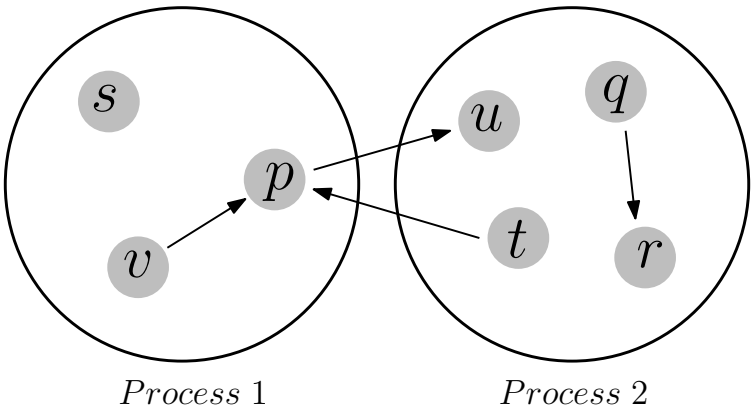


Figure 2.3: Calculating community membership information in a distributed setting involving two processes. Process 1 (on the left) computes the community for vertices v and p, while Process 2 (on the right) calculates community membership for vertices q and t.

following events occur. In process P_1,

$$C_p \leftarrow C_u,\ C_v \leftarrow C_p(= C_u),\ C_t \leftarrow C_p(= C_u),\ C_q \leftarrow C_r \tag{2.7}$$

In process P_2,

$$C_t \leftarrow C_p,\ C_q \leftarrow C_r,\ C_p \leftarrow C_u,\ C_v \leftarrow C_p(= C_u) \tag{2.8}$$

Figure 2.4 depicts the resulting communities in two distinct processes. Due to the processing of updates in different orders, the community assignment of Vertex t is no longer consistent across the processes.

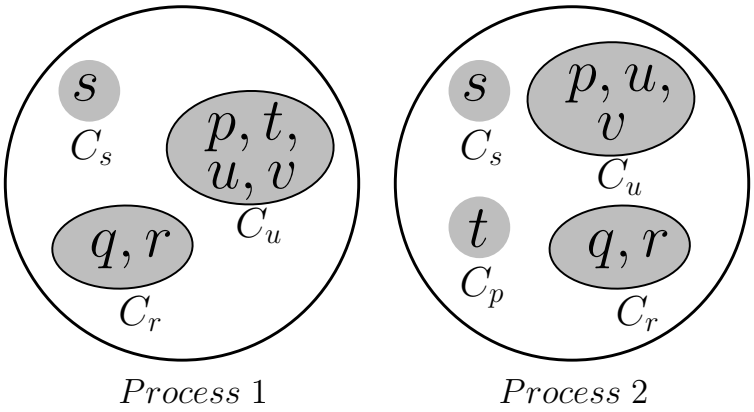


Figure 2.4: Non-uniform communities arise due to incorrect synchronization. In process 1 (left), community C_u comprises 4 vertices p, t, u, v, whereas in process 2 (right), community C_u consists of 3 vertices p, u, v.

2.4.3 Inactive vertices

The process of assigning vertices to communities continues over multiple iterations. In the initial few iterations, most vertices change communities before stabilizing in their final positions. Once a vertex moves

to a community and remains there for several subsequent iterations, it is less likely to undergo further changes. In later iterations, fewer updates to community assignments occur. This observation suggests that in each iteration, considering all vertices for community updates involves redundant activities that waste computational resources. It becomes essential to distinguish between vertices and select those more likely to change communities in subsequent iterations.

2.5 Solution Strategies: Our Heuristics

We devised the following heuristics to address the challenges outlined earlier.

2.5.1 Solution to Vertex Bouncing Problem

To tackle the vertex bouncing problem, we implemented *numeric ordering* during the synchronization step. In this approach, we accept one of the two moves illustrated in Figure 2.2—namely, $u \rightarrow v$ in process P_1 or $v \rightarrow u$ in process P_2—and discard the other. The decision to accept or discard a move is based on the numeric values of the IDs of the current communities of vertices u and v. We select the move from the community with the lower ID to the community with the higher ID. For example, 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 while ignoring the move of v to C_u.

2.5.2 Solution to Inconsistent Update Ordering

To ensure consistent community assignments for vertices across all processes, we implemented the *priority-based* community assignment heuristic. In this scheme, the owner process of a vertex makes the decision regarding the community assignment for that specific vertex. Each process respects the community assignment information received for vertices processed by other MPI processes. This straightforward yet effective approach addresses the challenge illustrated in Figure 2.4. Figure 2.5 showcases the communities resulting from the same vertex moves as depicted in Figure 2.3.

2.5.3 Solution to Inactive Vertices Problem

To avoid recomputing the community assignments for vertices unlikely to change communities, we introduce the distinction between *Inactive Vertices*, which are unlikely to change communities in the current iteration, and *Active Vertices*, which may move to different communities in the next iteration. Importantly, there is no deterministic way to decide which vertices will be active or inactive in the immediate next iteration. Empirical observation suggests that vertices changing communities in one iteration are likely to do so in the immediate next iteration. Moreover, their neighbors may also become active. Thus, before an iteration begins, we create a prediction list of vertices that may become active, derived from the community assignments of the immediate previous iteration. This prediction list includes vertices that changed communities in the previous

iteration and their immediate neighbors. Empirical observations indicate that when a vertex moves from one community to another, its immediate neighbors contribute to over 95% of the quality improvement for subsequent iterations [15].

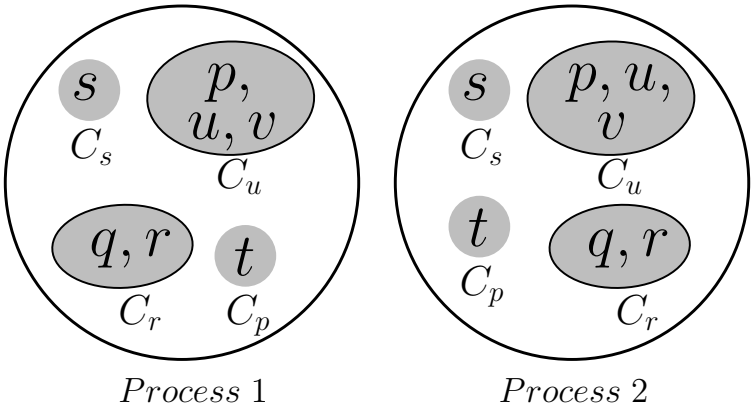


Figure 2.5: Consistent communities in two distributed processes (Process 1 and Process 2) due to synchronization based on priority ordering.

2.5.4 Our Parallel Algorithm Design of Distributed *Infomap*

Algorithm 2 delineates our blueprint for the distributed-memory parallel *Infomap* using MPI. Lines $1-6$ introduce the notations utilized throughout the algorithm. Lines $7-8$ calculate the PageRank using the power iteration method. Similar to the sequential Algorithm 1, the distributed Algorithm 2 commences with each vertex belonging to its exclusive community (line 9). At any juncture in the community discovery, a vertex may belong to a solitary community (distinct from the intermediate computational phase involving different MPI processes with diverse community membership information for a specific vertex). Lines $10-11$ evaluate the initial exit probability q_i for the module m_i, and line 12 determines the initial code length following equation 2.4. All MPI processes autonomously progress through the same computations described thus far until they engage in the parallel computation of community discovery across multiple iterations (lines $13-31$). Each MPI process operates on its individual Metis-partitioned vertex list concurrently (line 16), randomly selecting all vertices (random walk) and striving to identify the optimal module that minimizes the code length (lines $18-20$). Subsequent to computing the new module information for its designated sets of vertices, each MPI process synchronizes this information across all other MPI processes (line 21), ensuring the precision of community membership for subsequent iterations. Up to this point, the community discovery approach operates at the vertex level and spans multiple iterations. Following this, the community established in the vertex level phase transforms into supernodes, where each supernode may encompass one or more vertices (line 22). A parallel computation is executed to determine the optimal module for supernodes (lines $23-29$). Instead of vertices, each MPI process randomly designates its supernodes (lines $25-26$). Another synchronization of community information is conducted (line 30), this time at the supernode level.

The amalgamation of vertex-level and supernode-level community discovery persists until the algorithm attains a user-specified threshold γ between the previous code length and the new code length (line 31).

Algorithm 2: Distributed Infomap

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

1 m_i, i$^{\text{th}}$ module
2 q_i, exit probability of module m_i
3 γ, the minimum threshold for code length improvement
4 L_{old}, code length of previous iteration
5 L, code length of current iteration
6 P, number of MPI processes spawned
7 Initialize vertex visit rate, $p_{v_i} \leftarrow 1/N$
8 Compute ergodic vertex visit rate (PageRank), p_{v_i}, by power iteration
9 $M \leftarrow \{\forall m_i \,|\, (v_i \in m_i) \,\&\, (v_i \notin m_j),\ (0 \leq v_i < N) \,\&\, (v_i \in V)\}$
10 **for** $i = 0$ *to* $|M|$ **do**
11 Calculate exit probability, q_i
12 Initial code length $L \leftarrow L(M)$
13 **do**
14 $L_{old} \leftarrow L$
15 **for** *process* $p = 0$ *to* $P - 1$ ***in parallel*** **do**
16 Compute vertex indices range $[v_{start},\ v_{end}]$ from $metis - partitioned$ vertex list
17 **for** $j = [v_{start},\ v_{end}]$ **do**
18 Select randomly each vertex, v_j
19 $m_{new} \leftarrow FindBestModule(v_j)$
20 Compute L cumulatively
21 Synchronize $m_{new} \in M$ across P processes
22 $M \leftarrow CreateSuperNode()$
23 **for** *process* $p = 0$ *to* $P - 1$ ***in parallel*** **do**
24 Compute SuperNode indices $[m_{start},\ m_{end}]$ from $Active$ SuperNodes
25 **for** $j = [m_{start},\ m_{end}]$ **do**
26 Select randomly each SuperNode m_j
27 $m_{new} \leftarrow FindBestModule(m_j)$
28 $m_j \leftarrow \{m_{new} \,|\, \forall v_j \in m_j\}$
29 Compute L cumulatively
30 Synchronize $m_{new} \in M$ across P processes
31 **while** $(L_{old} - L) > \gamma$
32 **return** $|M|$

2.6 Experimental Setup

The majority of the experiments and corresponding results presented in this study were conducted on the Louisiana Optical Network Infrastructure (LONI) system [67]. The utilized computing cluster is QB2 [68], boasting a peak performance of 1.5 Petaflops. It comprises 504 compute nodes, each equipped with 20 processing cores, totaling over $10,000$ Intel Xeon processing cores. The cluster also features a 2.8 PB Lustre file system. Operating on the RedHat Enterprise Linux 6 Operating System, it incorporates a 56 Gb/sec (FDR) InfiniBand and a 1 Gb/sec Ethernet network.

2.7 Implementation

Our implementation is developed in C++ using the MPI framework with the g++ compiler. The source code of our implementation is available online [52]. The program supports networks in pajek (.net) format [20]. The main phase of the algorithm, i.e., the greedy optimization phase, runs in multiple iterations. In the initial iteration, each vertex represents its module. In each subsequent iteration, each process takes an almost equal chunk of vertices from the active vertices list. Each process computes the change in Minimum Description Length (MDL) for a potential move of that vertex along any of the links to which the vertex is connected. It then greedily selects the move to a module that minimizes MDL the most. Each processor compiles an information list of the vertices that have been moved from one module to another. During the synchronization phase, each process sends and updates the community information based on the received information. Each module is also transformed into what we term a "supernode," representing a group of vertices with the same module ID. All inter-edges between a pair of supernodes are consolidated into a single edge with a weight equal to the sum of all edges between that pair of supernodes. After creating a network with supernodes, the greedy optimization for reducing MDL is executed again at the supernode level, similar to the vertex-level optimization performed earlier. After each iteration, the list of active vertices for the next iteration is computed. This process continues until no further reduction in MDL occurs in successive iterations, i.e., until convergence is reached. The final output of the program includes the number of detected communities along with the final compressed value of the MDL.

2.8 Performance Comparison

We conducted a qualitative comparison between our distributed implementation of *Infomap* and the one designed by Bae et al. [15]. Additionally, we performed a parallel performance comparison against the distributed implementations [16], demonstrating that our work surpasses that implementation in terms of scalability. Unfortunately, the implementation by Zeng et al. [149] is not publicly available online. Consequently, we had to rely on the data provided in their paper [149] to evaluate the superiority of our work in terms of speedup gain in the subsequent discussion.

2.9 Dataset

We employed a network dataset of varying sizes, ranging from a network with 0.31 million vertices and 1.04 million edges to a larger network with 3 million vertices and 117 million edges. Table 2.3 provides a concise overview of the dataset, where columns 2 and 3 indicate the number of vertices and edges in the network, respectively. Our experiments with distributed *Infomap* utilized networks such as *Amazon, DBLP, YouTube, Wiki-topcats*, and *soc-Pokec*, all sourced from SNAP [84]. The selection of these networks is based on their well-defined community structures, making them suitable for the evaluation and comparison of our implementation.

Table 2.3: Network dataset for our experiments. We used several social and information networks.

Network	**# Vertices**	**# Edges**	**Description**
Wiki-topcats	1791489	28511807	Hyperlinks network from Wikipedia
soc-Pokec	1632803	30622564	Pokec online social network
Youtube	1134890	2987624	Youtube social network
DBLP	317080	1049866	CS bibliographical network
Amazon	334863	925872	Amazon co-purchased network

2.10 Evaluation

2.10.1 Quality analysis of the Detected Modules

Infomap consistently produces higher-quality communities compared to state-of-the-art techniques, as evident in various benchmark studies [81, 3]. To assess the quality of the detected communities, we consider Modularity, Conductance, and the convergence MDL value. Our comparison includes RelaxMap [15], known for achieving community quality comparable to the original Infomap. Table 2.4 presents Modularity and Conductance values for the shared memory-based Infomap [15]. While distributed-memory-based implementations may achieve quality comparable to their sequential counterparts at best, our comparative study with RelaxMap offers a meaningful qualitative analysis.

Table 2.4: Modularity and Conductance of the networks for the sequential Infomap

Network	**Modularity**	**Conductance**
soc-pokec	0.52	0.47
wiki-topcats	0.43	0.57
YouTube	0.39	0.56
DBLP	0.59	0.41
Amazon	0.77	0.23

2.10.1.1 Convergence of the Objective Function

The objective function of Infomap aims to minimize the Minimum Description Length (MDL). Achieving improvements in MDL within a distributed implementation poses challenges compared to sequential or shared-memory-based optimization. The compression outcome of a preceding move might not be accessible to other processors, potentially influencing decisions regarding MDL changes—contrary to sequential or shared implementations. Distributed implementations also face the risk of premature convergence, leading to suboptimal MDL improvements, as noted in [16]. Our optimization of the objective function in Equation 2.5 yields MDL improvements comparable to those observed in [15]. Table 2.3 presents the initial MDL values for the utilized networks. Figure 2.6 displays the converged MDL values. In all cases, the differences in MDL are negligible, with the largest variance occurring in the network *Wiki-topcats*, where the final MDL value exceeds that of [15] by only 0.39. This suggests that the detected communities after convergence closely resemble those found in [15]. Importantly, our algorithm avoids issues of under-clustering or over-clustering.

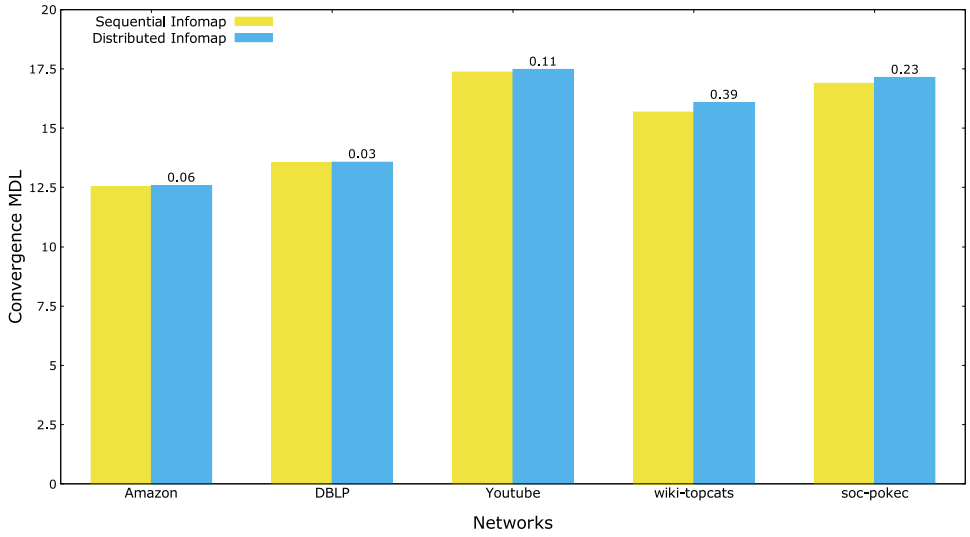


Figure 2.6: Comparison of MDL after convergence between sequential and distributed Infomap.

2.10.1.2 Modularity

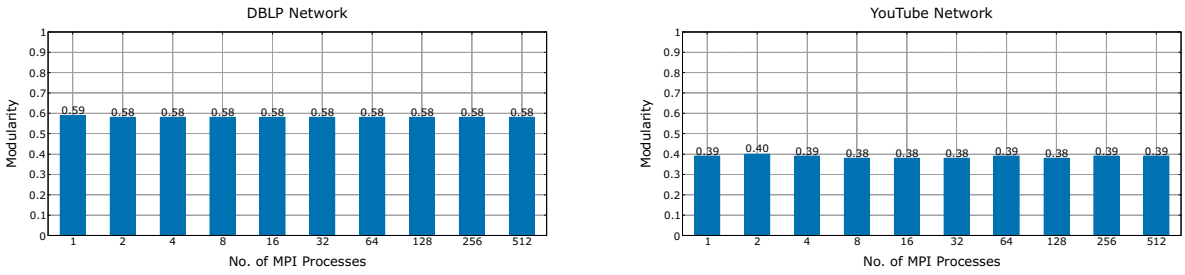
To assess the quality of the identified communities, we employed the modularity (Q) measure [99]. Modularity is indicative of how effectively a network is partitioned into communities. For a given network, the modularity score (Q) represents the fraction of edges within communities minus the expected value of the same quantity if the edges were randomly distributed in a network with the same degree sequence. The mathematical definition of this measure is provided in Equation 2.9.

$$Q = \sum_{i} (e_{ii} - a_i^2) \tag{2.9}$$

In the provided equation, e_{ii} represents the fraction of edges within communities, and a_i^2 denotes the expected value of the aforementioned quantity for a graph with the same degree sequence but random edges. A positive decimal value of Q indicates that the number of edges within communities exceeds the expected number. Typically, a value of Q in the range of 0.3-0.7 signifies significant community structure [85].

It is noteworthy that our approach does not optimize the modularity value for community detection, as is common in other community detection mechanisms [99, 34, 26]. Instead, we utilize the modularity measure (Q) as a quality metric to analyze the effectiveness of our detected communities, obtained by optimizing the *Map equation* as outlined in Equation 2.4. In Table 2.3, we present the quality measurement metrics (i.e., modularity, conductance) and their corresponding values from [15] for the dataset.

To evaluate the quality of modularity concerning the increasing number of processors, we examined whether the values fluctuate. Histogram plots in Figures 2.7a and 2.7b showcase modularity values across different numbers of MPI processes. For instance, in Figure 2.7a at the bar corresponding to 256 MPI processes, the modularity value is 0.58, with a difference of only 0.01 compared to sequential execution. Similarly, the modularity score obtained for the network *YouTube* running on 512 MPI processes matches the modularity score of the sequential execution, both being 0.39. Importantly, higher modularity values signify better communities. In summary, the modularity values for different networks are comparable to those from sequential or shared-memory-based *Infomap* implementations, and the quality of detected communities remains consistent across an increasing number of processors.



(a) DBLP Network

(b) YouTube Network

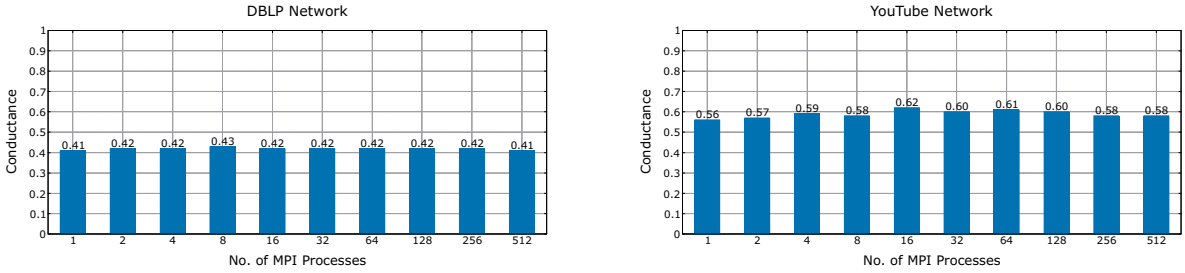
Figure 2.7: Illustration of preserved community quality in the distributed setting using modularity score. Changes in modularity values across different numbers of MPI processes for (2.7a) *DBLP* and (2.7b) *YouTube* networks. The numeric value atop each histogram bar in each plot represents the modularity of the discovered community in the run using the specified number of MPI processes on the x-axis. Higher modularity scores indicate better quality of the discovered communities.

2.10.1.3 Conductance

According to the study conducted by Yang et al. [144], when a network comprises well-separated disjoint communities, conductance provides the best quality analysis of the detected communities. For unweighted

networks, conductance measures the fraction of the total number of edges that point outside the community, and for weighted networks, it is the fraction of the total weight of such edges. In a directed network, $conductance = \frac{|{E_c}^{out}|}{|{E_c}^{in}|+|{E_c}^{out}|}$, and for an undirected network, $conductance = \frac{|{E_c}^{out}|}{2|{E_c}^{in}|+|{E_c}^{out}|}$. Inspired by the concept of electric conductivity, where higher conductivity values indicate connected paths and 0 or less conductivity means no connection or loosely coupled connection, high conductance implies that communities are not well-separated and disjoint; the portions of intra-edges and inter-edges are not well-separated. On the other hand, a low value of conductance indicates that communities are well-separated, and if not completely, they are highly disjoint. The smaller the value of conductance, the better the quality of the discovered community.

We employ similar concepts and plots as used for modularity in Section 2.10.1.3, with the only difference being that smaller conductance values from shared/sequential *Infomap* indicate higher quality in the detected communities. Figures 2.8a and 2.8b show minimal changes in conductance values across multiple MPI processes for the *DBLP* network and some minor fluctuations (ranging $0.01 - 0.06$) for the *YouTube* network compared to the study by [15]. We conclude that the quality of the detected communities using our distributed *Infomap* is comparable to the sequential *Infomap*.



(a) DBLP Network

(b) YouTube Network

Figure 2.8: Illustration of the preserved quality of discovered communities in the distributed setting using conductance. Change of conductance values across different numbers of MPI processes for (2.8a) *DBLP* network and (2.8b) *YouTube* network. The numeric value on top of each histogram bar in each figure demonstrates the conductance score for a specific number of MPI processes. The lower the value of conductance, the better the quality of the discovered communities.

2.10.2 Distributed Performance Analysis

2.10.2.1 Workload Balancing

In our initial network dataset partitioning design across MPI ranks/processes, we adopted a naive approach of distributing an approximately equal number of vertices across those ranks, as discussed earlier in Section 2.2.1. Figure 2.9 illustrates a highly uneven workload across multiple ranks, where the horizontal bar with different colors represents computation (various functions) or communication (different MPI data transfer calls). To

address this issue, we employed the *Metis* [77] graph partitioner to distribute the workload among processors. The goal is not only to ensure a balanced workload among individual MPI ranks but also to minimize edge-cut across the different partition sub-graphs, thereby reducing the vertex bouncing problem's impact, as discussed in Section 2.4. To achieve equal computational workload sharing, each processor receives a subset of vertices and corresponding edges returned by the *Metis* partitioner. Consequently, each processor takes nearly an equal amount of time to complete the algorithm's execution, as depicted in Figure 2.10. Performance profiling Figures 2.9 and 2.9 were generated using [122].

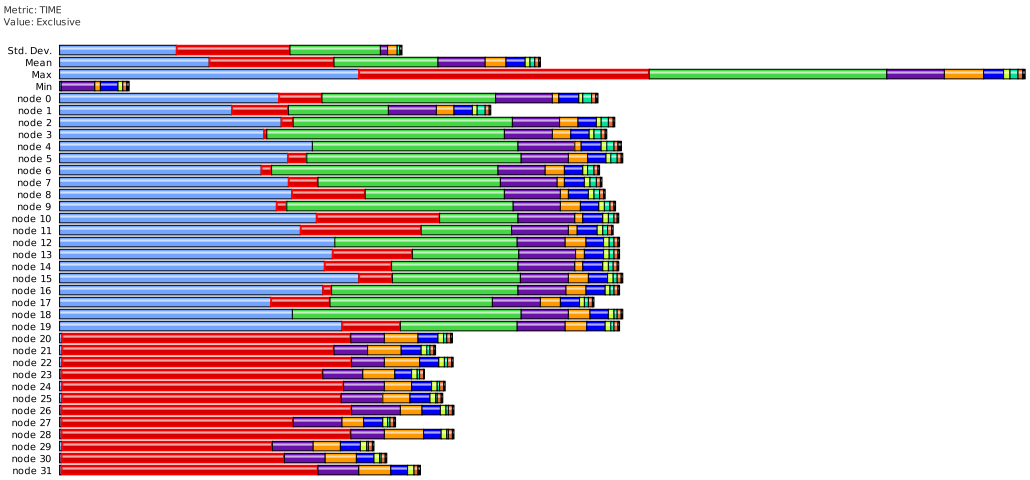


Figure 2.9: Workload imbalance resulting from naïve vertex distribution across MPI processes. Different colors in each horizontal bar represent the time spent by different computation/communication kernels. There are significant imbalances observed across different MPI processes

2.10.2.2 Speedup and Parallel Efficiency

We evaluate the speedup and time-performance using the networks listed in Table 2.3. Figure 2.11 illustrates the runtime of our algorithm for three different networks with varying numbers of processors. We observe substantial scalability, reaching up to 512 MPI processes for larger networks such as *wiki-topcats* and *soc-Pokec*. In the case of a relatively large network like *YouTube*, we achieve scalability up to 256 MPI processes. However, for smaller networks like *Amazon* and *DBLP*, the advantages of scalability are outweighed by the increased *MPI* communication cost across a higher number of processes. The communication cost is influenced by the interconnect infrastructure of the computation nodes, while the computation cost is determined by the volume of computation required due to the size of the network dataset. Larger networks involve more computation, providing the opportunity for higher scalability.

In Table 2.5, we present the maximum speedup achieved by our algorithm with varying numbers of MPI processes compared to sequential execution. The maximum speedup gains are observed for larger graphs

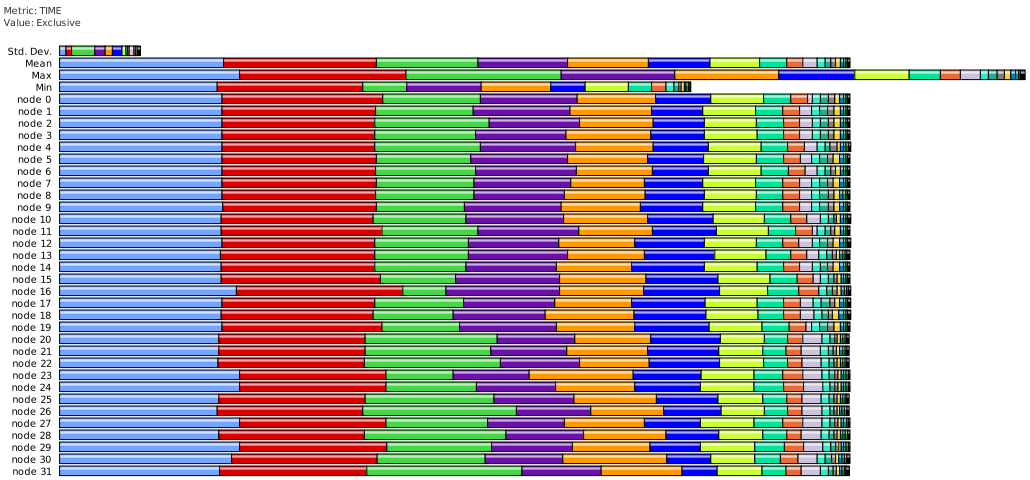


Figure 2.10: Balanced workload across processes resulting from workload distribution by *Metis* partitioner. Different colors in each horizontal bar represent the time spent by different computation/communication kernels. There are no notable imbalances observed for different kernels across different MPI processes.

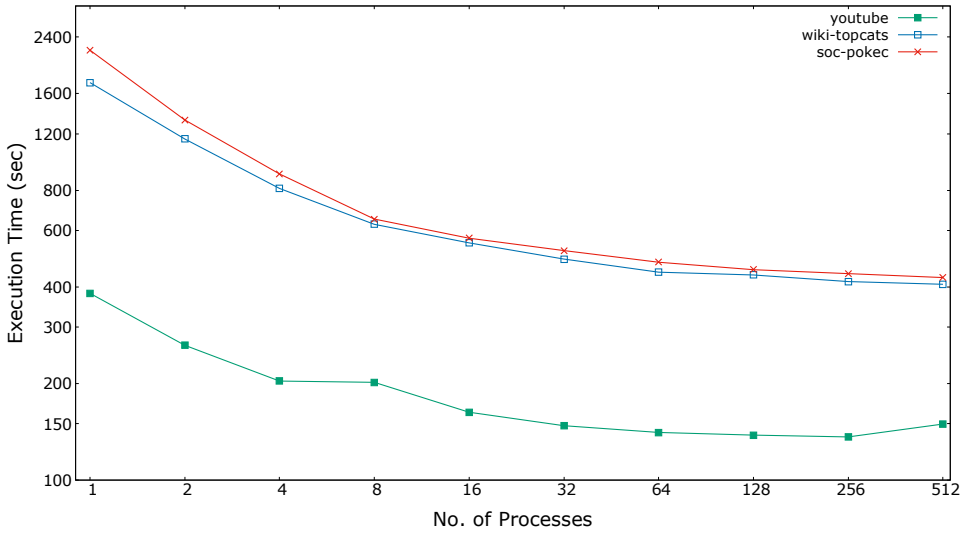


Figure 2.11: Reduction of processing time for networks of different sizes from a single process to 512 processes in distributed Infomap. The runtime scalability curves are drawn for the larger networks (YouTube, wiki-topcats, and soc-pokec).

(*wiki-topcats*, *soc-pokec*) with 512 MPI processes, as depicted in Figure 2.11. For the *YouTube* network, the maximum speedup gain is observed with 256 MPI processes. It's crucial to note that the achievable speedup from parallelizing a computational problem depends heavily on the type of problem being addressed. For instance, a well-designed parallel Breadth-First-Search [30] can achieve high scalability and significant speedup compared to sequential BFS traversal. However, the same may not hold true for Depth-First-Search.

In the case of the *Infomap* problem, we observe similar speedup gains compared to state-of-the-art techniques, as demonstrated in the work of [149], even with the use of thousands of processors.

Table 2.5: Speedup factors on various social and information networks. For larger networks such as soc-pokec and wiki-topcats, we observe better speedup due to having more computation.

Network	**Speedup**
Amazon	1.64
DBLP	1.92
Youtube	2.80
wiki-topcats	4.25
soc-pokec	5.10

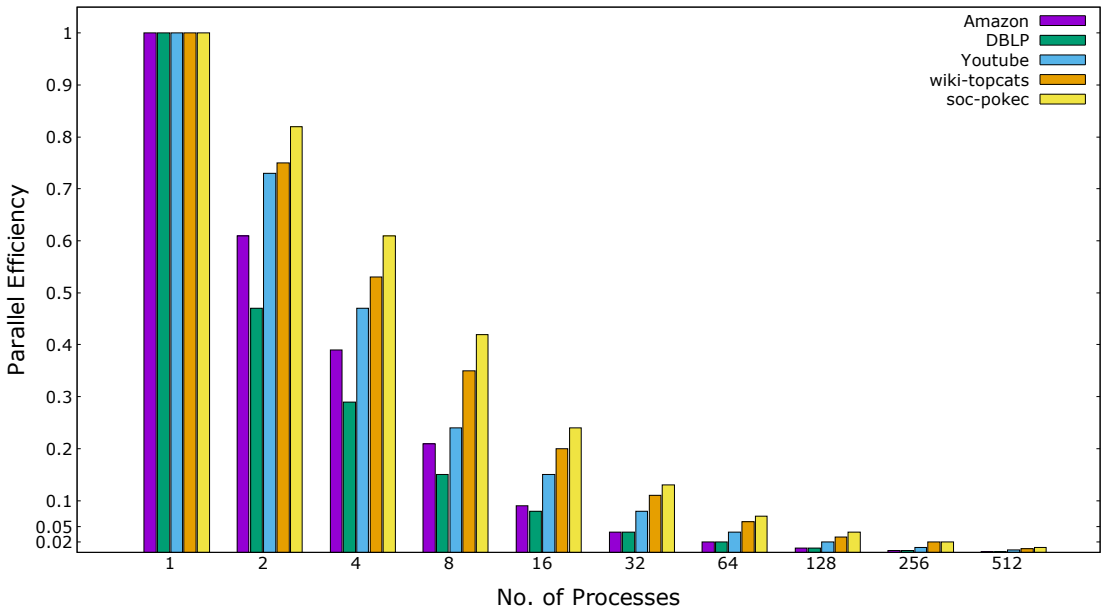


Figure 2.12: Parallel efficiency obtained against different numbers of MPI processes. Larger networks (soc-pokec, wiki-topcats) shows better parallel efficiency up to 512 MPI processes due to the reason of having more computation compared to the smaller networks (Amazon, DBLP).

We utilized parallel efficiency measures to conduct a performance analysis of our distributed implementation. The parallel efficiency, denoted as ϵ, compares the parallel runtime to the best possible runtime assuming perfect scalability [15]. The parallel efficiency is calculated using the formula $\epsilon = \frac{T_{seq}}{pT(p)}$, where p represents the number of parallel units, $T(p)$ is the time with p parallel units, and T_{seq} is the time of the sequential version.

Figure 2.12 illustrates parallel efficiency in the form of a histogram plot for various networks across different numbers of MPI ranks in the distributed platform. The less the change in the histogram bar height, the lower the efficiency gain obtained by increasing the number of processing units. In Figure 2.12, we observe that

the histogram bar for *soc-Pokec* has less change in height than others, followed by *Wiki-topcats* and the rest for an increasing number of processes. As there is more computation to be performed for larger networks, the parallel algorithm exhibits higher parallel efficiency than smaller ones. Parallel efficiency approaching 0 indicates that adding extra processing units or MPI processes for computation may not benefit the task of parallel computation.

2.11 Literature Review

Research on the design of parallel algorithms to minimize computation time for computational problems has gained significant attention in recent years. The emergence of supercomputers with thousands of processing cores and the tremendous growth of big data due to advancements in information technology are playing complementary roles in driving research in this direction. The development of parallel algorithms for graph data analysis, as evidenced by various studies [29, 131, 11, 94, 10, 6, 7, 9], is a crucial outcome of this trend. Parallelizing various community detection approaches mentioned in Table 2.1 in Chapter 2.1 is no exception.

Numerous sequential algorithms focus on modularity optimization [59, 34, 61, 92, 93, 108]. The work by [34] is a swift implementation of Newman et al.'s work [59]. Guimerá [61] posited that finding the modularity of a network is analogous to determining the ground-state energy of a spin system and demonstrated that random graphs and scale-free networks can exhibit modularity. Claire et al.'s study [92] applied modularity optimization in combination with Monte Carlo methods using simulated annealing. Andres et al.'s approach [93] is also grounded in the combination of modularity optimization with simulated annealing. Radicchi et al.'s work [108] follows the spirit of Girvan and Newman's work [59], employing a divisive hierarchical method based on the edge clustering coefficient, in contrast to edge betweenness in [59]. Blondel et al.'s well-known community detection approach [26] also known as Louvain algorithm is based on modularity maximization using a greedy agglomerative heuristic.

Various parallel implementations exist for the modularity-based approach of the Louvain method. Bhowmick et al. [25] provided an OpenMP implementation. Hiroaki et al. [124] demonstrated a fast modularity-based community detection by avoiding searching all the vertices in each iteration. Zhang et al. [154] introduced a distributed framework that speeds up the convergence rate by considering the most suitable candidate vertices to be processed in each iteration. GPU-based parallel Louvain is presented in the studies of Cheong et al. [33] and Naim et al. [95]. Staudt et al. [128, 129] used a combination of the Louvain algorithm and breadth-first search (BFS) for distributed-memory parallelization. Zeng et al. [147, 148] designed a parallel Louvain that can achieve high scalability over thousands of CPU cores. Recent work on parallel implementations of the Louvain algorithm includes Sattar et al. [118]. Sayan et al. [58] demonstrated a distributed+shared memory (MPI + OpenMP) based approach to the Louvain algorithm.

The study by Guimera et al. [61] demonstrated that a random network with an irregular community structure can still display a high modularity value. Consequently, relying on modularity for the process of detecting communities may not deliver high-quality clusters, and the detected communities may not reflect the actual communities. Another caveat of the modularity-based approach is that it may suffer from the *resolution limit* problem, and therefore, it may struggle to detect small communities, as described by Fortunato et al. [57].

The application of statistical inference and generative models for community inference in a network has garnered increased attention in recent years [111, 65, 100]. Among these models, the stochastic block model (SBM) [5, 45, 66] stands out as the most popular, despite its not-so-recent origin. The basic idea is to partition the network's vertices into B blocks, and a $B \times B$ matrix specifies the probabilities of edges existing between the vertices of each block. This model generalizes the community structure [56] by accommodating assortative connections. In this context, the task of detecting communities transforms into a process of statistical inference for the parameters of the generative model, given the observed data. The problem of network partitioning using a statistical inference model is discussed in the works of Tiago et al. [106, 104, 105]. Tiago's work on the stochastic block model for partitioning (often referred to as partitioning in the context of SBM) incorporates the degree-corrected model by Karrer et al. [76] for large-scale dynamic networks. The algorithm exhibits sub-quadratic complexity of $O(N \ln^2 N)$ for a sparse graph, where N is the number of vertices with $N \approx E$. The model presented by Peixoto [106] can function either as a greedy heuristic when partitioning at the block level or as the Markov Chain Monte Carlo (MCMC) method when sampling individual vertices. Peixoto provided an OpenMP-based implementation [103]. Another OpenMP-based work with a modified heuristic for fast network processing has emerged [48]. Distributed parallelization techniques [134, 133] on SBM in Python and *mpi4py* have also surfaced. Achieving raw performance speedup through parallelization in native code is challenging when using a scripting language such as Python. The significant computational slowness of Python compared to C or C++ has been highlighted by comparing three different versions of the baseline algorithm in the study of the streaming graph challenge [73].

As Lancichinetti et al. [81] empirically demonstrated, *Infomap* excels at discovering high-quality communities. This observation is further supported by a comprehensive comparative analysis conducted by Aldecoa et al. [3]. Bae et al. proposed a shared memory-based parallel execution model for *Infomap* [15]. While this model achieves high-quality communities similar to sequential *Infomap*, its scalability is limited, constrained by the number of physical cores and memory in a single machine. Bae et al. also introduced an asynchronous distributed memory-based implementation using the GraphLab framework [87] [16]. However, this distributed implementation demonstrates scalability only up to 128 processing cores. Zeng et al. presented an MPI-based distributed *Infomap* [149] that exhibits scalability for thousands of processors. Nevertheless, the achieved speedup, given the substantial number of processors used, is relatively low. Their work lacks a comprehensive quality analysis of the implementation compared to sequential *Infomap*, except for some small networks (e.g., DBLP, Amazon). It is crucial to emphasize that achieving high scalability in distributed community detection

is equally important as maintaining high quality. The exceptional quality of the detected communities is what sets *Infomap* apart from other approaches for community discovery [81, 3].

2.12 Concluding Remarks

We introduced a distributed *Infomap* algorithm with the aim of ensuring that the quality of detected communities matches that of the sequential algorithm. The task of identifying high-quality community structures within a distributed setting poses significant challenges. We successfully integrated heuristics to effectively address these challenges. To achieve high scalability, we employed cutting-edge workload balancing techniques across processes. In order to preserve accuracy, we implemented synchronization of module information after each iteration. While recognizing the potential for further performance improvements through thorough instrumentation and profiling of our implementation, we acknowledge the success of our approach in maintaining community detection quality.

Chapter 3: HyPC-Map, A Hybrid Memory Parallel Infomap

The serial *Infomap* algorithm faces challenges in scaling for large graphs, given its inherent sequential nature. Parallelizing the *Infomap* method is a complex undertaking. In this investigation, we present a hybrid (shared + distributed)-memory parallel approach for community detection in graphs using *Infomap*. Building upon our previous work on distributed *Infomap* [47], this hybrid design addresses the limitations of processing very large networks encountered in our distributed implementation. Through extensive micro-benchmarking and analysis of hardware parameters, we identify and mitigate performance bottlenecks. Additionally, we employ cache-optimized data structures to enhance cache locality. These optimizations collectively result in a more efficient and scalable community detection algorithm, named HyPC-Map, showcasing a 16× speedup (compared to its single-threaded execution) and a 25× improvement (against the original sequential *Infomap* implementation), all while maintaining the quality of the discovered community.

3.1 Introduction

While the literature on the problem of discovering communities is extensive [59, 34, 97, 26, 114, 98, 75, 106], it has recently garnered increased attention due to the proliferation of social (e.g., human contacts, social media friendships, disease spreading), biological (e.g., protein interaction, genomics), and other graph-related applications. The sheer volume of data that requires processing underscores the necessity for the development of parallel computational strategies in both homogeneous and heterogeneous computing platforms. In recent years, efforts have been made to parallelize various community detection algorithms. Among these algorithms, the Louvain method [26] has attracted perhaps the highest amount of attention, despite its *resolution limit* problem [57]. The Markov clustering technique [136] is another algorithm that has been parallelized [13]. Combining both shared and distributed memory parallelism has demonstrated high scalability in graph clustering applications, as seen in [13]. However, parallel algorithm design studies for *Infomap* [15, 16, 149, 46] did not leverage this approach. Therefore, in this study, we introduce a hybrid memory algorithm, which we call *HyPC-Map*, demonstrating that the hybrid-memory parallel model enhances performance gains over both sequential *Infomap* and our distributed *Infomap* design (Chapter 2), while also

enabling the processing of significantly larger networks than our previous work [46]. We summarize our contributions in this work as follows:

- We amalgamated both distributed-memory and shared-memory-based parallelism to formulate our hybrid parallel algorithm. The Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) frameworks were employed to implement the respective parallelism. Our hybrid parallel algorithm showcases superior scalability compared to related methods in the literature (e.g., [16, 150, 46]).
- We conducted thorough micro-benchmarking and scrutinized memory subsystems to pinpoint and mitigate performance bottlenecks. These analyses enable us to refine the algorithm design. Additionally, we leverage cache-optimized data structures to enhance cache locality.
- Our algorithm scales effectively to large graphs, surpassing the speedups achieved by state-of-the-art techniques grounded in information theory, all while maintaining solution quality—specifically, the quality of detected communities. Through experiments on diverse social and scientific graph datasets, our algorithm exhibits speedups of up to 16x compared to its sequential execution and up to 25x compared to the original sequential implementation by Rosvall [113].

3.2 Algorithmic Analysis and Performance Profiling

Our distributed *Infomap* algorithm achieves scalability with up to 512 MPI processes, as detailed in the preceding chapter (Chapter 2). In specific computation kernels of the algorithm, the application of distributed parallelism introduces notable communication costs across processes, which can outweigh the benefits of distributed computation. However, we have observed that these parts of the algorithm can still gain advantages from shared memory parallelism through the use of multiple threads. To substantiate our observation, we initially parallelize the computation of the steady-state vertex visit rate (i.e., PageRank [27]) as depicted in Algorithm 2, line 8, employing OpenMP threads within each distributed MPI process. Previously, each MPI process sequentially computed the steady-state vertex visit rate using the power iteration method. For larger networks, the results of OpenMP parallel PageRank computation have proven to be highly beneficial.

Figure 3.1 illustrates the breakdown of execution time for various networks after incorporating OpenMP parallel PageRank computation. Smaller networks, such as *Amazon* and *DBLP*, lack sufficient computation to fully leverage parallelism. Consequently, we did not observe significant performance gains beyond $16 - 32$ processes, leading to the omission of scalability curves for these networks in Figure 3.1. However, for larger networks, notable performance gains are evident with an increasing number of processors. For the *YouTube* network, which has approximately 1.1 million vertices and around 3 million edges, the execution time for a single MPI process is roughly 370 seconds, decreasing to around 70 seconds for 256 MPI processes. The scalability curves for more substantial networks, such as *Wiki-topcats* with 1.7 million vertices and 28 million

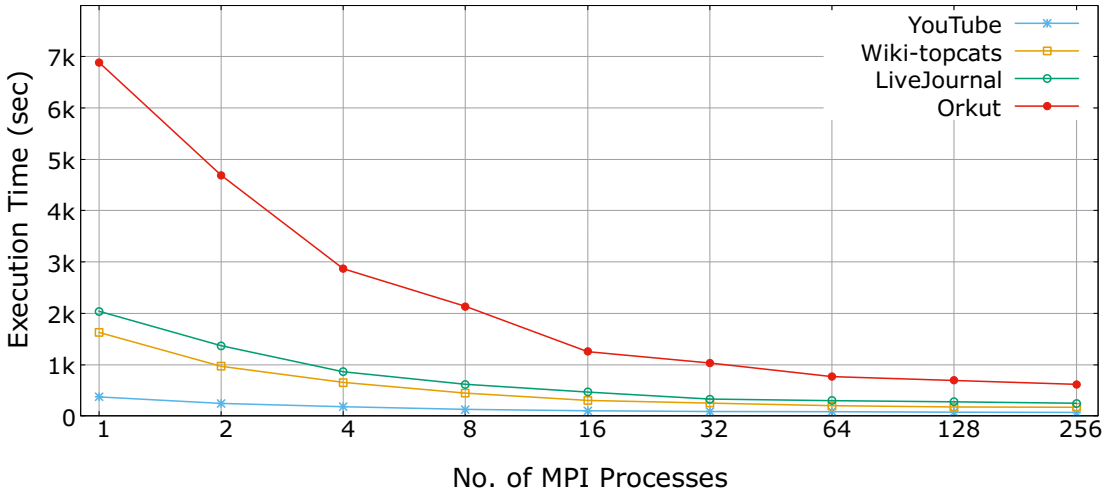


Figure 3.1: Runtime scalability for large networks with the PageRank kernel processed in parallel using shared-memory parallelism (OpenMP) within each MPI process.

edges, and *LiveJournal* with 4 million vertices and 34.6 million edges, exhibit steeper trends. For example, the *Wiki-topcats* network achieves a processing time of 171 seconds for 256 processes, compared to 1630 seconds in the sequential algorithm. In the case of the *LiveJournal* network, the sequential processing time is 2040 seconds, while the parallel processing time is 249 seconds for 256 processes. The largest network in our experiment is the *Orkut* social network, comprising 3 million vertices and 117 million edges. The sequential algorithm takes 6888 seconds to discover communities, whereas it takes only 615 seconds to achieve the same result with 256 processes. This marks a significant performance boost over sequential execution time.

In Figure 3.2, we present the performance gain in terms of speedup. For smaller networks in our dataset, the speedup gain aligns with state-of-the-art techniques. However, for a large network like *LiveJournal*, we achieved a significantly better speedup (8.18×) compared to the work of Zeng et al. [149], which attains a speedup of 3.05× despite utilizing thousands of processes. Their highest speedup, achieved for the *UK-2007* network, is 6.02×, whereas our highest speedup is 11.15× with the largest network, *Orkut*. This observation underscores the influence of network size on the speedup gain in our algorithm—larger networks exhibit higher speedup gains, as evident from the curves in Figure 3.2.

3.3 Optimizing Computational Kernels

Our initial attempt to integrate OpenMP multithreaded parallelism within the distributed MPI processes prompted us to conduct extensive profiling and instrumentation of the primary computational kernels in our *Infomap* algorithm. The implementation can be deconstructed into five major kernels, visualized in Figure

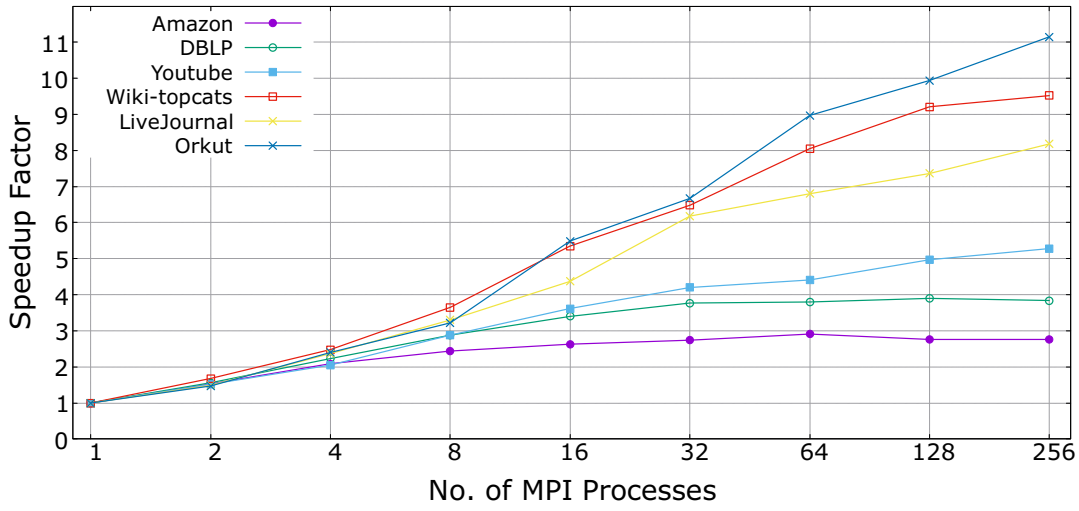


Figure 3.2: Speedup factor achieved for different networks. For larger networks, we observe better speedup such as 11.15× for the Orkut network.

Table 3.1: Performance micro-benchmark of insertion and read operations between c++ map vs unordered_map. There is a significant difference in insertion time between between c++ STL map and unordered_map across different number of entries.

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

3.3 as a stack graph indicating the percentage of their runtime for different networks.

The *PageRank* kernel computes the ergodic node visit frequency using the PageRank [27] algorithm through a power iteration method. The *CreateSuperNode* kernel generates super nodes from a group of vertices with the same module ID, where edges between super nodes represent combined edges with a sum of edge weights. The *UpdateMembers* kernel updates member vertices for each module/community after each iteration. The *FindBestModule* kernel involves finding the community in the individual vertex phase and in the super node phase.

From Figure 3.3, it is evident that the *FindBestModule* kernel is the most time-consuming part of the algorithm, accounting for as much as 89% of the execution time (Orkut network) and up to 74% (YouTube network). The performance of the data structure used inside the kernel significantly contributes to its efficiency.

To store, search, and process community memberships and the corresponding flows of the neighboring adjacency list of a vertex, a key-value-based data structure (map) is employed instead of a regular array (or vector). This choice is justified by the fact that one vertex may have neighboring vertices belonging to distinct communities, or more than one neighboring vertex may collapse into a single community as the algorithm progresses. The map data structure in C++ STL internally uses an *RB-tree* that maintains the ordering of the key, while the unordered_map data structure uses an array and hashing for storage.

Micro-benchmark analysis (Table 3.1) revealed a significant difference in insertion performance between map and unordered_map. The time taken for insertion of each entry (a key-value pair of integer and double value) is nearly half for unordered_map compared to map for varying numbers of entries.

This observation led to a performance improvement for the *FindBestModule* kernel, reducing the execution time from 1095 seconds to 1030 seconds for the Orkut network and from 55 seconds to 37 seconds for the YouTube network. However, the *FindBestModule* kernel still dominates. In an effort to further optimize this kernel, we selectively applied OpenMP multithreading within critical zones inside the kernel, resulting in a substantial performance gain, as evident from Figure 3.5. The execution time of the *FindBestModule* kernel decreased to 240 seconds from 1030 seconds for the Orkut network. Similar performance gains were observed for other networks, as illustrated in Figure 3.5.

3.4 Overview of the Algorithm

Analyzing the distinctions between our distributed-memory parallel *Infomap* and the hybrid approach becomes more comprehensible when examining Algorithm 3. Our parallel algorithm, *HyPC-Map*, is crafted using a combination of both shared memory and distributed memory, drawing inspiration from the map equation discussed in section 2.4 and the heuristics in section 2.5. Subsequently, we refine our algorithm through micro-benchmarking and hardware profiling. HyPC-Map can be segmented into the following principal steps, and the order of these steps is maintained during the actual computation.

1. Computing the ergodic node visit frequency (PageRank) for the network vertices inside each MPI process using a combination of OpenMP-based parallelism.

2. Performing community optimization for each of the subgraphs inside each MPI process in parallel. Each MPI process spawns t-number of OpenMP threads to compute communities for t-number of vertices concurrently. This process continues for multiple iterations.

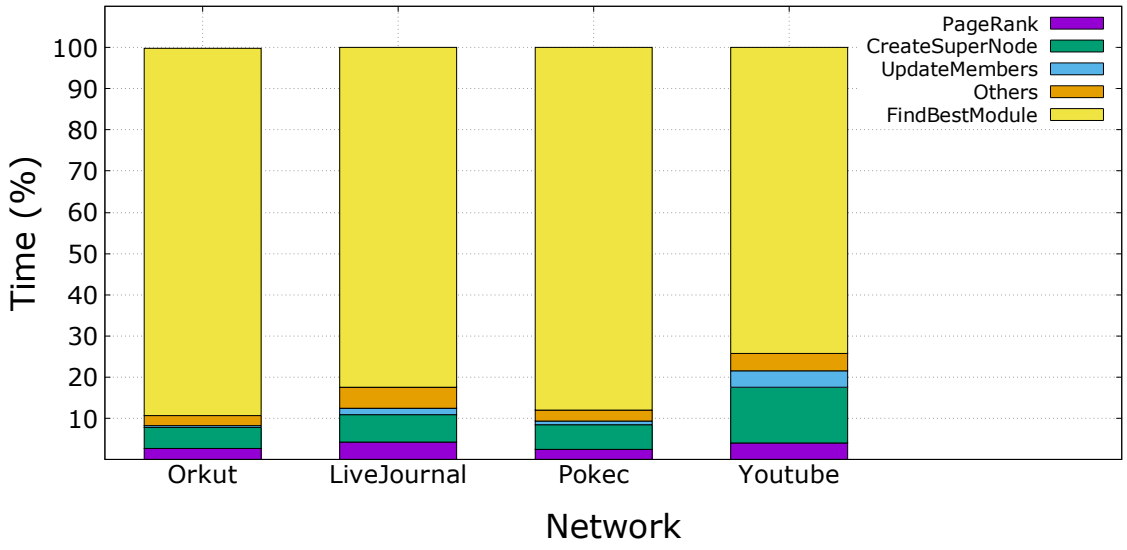


Figure 3.3: Operational kernels in our initial implementation of the distributed (MPI) *Infomap* algorithm. The percentage breakdown of runtime for four different networks is depicted, highlighting the *FindBestModule* kernel as the major portion of the execution time.

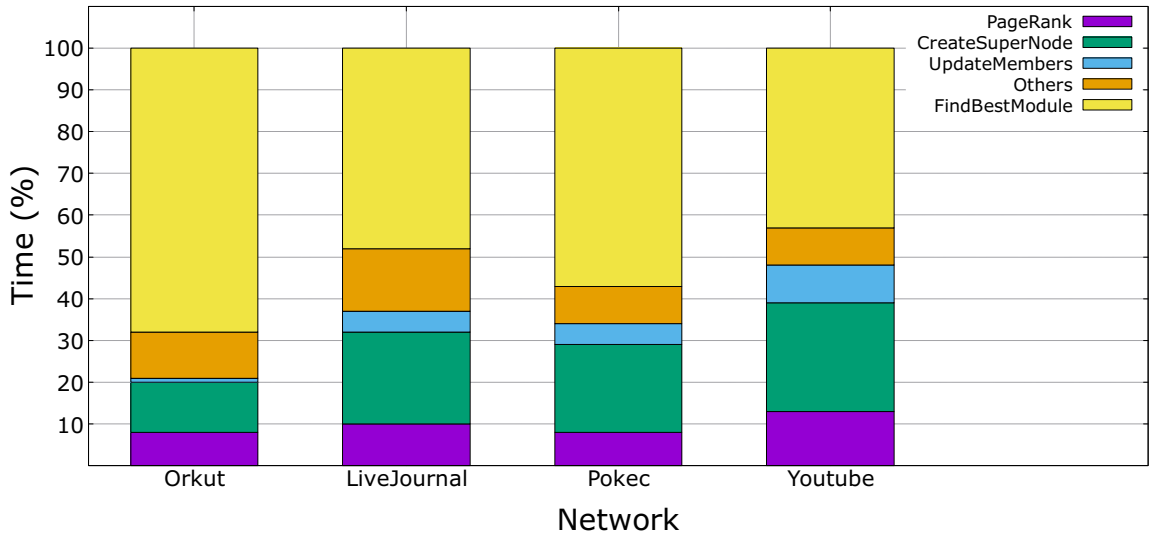


Figure 3.4: Runtime improvement of the operational kernels in Infomap achieved through the implementation of a cache-friendly data structure and the combination of distributed and shared memory parallelism.

3. Exchanging information about the discovered communities of its subgraph vertices with the rest of the MPI processes by each MPI process. This phase is referred to as the vertex-level synchronization step.

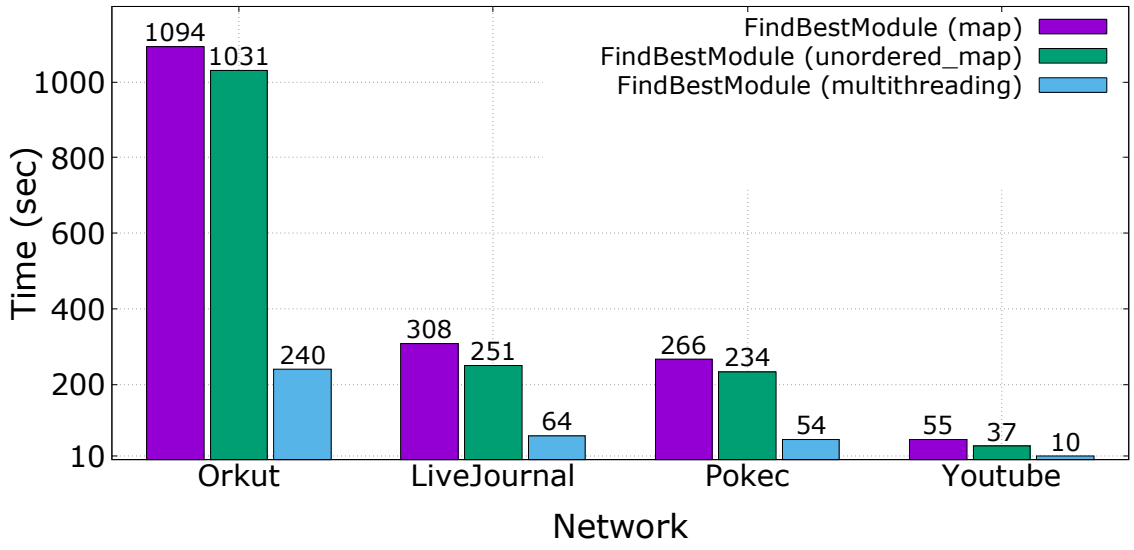


Figure 3.5: Runtime improvement of the operational kernel of Infomap by using cache-optimized kernel and multi-threading. Using cache-friendly data structure (unoreded_map) and multithreading both contribute to the optimization. However, OpenMP parallelism delivers significant kernel optimization, e.g., reducing *FindBestModule* runtime to 240 seconds from 1031 seconds of the unordered_map-based kernel.

4. Creating a super node of the modules it has with a combination of OpenMP parallelism for each MPI process. The edges across distinct vertices of any two supernodes are replaced by a single edge with accumulated edge weights.

5. Spawning t-number of threads by each MPI process to find communities of the t-number of supernode(s) concurrently. This process continues for multiple iterations until no more improvement in the code length occurs.

6. Synchronizing the community membership for the supernode(s) processed by each MPI process with other MPI processes to maintain uniform community information across the MPI processes.

Algorithm 3 outlines the design of the hybrid memory parallel *Infomap*. Lines $1-7$ introduce the notations used in the algorithm. Lines $8-9$ describe the calculation of *PageRank* through power iteration using OpenMP parallelism. Line 10 initializes the set of modules, $\boldsymbol{M}$. Lines $11-12$ compute exit probability q_i in parallel using t OpenMP threads. Line 13 initializes the code length. Lines $14-32$ conduct community discovery in multiple iterations until the change in code length falls below a certain threshold γ. For each process, p, the corresponding vertex range is computed (lines $16-17$) in parallel from the list of *Active* vertices (details in Section 2.5). Community discovery for vertices occurs in lines $18-21$, and community discovery for supernodes takes place in lines $27-30$ using t OpenMP threads inside each MPI process. Line 23 describes the creation of the supernode object consisting of the vertices of a module. The synchronization

Algorithm 3: Hybrid Infomap

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

1 m_i, i$^{\text{th}}$ 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, total MPI processes spawned
7 t, total OpenMP threads spawned
8 Initialize vertex visit rate, $p_{v_i} \leftarrow 1/N$
9 Compute ergodic vertex visit rate p_{v_i} by PageRank in $t - way$ parallel
10 $M \leftarrow \{\forall m_i | (v_\alpha \in m_i) \& (v_\alpha \notin m_j), V = \sum_1^N v\}$
11 **for** $m_i = 1$ *to* $|M|$ *in* $t - way$ *parallel* **do**
12 Calculate exit probability, q_i
13 Initial codelength $L \leftarrow L(M)$
14 **do**
15 $L_{old} \leftarrow L$
16 **for** *process* p *= 1 to* P *in parallel* **do**
17 Compute vertex indices range $[v_{start}, v_{end}]$ from $Active$ vertices list
18 **for** $j = [v_{start}, v_{end}]$, $t - way$ *parallel* **do**
19 Select randomly each vertex, v_j
20 $m_{new} \leftarrow FindBestModule(v_j)$
21 Compute L cumulatively
22 Synchronize $m_{new} \in M$ across P
23 $M \leftarrow CreateSuperNode()$, $t - way$ parallel
24 **for** *process* p *= 1 to* P *in parallel* **do**
25 Compute SuperNode indices $[m_{start}, m_{end}]$ from $Active$ SuperNodes list
26 **for** $j = [m_{start}, m_{end}]$, $t - way$ *parallel* **do**
27 Select randomly each SuperNode m_j
28 $m_{new} \leftarrow FindBestModule(m_j)$
29 $m_j \leftarrow \{m_{new} | \forall v_j \in m_j\}$
30 Compute L cumulatively
31 Synchronize $m_{new} \in M$ across P
32 **while** $(L_{old} - L) > \gamma$
33 **return** $|M|$

of the community memberships of the vertices occurs in lines 22 and 31. Line 33 returns the number of communities $|M|$ after the algorithm converges.

3.5 Experimental Settings

We implement our algorithm using the C++ programming language, MPI, and OpenMP frameworks. The code is compiled using the g++ compiler. The program is designed to support networks in Compressed Sparse Row (CSR) format. The source code of our implementation is accessible online [52].

3.5.1 Computational Infrastructure

We used large compute clusters to perform experimentation on our algorithm. We list the key specifications of those systems below.

LONI: The majority of our experiments were conducted on the LONI supercomputer [67]. Specifically, we utilized the QB2 computing cluster [68], which boasts a peak performance of 1.5 Petaflops. QB2 comprises 504 compute nodes, each equipped with 20 processing cores, resulting in over $10,000$ Intel Xeon processing cores and a 2.8 PB Lustre file system. The computing cluster operates on the RedHat Enterprise Linux 6 Operating System, features a 56 Gb/sec (FDR) InfiniBand, and utilizes a 1 Gb/sec Ethernet network.

For experiments involving the hybrid memory setting, we leveraged the LONI [67] clusters. To maximize speedup gains in each computing node, we employed 10 OpenMP threads in each of the MPI processes. In LONI clusters, each computing node consists of 20 processing cores. Consequently, we executed 2 MPI processes per computing node, each with 10 OpenMP threads, to achieve optimal performance. Due to the limitations on the number of computing nodes available for researchers in LONI, we were constrained to using a maximum of 128 computing nodes and could not test hybrid performance beyond 256 MPI processes.

NERSC Cori: To validate the micro-benchmark analysis and observations discussed in Section 3.3, we employed both the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC) and the LONI QB2 system.

In-house Server: To conduct the comparative analysis mentioned in Section 3.6.6.3, we utilized the compute server designated for research purposes, generously provided by the Department of Computer Science at the University of New Orleans (UNO). The experiments necessitated the installation of various third-party libraries and dependencies, including the *Graphlab PowerGraph* framework [88]. Unfortunately, this setup was not feasible on public servers due to user-privilege restrictions. The server is equipped with 32 Intel(R) Xeon(R) CPU E5-2683 v4 processors, each running at a clock speed of 2.10 GHz. It comprises two sockets, with each socket containing 16 processors, and boasts 512 GB of system memory.

3.5.2 Datasets Used in Experiments

We employed networks of varying sizes, ranging from approximately one million vertices and edges to over a hundred million edges. Table 3.2 provides a concise overview of the datasets, with columns two and

three indicating the number of vertices and edges in each network, respectively. These datasets consist of real-world networks exhibiting a power-law degree distribution, a common characteristic of social networks. The presence of such a power-law distribution allows us to assess our algorithm's performance in worst-case scenarios. All the networks in our datasets were sourced from SNAP [84]. These networks, known for their well-defined community structures, are suitable for evaluating and comparing our implementation using various quality metrics.

Table 3.2: Scale-free network datasets used for our experiments that exhibit power-law degree distribution. These are social and information networks with the largest one (Orkut) having 3M vertices and 117M edges.

Network	# Vertices	# Edges	Modularity	Conductance	Description
Amazon	334863	925872	0.77	0.23	Co-purchase network
DBLP	317080	1049866	0.59	0.41	CS bibliographical network
Youtube	1134890	2987624	0.40	0.58	Youtube social network
soc-Pokec	1632803	30622564	0.34	0.66	Pokec social network
Wiki-topcats	1791489	28511807	0.40	0.58	Wikipedia hyperlinks
LiveJournal	3997962	34681189	0.47	0.53	LiveJournal social network
Orkut	3072441	117185083	0.42	0.55	Orkut social network

3.6 Performance Evaluation

We assess the performance of our algorithm by considering both solution quality and parallel scalability, comparing it with other information-theoretic approaches and the Markov Clustering algorithm (MCL) for community detection.

3.6.1 Quality Analysis of Discovered Communities

To assess the quality of the detected communities, we employed metrics such as *modularity*, *conductance*, and *convergence MDL*. We conducted a comparison with the results obtained using the sequential *Infomap*. The values of *modularity* and *conductance* for the sequential *Infomap* are provided in columns 4 and 5 of Table 3.2.

3.6.2 Convergence of the Objective Function

The objective function of *Infomap* aims to minimize the Minimum Description Length (MDL). Achieving MDL improvement in a hybrid memory setting poses similar challenges. In distributed implementations, the risk of premature convergence can lead to less MDL minimization, as noted by [16]. Our optimization of the objective function 2.4 yielded results closely aligned with the MDL improvement reported in [15]. Figure 3.6 illustrates the final converged values of MDL. The differences in MDL are minimal across all cases, ranging from as low as 0.08% (Amazon) to a maximum of 3% (Wiki-topcats). Consequently, the

quality of the discovered communities post-convergence is comparable to that of *RelaxMap* [15]. The slight differences in MDL can be attributed to the fact that the compression outcome resulting from the community membership update for any preceding vertex may not be readily available to other processes in the distributed execution until the synchronization step. The decision to update MDL can be influenced by this, unlike in sequential/shared-memory implementations.

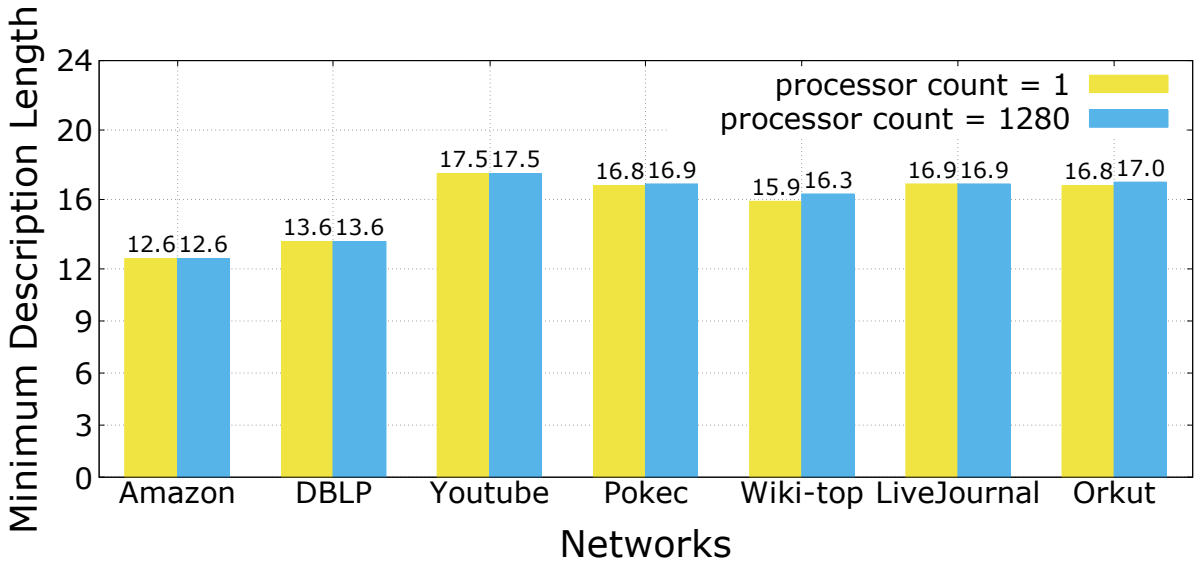


Figure 3.6: Illustration of the quality of discovered communities in terms of MDL. The numeric value atop each histogram bar in each figure represents the values of the quality metrics for 1 vs 1280 processors.

3.6.3 Modularity

For the dataset in Table 3.2, we provided the quality measurement metrics, such as modularity and conductance, along with their corresponding values obtained from the sequential *Infomap* [15]. As depicted in Figure 3.7, we observe minimal variations in the values of modularity across different networks presented in the figure. Specifically, we note no change in modularity for the *DBLP* network, no variation for the *LiveJournal* network, and a difference of less than 2% for 1280 processing cores in the case of the *Orkut* network.

3.6.4 Conductance

We assessed the quality of the identified communities in a manner analogous to our evaluation for modularity, extending our analysis to conductance. Examining Figure 3.8, we observe negligible distinctions between executions with 1 processor and 1280 processors for both the *DBLP* and *LiveJournal* networks. In the case of the *Orkut* network, the conductance value exhibits a variance of less than 2% for 1280 processors. Consequently, we can infer that the quality of the discovered communities in our *Infomap* implementation scales effectively with varying numbers of processors across different networks.

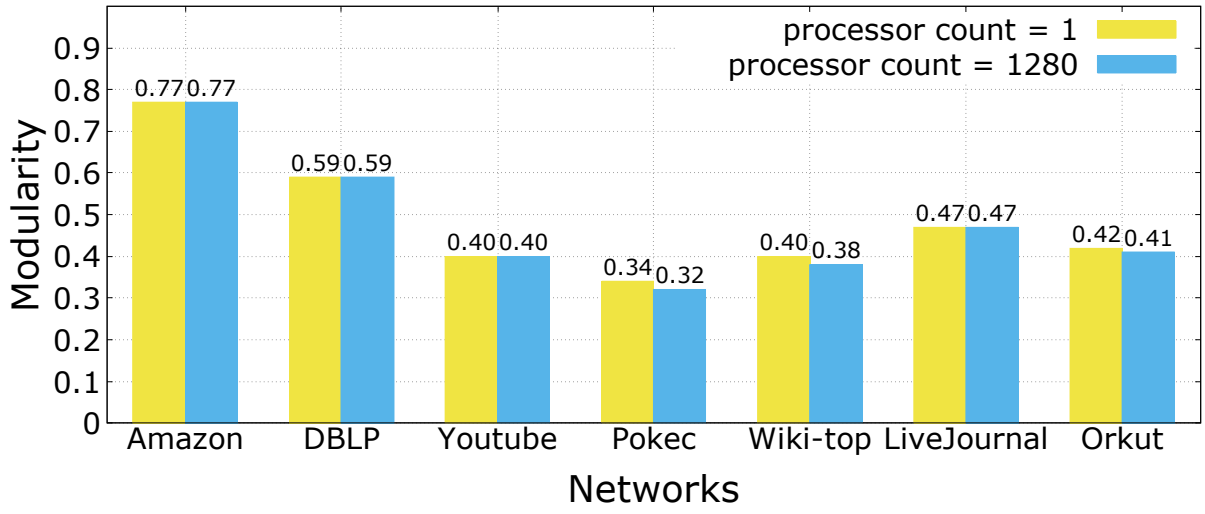


Figure 3.7: Illustration of the quality of discovered communities in terms of modularity. The numeric value on top of each pair of histogram bars represents the values of the quality metrics for 1 vs 1280 processors.

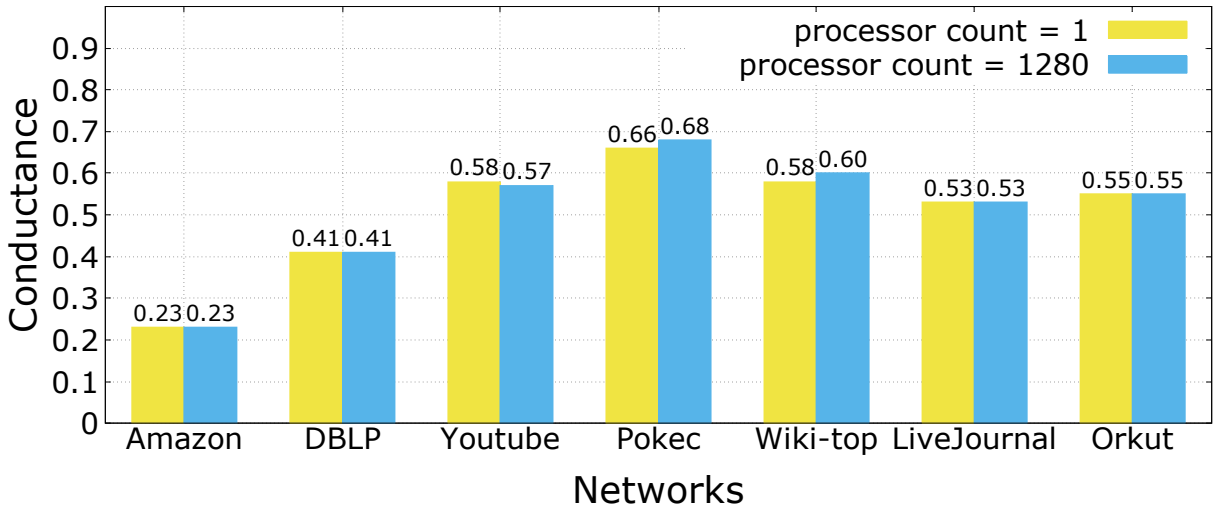


Figure 3.8: Illustration of community quality in terms of conductance. The numeric value on top of each pair of histogram bars represents the values of the quality metrics for 1 vs 1280 processors.

3.6.5 Normalized Mutual Information

Normalized mutual information (NMI) serves as a metric for comparing the quality of discovered communities against the ground truth communities in a given network. The mathematical expression is given by:

$$NMI(Y,C) = \frac{2 \times I(Y;C)}{|H(Y)| + |H(C)|} \tag{3.1}$$

where Y represents the truth community, C represents the computed community, $H(.)$ denotes entropy, and $I(Y;C)$ is the mutual information between Y and C.

Similar to modularity and conductance, NMI serves as a metric to evaluate scalability in terms of quality for different numbers of processors. The consistency of the quality across various processors is reported in Tables 3.3, 3.4, and 3.5, utilizing both real-world networks and synthetic networks with known truth partitions. As NMI requires a known truth partition, we utilized static graphs from the *MIT GraphChallenge network data sets* [139], as illustrated in Table 3.5.

Table 3.3: Scalability of HyPC-Map in terms of the quality metrics: Modularity

Network	1	20	40	80	160	320	640	1280
Amazon	0.23	0.23	0.23	0.23	0.23	0.23	0.23	0.23
DBLP	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41
LiveJournal	0.53	0.53	0.53	0.53	0.53	0.53	0.53	0.53
Orkut	0.55	0.51	0.55	0.55	0.54	0.56	0.54	0.56

Table 3.4: Scalability of HyPC-Map in terms of the quality metrics: Conductance

Network	1	20	40	80	160	320	640	1280
Amazon	0.23	0.23	0.23	0.23	0.23	0.23	0.23	0.23
DBLP	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41
LiveJournal	0.53	0.53	0.53	0.53	0.53	0.53	0.53	0.53
Orkut	0.55	0.51	0.55	0.55	0.54	0.56	0.54	0.56

Table 3.5: Demonstrating scalability of HyPC-Map in terms of Normalized Mutual Information (NMI) for different number of processors.

			NMI							
Network	**# Vertices**	**# Edges**	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

3.6.6 Parallel Performance

3.6.6.1 Speedup Gain

In Table 3.6, we present the achieved performance gain in terms of speedup. To the best of our knowledge, our parallel implementation has attained superior speedup compared to state-of-the-art information-theoretic approaches to community discovery. For smaller networks such as *Amazon*, *DBLP*, and *YouTube*, the speedup

Table 3.6: Speedup comparison with sequential (1-core/process) execution of our *HyPC-Map* (column 2) and with original sequential implementation of *Infomap* [113] by Rosvall et al. [114] (column 3).

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

gains are 2.78, 3.66, and 4.58, respectively. In the case of larger networks like *Wiki-topcats*, *Soc-pokec*, and *Orkut*, the speedup gains are 10.52, 12.52, and 16.16, respectively. These speedups significantly surpass those achieved by state-of-the-art implementations [149, 150]. Furthermore, we compare the speedup with the original sequential implementation of *Infomap* [113] by Rosvall et al. [114]. We observe even more substantial speedup, reaching as high as $\sim$ 25X for the *LiveJournal* network and $\sim$ 21.4X for the *Orkut* network—both of which are large networks. This underscores the advantages of using cache-optimized data structures and an efficient community optimization kernel that reduces sequential computation time. The experiments encompass various numbers of MPI processes, ranging from 1 to 128, with each process spawning 10 OpenMP threads. This configuration allows us to leverage all available processors within a single QB2 [68] compute node. The highest speedup is achieved when utilizing 1280 processors (128×10).

3.6.6.2 Scalability Analysis

We present a runtime comparison of our implementation for three different large networks. For the *Orkut* network in Figure 3.9, the runtime on a single processor, initially 2836 seconds, is reduced to 176 seconds with 1280 processors. For the *LiveJournal* network, the runtime decreases to 104.7 seconds with 1280 processors, down from a single-processor runtime of 840 seconds. Similarly, for the *Pokec* network, the runtime is as low as 63 seconds with 1280 processors, while the single-core runtime is 787 seconds.

3.6.6.3 Comparison with state-of-the-art techniques

In Table 3.7, we conducted a comprehensive comparison between *HyPC-Map* and state-of-the-art techniques, including the original *Infomap* and parallel algorithms developed up to the time of this study. The table includes an overview of the strengths and weaknesses of each approach. We specifically considered *GossipMap* and *Distributed Infomap* for comparison since both employ distributed-memory kernels. Despite employing 4096 processing units, *Distributed Infomap* achieves a maximum speedup of 6.02×. Its implementation is not publicly available as well. Therefore, we opted to compare with *GossipMap*. *GossipMap* reformulates

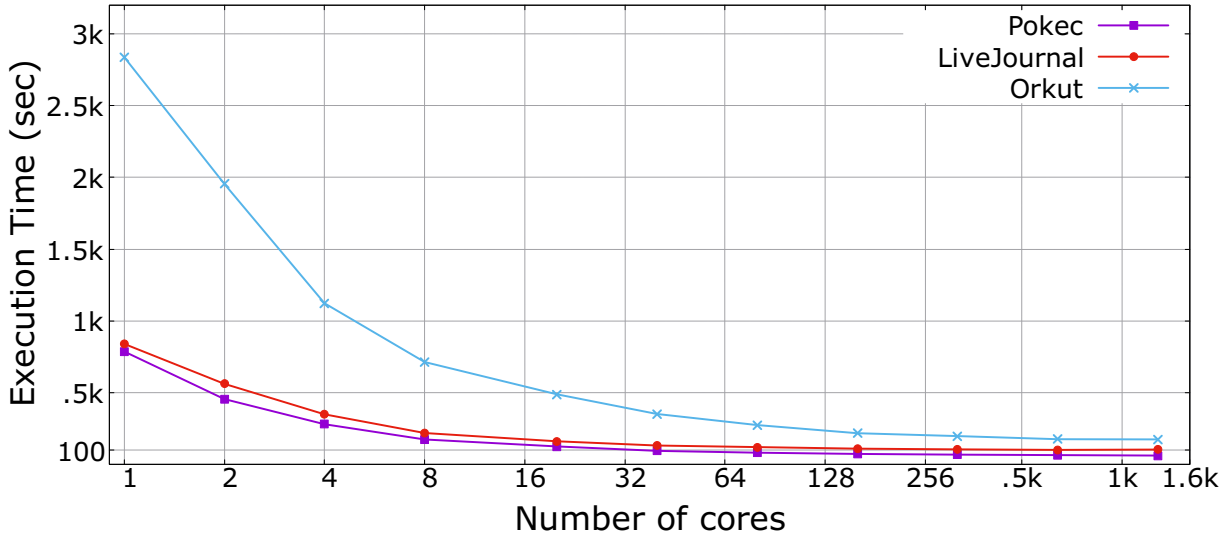


Figure 3.9: Illustrating the scalability of the execution time for *Orkut*, *LiveJournal*, and *Pokec* network.

Table 3.7: Comparison of *HyPC-Map* with state-of-the-art techniques

Work Name	**Type**	**Strength**	**Weakness**
Infomap [114]	Sequential	High accuracy	Computationally expensive
RelaxMap [15]	Shared-memory parallelism	High accuracy	Scalability limited to a single node
Gossipmap [16]	Asynchronous distributed memory parallelism	Asynchronous	Scalability up to 128 parallel units
Distributed *Infomap* [46]	Synchronous distributed memory parallelism	Scales to 512 processors	Speedup up to $\sim$ 5X
Distributed *Infomap* [149]	Synchronous distributed memory parallelism	Scales to $\sim$ 4k processors	Speedup up to $\sim$ 6X
HyPC-Map [49]	Synchronous hybrid memory parallelism	High accuracy & speedup	

the map equation for incremental computation and local evaluation, leveraging an asynchronous gossiping protocol for information approximation. The reported scalability for *GossipMap* is up to 128 parallel units.

We conducted experiments on our local computing server due to numerous dependencies for *GossipMap*, as discussed in section 3.5.1. The comparison involved *GossipMap*, single-threaded distributed *HyPC-Map*,

and multi-threaded distributed *HyPC-Map*. Figures 3.10, 3.11, and 3.12 present runtime and scalability comparisons for three different networks between *GossipMap*, single-threaded distributed *HyPC-Map*, and multi-threaded distributed *HyPC-Map*.

Figure 3.10 highlights the efficiency of *HyPC-Map*, with single-processor runtimes of 730, 660, and 600 seconds compared to *GossipMap*'s runtimes of 6734, 4316, 5800 seconds for the *LiveJournal*, *Pokec*, and *Wiki-topcat* networks, respectively. Table 3.8 details the runtime differences using 16 or 32 processing units between the distributed and hybrid forms of *HyPC-Map*. The hybrid form achieves superior runtime and parallel efficiency, reducing communication overhead during synchronization among fewer distributed processes while using the same number of processing units.

Ultimately, the hybrid form of *HyPC-Map* surpasses *GossipMap* in both execution time and relative parallel efficiency ($\varepsilon_r = \frac{p_1 T(p_1)}{p_2 T(p_2)}$). The relative parallel efficiency of *GossipMap* decreases more significantly with an increasing number of processing units. Here, $T(p_1)$ and $T(p_2)$ represent the execution times for p_1 and p_2 parallel units, respectively. Table 3.8 provides a detailed comparison between *GossipMap* and *HyPC-Map* in terms of relative efficiency (ε_r) for various scenarios.

It is crucial to note that the primary goal of *HyPC-Map* is to achieve superior performance and scalability for rapid community discovery. The hybrid form effectively bridges the gap between high communication costs due to synchronization and reduced quality due to asynchronous community optimization. The quality comparison between *GossipMap* and *HyPC-Map* in terms of MDL is illustrated in Figure 3.13, demonstrating comparable values.

Table 3.8: Relative efficiency ε_r between *GossipMap* and *HyPC-Map*

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

3.6.6.3.1 Comparison with other community discovery strategies HipMCL [13] is a parallel community discovery algorithm that utilizes the Markov Clustering algorithm (MCL) [136] as its core. Lancichinetti et al. [81] conducted various comparisons among Infomap, Louvain, and MCL in their work, employing different benchmarks such as GN, LFR, and random graphs. Demonstrated by Azad et al. [13], HipMCL, as a state-of-the-art clustering technique, addresses the performance and memory limitations of MCL. Its community optimization kernel employs SpGEMM (sparse matrix-matrix multiplication). The SpGEMM kernel implements a sparse form of SUMMA [135] for distributed graph computation in adjacency matrix

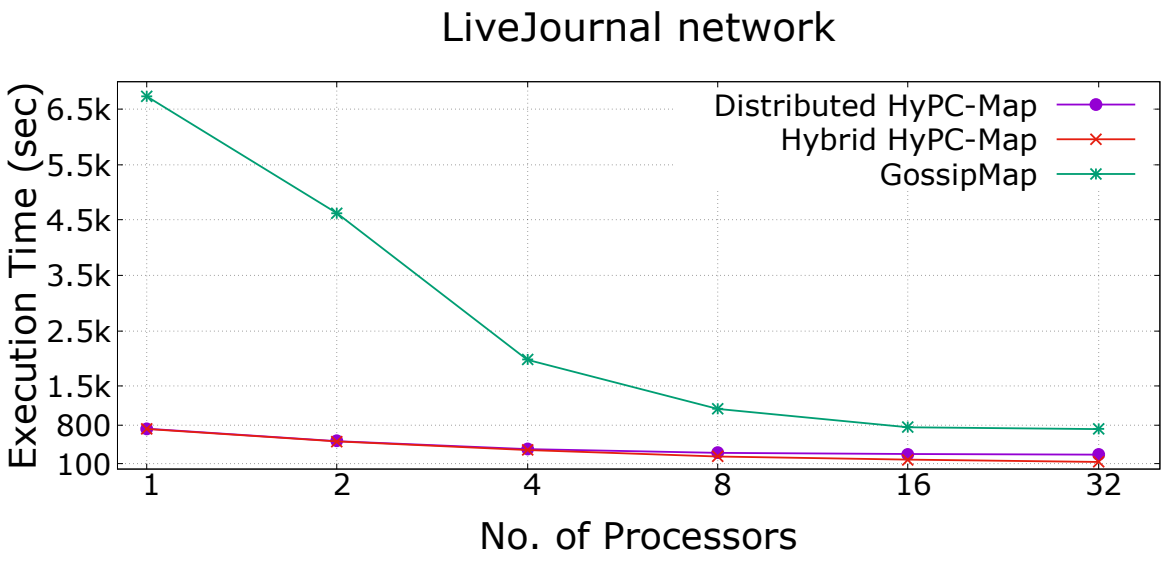


Figure 3.10: Runtime comparison for *LiveJournal* network between GossipMap and HyPC-Map (single-thread distributed and multi-thread distributed).

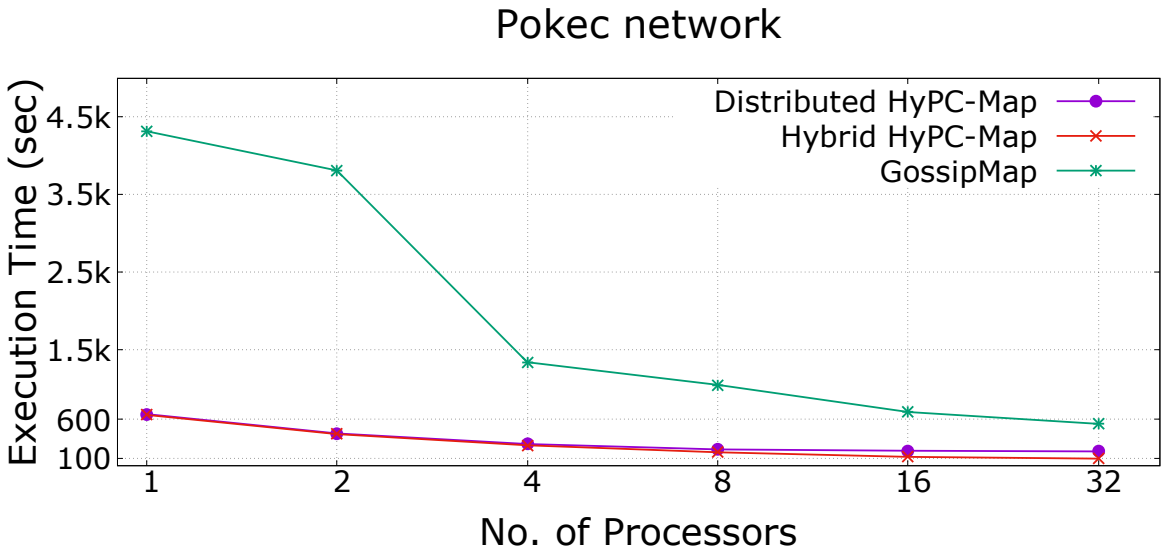


Figure 3.11: Runtime comparison for *soc-Pokec* network between GossipMap and HyPC-Map (single-thread distributed and multi-thread distributed).

form, with the requirement that the number of MPI processes be a perfect square number to form a square grid (e.g., 1×1, 2×2, 4×4).

In our comparison between HipMCL and *HyPC-Map*, we observed that *HyPC-Map* outperforms in terms of memory requirement and runtime performance. HipMCL maintains three matrices in the SpGEMM kernel,

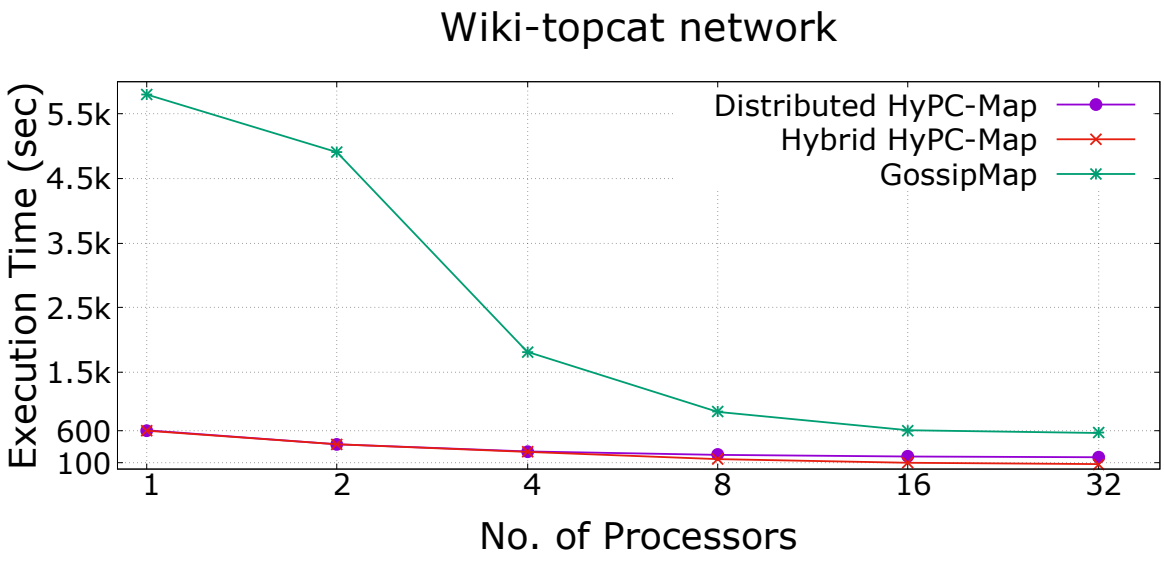


Figure 3.12: Runtime comparison for *wiki-topcats* network between GossipMap and HyPC-Map (single-thread distributed and multi-thread distributed).

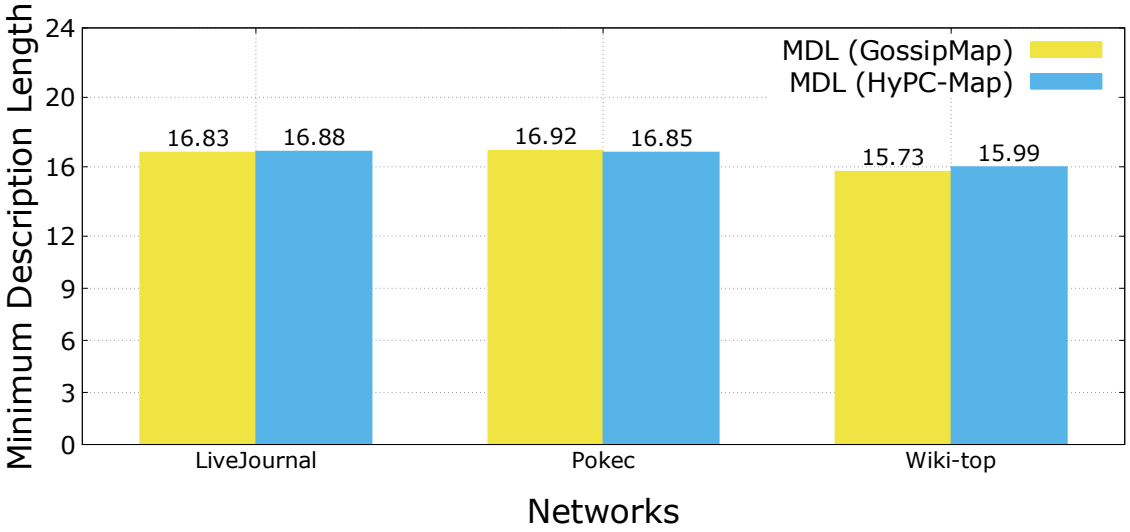


Figure 3.13: We observe similar MDL for 3 different networks after the convergence of both the *GossipMap* and *HyPC-Map*.

which is significantly higher than what *HyPC-Map* requires. Consequently, larger networks from our datasets in this study could not be processed by HipMCL, utilizing all available 128 GB memory of the NERSC Cori Haswell node, due to memory limit exceeding (MLE), as listed in Table 3.9. This limitation was observed when using a single compute node (e.g., *Youtube*, *Orkut*) and four compute nodes (e.g., *Orkut*). HipMCL takes a substantially longer time to process real-world social networks listed in Table 3.9, which follow a power-law degree distribution. Despite utilizing all 32 processors of the Haswell compute node and multiple

compute nodes (e.g., 4, 16), HipMCL still lags behind the single-node runtime of *HyPC-Map*.

Fortunato et al. [57] discussed the resolution limit problem inherent in *modularity*-based community detection strategies (e.g., Louvain). A parallel implementation of the *modularity*-based algorithm inherits this problem, along with additional challenges arising from parallelization. Lancichinetti et al. [81] provided detailed comparisons between sequential Louvain and Infomap in their work.

Table 3.9: Execution performance comparison between *HipMCL* [13] and *HyPC-Map*. MLE: Memory Limit Exceeded

	HyPC-Map (sec)	**HipMCL (sec)**		
Network	1 Compute Node	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

3.7 Concluding Remarks

HyPC-Map combines the advantages of both distributed- and shared-memory parallelism, resulting in superior scalability performance compared to state-of-the-art techniques. Moreover, our algorithm exhibits greater efficiency when using a single processing unit than other prominent map-based algorithms [113, 16]. *HyPC-Map* achieves significantly higher parallel performance than other map-based parallel algorithms in the literature. Despite achieving such speedup, HyPC-Map does not compromise on maintaining the quality. The modularity, conductance, and MDL scores attest to the high quality of detected communities, which closely resemble those found by sequential *Infomap*. We believe *HyPC-Map* holds potential for analyzing emerging large-scale social, information, and scientific networks.

Chapter 4: Fast Infomap with Accelerated Hash Accumulation

The information-theoretic community discovery method, commonly known as *Infomap*, is renowned for providing superior quality results in the Lancichinetti–Fortunato–Radicchi (LFR) benchmark compared to modularity-based algorithms. To address the computational challenges posed by the analysis of massive graphs arising from the exponential growth of information in various domains such as bio-sciences, social sciences, and business, parallel algorithms have been developed for *Infomap*. State-of-the-art techniques in information-theoretic community discovery often rely on hash tables to store vertex neighborhood flow information. However, these operations can be computationally expensive due to collision handling and CPU branch mispredictions. Recently, the Accelerated Sparse Accumulation (ASA) hardware accelerator for hash accumulation has been introduced, specifically designed for sparse matrix-matrix multiplication (SpGEMM). We extend the interface of the ASA accelerator and demonstrate that, for state-of-the-art parallel *Infomap*, utilizing the accelerator for hash accumulation with fast on-chip memory can overcome the performance bottlenecks associated with software hash tables. This approach achieves a speedup of 5.56× while reducing the number of branch mispredictions by 59%, the CPI rate by 21%, and the total number of instructions by 24%.

4.1 Introduction

Community discovery is a widely applied task involving the grouping or clustering of entities with similar characteristics [59, 34, 108, 61, 92, 93, 97, 26, 114, 98]. Applications of community discovery span various domains, such as identifying people with similar interests in social networks, marketing products based on consumer categories, clustering proteins with similar functions, detecting web spam in cybersecurity, and more. The significant expansion of social, biological, professional, and traffic networks in recent years has spurred research into parallel algorithm designs for community discovery [106, 124, 95, 15, 16, 149, 118, 150, 13, 46]. The adoption of an *information-theoretic* approach, commonly referred to as *Infomap* [114], has been demonstrated to yield higher-quality discovered communities in separate studies [81, 3] and experimental LFR benchmarks [83]. Numerous shared-memory and distributed memory-based parallel algorithms have

been developed for Infomap [15, 16, 149, 46, 49].

The study discussed in Chapter 3 [49] devised a parallel version of Infomap, leveraging both shared-memory and distributed-memory parallelism. This implementation achieved a speedup of 25× compared to the sequential counterpart of Infomap [114]. An essential phase in determining the community membership of a vertex involves computing and accumulating information regarding neighboring vertices. Whether in sequential [114] or parallel implementations [15, 16, 46, 49], all versions of Infomap utilize software hash tables for storing information about neighboring vertices. Subsequent sections will demonstrate that software hash accumulation consumes up to $50 - 65\%$ of the total execution time, representing a significant performance bottleneck in hardware resource utilization, specifically due to stalls resulting from branch misprediction.

We illustrate that an accelerator designed for hash accumulation with fast content-addressable memory (CAM) can effectively address challenges in software hash tables and narrow the gap between achievable and utilized hardware resources. To the best of our knowledge, none of the existing works [114, 15, 16, 149, 150, 46, 49] on Infomap community detection has explored hardware acceleration to expedite hash operations. Chao et al. [151] introduced an accelerator for hash accumulation (ASA) specifically tailored for SpGEMM computation. In this paper, we extend the ASA interface [151] beyond its original SpGEMM context, enabling any application with a high volume of hash lookup and accumulation to benefit from ASA. To showcase this, we enhance the parallel Infomap [49] implementation with ASA-accelerated hashing operations, demonstrating that hash operations achieve speedups ranging from 3.28× to 5.56× by utilizing ASA. The limited capacity of the CAM may raise concerns when storing hash tables for large networks. However, real-world social and metagenome networks exhibit power-law degree distributions and sparsity, as demonstrated in Figure 4.4 in Section 4.4 for social networks. Applications such as metagenome assembly [90] or clustering protein sequences [86] deal with similar network characteristics. In Section 4.4, we demonstrate that the accelerator's memory's limited capacity is not a hindrance when processing sparse networks, as 99% of the adjacency list (vertex neighbors) fits entirely within an 8KB CAM. Following is the summary of our contribution:

- We extend the ASA interface initially designed for accelerating SpGEMM computation by Chao et al. [151] and showcase its adaptability for applications involving a substantial volume of hash operations. To our knowledge, this marks the first instance where an accelerator is employed to enhance hash operations for Infomap community discovery.
- In the context of the Infomap application, ASA achieves a reduction of 59% in branch misprediction, a decrease in the CPI rate by 21%, and a decline in the total number of instructions by 24%. This is achieved by eliminating the computationally expensive software hash accumulation and collision handling operations.

- We illustrate that the on-chip CAM's limited capacity in ASA is not a hindrance when dealing with large social and biological networks. As detailed in Section 4.4, we demonstrate that over 99% of the vertices can be effectively processed using only 8KB of CAM per core.

4.2 Background

In this section, we provide some background on the application of the information-theoretic approach to community detection and the motivation for an accelerator for hash accumulation.

4.2.1 Components of A Parallel Infomap Algorithm

In [49], an effective parallel implementation of Infomap, referred to as *HyPC-Map*, is introduced. HyPC-Map consists of four primary compute kernels, which we will briefly outline to provide context for our subsequent methods and contributions.

PageRank: This kernel calculates the ergodic vertex visit probability (PageRank) for all vertices, considering teleportation. The PageRank [27] is computed through the power iteration method. The ergodic vertex visit frequencies are utilized to determine both the module stay probability $p^i_{\circlearrowright}$ and the exit probability of a vertex $q_i\curvearrowright$ from module i.

FindBestCommunity: This compute kernel is tasked with greedily determining the optimal community for each vertex. It functions in both the vertex-level phase and the super node-level phase. In the vertex-level phase, it iterates through each vertex, selecting the merge with a neighboring vertex that maximally reduces the MDL in Equation (2.4). In the super node-level phase, the groups of vertices produced in the vertex-level phase are delivered to this kernel in the form of a structure known as a super node.

Convert2SuperNode: The sets of vertices produced during the vertex-level phase in the *FindBestCommunity* kernel are denoted by a structure named a super node. Within a super node, the constituent elements encompass all the vertices belonging to a specific group. These member vertices may exhibit connections to other super nodes via edges. In cases where multiple vertices within one super node are linked to another super node, a solitary super edge is established, amalgamating the associated edge weights.

UpdateMembers: Following the determination of new community memberships by the *FindBestCommunity* kernel for an individual vertex or a set of vertices, the community membership field for each of the vertices is subsequently revised.

4.2.2 Motivation for Accelerator

To analyze the computational costs of the primary kernels in parallel *Infomap*, we conducted experiments on large networks, and the results are depicted in Fig. 4.1. The breakdown indicates that the *FindBestCommunity* kernel (yellow bar) is the most time-consuming component in *Infomap*, constituting 70% to 90% of the entire application. Moreover, we observe that software hash operations (orange bar) account for as much as 50% to

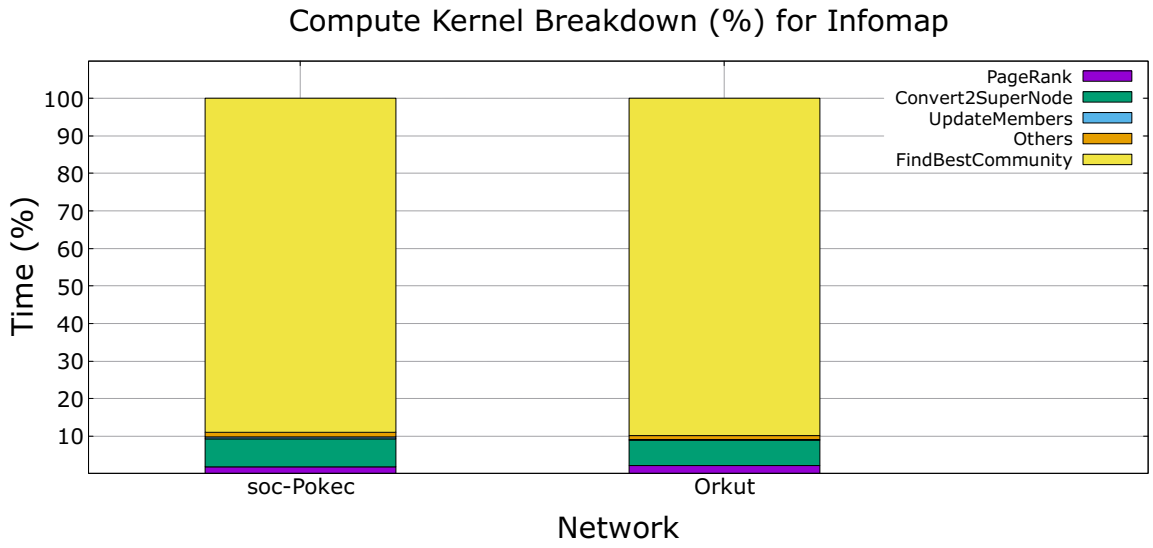


Figure 4.1: The kernel breakdown of the Infomap application in native execution for large networks (Pokec and Orkut). The majority of time is spent on *FindBestCommunity* kernel.

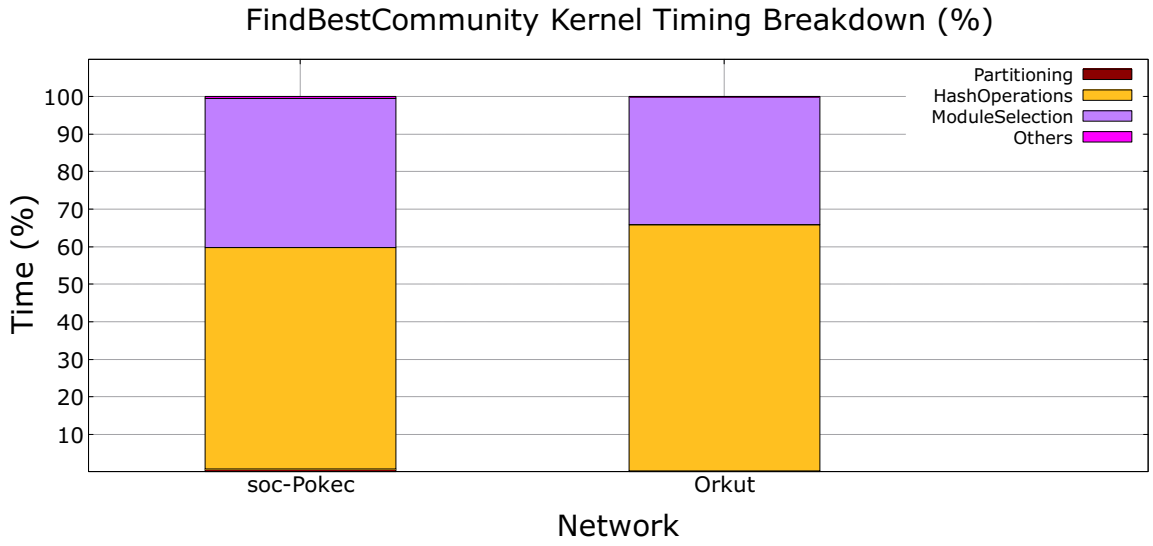


Figure 4.2: A further breakdown of the *FindBestCommunity* kernel shows hash operations taking 50% to 65% of the kernel computation time.

65% of the *FindBestCommunity* kernel (see Fig. 4.2). For simplicity, all plots shown in Fig. 4.1 and 4.2 represent single-core executions of the application.

This heavy reliance on software hash tables arises from how the *Infomap* implementation decides to change modules for an arbitrary vertex. As discussed in the extended version of the *Map Equation* [114, 46], it

Algorithm 4: FindBestCommunity

```
   Data: A vertex/supernode v_i of graph G(V, E)
   Result: New community m_new for vertex v_i
 1 std::unordered_map<pair<int, double≫ outFlowtoModules
 2 std::unordered_map<pair<int, double≫inFlowFromModules
 3 numModlinks ← 0
 4 for (linkIt ← v_i.outLinks.begin()  to  v_i.outLinks.end() ) do
 5     newModId ← node.at(linkIt → first).modId
 6     if (outFlowtoModules.count(newModId) > 0) then
 7         outFlowtoModules[newModId] + = linkIt → second
 8     else
 9         outFlowtoModules[newModId] ← linkIt → second
10         inFlowFromMod[newModId] ← 0.0
11         numModLinks ← numsModLinks + 1
12 Accumulate incoming flow in inFlowFromModules (as in Ln. 4 − 11)
13 bestDiffCodelen ← 0.0
14 for ( it ← outFlowtoModules.begin()  to  outFlowtoModules.end() ) do
15     newModId ← it → first
16     outFlowToNewMod ← it → second
17     inFlowFromMod ← inFlowFromModules[newModId]
18     diffCodeLen ← calc(outFlowToNewMod,  inFlowFromMod)
19     if (diffCodelen < bestDiffCodelen) then
20         bestDiffCodelen ← diffCodelen
21         bestModuletoMove ← newModId
22 return bestModuletoMove
```

is sufficient to keep track of the exit probability $q_{i\curvearrowright}$ of a module i and the stay probability of the module $\sum_{\alpha \in i} p_\alpha$ to determine the next possible move for a vertex or super node. The exit probability and the stay probability are updated based on the incoming and outgoing flow information between modules. A closer look at the *FindBestCommunity* kernel is provided in Algorithm 4.

In the module selection process for each vertex outlined in Algorithm 4, every vertex or supernode manages a pair of hash tables to store incoming and outgoing flow from or to neighboring vertices or supernodes, respectively. This flow information is vital for determining the best reduction in MDL during the processing of each vertex. However, the frequent utilization of the hash table in this context makes it a computationally expensive phase (orange bar) of the *FindBestCommunity* kernel. The software hash table, implemented using the C++ standard library unordered map, faces challenges such as high latency-bound memory access resulting from branch misprediction and collision chaining.

At the initiation of the *FindBestCommunity* phase (as outlined in Algorithm 4), each vertex is initially assigned to its individual community or module. Throughout multiple iterations of *FindBestCommunity*, vertices may

undergo transitions from one module to another in a greedy optimization manner. Such transitions can occur by following connecting edges or through random teleportation. Vertices may have neighboring vertices or none, and these neighbors can belong to the same or different modules. The decision for a vertex to move to another module is based on achieving the maximum compression of the minimum description length (MDL), as described in Equation (2.4). The MDL minimization, associated with a vertex's move to a module, is influenced by the edge flow to or from other neighboring vertices [114]. This flow to or from a module is expressed as a function of the ergodic node visit frequency of the vertex itself and the edge weight to or from that module.

In Algorithm 4, lines $1-2$ declare two hash tables for storing the outgoing flow to other modules and the incoming flow from other modules. Lines $4-11$ iterate over the adjacency neighbors, storing and accumulating the module ID and corresponding outgoing flow as a $<$ key, value $>$ pair. Line 12 performs similar actions for the incoming flow from the modules to the current vertex. For simplicity, the flow coming from teleportation is omitted from the presented algorithm snippet. Lines $14-21$ iterate over the $<$ key, value $>$ pair, computing the difference in code length for a module (*newModId*). The difference in code length is computed by the function *calc* in line 18. If moving to a module reduces the code length by more than the reduction observed so far, the change in code length is recorded along with the *moduleId*. This detailed examination of the algorithm for *FindBestCommunity* reveals that most of the operations involve hash table insertions, lookups, and accumulations of flow values.

4.2.3 Pin and ZSim

To simulate our hardware accelerator ASA, we utilize ZSim [115], a Pin-based simulation infrastructure and tool. Pin is a program designed for instrumenting executables on Linux, Windows, and macOS for Intel (R) IA-32, Intel64, and Itanium (R) processors. Pin acts as a dynamic instrumentation tool, intercepting the application binary during execution and injecting instrumentation code snippets at specified locations. It allows the inspection of program context information, such as register states, storing and restoring this information when necessary to ensure the original execution flow remains unaffected by the instrumentation. Pintools are commonly employed for hardware simulation, as they can capture a natively executed instruction stream and then replay it on a simulated architecture.

Moreover, we implemented modifications in ZSim to simulate the ASA. A software representation of the ASA architecture is directed through custom instrumentation of the *xchg* instruction, which is not commonly generated by x86 compilers. We introduce *xchg* instructions with different registers to distinguish between the insertion of key-value pairs into the CAM (content addressable memory) and loading data from the CAM. These operations are assigned latencies and utilize relevant ports within ZSim's out-of-order core model to accurately replicate the time spent executing ASA instructions. Finally, ZSim reads the register values to

update the CAM state, a crucial aspect, for instance, in determining whether an ASA insertion might lead to overflow.

4.3 Methodology

We introduced the *FindBestCommunity* kernel with software hash in Section 4.2. In this section, we outline the design changes for the *FindBestCommunity* kernel using ASA. The corresponding pseudocode is provided in Algorithm 5. Additionally, in Fig. 4.3, we illustrate the generalized block diagram of the ASA micro-architecture. The original work on ASA [151] extensively covered the ASA micro-architecture for SpGEMM computation. Here, we focus on the API calls relevant to our specific context.

Algorithm 5: FindBestCommunity_ASA

Data: A vertex/supernode v_i of graph $G(V, E)$
Result: New community m_{new} for vertex v_i

```
1  std::vector<pair<key, value≫ nonoverflowed_pairs
2  std::vector<pair<key, value≫ overflowed_pairs
3  tid ← omp_get_thread_num()
4  numModlinks ← 0
5  for (linkIt ← v_i.outLinks.begin()  to  v_i.outLinks.end() ) do
6  |   k ← node.at(linkIt → first).modId
7  |   accumulate(tid, hash(k), k, linkIt → second)
8  gather_CAM(tid, nonoverflowed_pairs, overflowed_pairs)
9  if (!overflowed_pairs.empty()) then
10 |   sort_and_merge(nonoverflowed_pairs, overflowed_pairs)
11 Accumulate incoming flow (as in Ln. 5 − 10)
12 Iterate over the merged vector and record the module (bestModuletoMove) that minimizes code
     length most
13 return bestModuletoMove
```

4.3.1 Hash Accumulation

The software hash accumulation in lines $4-11$ of Algorithm 4 is substituted with the ASA accumulation call in line 7 of Algorithm 5. As each thread possesses its own core-local CAM, the *accumulate* API call takes four parameters: the thread ID (*tid*), the module ID (k), the hashed module ID (*hash(k)*) used to index the CAM entry, and the flow value ($linkIt \rightarrow second$) accumulated in the corresponding CAM entry. The *accumulate* call can yield three possible outcomes. If the key (k) is hashed to a unique index, a new entry is created in the cache. If the key already exists in the cache, the passed argument value is added to the partial sum. If the key is not found, and there is no space available in the cache, an entry is evicted based on an LRU policy and stored in a queue buffer.

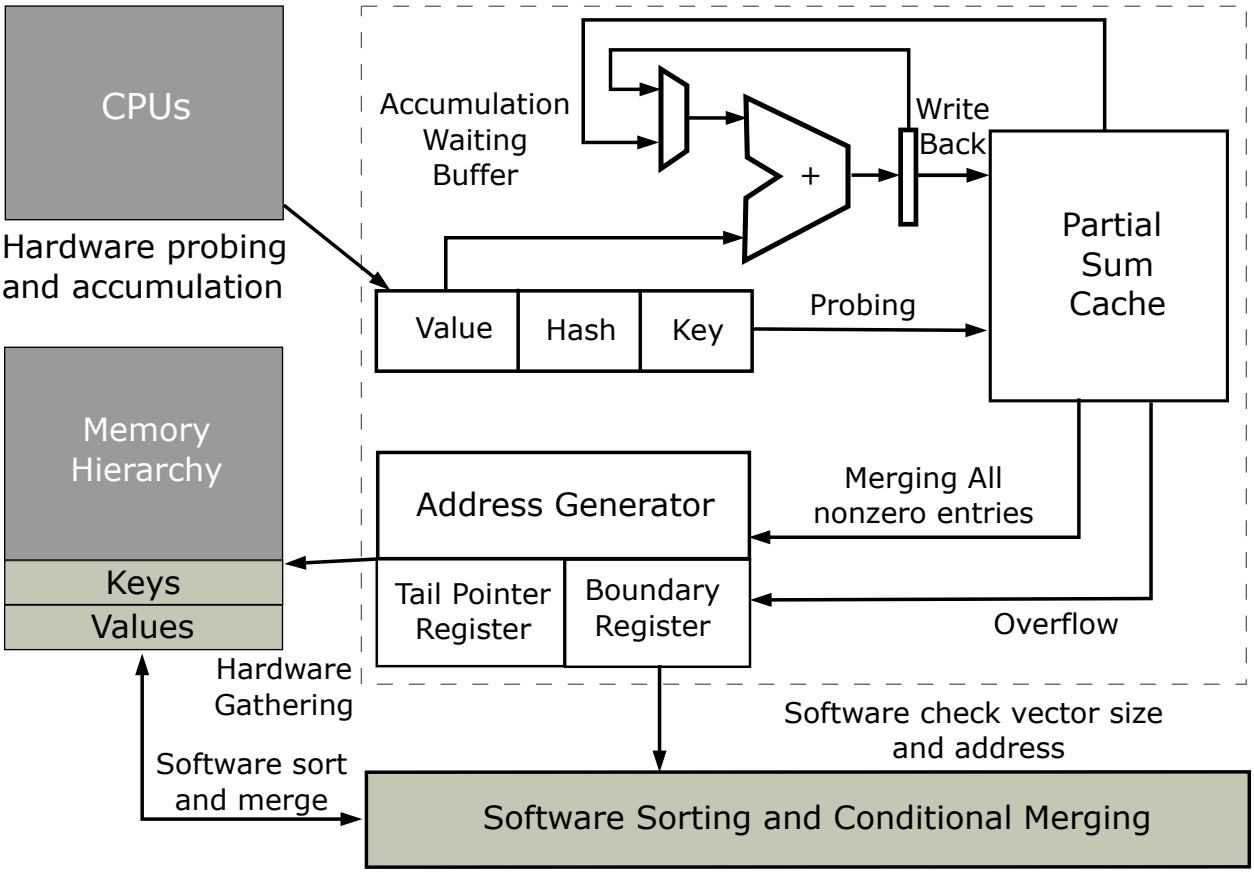


Figure 4.3: Generalized ASA micro-architecture block diagram. Different modules of the architecture and their functionalities are described in Chao et al. [151] and therefore skipped here for brevity.

4.3.2 Gather CAM Entries

The API call *gather_CAM* in line 8 of Algorithm 5 accepts the thread ID (*tid*) and references to the vectors for copying the CAM entries back to memory. The vector $nonoverflowed\_pairs$ receives the contents of the CAM, while the vector $overflowed\_pairs$ obtains the content of the overflowed queue buffer.

4.3.3 Sorting and Merging

In certain situations, an overflow may occur because of the restricted cache entries (CAM size). When the overflow buffer $overflowed\_pairs$ is not empty, the overflowed pairs are appended to the end of $nonoverflowed\_pairs$ and subsequently sorted based on their keys. Following that, values associated with identical keys are merged. This procedure is outlined in lines $9-10$ of Algorithm 5.

4.4 Evaluation

We opt for C++ in our implementation and utilize the g++ 7.5.0 compiler for the construction and integration of the ASA accelerator into HyPC-Map. The experiments and simulations are conducted on a Linux system featuring an Intel 2.6 GHz 64−bit processor, equipped with 16 physical cores distributed across 2 sockets, with 8 physical cores in each socket. To align the native configuration's CPU clock frequency with that of ZSim's simulated CPUs, the *scaling_governor* is configured to *performance*, ensuring a consistent (non-turbo mode) native CPU clock frequency. Our hardware architecture simulations are carried out using ZSim [115]. The datasets utilized in this paper are sourced from SNAP [84] and detailed in Table 4.1.

Table 4.1: Scale-free network datasets used for our experiments that exhibit power-law degree distribution. These are social and information networks with the largest one (Orkut) having 3M vertices and 117M edges.

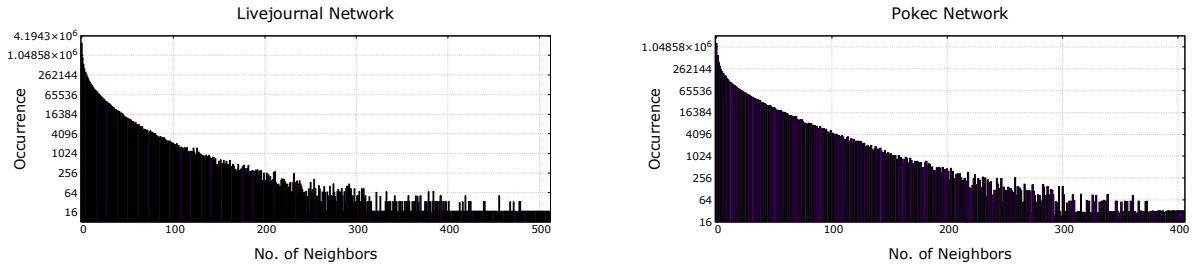
Network	**# Vertices**	**# Edges**
Amazon	334863	925872
DBLP	317080	1049866
YouTube	1134890	2987624
soc-Pokec	1632803	30622564
LiveJournal	3997962	34681189
Orkut	3072441	117185083

4.4.1 Utilizing Limited CAM Capacity

A balance exists between the on-chip memory cost of the ASA accelerator and the capacity to accommodate hash table entries. Fortunately, owing to the power-law degree distribution observed in scale-free networks, the majority of vertices have only a limited number of neighbors. A few vertices may possess more than thousands of neighbors, as illustrated in Figure 4.4 for the three large social networks. In Figure 4.5, we demonstrate that allocating 8KB of memory per core proves adequate to encompass the neighborhood lists of 99% of the vertices across all the social networks depicted in the plots. This insight can be leveraged for biological networks, as they exhibit analogous sparsity and degree distribution patterns.

4.4.2 Validation of Native vs Baseline

Before assessing the application performance of Infomap with hardware-accelerated hash compared to software hash, we validate the simulated performance of Infomap with the software hash implementation [49] (referred to as *Baseline*) using *ZSim* against its performance from native execution on the same machine (without *ZSim*). The accuracy of *ZSim* [115] has been confirmed on an Intel *Westmere* architecture with an average error of $\sim$ 10%. For our experiments, validation is conducted on an Intel *Ivy Bridge* architecture. The configurations for native hardware and *ZSim* simulation are detailed in Table 4.2 in columns 2 and 3, respectively. It's important to note that the $L3$ cache size of 20MB for the *Native* configuration (column 2)



(a) LiveJournal (b) Soc-pokec

Figure 4.4: Illustration of the degree distribution in scale-free social networks characterized by a power-law degree distribution. While a few vertices exhibit high neighbor counts, the majority of vertices (in this instance, hundreds of thousands of them) have either one or a few neighbors. This is evident in the *LiveJournal* network (Figure 4.4a) and the social *Pokec* network (Figure 4.4b).

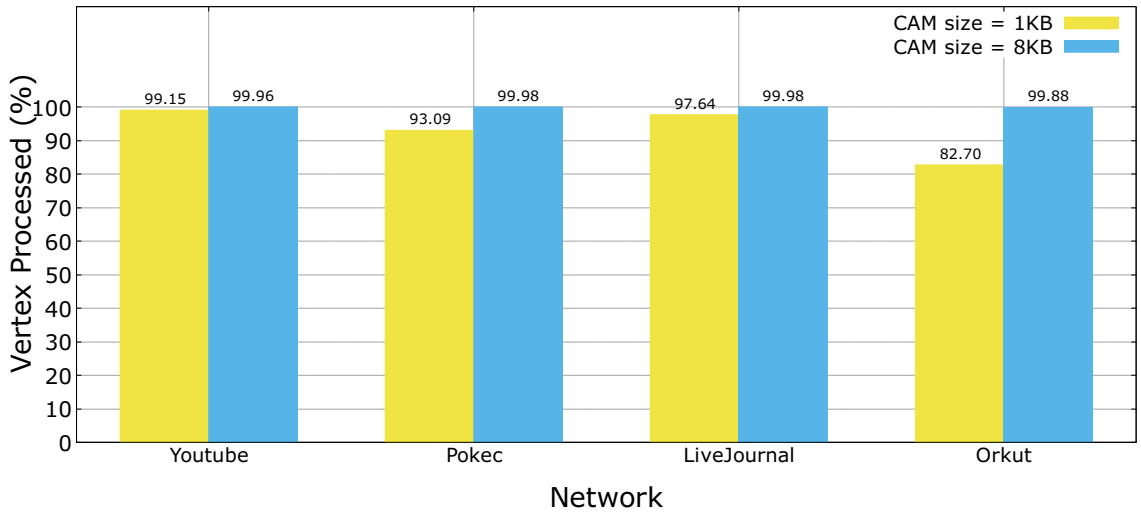


Figure 4.5: Harnessing the power-law degree distribution inherent in real-world networks to minimize Content-Addressable Memory (CAM) storage needs. The nature of power-law degree distribution enables even modest-sized CAMs to store the majority of graph neighbor lists. For instance, a core-local CAM with a capacity of 1KB can contain the neighbor lists of over 82% of the vertices of the Orkut network without reaching overflow. Scaling up to a capacity of 8KB covers more than 99% of the vertices in the social network datasets employed in this study.

cannot be replicated for *Baseline* (column 3) since *ZSim* requires power-of-2 cache sizes. The validation results for single-core execution in native versus *Baseline* are presented in Table 4.3. The *FindBestCommunity* kernel is executed for multiple iterations, and the runtime for each iteration is listed in each row of the table for both native and *Baseline* executions, along with the percentage difference in column 4. The average error is $\sim$ 12.7% between native and *Baseline*. Similarly, Table 4.4 presents execution times for native versus *Baseline* execution for 2 processing cores. The disparity in runtime between Native and *Baseline* could arise from differences in LLC ($L3$) cache sizes or simulation errors induced by *ZSim*.

Table 4.2: Machine configurations for Native vs Baseline validation. The only difference is in L3 cache size between Native and Baseline.

Item	**Native**	**Baseline**
Processor	8 cores, 2.6GHz	8 cores, 2.6GHz
L1 instruction cache	32KB	32KB
L1 data cache	32KB	32KB
L2	private 256KB	private 256KB
L3	shared 20MB	shared 16MB
Main Memory	$DDR3-1333$MHz, CL 1600MT/s	$DDR3-1333$MHz, $CL10$ 1600MT/s

Table 4.3: Runtime comparison in different iterations between Baseline and native using single processing core for the YouTube social network. The runtime difference (in %) is well within the limit as reported by ZSim [115].

Iteration no.	**Native (sec)**	**Baseline (sec)**	**(% diff)**
1	8.426	9.254	10
2	6.580	7.201	9
3	5.151	5.739	11
4	3.452	3.910	13
5	2.272	2.605	15
6	1.614	1.859	15
7	1.180	1.369	16

Table 4.4: Runtime comparison between baseline and native in different iterations using 2 processing cores for the YouTube social network. The runtime difference (in %) is within the limit as reported by ZSim [115].

Iteration no.	**Native (sec)**	**Baseline (sec)**	**(% diff)**
1	5.676	5.572	2
2	4.072	4.055	1
3	3.275	3.186	3
4	2.048	2.026	1
5	1.238	1.466	18

Table 4.5: Time spent on hash operations for Baseline vs ASA

Network	**Baseline (sec)**	**ASA (sec)**
Amazon	4.73	1.44
DBLP	7.35	1.86
YouTube	52.38	11.15
soc-Pokec	508.97	91.46
Orkut	1846.70	379.97

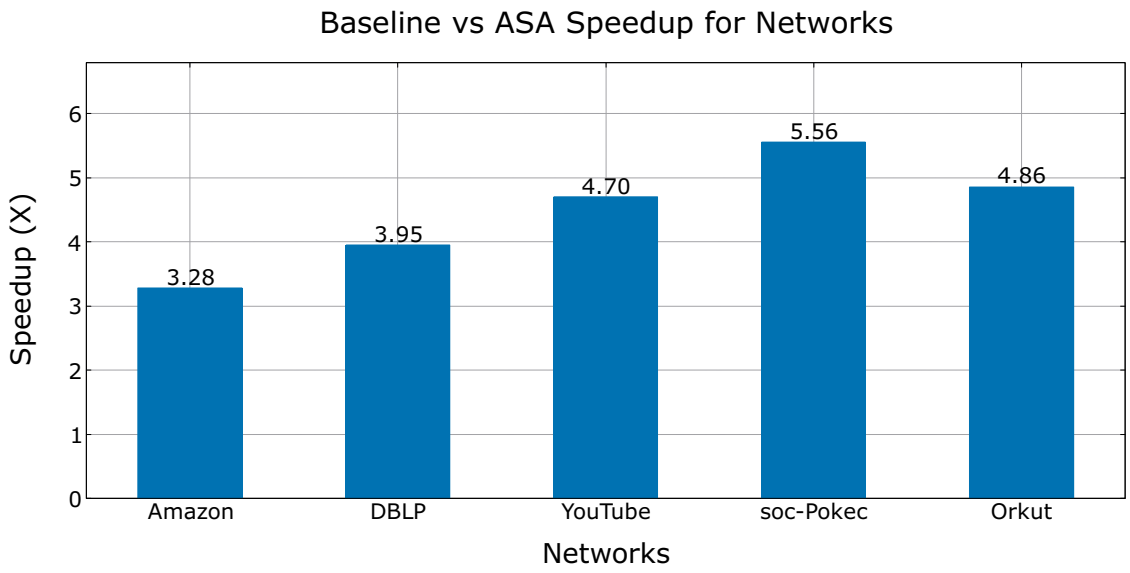


Figure 4.6: Comparison of speedup between Baseline and Accelerator for Hash Accumulation (ASA) across various networks. ASA significantly reduces the time required for hash operations compared to the Baseline software hash. In single-core experiments, the speedups are 3.28× for the *Amazon* network, 3.95× for the *DBLP* network, 4.70× for the *YouTube* network, 4.86× for the *Orkut* network, and 5.56× for the *soc-Pokec* network.

4.4.3 Performance Evaluation

The time spent on *HashOperations* is documented in Table 4.5, where *Baseline* is in column 2, and *ASA* is in column 3. Additionally, in Fig. 4.6, we depict the speedup achieved by *ASA* over *Baseline* for hash operations across various networks in single-core executions. The most substantial single-core performance gain, 5.56×, is observed for the *soc-Pokec* network. Similarly, *ASA* demonstrates gains of 3.28×, 3.95×, 4.7×, and 4.86× over *Baseline* for the *Amazon*, *DBLP*, *YouTube*, and *Orkut* networks, respectively. Figures 4.7 and 4.8 illustrate the performance breakdown of computational kernels between *Baseline* and *ASA* for multi-core executions. We observe a $(68-70)\%$ reduction in *HashOperations* computation time from *Baseline* to *ASA* for multi-core execution in the *Amazon* network (Figure 4.7). Similarly, we observe a $(75-77)\%$ reduction in *HashOperations* time for the *DBLP* network (Figure 4.8).

The performance enhancement observed from *Baseline* to *ASA* can be attributed to two main factors. Firstly, ASA reduces the average number of instructions compared to the software hash implementation. Software hash tables involve collision chaining or linear probing logic to address collisions, which necessitates the execution of additional instructions. ASA's extension to the ISA provides a single CPU instruction for hash lookup and accumulation. In Fig. 4.9, we observe a reduction of up to 24% in the total number of instructions for the *FindBestCommunity* kernel for some larger networks. Figure 4.10 illustrates a 12% reduction for the *Amazon* network, and Figure 4.11 shows a 15% reduction for the *DBLP* network in the average number of

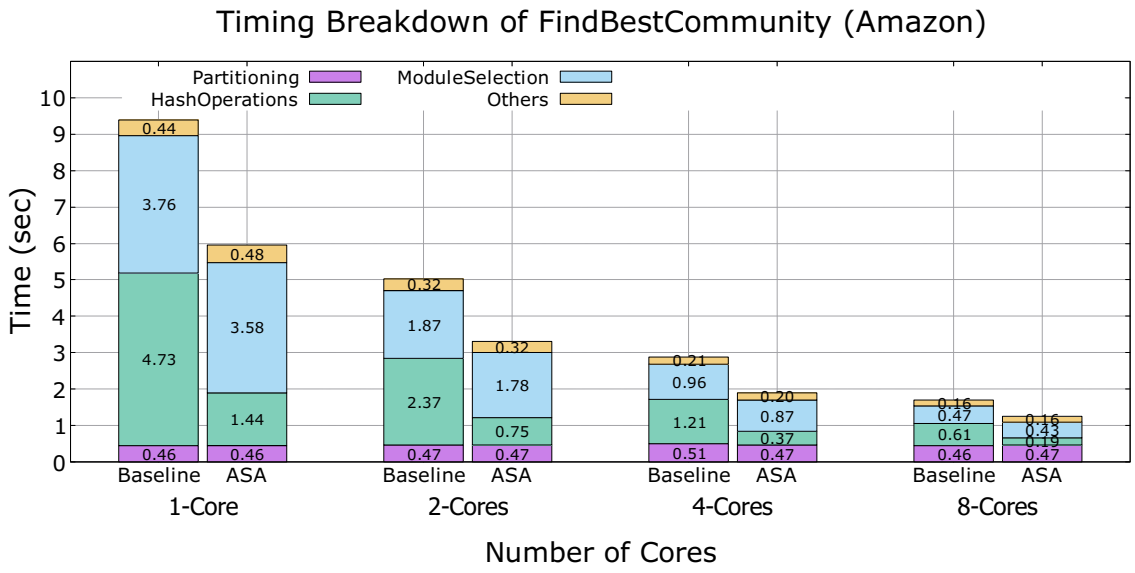


Figure 4.7: Breakdown of the execution time for the simulated kernel (*FindBestCommunity*) in the *Amazon* network. The timing breakdown illustrates the reduction in execution time for *HashOperations* across varying numbers of processing cores. In the single-core setting, the *HashOperations* time decreases from 4.73 seconds to 1.44 seconds. In 2-core setting, the *HashOperations* time decreases from 2.37 seconds to 0.75 seconds.

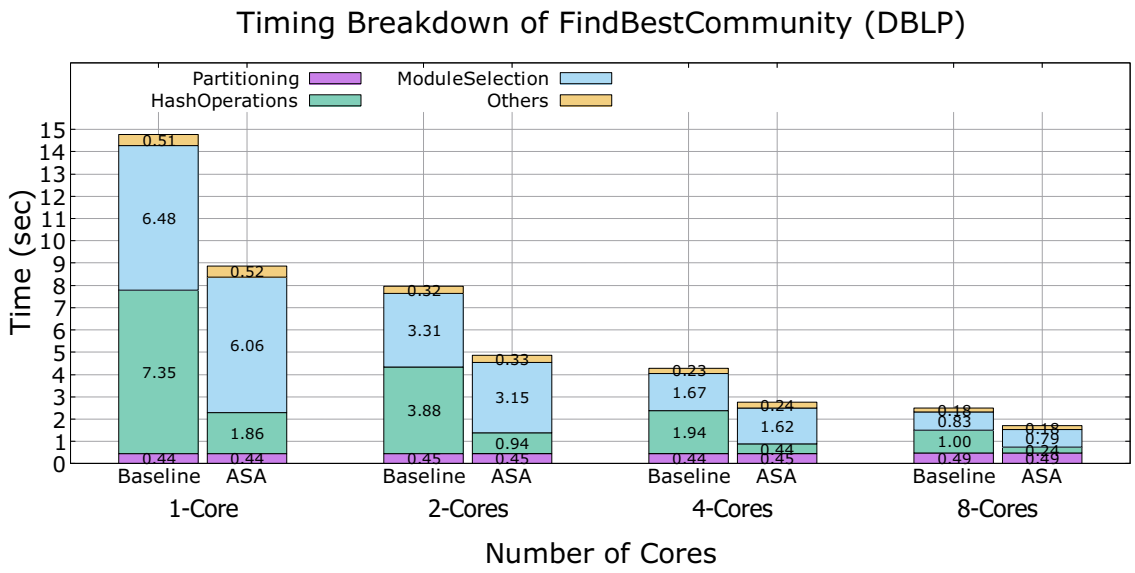


Figure 4.8: Breakdown of the execution time for the simulated kernel (*FindBestCommunity*) in the *DBLP* network. The timing breakdown illustrates the reduction in execution time for *HashOperations* across varying numbers of processing cores. In the single-core setting, the *HashOperations* time decreases from 7.35 seconds to 1.86 seconds. In 2-core setting, the *HashOperations* time decreases from 3.88 seconds to 0.94 seconds.

instructions per core from *Baseline* to *ASA* during multi-core executions for the *FindBestCommunity* kernel. The aforementioned statistics for the reduced number of instructions in ASA include the instructions for

handling overflow (lines 9 – 10 of Algorithm 5). For the *soc-Pokec* network, it constitutes only 9.86% of the ASA computation time (Table 4.5, column 3), and for the *Orkut* network, it represents only 13.31% of the ASA computation time to handle overflow.

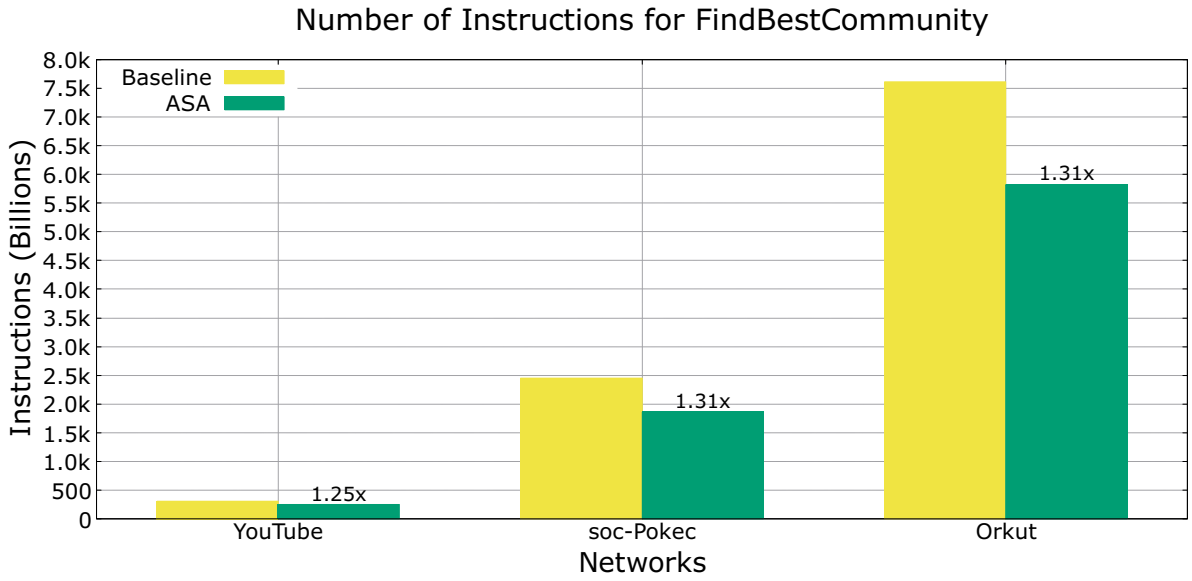


Figure 4.9: Performance comparison metric (total instructions) for large networks (Orkut, soc-Pokec, and YouTube). The total number of instructions shows a reduction from the *Baseline* to *ASA*. In the case of the *soc-Pokec* network, the total number of instructions decreases to 1.8 trillion with *ASA* from the original of 2.4 trillion in the *Baseline*.

Additionally, ASA's performance improvement is largely attributed to a significant reduction in the number of branch mispredictions in the software hash table. Branch mispredictions can incur substantial costs as the CPU core needs to flush all partially executed instructions from the incorrect branch from its pipeline and restart execution on the correct branch. Fig. 4.12 illustrates a reduction of up to 59% in the number of mispredicted branches for larger networks. Fig. 4.13 showcases a 40% reduction for the *Amazon* network, and Fig. 4.14 reveals a 46% reduction for the *DBLP* network in the average number of branch mispredictions per core for experiments conducted with different numbers of processing cores.

Moreover, addressing collisions in a software hash table often leads to irregular memory access patterns that are challenging for hardware prefetchers to predict, such as following pointers connecting entries that hash to the same bucket. This situation can potentially cause memory latency stalls. Reducing the number of branch mispredictions and irregular memory accesses resulting from hash collisions leads to a lower CPI for ASA compared to the Baseline. Fig. 4.15 illustrates a $(18 - 21)\%$ reduction in CPI for some larger networks (*YouTube, soc-Pokec, and Orkut*) in single-core execution. Similarly, in multi-core execution, Fig. 4.16 shows

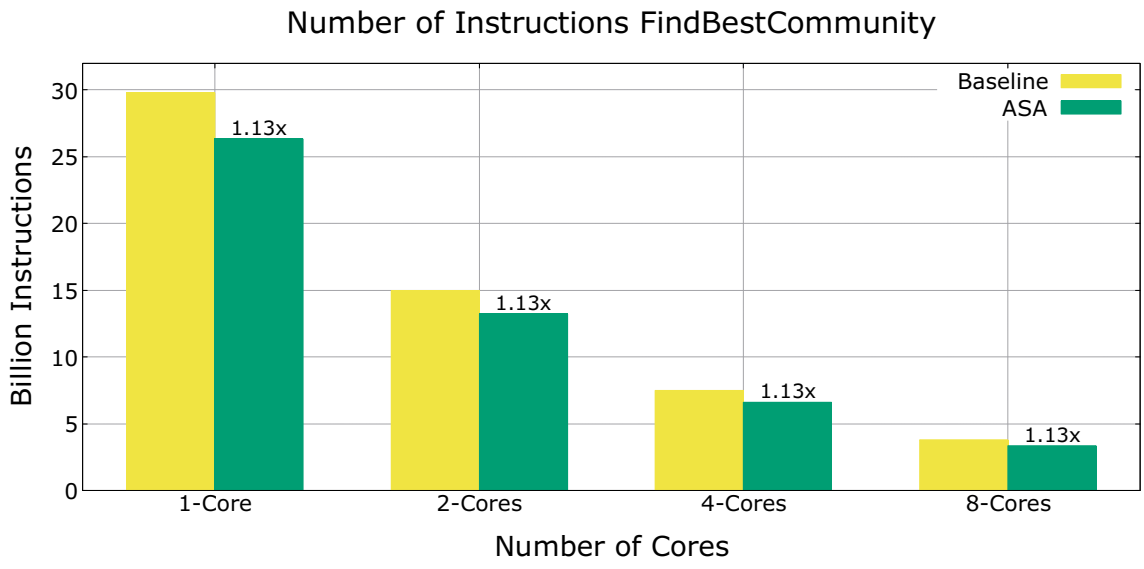


Figure 4.10: The average number of instructions per core decreased from *Baseline* to *ASA* for the *Amazon* network. The reduction factor is 1.13× from *Baseline* to *ASA* and remains consistent across multi-core executions.

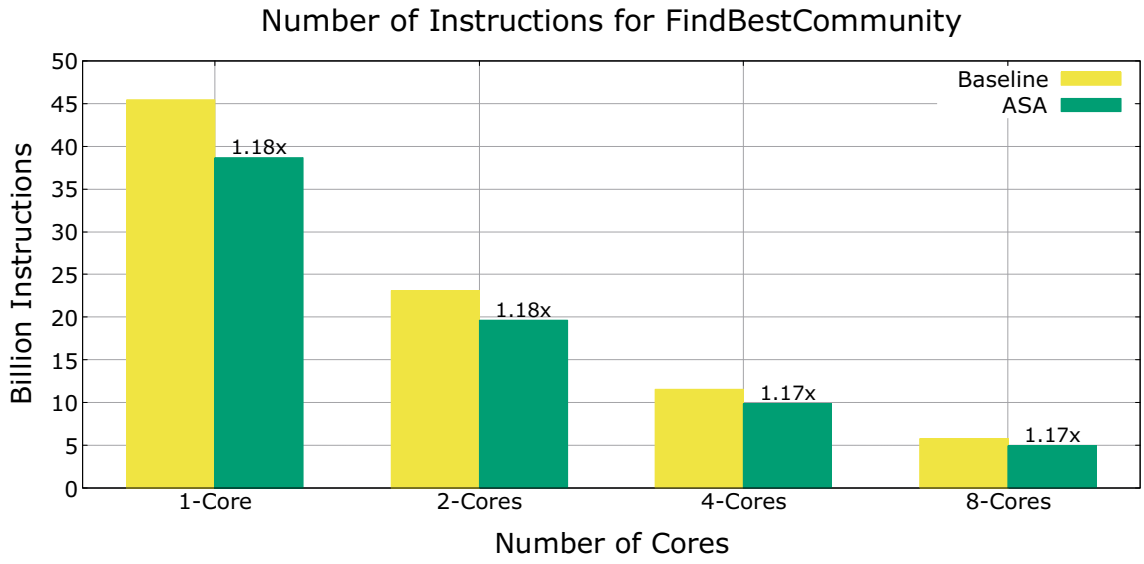


Figure 4.11: The average number of instructions per core decreased from *Baseline* to *ASA* for the *DBLP* network. The reduction factor is 1.17×, and this improvement remains consistent across multi-core executions.

a 20% reduction in CPI rate for the *Amazon* network, and Fig. 4.17 demonstrates a 21% reduction in CPI rate for the *DBLP* network, on average per core from Baseline to ASA.

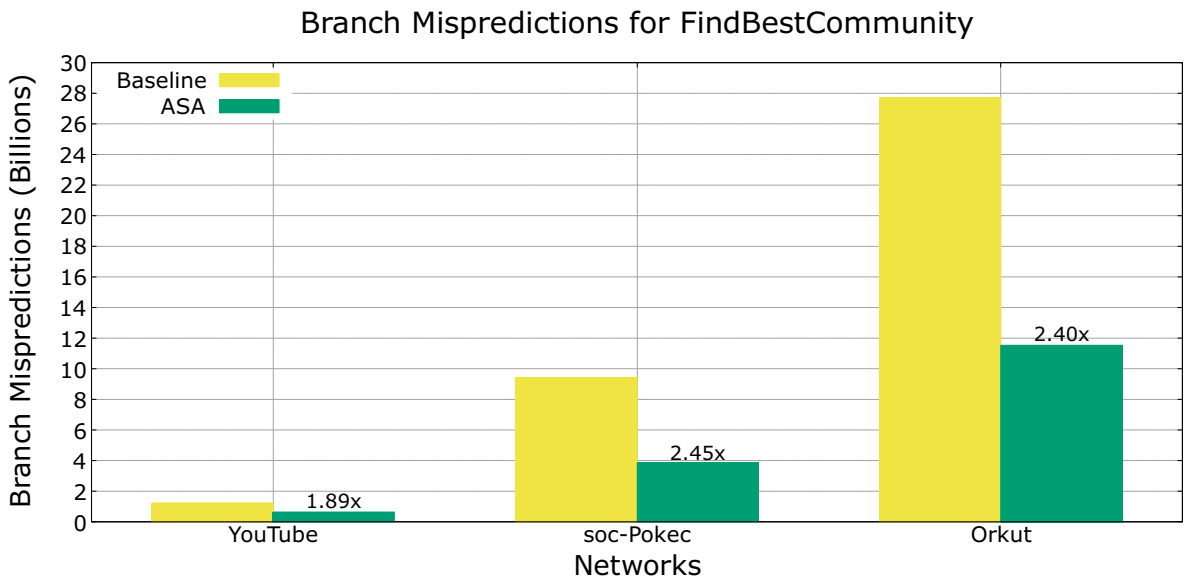


Figure 4.12: The decrease in the number of mispredicted branches from *Baseline* to *ASA* for large networks (Orkut, soc-Pokec, and YouTube). In the case of the *Orkut* network, it decreases to 11.55 billion in *ASA* from the initial 27.69 billion in *Baseline*.

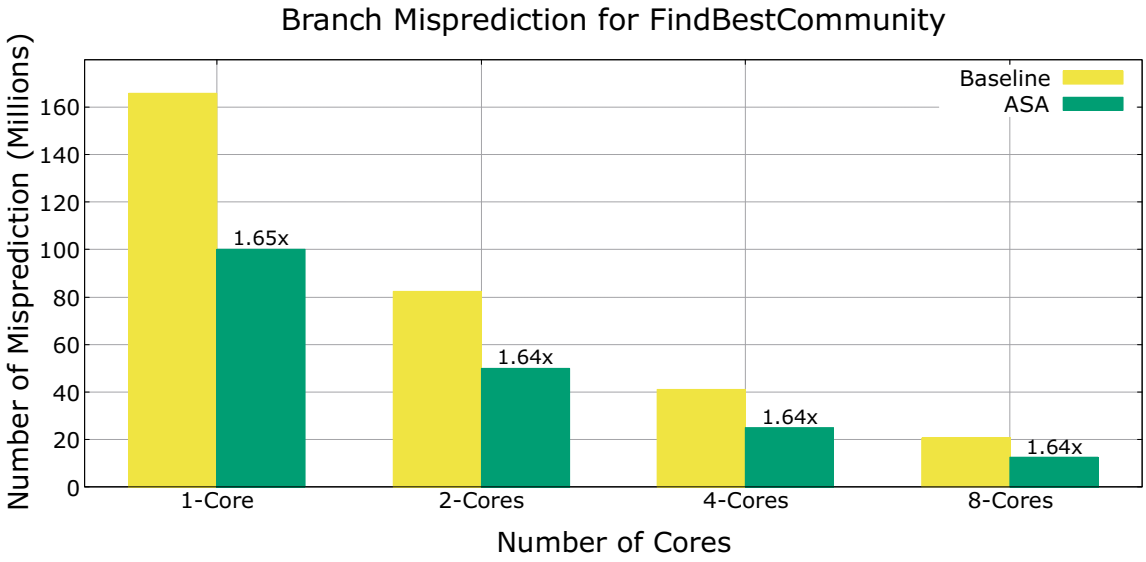


Figure 4.13: The average reduction in the number of branch mispredictions per core from *Baseline* to *ASA* for the *Amazon* network. The reduction factor is 1.64× and remains relatively consistent across multi-core executions.

4.5 Related Work

Leveraging a hardware accelerator for efficient graph mining applications stands as a crucial aspect of software-hardware co-design for graph algorithms. In a broader sense, the term graph data mining

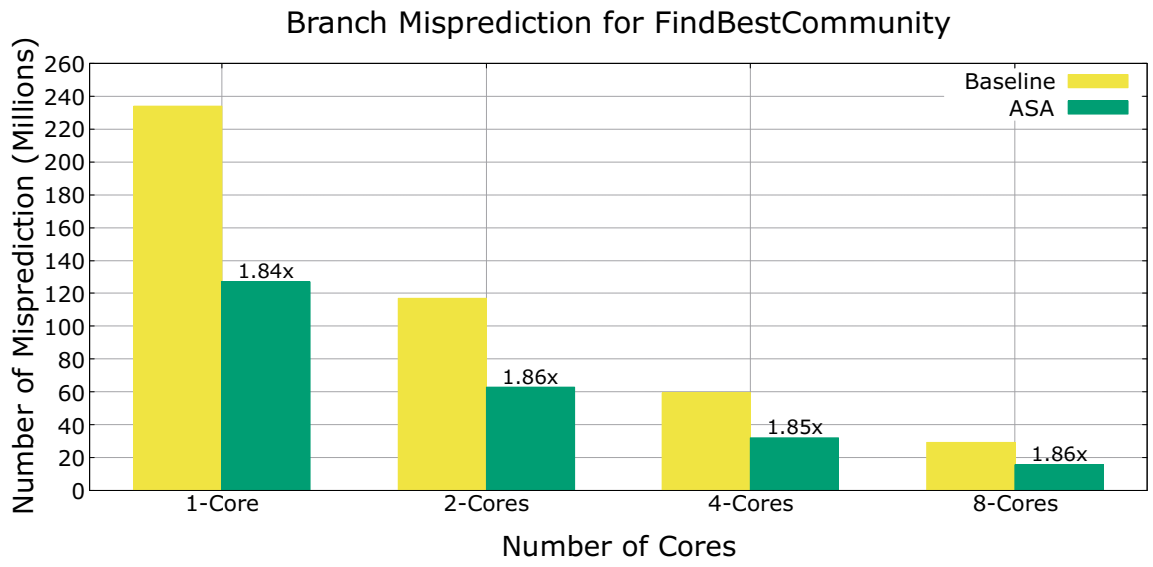


Figure 4.14: The average reduction in the number of branch mispredictions per core from *Baseline* to *ASA* for the *DBLP* network. The reduction factor is 1.86× and remains relatively consistent across multi-core executions.

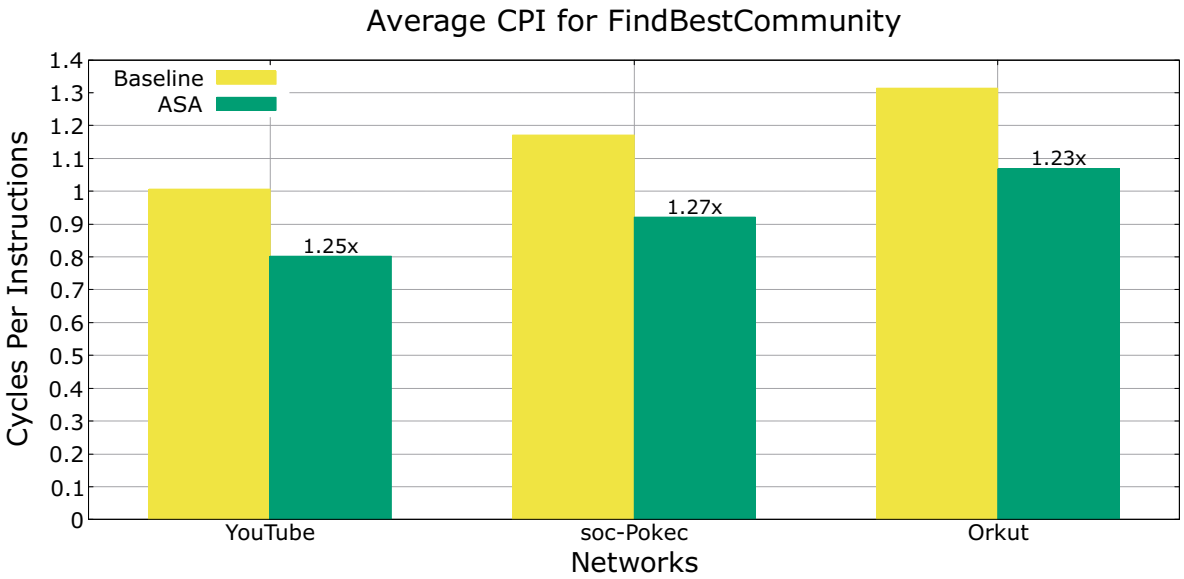


Figure 4.15: Reduction in the average cycles retired per instruction (CPI) from *Baseline* to *ASA* for the large networks (Orkut, soc-Pokec, and YouTube). In the case of the *Orkut* network, CPI decreases to 1.08 in *ASA* from the initial 1.32 billion in *Baseline*

encompasses strategies aimed at discovering structural information, such as communities, cliques, motifs, k-trusses, and other patterns within graphs. The process of graph data mining on real-world datasets encounters challenges like the under-utilization of computing resources due to the random access patterns inherent in the irregular graph structure and the significant load imbalance caused by the power-law degree

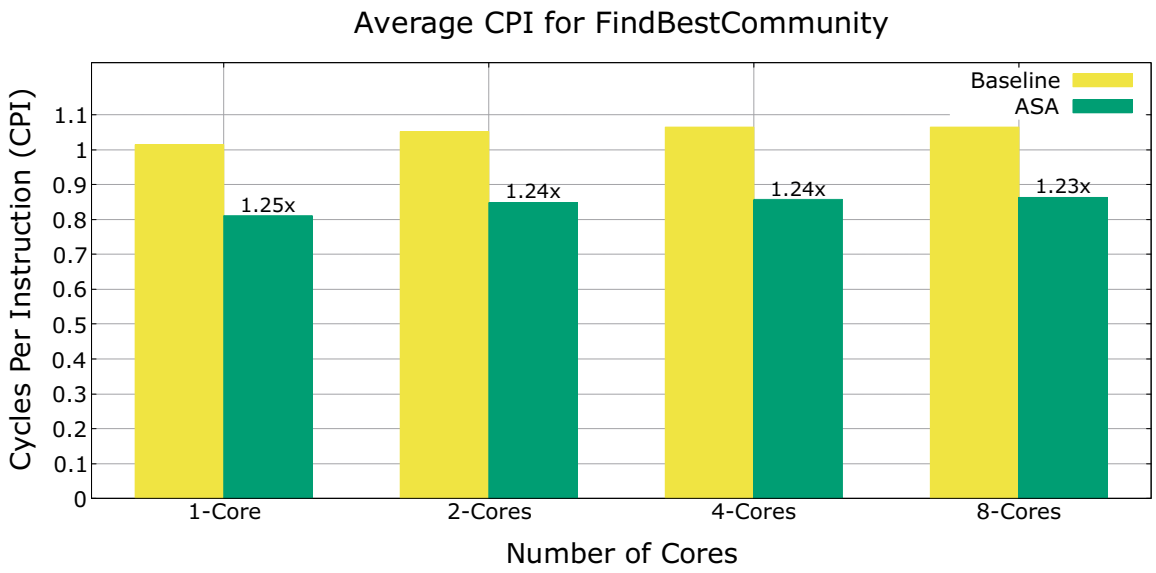


Figure 4.16: The average CPI (Cycles Retired per Instruction) per core decreases from Baseline to ASA for the *Amazon* network. The reduction factor is 1.24× and remains relatively consistent across multi-core executions.

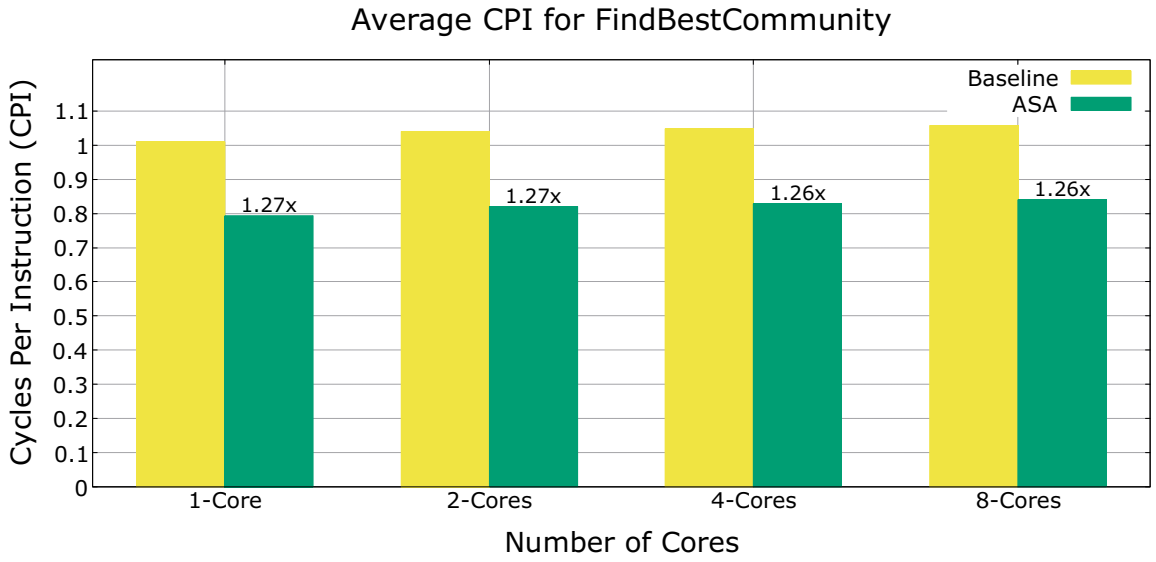


Figure 4.17: The average CPI (Cycles Retired per Instruction) per core decreases from Baseline to ASA for the *DBLP* network. The reduction factor is 1.27× and remains relatively consistent across multi-core executions.

distribution. Additionally, the set of potential patterns can grow exponentially with the size of the input graph. All these factors underscore the need for a hardware accelerator in graph mining. While there exist mature computational kernels and accelerators for various machine learning algorithms, the field of graph

data mining has yet to reach a similar level of maturity.

One accelerator for graph mining, Gramer [146], is grounded in the observation that a small fraction of vertex and edge data generates the majority of random memory requests. Gramer implements an intelligent cache hierarchy where the most frequently accessed data/pattern resides in the top level with a no-eviction policy, and the second level is on-chip memory with a lightweight replacement policy. The challenge in this approach lies in the identification and management of *valuable* data. Flexminer [32], another graph mining accelerator equipped with its compiler program, operates on dedicated on-chip storage and memoizes reusable connectivity information on a connectivity map (c-map), yet it does not address the rapid accumulation of values against keys. Another work, IntersectX [109], devises a stream-based ISA extension, following the observation that many graph pattern mining applications rely on the intersection of two edge lists as a core computation. Yet another accelerator named SISA [24] identifies a similar set of operations (union, intersection) and designs a set-centric ISA extension to process with on-chip memory. Rao et al. [110] propose an accelerator, *SparseCore*, for sparse tensor and graph pattern computation for streaming data or sparse vectors. Adam et al. [12] propose a hardware accelerator with dedicated hardware units to manage the irregular data movements of graph computation in Graph Neural Network (GNN), which can be applied to solve community detection problems. Notably, none of the literature [146, 32, 109, 110, 24, 12] addresses the challenge of accelerating key-value matching and accumulation for vertex-neighborhood connectivity, a concern tackled in our work.

Yang et al. [145] developed a hash accelerator using FPGA on-chip SRAM, but its scalability is restricted to 16 processing engines (PE). In contrast, Zhang et al. [152] devised a hash accelerator through ISA extensions and hardware modifications, managing most operations within the accelerator and only resorting to software for rare cases. The study [152] introduced two implementations of the hash accelerator, namely, Flat-HTA and Hierarchical HTA. However, Chao et al. [151] demonstrated that both versions could be surpassed by ASA, as evidenced in their comparison involving the SpGEMM computation.

4.6 Concluding Remarks

Community discovery stands as a widely adopted application for uncovering prominent motifs in social and relational networks. The integration of hardware accelerators to expedite specific aspects of the software kernel profoundly influences hardware-software co-design. To our knowledge, our work marks the pioneer effort to introduce a hardware accelerator for sparse accumulation in the context of information-theoretic community discovery. We assert in this investigation that prevailing implementations of *Infomap* employing a conventional software hash kernel encounter limitations with general-purpose CPUs. Our study reveals that, in addition to algorithmic optimizations, there exists an untapped potential to achieve superior performance throughput by employing an accelerator capable of executing specific computations more efficiently than conventional general-purpose hardware. Existing accelerators tailored for graph pattern mining fall short in

addressing the acceleration of key-value lookup and accumulation. Through this research, we demonstrate that the ASA accelerator not only accelerates the SpGEMM kernel but also enhances the performance of other applications heavily reliant on hash lookup and accumulation. This improvement is achieved by mitigating the number of branch mispredictions, average CPI, and the total number of instructions.

Chapter 5: Fast Parallel Index Construction for k-truss-based Local Community Detection

Identifying cohesive subgraphs stands as a fundamental graph analysis kernel widely employed in the study of social and biological networks. Various approaches, including clique identification, community discovery, and truss decomposition, aim to unveil insightful substructures within a network. However, the computational intractability of finding cliques poses challenges in identifying cohesive subgraphs within large networks. One viable solution is k-truss decomposition, a more relaxed alternative to finding cliques that can be solved in polynomial time. Unlike global community detection, which involves breaking down the entire graph into disjoint communities, local or goal-oriented community search focuses on identifying the community of interest for a specific entity. In this study, we propose a parallel k-truss-induced community discovery technique capable of detecting local communities in polynomial time. Previous studies primarily explored k-truss-induced local community formation in a serial setting, rendering them unsuitable for large graphs. In this paper, we devise a parallel k-truss-induced local community construction method utilizing multi-core parallelism. To the best of our knowledge, this marks the inaugural attempt to parallelize this algorithmic approach, coupled with a thorough performance analysis. Our experiments reveal a notable performance enhancement, with speedups ranging from 19× to 55× for graphs featuring hundreds of millions to billions of edges, leveraging NERSC Perlmutter compute nodes.

5.1 Introduction

Community discovery stands as a widely adopted application for the categorization or clustering of entities with shared attributes [26, 114, 98, 46, 49]. This application finds applications in various contexts, such as identifying groups of individuals with similar interests in social networks, marketing products to consumer groups based on their categories, clustering proteins with similar characteristics, and elucidating the functionality of unknown proteins, as well as cyber-security tasks like web spam detection. In many real-world scenarios, the emphasis is on ascertaining the communities to which an entity (represented as a vertex in a graph) belongs, rather than uncovering the disjoint communities of the entire graph [2, 70]. For example, a user on a social network may be more interested in the social groups or communities they engage with, rather than all communities within the network. This user-centric, personalized search is more

meaningful as the communities a user participates in provide insights into their social or behavioral context. While the disjoint community problem typically employs a global criterion [50, 53] or optimization function to unveil all eligible communities, the overlapping community problem involves constructing and maintaining an index-based structure to retrieve community subgraphs containing the query vertex [2]. We refer to the latter as a local or goal-oriented community search. A fundamental distinction between these problems is that in global community discovery, a vertex is associated with only one community at a time (disjoint), whereas in local community discovery, a vertex may simultaneously belong to multiple communities (overlapping). In Chapter 1, we provided an illustration (Figure 1.2) depicting the difference between disjoint and overlapping community membership.

Several models for goal-oriented local community discovery have been proposed based on graph motifs, such as *k-core* [17, 117, 127], *clique*/*quasi clique* [36, 132], and *k-truss* [69, 137, 2]. A k-truss-oriented index construction for local community search offers advantages over other methodologies. Notably, the commonly used cohesive subgraph, *clique* [89], suffers from being excessively restrictive (encompassing every vertex within a 1-distance) and exhibiting extremes of either too common (small clique) or too rare (large clique) occurrences in real-world scenarios. Moreover, solving the *clique* problem is not polynomially tractable [28]. The *k-core* problem, while polynomially solvable, has the drawback of lacking cohesion, an essential property of community subgraphs [35]. On the other hand, *k-truss*, a relaxed version of *clique*, can be computed in polynomial time. Utilizing a higher-order graph motif of triangle connectivity as the fundamental unit for defining a community, *k-truss* moves beyond primitive features like vertex sets or edge sets, enabling a comprehensive model of multiple overlapping communities.

Recent investigations have explored goal-oriented community search based on *k-truss* [70, 2, 62]. A key limitation of existing studies on *k-truss*-oriented index construction or community search lies in the sequential nature of their algorithms. One essential sub-problem within this formulation of local community search is *k-truss* decomposition, a well-explored challenge for parallel algorithm design. Numerous works have addressed parallel *k-truss* decomposition in both shared-memory settings [126, 72, 140] and distributed-memory settings [102, 31, 42]. GPU-based studies [138, 39, 4] for *k-truss* decomposition also exist. Akbas et al. [2] proposed *EquiTruss*, a *k-truss*-based index structure outperforming *TCP-Index* by Huang et al. [70] in the context of building an index for local community search. However, both studies are sequential and exhibit limited scalability. In response, we introduce a shared-memory parallel algorithm designed for a multicore setting using the *EquiTruss* formulation, specifically tailored for large graphs. We observe that *EquiTruss* can be computationally expensive for larger graphs, as evident in Figure 5.1. While parallel algorithms exist for *k-truss* decomposition, none have addressed parallel *EquiTruss*. Hence, our focus in this study is on designing a scalable algorithm for the *EquiTruss* problem.

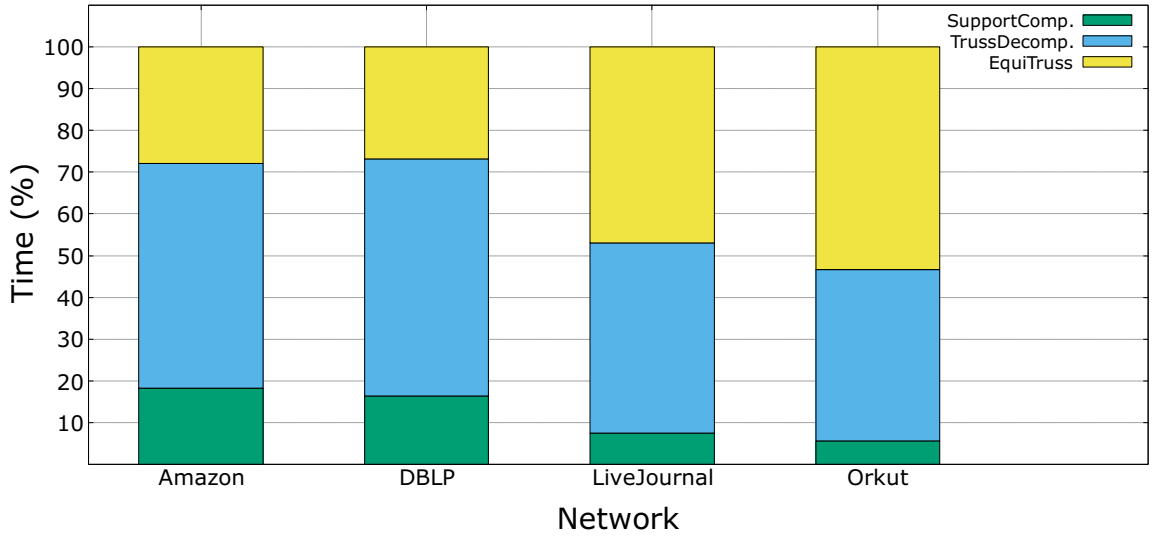


Figure 5.1: Percentage breakdown of compute kernel timing for our initial implementation based on *EquiTruss*. The computational cost of computing *EquiTruss* (depicted in yellow) is comparable to that of *k-truss* decomposition (in blue) for large graphs.

We parallelize the construction of *EquiTruss* using a connected component (CC) algorithm [123], denoted as *Baseline EquiTruss*. Subsequently, we enhance execution time by introducing cache-optimized storage and extraction of neighborhood information, referred to as *C-Optimal EquiTruss*. Finally, we leverage the state-of-the-art sampling-based parallel connected component algorithm *Afforest* [130] to construct supernodes in *EquiTruss*, denoted as *Afforest EquiTruss*, which surpasses the performance of the earlier two versions. Our contributions are summarized as follows:

- We uniquely recognize the construction of the k-triangle-induced index (*EquiTruss*) as a connected component problem on a graph, where edges are considered entities instead of vertices. The interconnection between edges is established based on k-triangle connectivity.
- We develop a parallel approach for constructing the supergraph (index) of *EquiTruss* using OpenMP, ensuring no loss of accuracy. As far as we know, our innovative algorithm stands as the initial parallel algorithm designed for constructing such index structures to support local community searches.
- In our parallel implementations of *EquiTruss*, we employ the state-of-the-art connected components approach, Afforest [130], along with the previously leading connected component approach, Shiloach-Vishkin (SV) [123]. We provide a comparative analysis of their performance.
- We generate the supergraph using a combination of parallel supernode and parallel superedge formulations, achieving a speedup of up to 30× on the *NERSC Perlmutter* compute node compared to the sequential counterpart and up to 55× compared to the *Baseline EquiTruss*.

5.2 Background

Our parallel algorithm design follows the sequential *EquiTruss* approach [2]. The notations used in this paper are outlined in Table 5.1. Additionally, Table 5.2 provides an overview of our various *EquiTruss* implementations. Subsequently, we introduce a few pertinent definitions and discuss the *EquiTruss* index construction strategy.

Table 5.1: Notations and abbreviations to describe the work of our parallel *EquiTruss*

Notation	**Description**
G (V, E)	A simple undirected graph, G
$\mathbb{G}(\mathbb{V}, \mathbb{E})$	A summary/supergraph $\mathbb{G}$
$\mathbb{V}$	Set of supernode(s)
$\mathbb{E}$	An edge list of superedge(s)
τ	A dictionary storing trussness for edge $e \in E$
Π	A dict. for parent component ID of $e \in E$
$support(e)$, $\lvert\Delta\rvert_e$	No. triangles having e as their constituent edge
k	Trussness of an edge
Φ_k	An edge set of same trussness k
ν	A supernode satisfying k-triangle connectivity
$\Delta_s \leftrightarrow \Delta_t$	Δ_s and Δ_t are triangle connected
$e \leftrightarrow e'$	Edges e and e' are triangle connected
$e \overset{k}{\leftrightarrow} e'$	Edges e and e' are k-triangle connected
CC	Connected Component
SV	Shiloach-Vishkin algorithm
LP	Label Propagation algorithm

Table 5.2: Listing of different algorithmic implementations/optimizations and corresponding descriptions

Name	**Description**
Original *EquiTruss*	Our C++ implementation based on work [2]
Baseline *EquiTruss*	Our shared-memory-parallel *EquiTruss* based on Shiloach-Vishkin CC algorithm
C-Optimal *EquiTruss*	Our memory and computation optimized *EquiTruss* from its predecessor Baseline
Afforest *EquiTruss*	Our shared-memory-parallel *EquiTruss* based on Afforest [130] CC algorithm

5.2.1 Preliminaries

The task of constructing the *EquiTruss* index for an online community search assumes that the graph $G(V, E)$ is simple, undirected, and unweighted, where $|V|$ represents the number of vertices and $|E|$ represents the number of edges. The following definitions are pertinent to the context of *EquiTruss*.

Definition 3 (Triangle) *Given vertices u, v, w s.t.* (u, v), (v, w), *and* (u, w) *are edges in E, a triangle* Δ *is a set of the three edges forming a cycle, i.e.,* $\Delta = \{(u, v), (v, w)(u, w)\} \subseteq E$.

Definition 4 (Support) *The support of an edge, e, is the number of triangle(s) having e as their constituent edge. We denote the support of e as* $|\Delta|_e$ *or support(e).*

Definition 5 (k-truss [35, 71, 2]) *A k-truss is a subgraph such that each edge has a support of at least* $k - 2$ *within the subgraph. Formally, given a subgraph* $G'(V', E') \subseteq G$, G' *is a k-truss if* $|\Delta|_e \geq k - 2$ *for all* $e \in E'$. *A maximal k-truss is a k-truss that is not a proper subgraph of another k-truss: formally, there exists no subgraph* G'' *such that* G' *and* G'' *are k-trusses and* $G' \subsetneq G''$.

Definition 6 (Trussness [71, 2]) *Given an edge* $e \in E$, *the trussness of an edge,* $\tau(e)$ *is defined to be the largest k such that there exists a k-truss in G that contains e. The trussness of a graph* $\tau(G)$ *is defined as* $\min_{e \in E} \tau(e)$.

Definition 7 (Triangle Adjacency [71, 2]) *Two triangles* Δ_1 *and* Δ_2 *are adjacent if they share a common edge, i.e.,* $\Delta_1 \cap \Delta_2 \neq \emptyset$.

Definition 8 (Triangle Connectivity [71, 2]) *Given* 2 *triangles* Δ_s *and* Δ_t *within G, they are triangle connected, i.e.,* $\Delta_s \leftrightarrow \Delta_t$ *if there exists a sequence of triangles,* $\Delta_1, \ldots, \Delta_n$ *in G with* $n \geq 2$ *such that* $\Delta_1 = \Delta_s$, $\Delta_n = \Delta_t$, *and for* $1 \leq i < n$, $\Delta_i \cap \Delta_{i+1} \neq \emptyset$. *If* $e \in \Delta_s$ *and* $e' \in \Delta_t$, *then* e, e' *are triangle connected or* $e \leftrightarrow e'$. *If all edges in the path between* $e \leftrightarrow e'$ *have trussness of k, then* $e \overset{k}{\leftrightarrow} e'$.

Definition 9 (k-truss Community [71, 2]) *For an integer* $k \geq 3$, *a subgraph* $G' \subseteq G$ *is a k-truss community if* G' *is a k-truss and for all* e, $e' \in E'$, $e \overset{k}{\leftrightarrow} e'$.

The goal of the *EquiTruss* algorithm is to create a *summary graph* $\mathbb{G}(\mathbb{V}, \mathbb{E})$ that will enable the fast construction of the k-truss communities associated with a given vertex.

Definition 10 (Supernode) *A supernode* $\nu \in \mathbb{V}$ *is a set of edges in E such that*

1. *For all* e_1, $e_2 \in \nu$, $\tau(e_1) = \tau(e_2)$,
2. *For all* e_1, $e_2 \in \nu$, $e_1 \leftrightarrow e_2$ *in the maximal k-truss of G,*
3. *The supernode* ν *is maximal, i.e., there does not exist an edge* $e \in G \setminus \nu$ *such that* $\tau(e) = \tau(\nu)$ *and* $e \leftrightarrow \nu$.

Note that due to the maximality requirement of the supernodes, the set of supernodes $\mathbb{V}$ partitions E.

Definition 11 (Superedge) *Given supernodes ν_1, $\nu_2 \in \mathbb{V}$, we say there exists a superedge between them if $\nu_1 \leftrightarrow \nu_2$ in the κ-truss where $\kappa = \min(\tau(\nu_1), \tau(\nu_2))$ and $\tau(\nu_1) \neq \tau(\nu_2)$.*

5.2.2 Index Construction Method

In this section, we delve into the index construction phase of the *EquiTruss* approach, and the corresponding pseudocode is outlined in Algorithm 6. The resulting index serves as the primary data structure for retrieving all communities associated with a query entity (vertex). The algorithm takes a graph $G(V, E)$ as input and outputs a supergraph with supernodes connected by superedges. Additionally, a pre-computed dictionary τ is provided, containing edges along with their corresponding k-trussness, derived from a k-truss decomposition technique.

The initialization phase (lines $1-5$) establishes an initially empty list of supernode IDs, which will be utilized for superedge computation later in the algorithm. The entire edge set E is then partitioned into subsets based on their respective trussness values, denoted as k (lines $4-5$). An iterative traversal begins, ranging from $k_{min} \geq 3$ to k_{max} (line 7). For each edge set Φ_k with a specific trussness k, edges are retrieved (line 8), converted into a supernode with a sequentially assigned supernode ID, and added to the set of supernodes $\mathbb{V}$ (lines $9-11$).

For an edge within a supernode ν, a Breadth-First Traversal (BFS) is conducted to establish connections with other edges belonging to the same supernode, adhering to *k-triangle* connectivity (lines $13-24$). In other words, all edges forming a triangle connection with the current edge e, sharing the trussness k of e, are incorporated into the supernode ν containing e. This process is elucidated in lines $20-23$ and $26-29$ of Algorithm 6.

If an edge $e\prime$ establishes *k-triangle* connectivity with e and $\tau(e\prime) > k$, an entry is appended to the list of supernodes connected to e (lines $31-32$). When the list for $e\prime$'s supernode is processed, a superedge entry is generated to link the supernode containing $e\prime$ to the supernode containing e (lines $17-19$). A schematic representation of supernodes, superedges, and the summary graph structure is provided in Figure 5.2.

5.3 Methodology

5.3.1 Overview of the parallel algorithm

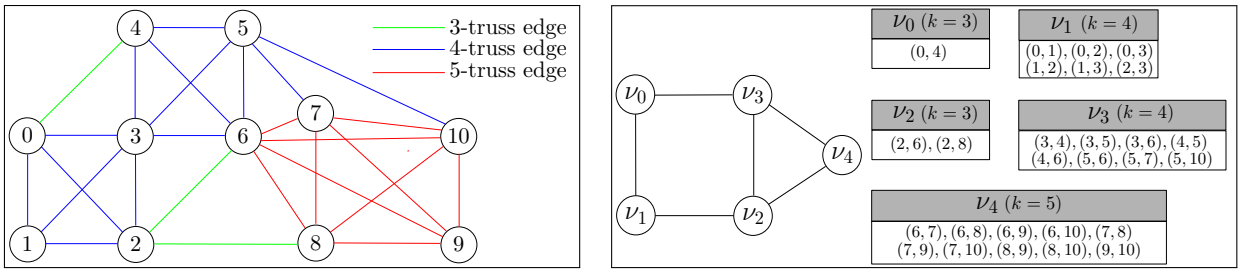
We decompose our parallel index construction method into three distinct algorithmic components. Algorithm 7 elaborates on the procedure for concurrently generating the set of supernodes. Subsequently, in Algorithm 8, we delve into our parallel algorithmic design for constructing the set of superedges denoted as $\mathbb{E}$. Lastly, in Algorithm 9, we expound on our parallel approach to crafting the supergraph $\mathbb{G}(\mathbb{V}, \mathbb{E})$.

Algorithm 6: Construct Index for *EquiTruss* [2]

Data: A graph, $G(V, E)$ and a dictionary of edges, τ, with their k-truss values
Result: A supergraph, *EquiTruss*: $\mathbb{G}(\mathbb{V}, \mathbb{E})$

for $e(u, v) \in E$ **do**
  $e.processed \leftarrow FALSE$
  $e.list \leftarrow \emptyset$
  if $(\tau(e) = k)$ **then**
    $\Phi_k \leftarrow \Phi_k \cup e$
$spNdID \leftarrow 0$
for $k = k_{min}$ *to* k_{max} **do**
  while $(\exists_e \in \Phi_k)$ **do**
    $e.processed \leftarrow TRUE$
    Create a supernode ν, where $\nu.spNdID \leftarrow spNdID++$
    $\mathbb{V} \leftarrow \mathbb{V} \cup \{\nu\}$
    Initialize an empty queue, Q
    $Q.enqueue(e)$
    while $(Q \neq \emptyset)$ **do**
      $e(u, v) \leftarrow Q.dequeue()$
      $\nu \leftarrow \nu \cup \{e\}$
      for $ID \in e.list$ **do**
        Create a superedge (ν, μ), where μ is an existing supernode with $\mu.spNdID = ID$
        $\mathbb{E} \leftarrow \mathbb{E} \cup \{(\nu, \mu)\}$
      for $w \in N(u) \cap N(v)$ **do**
        if $\tau(u, w) \geq k$ *and* $\tau(v, w) \geq k$ **then**
          ProcessEdge((u, w), spNdID, Q)
          ProcessEdge((v, w), spNdID, Q)
      $\Phi_k \leftarrow \Phi_k - \{e\}$
Procedure `ProcessEdge`$((u, v), spNdID, \&Q)$
  if $(\tau(u, v) = k)$ **then**
    if $(u, v).processed = FALSE$ **then**
      $(u, v).processed \leftarrow TRUE$
      $Q.enqueue((u, v))$
  else
    if $(spNdID \notin (u, v).list)$ **then**
      $(u, v).list \leftarrow (u, v).list \cup \{spNdID\}$

Supernode Creation: In Algorithm 7, we illustrate the process of creating supernodes using the Shiloach-Vishkin (SV) [123] approach for parallel connected components (CC). While there exist alternative methods for parallel CC, such as Label Propagation [143, 116] or *BFS*, we opt for SV [123] for executing our edge-induced CC to construct supernodes. SV offers linear work efficiency like LP but is independent of the graph diameter (D). Variants [21, 125] of parallel CC using *BFS* exhibit linear work efficiency, but the parallelism is constrained by the growing number of connected components. It is crucial to highlight that



(a) (b)

Figure 5.2: Visualization of the construction process of the summary graph by *EquiTruss*. The presented graph example is sourced from [2]. In Figure 5.2a, edges belonging to different *k-trussness* values are color-coded, with the red edges representing members of the 5-truss subgraph but not the 4-truss (blue) or 3-truss (green) subgraphs. Following *Definition 8*, supernodes ν_0, ν_1, ν_2, ν_3, and ν_4 are formed, as depicted in Figure 5.2b. Superedges are subsequently established between these supernodes according to *Definition 9*.

our edge-based CC, incorporating *k-triangle* connectivity, aligns seamlessly with the SV formulation for establishing supernodes. Additionally, we explore a cache-optimized variant of SV and the state-of-the-art CC approach, Afforest [130], to showcase improved runtime in both sequential and parallel executions.

Algorithm 7 takes the original graph $G(V, E)$ and a dictionary of edges with their k-trussness, τ, as input. The algorithm begins by initializing each edge to its own parent component (ln. $1-2$) and grouping edges based on their trussness (ln. $3-5$). Similar to Algorithm 6, all different subsets of edges based on their trussness are iteratively processed from $k_{min} \geq 3$ to k_{max} (ln. 6). All edges under certain truss groups are processed in parallel (ln. 10) to identify the other edges forming triangles with edge $e(u, v)$ (ln. 11). We opted for the SV approach because it is highly amenable to parallelism and theoretically works well independent of graph topology [130]. SV consists of two alternating phases, *hooking* and *shortcut*. The hooking phase (ln. $12-20$) connects the edge e_1 (ln. 16) to the same parent component as e if the condition for k-triangle connectivity (ln. 15) is satisfied. A similar action is performed for edge e_2 (ln. $18-19$). In either case, the boolean variable *hooking* is set (ln. $17, 20$) to indicate a successful attempt to connect the edge to the parent component of e, triggering another round of *hooking* and *shortcut* phases. The shortcut phase (ln. $21-23$) runs in parallel across all edges in a Φ_k set, with continuous linking up (ln. $22-23$) to the parent until all edges under a specific component are directly connected to the root. It is important to note that both the *hooking* and the *shortcut* phases have a benign race condition that does not affect correctness.

Creating Superedge: Algorithm 8 illustrates the design considerations for creating superedges in parallel. A list or vector of subsets of superedges is allocated (ln. 1), with a size equal to the number of available parallel threads. Each thread can contribute to its own subset of superedge(s), mitigating race conditions. Both Algorithm 7 and Algorithm 8 are consecutively invoked on the same Φ_k set. All edges in the Φ_k set are processed in parallel to identify their triangle-composing edges (e_1 and e_2), retrieve their trussness from τ, and compute the minimum trussness (ln. $3-8$). A superedge is then established between the supernode

Algorithm 7: Construct SuperNode(s) in parallel

Data: A graph $G(V, E)$ and a dictionary of edges with their k-truss values
Result: A dictionary of edges, Π, with each edge having their supernode ID/parent component ID assigned

```
   /⋆ Each edge initially forms its own component ⋆/
1  for e(u, v) ∈ E do
2      Π(e) ← e
   /⋆ Group edge set, E, into different subsets based on their trussness, e.g., k = 3, 4, ..., k_max ⋆/
3  for e(u, v) ∈ E do
4      if (τ(e) = k) then
5          Φ_k ← Φ_k ∪ e
   /⋆ Run ShiloachVishkin (SV) connected component for each Φ_k set ⋆/
6  for k = k_min to k_max do
7      hooking ← true
8      while (hooking) do
9          hooking ← false
           /⋆ Hooking phase for SV ⋆/
10         for e(u, v) ∈ Φ_k in parallel do
11             Compute a list of all common neighbors, W, that make triangle(s) with e
12             for (w ∈ W) in parallel do
13                 e_1 ← (u, w) ∈ E
14                 e_2 ← (v, w) ∈ E
15                 if (Π(e) < Π(e_1) and Π(e_1) = Π(Π(e_1)) and τ(e) = τ(e_1)) then
16                     Π(Π(e_1)) ← Π(e)
17                     hooking ← true
18                 if (Π(e) < Π(e_2) and Π(e_2) = Π(Π(e_2)) and τ(e) = τ(e_2)) then
19                     Π(Π(e_2)) ← Π(e)
20                     hooking ← true
           /⋆ Shortcut phase for SV ⋆/
21         for e ∈ Φ(k) in parallel do
22             while (Π(Π(e)) ≠ Π(e)) do
23                 Π(e) ← Π(Π(e))
```

containing the current edge e with trussness k and the supernode containing the edge e_1 or e_2, where both have a minimum trussness $k_1 < k$ or $k_2 < k$. A thread creating a superedge adds it to its subset of superedge(s) (ln. 10, 12).

Creating Supergraph (Index): Algorithm 9 addresses the parallel merging of the thread-local subset of superedges, constructed in Algorithm 8, to form the supergraph $\mathbb{G}(\mathbb{V}, \mathbb{E})$. A list, *sm_graph*, is allocated with a size equal to the total number of threads (ln. 1). Each thread possesses a thread-local vector of vectors of superedge ID1, ID2, denoted as *sm_graph_t* (ln. 6), where the outer vector has entries equal to the total number of threads, and *ID1* and *ID2* represent the supernode IDs. All superedges from each thread-local subset, constructed in Algorithm 8, are hashed to the vector corresponding to the destination thread (ln.

Algorithm 8: Create SuperEdge(s) in parallel

Data: $\Phi(k)$ set of current k from Algo. 7
Result: A vector/list of thread local superedge subsets

```
1  Allocate vector<set<compID1, compID2≫sp_edges, a vector of size = number of threads
2  for e(u, v) ∈ Φ(k) in parallel do
       /⋆ W is the list of neighbor(s) forming triangle(s) with e ⋆/
3      for w ∈ W in parallel do
4          tid ← get_thread_id
5          e_1 ← (u, w) ∈ E
6          e_2 ← (v, w) ∈ E
7          k ← τ(e), k_1 ← τ(e_1), k_2 ← τ(e_2)
8          lowest_k ← min(k, k_1, k_2)
           /⋆ Create superedge downward, k > k_1 ⋆/
9          if k > lowest_k and lowest_k = k_1 then
10             sp_edges[tid].insert({Π(e_1), Π(e)})
           /⋆ Create superedge downward, k > k_2 ⋆/
11         if k > lowest_k and lowest_k = k_2 then
12             sp_edges[tid].insert({Π(e_2), Π(e)})
```

Algorithm 9: Construct SuperGraph in parallel

Data: A vector/list of thread local superedge subsets **sp_edges** from Algo. 8
Result: A complete list of superedges from merging thread local superedge subsets

```
1  Allocate, a list sm_graph of size = num_threads
2  Allocate vector<vector<{ID1, ID2} ≫combined_sm_graph_t(num_threads)
3  Allocate a contiguous buffer, final_sp_graph, of type <ID1, ID2>and size = total_num_sp_edges
4  Inside each thread t in parallel
5  {
6      Allocate thread-local vector<vector<{ID1, ID2} ≫ sm_graph_t(num_threads)
7      for each superedge ∈ sp_edges[t] do
8          ID1 ← superedge.ID1
9          ID2 ← superedge.ID2
10         dest_t ← (hash(ID1, ID2))%num_threads
11         sm_graph_t[dest_t] ← superedge
12     sm_graph[t] ← sm_graph_t
13     for sm_t ∈ sm_graph do
14         Copy all sm_t[t] into combined_sm_graph_t[t]
15     sort combined_sm_graph_t[t]
16     remove duplicates from combined_sm_graph_t[t]
       /⋆ Parallel reduction ⋆/
17     total_num_sp_edges += combined_sm_graph_t[t].size()
18 }
19 Merge combined_sm_graph_t[t] into final_sp_graph in parallel
```

7 − 11). Subsequently, each thread combines (ln. 13 − 14) all its corresponding superedges, annotated by

all threads, into *combined_sm_graph_t*, which was allocated in ln. 2, and removes duplicates (ln. $15-16$). Finally, all threads merge their superedge(s) to obtain the final supergraph (ln. 19).

5.3.2 Algorithm Complexity Analysis

The time complexity for computing support/triangles is best achieved with $\mathcal{O}(|E|^{1.5})$ [119]. In Algorithm 6, supernodes are computed using *BFS*, which has a time complexity of $\mathcal{O}(|V| + |E|)$ for a graph $G(V, E)$ with the number of vertices $|V|$ and the number of edges $|E|$. However, for the edge-induced graph of *EquiTruss*, where the constituent components of supernodes are edges from the original graph $G(V, E)$, the time complexity is $\mathcal{O}(|E| + |E|^{1.5})$, with $|E|^{1.5}$ being the maximum number of triangles possible for a graph with $|E|$ edges [44]. The time complexity of the CRCW (concurrent read, concurrent write) based Shiloach-Vishkin CC is $\mathcal{O}(\frac{|E| \log |V|}{p} + \log |V|)$ for p parallel processing units [64]. In the case of Algorithm 7 for the edge-induced graph of *EquiTruss*, the time complexity using p threads is $\mathcal{O}(\frac{|E|^{1.5} \log |E|}{p} + \log |E|)$. The majority of component identification work for Afforest is proportional to $\mathcal{O}(|V|)$ [130]. The edge-induced graph of *EquiTruss* would take $\mathcal{O}(|E|)$ time, with an additional $\mathcal{O}(|E|^{1.5})$ time complexity to compute triangles. Therefore, the time complexity is $\mathcal{O}(\frac{|E|^{1.5}+|E|}{p})$ for p parallel units. The space requirement for both groups of Algorithm 6 and Algorithms 7, 8, 9 is proportional to the number of edges in the original graph $G(V, E)$, i.e., $\mathcal{O}(|E|)$ for storing the relevant dictionary and data structure. Additionally, there is an extra memory requirement for storing the summary graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$, making the total space complexity $\mathcal{O}(|E| + |\mathbb{E}|)$.

5.3.3 Optimization of Compute Kernel

Our implementation of *Baseline EquiTruss* is decomposed into distinct computational kernels, as elucidated in Section 5.4. These kernels encompass *Support*, *Initialization*, *SpNode*, *SpEdge*, *SmGraph*, and *SpNodeRemap*. Among these, the *SpNode* kernel (outlined in Algorithm 7) emerges as the most computationally intensive, prompting our efforts to enhance its efficiency through several optimizations. To streamline storage and operations, we leverage the *CSRGraph* class from the *GAP* Benchmark Suite [23]. Instead of searching for trussness (k) throughout the entire edge set in a dictionary/hashmap for each edge (as denoted in lines 4, 15, and 18 in Algorithm 7), we narrow down the search to the neighborhood list using *CSR* storage from *GAP*. Furthermore, we replace the hashmap employed for storing and retrieving parent component/supernode IDs for the entire edge set with a contiguous memory buffer to enhance efficiency.

The connected component (CC) framework of *Shiloach-Vishkin* (*SV*) from *GAP* has been customized to accommodate our unique scenario, where we consider an edge as an entity within the connected component, deviating from the conventional use of vertices in SV connected components. In this adapted *SV* design, further processing is skipped if $\Pi(e) = \Pi(e_1)$ (as seen in lines 15 or 18 in Algorithm 7). This adaptation yields an optimal configuration for constructing *SpNode* through the Shiloach-Vishkin CC algorithm, denoted as *SpNode C-Optimal*. Afforest [130] enhances the SV algorithm for CC by adjusting the convergence logic,

applied independently to distinct subgraphs. It employs component approximation through subgraph sampling to minimize edge processing while ensuring an exact solution. The original SV algorithm's two phases (*hooking* and *shortcut*) are modified into corresponding *link* and *compress* phases to prevent concurrent parallel units from overriding each other's work. Similar to the *SpNode C-Optimal*, we have adapted the *Afforest* implementation from *GAP* to suit our specific case in the connected component algorithm.

5.4 Performance Evaluation

5.4.1 Experimental Settings

Our algorithm was coded in C++, employing the OpenMP framework for multi-threading, and compiled using the GNU g++ compiler. The computations were conducted on the *Perlmutter* CPU compute node at the National Energy Research Scientific Computing Center (NERSC). This CPU node is equipped with 2 AMD EPYC 7763 CPUs, each featuring 64 cores and a base frequency of 2.45 GHz. Additionally, it has 512 GB of DDR4 memory and a memory bandwidth of 204.8 GB/s per CPU. The undirected network datasets, as detailed in Table 5.3, were sourced from SNAP [84].

Table 5.3: The social and information network datasets used for our experiments of sequential and parallel *EquiTruss* approaches

Network	**# Vertices**	**# Edges**
Amazon	334863	925872
DBLP	317080	1049866
YouTube	1134890	2987624
LiveJournal	3997962	34681189
Orkut	3072441	117185083
Friendster	65608366	1806067135

5.4.2 Effect of Compute Kernel Optimization

Figure 5.3 provides a breakdown of time percentages for operational kernels in the parallel *EquiTruss* across different networks. It is evident from the illustration that the construction of supernodes (*SpNode* in Figure 5.3) constitutes the most resource-intensive aspect of the overall algorithm. This kernel accounts for up to 79% and 87% of the total index construction time for the *YouTube* and *Orkut* networks, respectively. The second most resource-intensive kernel is the creation of superedges, as outlined in Algorithm 8, ranging from as low as 6% for the *DBLP* network to 10% for the *YouTube* network in terms of the overall time.

Figure 5.4 visually demonstrates the enhanced performance, measured in terms of speedup, achieved through our optimizations in Algorithm 7 across *SpNode Baseline*, *SpNode C-Optimal*, and *SpNode Afforest*. The supernode construction time notably decreases from 8655 seconds in the *Baseline* approach to 2093 seconds

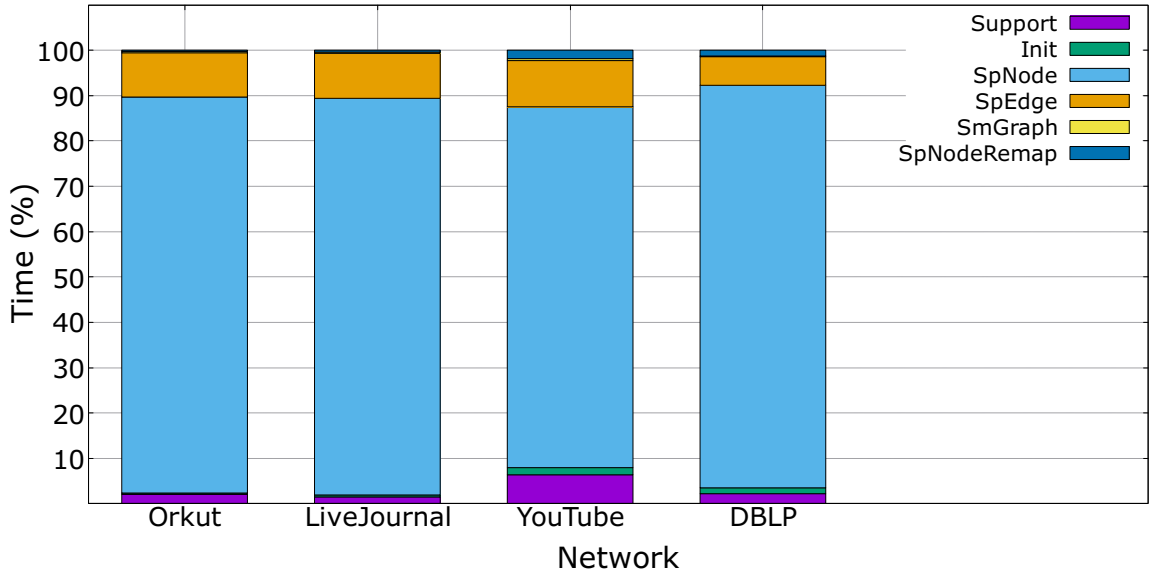


Figure 5.3: Functional components in the Baseline version of the parallel *EquiTruss* algorithm. The distribution of run time percentages in a single-threaded execution across four diverse networks is presented. Notably, the *SpNode* kernel constitutes the predominant portion, ranging from $(79 - 89)\%$ of the overall execution time.

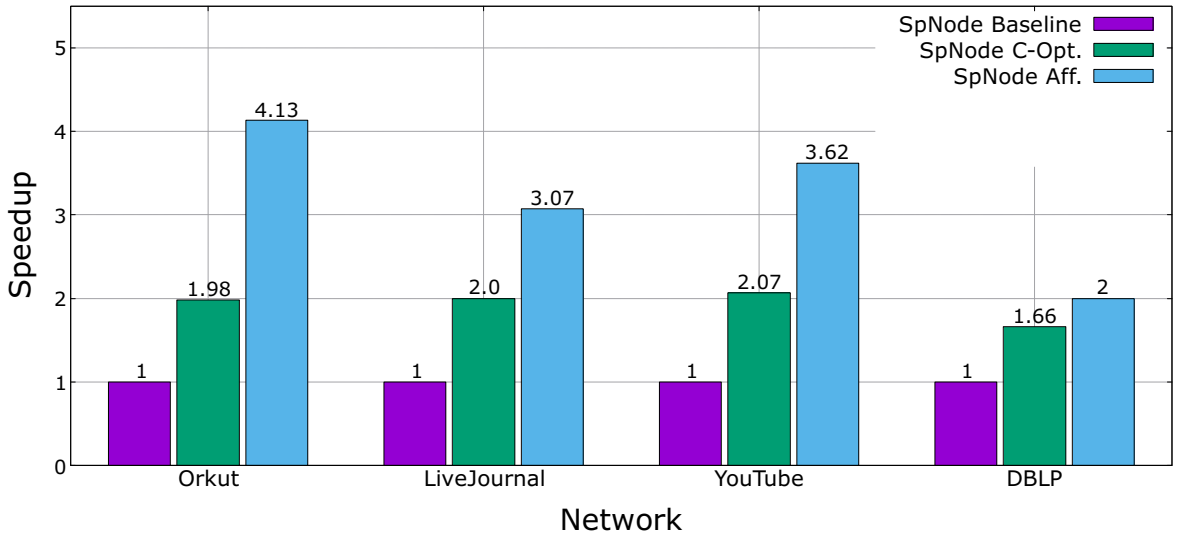


Figure 5.4: Enhancement in single-threaded execution time through speedup of the primary operational kernel. The improvement in single-threaded execution time, measured in terms of speedup, for the primary operational kernel of EquiTruss is achieved by employing cache-optimized data structures and the Afforest connected component algorithm.

in *SpNode Aff.* and 4371 seconds in *SpNode C-Opt.*, resulting in speedups of 4.13× and 1.98× respectively, for the *Orkut* network in a single-thread execution. The optimization of Afforest for connected components over Shiloach-Vishkin (SV) exhibits significantly improved performance, as depicted by the blue bar in Figure 5.4.

Similarly, for the *LiveJournal* network, the supernode construction time reduces to 453 seconds in *SpNode Aff.* and 696 seconds in *SpNode C-Opt.*, compared to the *Baseline SpNode* computation time of 1393 seconds, resulting in speedups of 3.07× and 2× respectively.

5.4.3 Performance Analysis

Comparison with State-of-the-art: We acquired the initial Java implementation of the sequential EquiTruss proposed by Akbas et al. [2] to conduct a comprehensive comparison with our implementations. The runtime evaluation is presented in Table 5.4. In the case of the *LiveJournal* network, the serial Java code's index construction phase (SpNd, SpEdge, and SmGraph) surpasses our Baseline by 3.3×, C-Opt. EquiTruss by 1.8×, and Afforest EquiTruss by 1.3× in sequential settings. Our parallel versions (128-thread) exhibit substantial improvements, being 11.55× faster (Baseline), 20.59× faster (C-Opt. EquiTruss), and 29.56× faster (Afforest EquiTruss) than the sequential Java implementation. Notably, for larger networks like *Orkut* with $117M$ edges, the sequential Java code encounters memory limitations, while all our implementations in Table 5.4 adeptly handle billion-size graphs (e.g., *com-Friendster*). To assess the accuracy of the constructed supernodes or supergraphs, we conducted a thorough comparison of the total number and constituent components (constituent edges) of supernodes and superedges between the sequential Java code by Akbas et al. [2] and our implementations in both sequential and parallel setups. Remarkably, the results are identical in all cases. *EquiTruss* or parallel *EquiTruss* relies on deterministic sub-kernels: k-truss connected components. With no approximation at any stage, the formulation of k-triangle connectivity ensures the exactness of the connected components (supernodes). Consequently, we only report the number of supernodes and superedges in Table 5.5, as the accuracy remains consistently at 100% for all cases.

Table 5.4: Comparison of the total runtime for key computational phases (SpNd, SpEdge, and SmGraph) in the construction of the *Index*. This comparison is conducted in a single-threaded environment, contrasting our implementations with the respective computational phases of the original Java implementation by Akbas et al. [2].

Network	Baseline (sec)	C-Opt. (sec)	Aff. (sec)	Akbas et al. [2] (sec)
Amazon	6.77	3.96	3.24	1.46
DBLP	10.92	7.37	6.57	2.33
LiveJournal	1549	851	608	467
Orkut	9631	5268	2990	MLE

Speedup: Table 5.6 presents the speedup enhancements for our parallel *Baseline EquiTruss*, an improved version (*C-Opt. EquiTruss*) over the Baseline, and a superlative version *Afforest EquiTruss*. The speedup for the *Baseline* is 13.92×, 27.31×, and 29.63× for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. *C-Opt. EquiTruss* demonstrates 8.82×, 22.25×, and 22.61× speedup over the single-threaded counterpart for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. Likewise, using *Aff. EquiTruss*, we observe

7.06×, 19.55×, and 18.27× speedup over the single-threaded counterpart for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. In each case, the maximum speedup occurs when employing the maximum number of threads (i.e., physical cores) in a compute node, which is 128. The *Baseline* version attains a superior speedup, being less efficient but performing more computation than the other two versions. It's noteworthy that the *Baseline* version still achieves significantly lower run-time compared to our C++ implementation of *EquiTruss* based on Akbas et al. [2] (*Original EquiTruss* in Table 5.2). Focusing on the speedup gain from the sequential *Baseline* to our optimized version (*Aff. EquiTruss*) with 128 threads, it reaches 16.10×, 47.8×, and 55.24× for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. These substantial speedup gains underscore the effectiveness of our parallel implementation over the sequential versions.

Table 5.5: Quantifying the count of supernodes and superedges within summary graphs across various networks. The findings are verified under both sequential and parallel conditions, and comparisons are made against the C++ implementation of the work by Akbas et al. [2].

Network	**No. of Supernodes**	**No. of Superedges**
Amazon	115060	103513
DBLP	126904	105409
YouTube	400408	940550
LiveJournal	4765102	13405280
Orkut	17227001	76631446

Table 5.6: Contrast between the elapsed time for the slowest execution (1-thread) and the fastest execution time (128-thread) in seconds, along with the associated speedup (X) for various optimized versions of our parallel *EquiTruss*.

	Base. Eq.			**C-Opt. Eq.**			**Aff. Eq.**		
Network	1t	128t	(**X**)	1t	128t	(**X**)	1t	128t	(**X**)
Amzn.	7.26	0.52	13.86	4.45	0.46	9.7	3.74	0.40	9.16
DBLP	11.52	0.62	18.53	7.96	0.51	15.52	7.16	0.49	14.46
YouTb.	36.56	2.62	13.92	21.60	2.44	8.82	16.07	2.27	7.06
LvJrnl.	1.6k	58.34	27.31	895.03	40.21	22.25	651.69	33.33	19.55
Orkut	9.9k	334.89	29.63	5.5k	245.97	22.61	3.2k	179.64	18.27

Strong Scalability: Figures 5.5, 5.6, and 5.7 depict the strong scalability trends for an increasing number of threads, ranging from 1 to 128. Each figure contains three distinct curves representing scalability for three different design phases (*Baseline EquiTruss*, *C-Opt. EquiTruss*, and *Aff. EquiTruss*) of the parallel *EquiTruss* solution. In Figure 5.5 for the *Orkut* network, the execution time diminishes from 3283 seconds to 179 seconds with *Aff. EquiTruss* using 128 threads. Likewise, Figure 5.6 illustrates the runtime scalability for the *LiveJournal* network across the three design phases of the *EquiTruss* problem. The runtime decreases from 895 seconds with a single thread to 40 seconds with 128 threads for *C-Opt. EquiTruss* (blue curve). Finally, Figure 5.7 showcases the execution time reduction from 36.56 seconds to 2.62 seconds for the *YouTube*

network using the *Baseline* version of *EquiTruss*.

Figure 5.8 exhibits strong scalability for the *SpNode* construction runtime concerning the billion-size *Friendster* network. Due to a regular compute node's 12-hour occupancy limit on NERSC *Perlmutter* supercomputer, only the *SpNode* construction cost is shown. In this Figure, the *SpNode* computation time for *C-Opt. EquiTruss* cannot be displayed for single-thread and 2-thread scenarios due to the occupancy hour limit.

In Figures 5.9 and 5.10, we illustrate the runtime reduction for the three major kernels outlined in Algorithm 7, 8, and 9 across our three different versions of parallel *EquiTruss* using 1, 8, 32, and 128 threads. The *SpNode* kernel (light purple) dominates over the other two kernels, *SpEdge* (light green) and *SmGraph* (light blue), in a single thread. However, it significantly diminishes along with the other two kernels as the number of parallel threads increases, becoming negligible with 128 threads for both example networks (Figures 5.9 and 5.10).

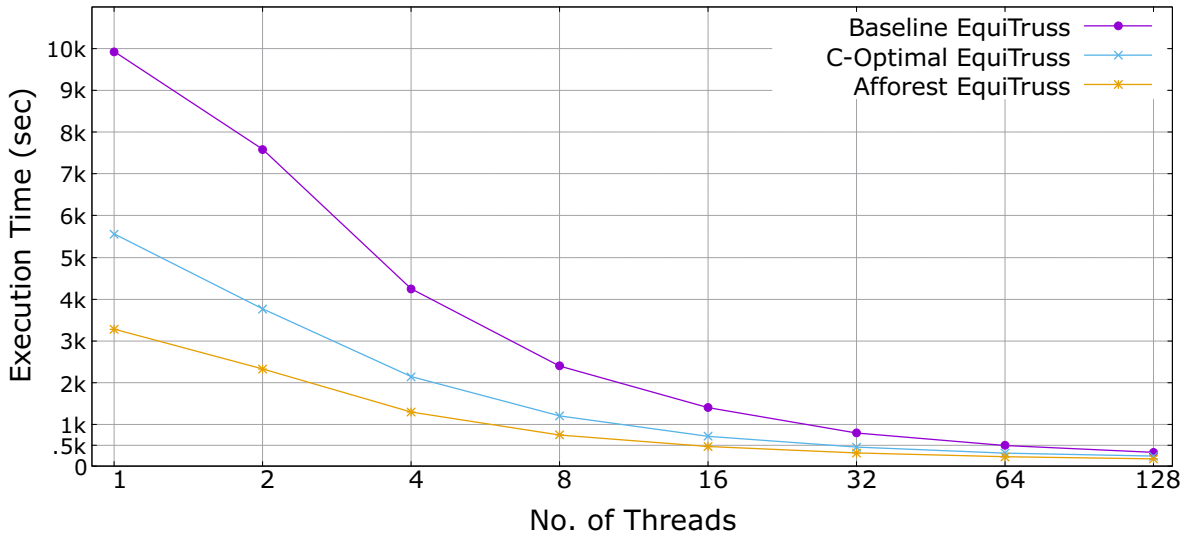


Figure 5.5: Demonstrating the scalability and reduction in runtime across 3 distinct design phases of the parallel *EquiTruss* algorithm for the *Orkut* network. Notably, the sequential execution times decrease from 9924, 5561, and 3283 seconds to 334, 245, and 179 seconds, respectively, for the Baseline, C-Opt. EquiTruss, and Afforest EquiTruss versions when utilizing 128 threads on the *Orkut* network.

Parallel Efficiency: Figures 5.11, 5.12, and 5.13 depict parallel efficiency using histogram plots for three different networks. Parallel efficiency (ε) measures how well an algorithm scales in parallel, comparing parallel runtime to the sequential runtime under the assumption of perfect scalability [15]. The formula for parallel efficiency is given by $\varepsilon = \frac{T_{seq}}{pT(p)}$, where p is the number of parallel units, $T(p)$ is the time with p parallel units, and T_{seq} is the sequential runtime.

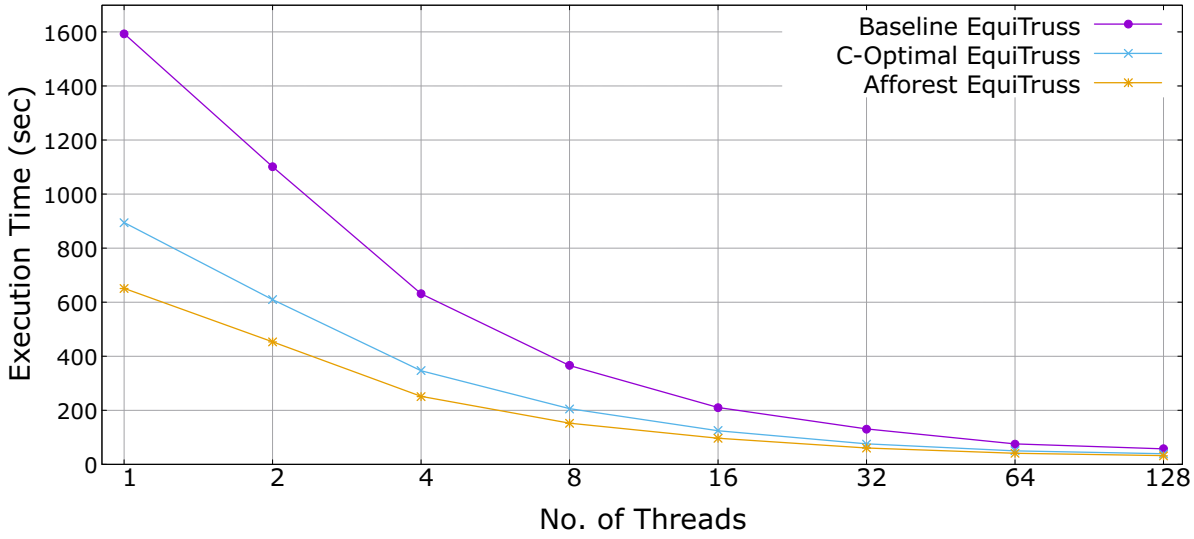


Figure 5.6: Illustrating the scalability and reduction in runtime across 3 distinct design phases of the parallel *EquiTruss* algorithm for the *LiveJournal* network. For instance, the execution times decrease from 1593, 895, and 651 seconds to 58, 40, and 33 seconds, respectively, for the Baseline, C-Opt. EquiTruss, and Afforest EquiTruss versions when utilizing 128 threads on the *LiveJournal* network.

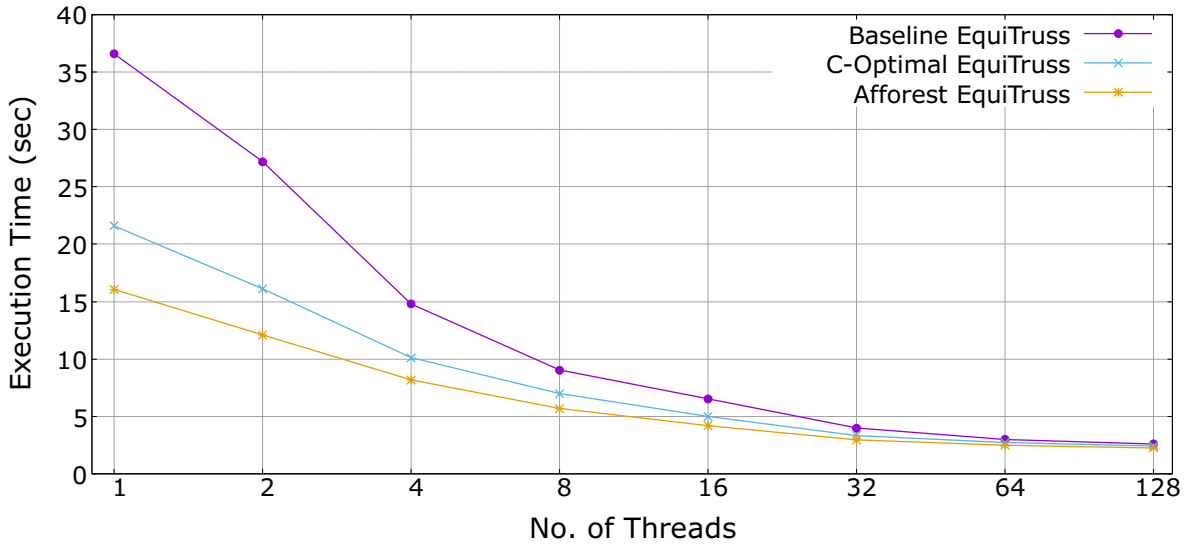


Figure 5.7: Demonstrating the scalability and reduction in runtime across 3 distinct design phases of the parallel *EquiTruss* algorithm for the *YouTube* network. For example, the execution times decrease from 6253, 3586, and 2093 seconds to 131, 102, and 90 seconds, respectively, for the Baseline, C-Opt. EquiTruss, and Afforest EquiTruss versions when employing 128 threads on the *YouTube* network.

In each histogram plot, three bars are grouped together, representing our three versions of the *EquiTruss* implementation. In Figure 5.11, we observe 70% parallel efficiency for *Aff. EquiTruss* and 73% parallel efficiency for *C-Opt. EquiTruss* using 2 threads for the *Orkut* network. For the same network, these corresponding

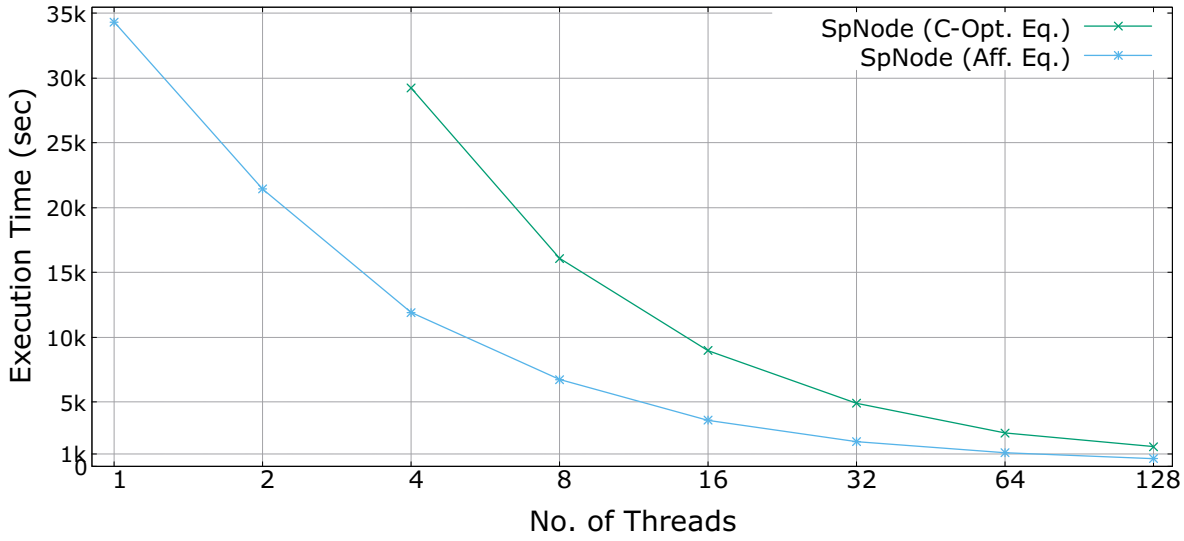


Figure 5.8: Reduction in execution time for the *SpNode* kernel on the billion-size *Friendster* network utilizing *C-Opt. EquiTruss* and *Aff. EquiTruss*. In the case of *Aff. EquiTruss*, the single-thread run-time of 34332 seconds diminishes to just 612 seconds when employing 128 threads.

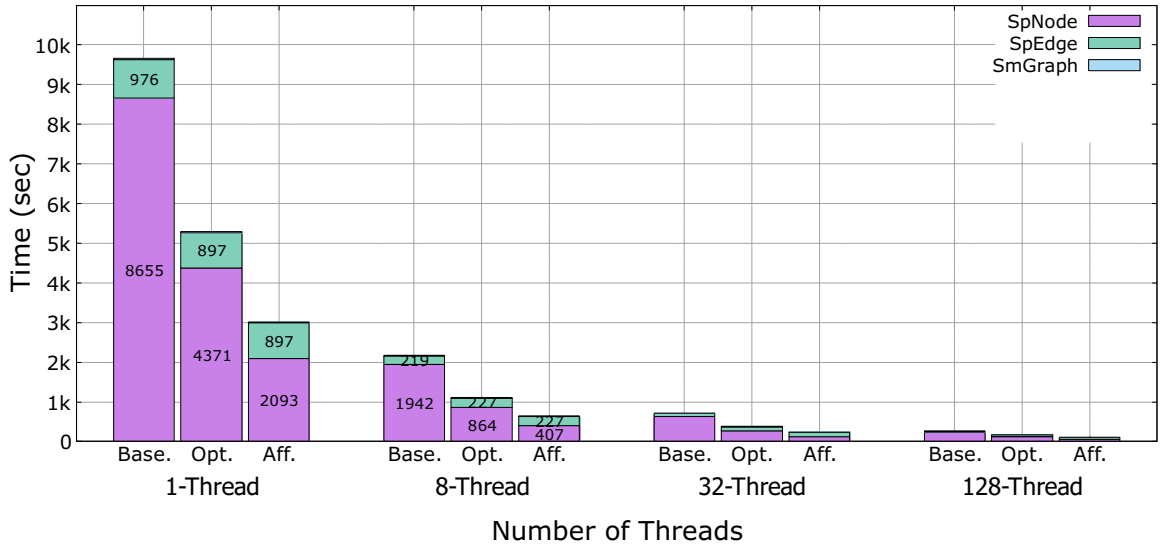


Figure 5.9: Breakdown of execution time for the major compute kernels (*SpNode, SpEdge, SmGraph*) on the *Orkut* network. The time reduction for these kernels is depicted across varying thread counts (1, 8, 32, and 128). Specifically, the *SpNode* execution time decreases from 2093 seconds in the single-threaded setting to 407 seconds with 8 threads, further to 127 seconds with 32 threads, and eventually to 60 seconds with 128 threads for the *Afforest EquiTruss* on the *Orkut* network.

parallel efficiencies decrease to 22% and 27%, respectively, with 64 threads, and to 14% and 17%, respectively, with 128 threads. The utilization of 128 threads across our diverse EquiTruss versions demonstrates the potential for even greater scalability in a shared-memory system with a higher number of available threads.

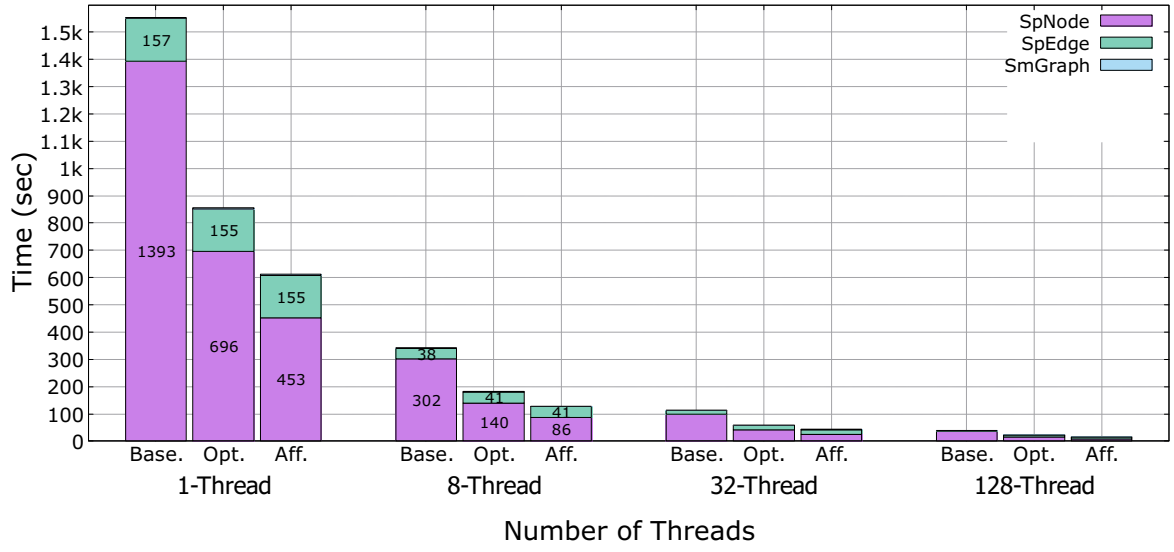


Figure 5.10: Breakdown of execution time for the major compute kernels (*SpNode, SpEdge, SmGraph*) on the *LiveJournal* network. The time reduction for these kernels is depicted across varying thread counts (1, 8, 32, and 128). Specifically, the *SpNode* execution time decreases from 696 seconds in the single-threaded setting to 140 seconds with 8 threads, further to 42 seconds with 32 threads, and eventually to 16 seconds with 128 threads for the *C-Opt. EquiTruss* on the *LiveJournal* network.

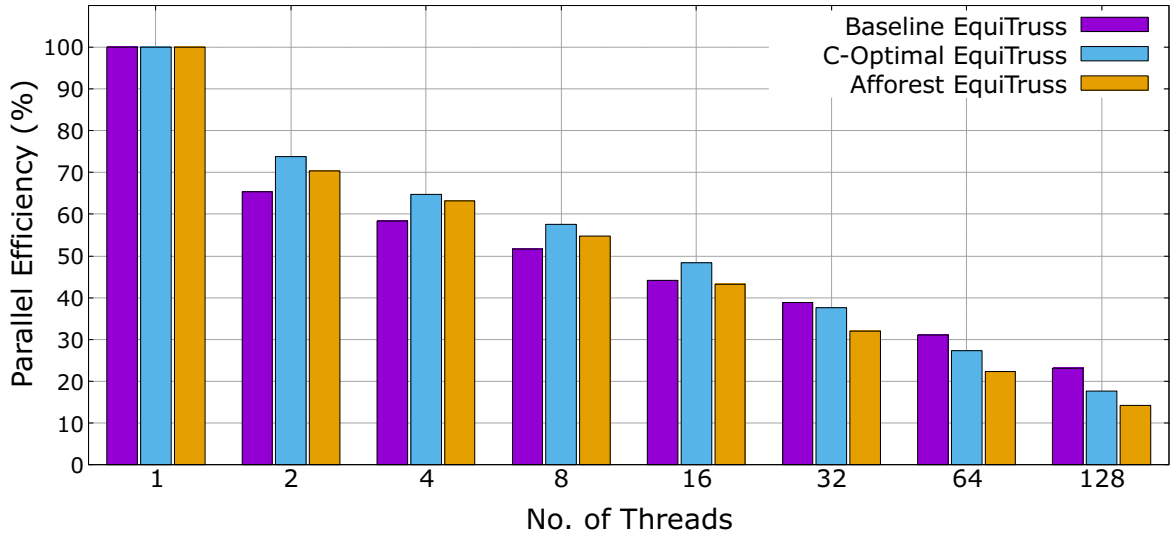


Figure 5.11: Demonstrating parallel efficiency with 3 distinct designs of the parallel *EquiTruss* on the *Orkut* network. For instance, the parallel efficiencies are 38.89%, 37.66%, and 32%, respectively, for the baseline *EquiTruss*, C-Opt. *EquiTruss*, and Afforest *EquiTruss* when utilizing 32 threads.

5.5 Community Search

Until the previous section (5.4), we have discussed the outcome of our parallel index construction for community search. The goal of the constructed index is to facilitate the local community search in a coherent

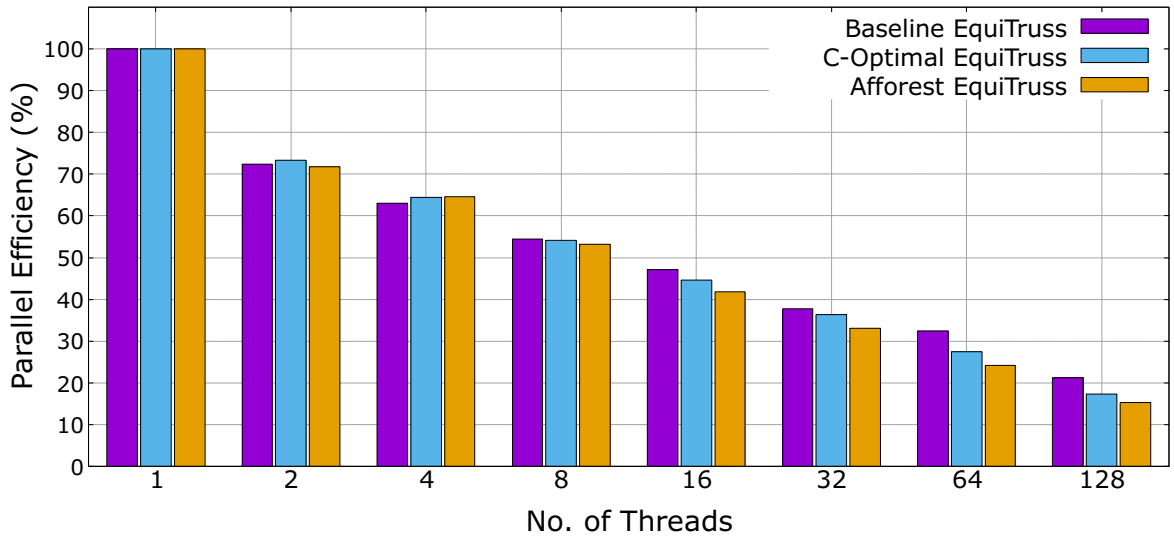


Figure 5.12: Demonstrating parallel efficiency with 3 distinct designs of the parallel *EquiTruss* on the *LiveJournal* network. For instance, the parallel efficiencies are 37.75%, 36.41%, and 33.14%, respectively, for the baseline *EquiTruss*, C-Opt. *EquiTruss*, and Afforest *EquiTruss* when utilizing 32 threads.

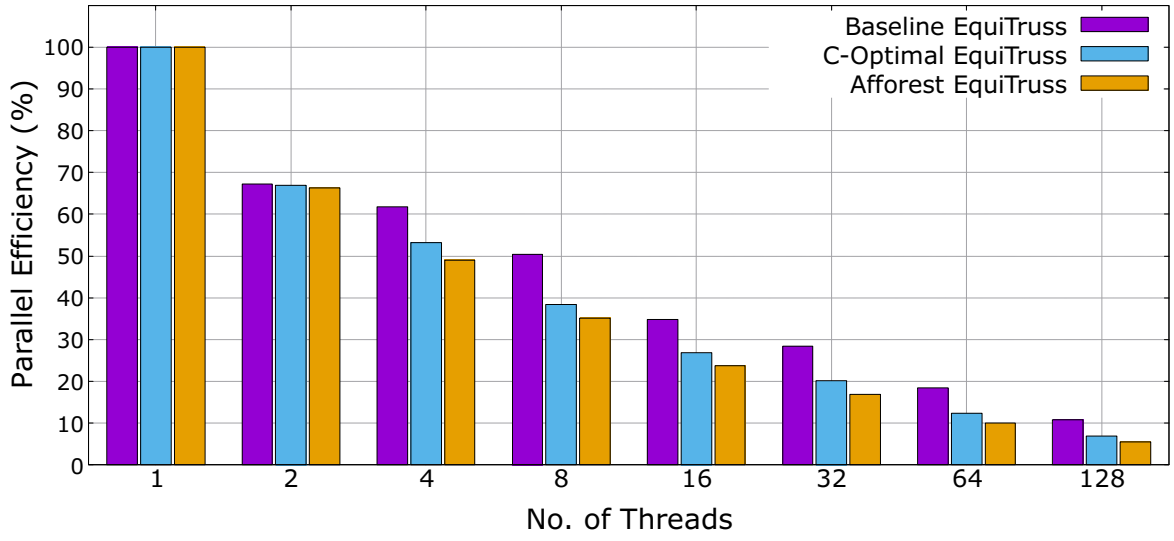


Figure 5.13: Demonstrating parallel efficiency with 3 distinct designs of the parallel *EquiTruss* on the *YouTube* network. For instance, the parallel efficiencies are 28.38%, 20.12%, and 16.84%, respectively, for the baseline *EquiTruss*, C-Opt. *EquiTruss*, and Afforest *EquiTruss* when utilizing 32 threads.

manner. In this section, we discuss the methodology for our parallel community search. We then present the outcome and comparison with sequential and parallel execution.

Algorithm 10: vertex $q \in G(V, E)$ to list of supernode(s) $\in \mathbb{G}(\mathbb{V}, \mathbb{E})$

Data: Graph $G(V, E)$ in CSR format
Result: A dictionary of the list of supernode(s) for all the vertices $v \in G(V, E)$

```
1 Allocate a vector of vectors of size = numVertices ∈ G(V, E)
2 for u ∈ G(V, E) in parallel do
3     for v ∈ neighbors(u) do
4         e ← (u, v)
5         Π_e ← Π(e)
6         vertex2SuperGraphNodes[u].push_back(Π_e)
```

5.5.1 Parallel Community Search Methodology

Before running the community search, we construct an auxiliary data structure where a mapping between the original graph vertices to the summary graph supernode(s) (Algorithm 10). We allocate a vector of size equal to the total number of vertices of the original graph (line 1). Each of those entries represents the corresponding vertex and can be indexed by that vertex label. We can store and retrieve the supernode(s) that the corresponding vertex belongs to in a vector of supernode ID(s). We traverse each vertex of the original graph (line 2) in parallel, find their neighbor(s) (line 3), and corresponding edge (line 4). We then find the supernode ID of that edge (line 5) and push the supernode ID to the vector of the corresponding vertex (line 6).

Next in Algorithm 11 we present our parallel approach to community search utilizing the summary graph we constructed and described in Algorithms 6, 7, 8, and 9. The community search is conducted similarly to the Breadth First Search (BFS) traversal on a graph. The community search function receives 2 arguments - a query vertex q and a trussness k. We allocate a vector with a size equal to the number of supernodes (s) in the summary graph to track the visited supernode during the community search (line 1 in Algorithm 11). The supernode(s) to the query vertex q is retrieved (line 2 in Algorithm 11) from the auxiliary data structure $vertex2SuperGraphNodes$ constructed in Algorithm 10. In line 3, we declare a vector of vector, C, to store the retrieved edge IDs (constituent components of the community). In line 4, we traverse each of the supernode(s). In lines $5-9$ use the supernode, $snID$, to access the index and corresponding supernode structure in the summary graph stored in CSR format. If the trussness of the starting edge of the supernode, $\tau(e)$, is greater than or equal to the query trussness k, the BFS-based community search can proceed (line 10). To facilitate the parallel BFS, we use a vector ($frontier$) instead of a *queue* used in the traditional BFS search (line 11). To prevent race conditions occurring from multiple threads reading/writing the same $frontier$ vector, we keep an additional vector $new\_frontier$ (line 12) for all threads to write their newly found supernodes at the end of the BFS search. In line 13, 14, operations on $snID$ are done to kickstart the BFS search. We allocate a vector (line 15) to store all the supernode(s) (and corresponding edge IDs from the original graph). The discovery of the new community will continue as long as there are newly found supernode(s) (line 16). Each thread proceeds with its supernode ($spndU$) in parallel (line 17), allocates its

Algorithm 11: Community Search in parallel

Data: A query vertex q from the original graph $G(V, E)$ and a trussness k
Result: A dictionary of edges grouped by their trussness, C

```
1  Allocate a global boolean vector visited of size = num_of_SpNode
2  Get supernode(s) of vertex q, snIDs ← vertex2SuperGraphNodes[q]
3  vector<vector<int ≫ C /⋆ for storing all retrieved local communities ⋆/
4  for (snID ∈ snIDs) do
5      Get the index (split_index) of the supernode (snID) from the CSR summary graph
6      startA ← splitters[split_index]
7      endA ← splitters[split_index + 1]
8      index ← sortedSNData[startA]
9      e(u, v) ← edgelist[index]
10     if (τ(e) ≥ k and visited[snID] = False) then
11         vector<int> frontier
12         vector<int> new_frontier
13         visited[snID] ← true
14         frontier.push_back(snID)
15         vector<int> comm /⋆ for storing communities from all threads ⋆/
16         while (!frontier.isEmpty()) do
17             for (spndU ∈ frontier) in parallel do
18                 vector<int> local_new_frontier
19                 vector<int> local_comm_edges
20                 Store all the edge IDs between startA and endA to the local_comm_edges of the
                    master thread
21                 for (spndV ∈ neighbors[spndU]) do
22                     Get index (splt_indx) of supernode (spndV) from summary graph
23                     s_A ← splitters[splt_indx]
24                     e_A ← splitters[splt_indx + 1]
25                     indx ← splitters[s_A]
26                     ê ← edgelist[indx]
27                     if (τ(ê) ≥ k and visited[spndV] = False) then
28                         Atomically capture spndV and add it to local_new_frontier
29                         Store all the edge IDs between s_A and e_A to the local_comm_edges of the
                            corresponding thread
30                         Add spndV to local_new_frontier
31             Copy thread-local local_new_frontier into new_frontier in parallel
32             Copy thread-local local_comm_edges into comm in parallel
33             Swap frontier with an empty vector
34             Swap frontier and new_frontier
35             Add comm to C
```

local vector ($local\_new\_frontier$) for storing the discovered thread-local supernode ID(s), and corresponding edge ID(s) (line 18, 19). The master thread (thread 0) stores (line 20) the corresponding edge ID(s) in the range $startA$ to $endA$ to its thread local vector ($local\_comm\_edges$). For each neighboring supernode, $spndV$ (in line 21), the range of edges of the original graph is accessed in lines $22 - 26$ similar to lines $5 - 9$. If the

trussness of the newfound supernode is $\geq k$, we then atomically capture the corresponding supernode and add the corresponding edges to the thread-local $local\_new\_frontier$ and $local\_comm\_edges$ (line $27-30$). The reason for atomic capture is to prevent data duplication. It may happen that the same supernode, $spndV$, and corresponding edges have been discovered by multiple threads. After all the threads participating in parallel community search reach the barrier point, all the thread-local frontier elements are copied in parallel to the designated spot in the $new\_frontier$ in line 31. Similarly, we compute the designated spot for each thread to copy the thread-local community edges and then copy those edges in parallel to the vector $comm$ (line 32). We then swap $frontier$ and $new\_frontier$ for the next iteration of parallel BFS search in line 33, 34 and add the discovered communities ($comm$) in the current BFS search phase to the global community vector C in line 35. While one might argue our choice of using regular parallel BFS search instead of direction-optimized BFS [22], we like to point out that the direction-optimized BFS takes advantage of the entire graph traversal where at a certain point all the leaf nodes try to locate its parent node (reverse search) in parallel. However, our aim is not to conduct the traversal on the entire summary graph, rather it is limited to the search of the supernodes having query vertex q in their constituent edges and trussness $\geq k$. Therefore, the parallel work efficiency achieved from bottom-up search in [22] is not realizable in our case.

5.5.2 Performance Evaluation

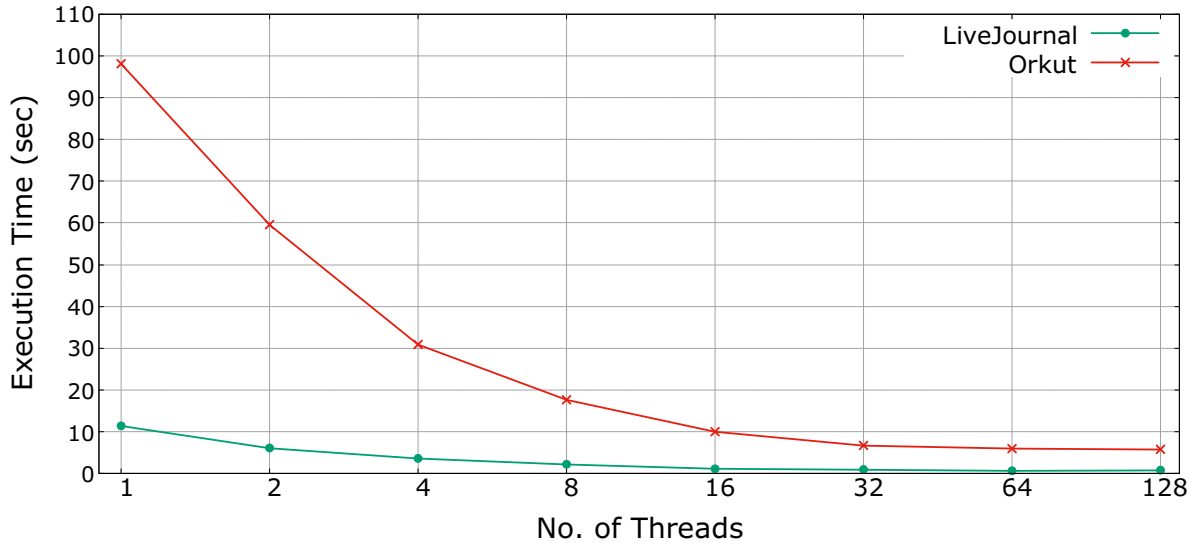


Figure 5.14: Illustrating runtime scalability of our parallel community search (Algorithm 11) for larger networks (*LiveJournal* and *Orkut*) using increasing number of threads. While it takes on average 98 seconds for community search on *Orkut* network using 1 thread, community search time reduces to 5.76 seconds using 128 threads.

We measure the performance of our parallel community search described in Algorithm 11 using *DBLP*, *YouTube*, *LiveJournal*, and *Orkut* network. The data reported here are based on the average of 100 execution

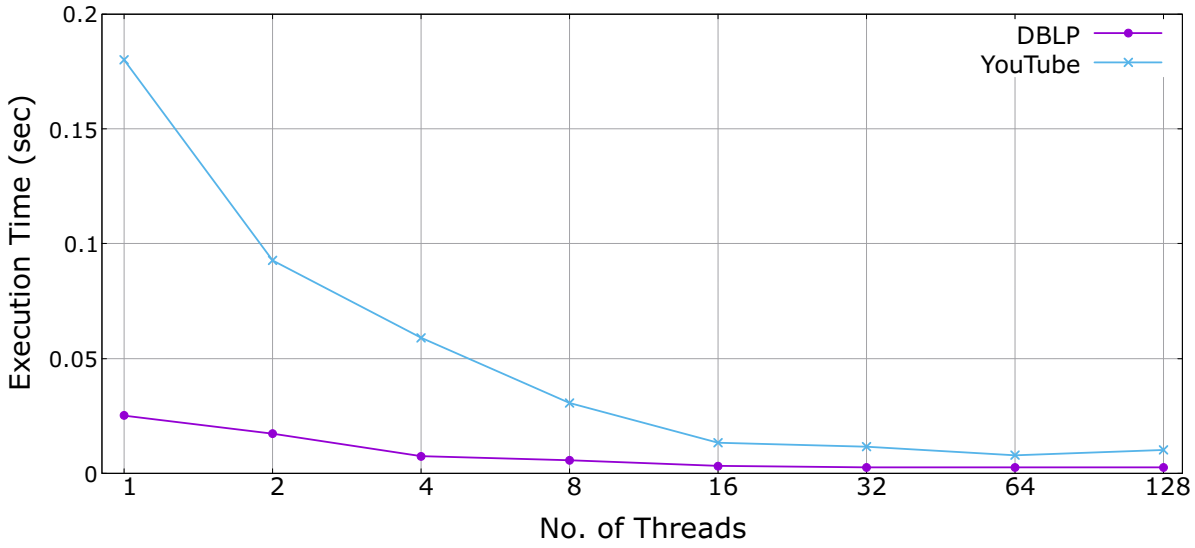


Figure 5.15: Illustrating runtime scalability of our parallel community search (Algorithm 11) for mid-size networks (*DBLP* and *YouTube*) using increasing number of threads. While it takes on average 0.18 second for community search on *YouTube* network using 1 thread, the search time reduces to 0.007 seconds using 64 threads.

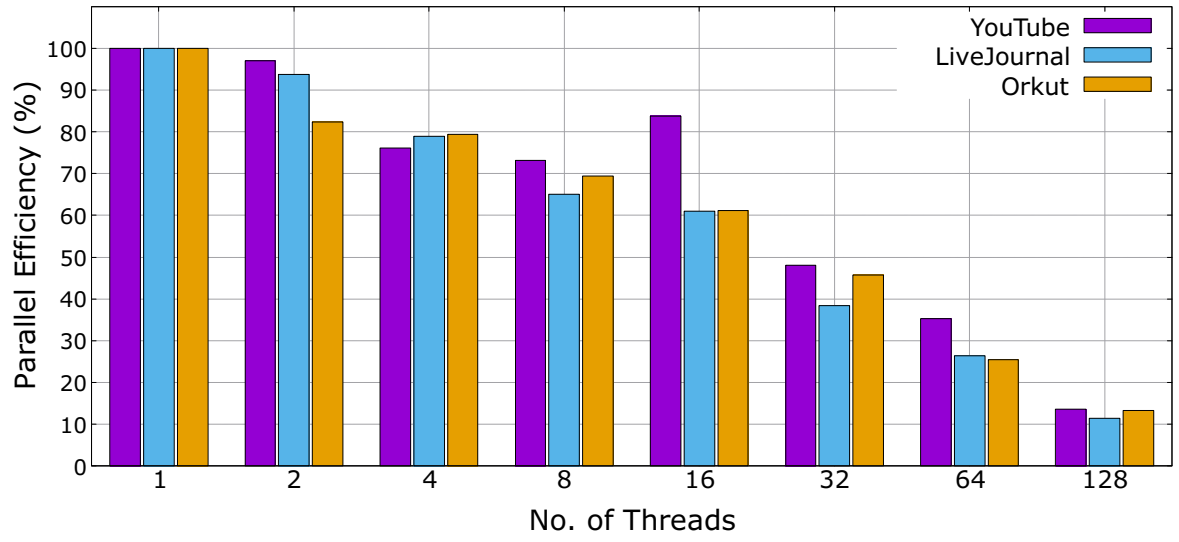


Figure 5.16: Illustrating parallel efficiency for 3 different networks (*YouTube*, *LiveJournal*, and *Orkut*) for community search using Algorithm 11. The parallel efficiencies are 82.34%, 79.38%, and 69.36%, respectively, using 2, 4, and 8 threads for the *Orkut* network.

of Algorithm 11 where the query vertices are picked randomly from the set of the vertices and the trussness of 3 is passed by default to allow the growth of the community to the maximum size possible for the given query vertex in the graph. In Figures 5.14 and 5.15 we illustrate the runtime scalability of our parallel local community search approach. As shown in Figure 5.14, the single thread runtime of 11.41 seconds

Table 5.7: Speedup comparison between sequential and parallel execution of community search (Algorithm 11) and state-of-the-art sequential community search by Akbas et al. [2] on experiment datasets.

Network	**Speedup** (vs sequential self)	**Speedup** (vs Akbas et al. [2])
DBLP	9.30	5.70
YouTube	22.58	10.51
LiveJournal	16.91	11.91
Orkut	17.01	MLE

for *LiveJournal* network reduces to 1.17 seconds using 16 threads and to 0.77 seconds using 128 threads. Similarly, the single thread runtime of 98 seconds reduces to 10.01 seconds and 5.76 seconds using 16 and 128 threads, respectively, for *Orkut* network. The single thread runtime of 0.18 seconds reduces to 0.013 seconds and 0.010 seconds using 16 threads and 128 threads, respectively, for *YouTube* network (Figure 5.15). We observe the minimum execution times of community search using 64 threads for *YouTube* (0.007 seconds) and *LiveJournal* (0.67 seconds) networks. The 128 thread execution times for *YouTube* is 0.010 seconds and for *LiveJournal* is 0.78 seconds. One possible reason for this could be there is not enough work to perform when going from 64 thread to 128 threads. Another related reason is the *Perlmutter* regular compute node has 2 AMD Milan CPUs, each having 64 cores. Going from 64 threads to 128 threads may incur the interconnect delay due to remote memory access. This effect is also observed in the reduction of execution runtime slowing down from 64 threads to 128 threads for *DBLP* and *Orkut* networks as well. In Figure 5.16, we illustrate the parallel efficiency of Algorithm 11 for *YouTube*, *LiveJournal*, and *Orkut* networks. The parallel efficiencies are 97%, 93.79%, and 82.33%, respectively, for *YouTube*, *LiveJournal*, and *Orkut* networks using 2 threads. Similarly, the parallel efficiencies are 83.77%, 60.94%, and 61.18%, respectively, for *YouTube*, *LiveJournal*, and *Orkut* networks using 16 threads. In Table 5.7, we show the maximum speedup observed for parallel community search against the sequential execution of its own self in column 2, and against the Java implementation of the community search in the work by Akbas et al. [2]. The maximum speedup observed for *YouTube* network is 22.58× using 64 threads against the sequential execution of Algorithm 11. The maximum speedup observed against the sequential community search in the study [2] is 10.51×. Similarly, the maximum speedup observed for *LiveJournal* network is 16.91× and 11.91× against the sequential execution of Algorithm 11 itself and against the java code from the study [2]. The reason for a higher sequential runtime of Algorithm 11 can be attributed to extra code and memory allocation for facilitating the parallel execution where each thread instantiates its own thread-local variable and merges to the global shared-memory space. That extra code is avoided in the implementation of community search in work [2]. We could not measure the speedup of the community search for *Orkut* network against the work [2] because of their code running out of memory in *Perlmutter* node with 512 GB memory.

5.6 Other Related Work

Some early investigations [101, 40] into clique-based overlapping community exploration rely on the clique percolation method, wherein, following the identification of k-cliques, all adjacent k-cliques (sharing $k-1$ nodes) are merged. Zhang et al. [153] introduce clique percolation clustering for detecting overlapping communities in PPI networks. Kumpula et al. [79] propose a clique-based approach applicable to both weighted and unweighted graphs. However, these approaches exhibit constraints related to clique size. Maity et al. [91] extend the work of [101] to complete graphs but inherit the associated limitations. Community search methods [121, 141] relying on maximal cliques face computational intractability. Local community search techniques based on *k-core* structures [17, 142] optimize metrics such as density, modularity, or conductance but struggle to exclude non-relevant vertices and cannot identify communities with overlapping memberships. Cui et al. [37] propose an online community search using a community model named α-adjacency-γ-quasi-k-clique, proven to be NP-hard [69], and the proposed approximation [37] does not offer promising solution quality. Truss-based community search, such as *TCP-Index* [69], maintains trussness information in groups of tree-structured indexes known as maximum spanning trees (MST). However, *TCP-Index* has limitations, including the need to redundantly maintain constituent edges of a graph G in multiple MSTs, and a resource-intensive truss reconstruction phase during community search. The approach in [2] circumvents these limitations by preserving an edge in a supernode structure but is constrained in scalability due to the sequential nature of the algorithm.

5.7 Concluding Remarks

Developing parallel algorithms for local community discovery is less explored compared to global community discovery. Previous studies have explored constructing community subgraphs using higher-order graph primitives like *cliques*, *quasi clique*, or *k-core*. However, an alternative strategy, *k-truss* decomposition, addresses inherent challenges such as computational intractability and lack of cohesiveness in these approaches. Leveraging the promising cohesiveness of k-triangle-connected subgraph structures, we integrate it with state-of-the-art parallel connected component methods in our formulation of the parallel *EquiTruss* algorithm for shared-memory environments. The parallel *EquiTruss* algorithm demonstrates effective scalability on large systems and datasets. It achieves a remarkable speedup of up to 55× when processing billion-size graphs on 128 physical cores of the NERSC Perlmutter compute node with 512 GB of memory.

Chapter 6: Conclusion

Graph algorithms play a crucial role in analyzing relationships among entities, yet these algorithms often suffer from being memory-bound with poor data locality. The continuous advancement of computing platforms into the exascale era, accompanied by faster chips and supercomputers, exacerbates the growing performance gap between computer memory and processing speed. This gap is particularly pronounced in the context of graph algorithms, where the theoretical and actual performance disparities continue to widen. Furthermore, many well-established graph algorithms still rely on sequential processing, struggling to handle the ever-increasing size of modern graph datasets.

Community discovery, involving the identification of structural patterns or motifs in a graph, holds significance in various domains such as social, biological, and professional network analysis. Unfortunately, existing community discovery approaches, developed over the past decade, face scalability challenges when confronted with massive-scale datasets. The primary objective of this dissertation research is to formulate novel and scalable algorithms for community discovery, capable of leveraging modern multicore/many-core architectures to process extensive graph data efficiently. Two parallel algorithms are presented, each addressing a major type of community discovery: global disjoint community discovery and local overlapping community discovery. Empirical evidence demonstrates the substantial performance gains achieved by our parallel algorithms, surpassing previous approaches. Throughout the design of these parallel algorithms, various challenges—both problem-specific and common in parallel computing—arose. To tackle these challenges, we adopted a three-pronged strategy: i) an algorithmic solution for parallel local community discovery, ii) a computational heuristics + frameworks-based solution for parallel global community discovery, and iii) an accelerator-based solution for a fast hash accumulator to enhance the efficiency of community discovery. Importantly, these solution strategies offer applicability to other graph problems within the parallel computing domain. Rigorous performance profiling of our algorithmic implementations guided the development of a software-hardware co-design strategy, aimed at minimizing the performance gap between theoretical and actual computational kernels. Insights gained from our performance modeling led to the proposal of a generalized architecture design for an accelerator capable of mitigating low throughput issues associated with hash-based graph kernels.

In the future, I would like to utilize the findings of this dissertation research to design novel scalable solutions for dynamically evolving relations among entities in analyzing social/traffic network data. Additionally,

I intend to explore GPU-based architecture for our parallel algorithms design for similar kinds of graph algorithms that we delve into in this dissertation research. I intend to continue research on analyzing how different algorithms and data structures are utilizing the HPC resources, what are their computational behavior, and memory usage patterns. The categorization of different applications/algorithms based on their operation behaviors, for instance, whether a certain application is CPU bound (heavily uses CPU's processing power) or memory bound (irregular memory access pattern or limited performance from memory bandwidth) can help to determine what kind of computer architecture can be used to ensure maximum application performance. My responsibilities as a graduate researcher comprised design, simulation, and empirical analysis on accelerators (additional dedicated computing architecture in CPU for faster processing of certain operations) for graph applications with applications fields in biological research, social network analysis, business predictions, etc. My goal is to further expand the domain of applications and categorizations for high-performance architectural design for exascale computing.

Appendix A: Publications from Dissertation Research

Parts of this dissertation work have been published in conferences and workshops on parallel processing, big data analysis, and emerging high-performance computing architecture, as well as under preparation for submission. The list of publications is mentioned below.

1. Refereed Conference & Workshop Contributions

   [**a1**] **Md Abdul Motaleb Faysal**, Maximilian Bremer, Cy Chan, John Shalf, and Shaikh Arifuzzaman, "Fast Parallel Index Construction for Efficient K-truss-based Local Community Detection in Large Graphs", International Conference on Parallel Processing (ICPP), 2023

   [**a2**] **Md Abdul Motaleb Faysal**, Maximilian Bremer, Shaikh Arifuzzaman, Doru Popovici, John Shalf and Cy Chan, "Fast Community Detection in Graphs with Infomap Method using Accelerated Sparse Accumulation", Accelerators and Hybrid Emerging Systems (AsHES) in IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW), 2023

   [**a3**] **Md Abdul Motaleb Faysal**, Shaikh Arifuzzaman, Cy Chan, Maximilian Bremer, Doru Popovici and John Shalf, "HyPC-Map: A Hybrid Parallel Community Detection Algorithm Using Information-Theoretic Approach", IEEE High Performance Extreme Computing (HPEC) Conference, 2021

   [**a4**] **Md Abdul Motaleb Faysal** and Shaikh Arifuzzaman, "Distributed Community Detection in Large Networks using An Information-Theoretic Approach", In proc. of 2019 IEEE International Conference on BigData (BigData 2019), pages 4773–4782, IEEE, December 2019

2. Other Refereed Conference & Workshop Contributions

   [**b5**] **Md Abdul Motaleb Faysal** and Shaikh Arifuzzaman, "Fast Stochastic Block Partitioning using a Single Commodity Machine", In proc. of 2019 IEEE International Conference on BigData (BigData 2019), pages 3632–3639, IEEE, December 2019

   [**b6**] **Md Abdul Motaleb Faysal** and Shaikh Arifuzzaman, "A Comparative Analysis of Large-scale Network Visualization Tools", In Proceeding of 2018 IEEE International Conference on BigData (BigData 2018), pages 4837–4843, Seattle, WA, USA, IEEE, Dec 2018

[**b7**] Shaikh Arifuzzaman, Naw Safrin Sattar, **Md Abdul Motaleb Faysal**, “Parallel Algorithms for Mining Large-scale Time-varying (Dynamic) Graphs” Nov 2018, In PDSW-DISCS Workshop in SC’18, Dallas, TX, USA, Nov 2018

[**b8**] Naw Safrin Sattar, **Md A. M. Faysal**, Minhaz Zibran, Shaikh Arifuzzaman, and Md Rakibul Islam. Data Mining in-IDE Activities: Why Software Developers Fail, In Proceedings of the 27th International Conference on Software Engineering and Data Engineering (SEDE), USA, 2018.

3. Poster Presentation

[**c9**] **Md Abdul M Faysal**, Shaikh Arifuzzaman, “Scalable Algorithm Design and Performance Analysis for Graph Motifs Discovery”, IEEE International Symposium on Parallel and Distributed Processing (IPDPS) Ph.D. Forum, 2023

[**c10**] Shaikh Arifuzzaman, **Md Abdul M Faysal**, Hasan Arikan, Doru Popovici, Max Bremer, and John Shalf, "Does One Size Fit All?: A Case Study of Performance Portability of Triangle Counting in Graphs on Kokkos", Sustainable Research Pathways (SRP) Program, Summer 2023, Lawrence Berkeley National Laboratory (LBNL)

[**c11**] **Md A M Faysal**, Shaikh Arifuzzaman, Cy Chan, Maximilian Bremer, Doru Popovici, John Shalf, “Fast Hash Accumulation for Information-Theoretic Community Discovery”, 2021 CS Summer Student Program (CSSP), Lawrence Berkeley National Laboratory (LBNL)

[**c12**] **Md Abdul Motaleb Faysal** and Shaikh Arifuzzaman, “Fast Stochastic Block Partitioning”, InnovateUNO-2019, The University of New Orleans

The publication [**a4]** contributes in writing Chapter 2. The publications [**a3]** and [**a2]** constitute Chapter 3 and Chapter 4, respectively. The publication [**a1]** constitutes Chapter 5. Although, the publications [**b5]** and [**b6]** are not included in this dissertation write-up, the research conducted as part of those publications worked as the stepping stones for this doctoral research. Publication [**b6]** helped visualize and analyze networks to gain valuable insights from different network topologies. In publication [**b5]** we investigated another shared-memory parallel community discovery strategy to find similarity and dissimilarity with *Infomap* [114] and *Louvain* [26] community discovery approaches. The publications [**b7]** and [**b8]** are also relevant to this dissertation research.

A.1 Co-authorship

In this doctoral dissertation, I have presented my research conducted under the supervision of Dr. Shaikh Arifuzzaman. The other mentors and collaborators during the span of my doctoral research are Maximilian Bremer, Cy Chan, John Shalf, and Doru Thom Popovici from Lawrence Berkeley National Laboratory. In this dissertation write-up, we have given citations to the work and ideas or techniques that are not products

of my work. Additionally, any contents from other research articles and materials that are presented in this dissertation as part of the discussion of relevant background are given proper credit without claiming ownership. If any concepts or ideas are well-known among the corresponding research community and if tracing down the original intellectual source is impractical, we present the concepts/ideas by paraphrasing while acknowledging the intellectual ownership belongs to their original creators.

While the findings of this dissertation are my direct intellectual contribution, I acknowledge the valuable comments and feedback from my collaborators for shaping the direction and progression of this doctoral research. My doctoral supervisor Dr. Shaikh Arifuzzaman abundantly guided and supervised by brainstorming and making himself available whenever needed during the whole span of my doctoral research and ensuring my timely progress towards the goal. As part of the research collaboration with the Lawrence Berkeley National Lab (LBNL), some portion of this dissertation involved joint work with our collaborators/researchers to make co-authored publications. Our work for Hash-based hardware accelerator design has constituent parts of application software and a library interface with Zsim [115] simulator for architectural simulation. Maximilian Bremer was the lead developer for that interfacing library (*libasa*). In our work of parallel *EquiTruss*, Bremer and I also co-developed the capability/interface so that the parallel *EquiTruss* application can invoke the functionalities/graph kernels of the GAP benchmarking suites [23]. Cy Chan mentored and shared valuable research guidelines during my algorithm design for parallel equitruss [54]. John Shalf and Cy Chan mentored the experimental phase for performance modeling for HipMCL [13] and HyPC-Map [49]. John Shalf provided us access to NERSC supercomputing resources (NERSC Cori and NERSC Perlmutter) used during our experimental evaluations for the works **[a1]**, **[a2]**, and **[a3]**. Doru Popovici contributed to the writing of the manuscripts for our publications **[a2]** and **[a3]** while providing valuable insights for the formulation of the experimental evaluation.

Bibliography

[1] K. Academy, "Information entropy." [Online]. Available: https://www.khanacademy.org/computing/computer-science/informationtheory/moderninfotheory/v/information-entropy

[2] E. Akbas and P. Zhao, "Truss-based community search: A truss-equivalence based indexing approach," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1298–1309, aug 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137640

[3] 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

[4] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss decomposition on gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[5] C. J. Anderson, S. Wasserman, and K. Faust, "Building stochastic blockmodels," *Social Networks*, vol. 14, no. 1, pp. 137 – 161, 1992, special Issue on Blockmodels. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0378873392900172

[6] S. Arifuzzaman and M. Khan, "Fast parallel conversion of edge list to adjacency list for large-scale graphs," in *Proceedings of the 23rd High Performance Computing Symposium (HPC 2015), Alexandria, VA, USA*, April 2015, pp. 17–24.

[7] S. Arifuzzaman, M. Khan, and M. Marathe, "A space-efficient parallel algorithm for counting exact triangles in massive networks," in *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications (HPCC 2015), New York City, USA*, August 2015, pp. 527–534.

[8] ——, "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

[9] ——, "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

[10] S. Arifuzzaman, M. Khan, and M. V. Marathe, "PATRIC: a parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM 2013), San Francisco, CA, USA*, October 2013, pp. 529–538.

[11] S. Arifuzzaman and B. Pandey, "Scalable mining, analysis, and visualization of protein-protein interaction networks," *International Journal of Big Data Intelligence (IJBDI)*, vol. 6, no. 3/4, 01 2019.

[12] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[13] 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

[14] 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

[15] 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.

[16] 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.

[17] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data Mining and Knowledge Discovery*, vol. 29, no. 5, pp. 1406–1433, Sep 2015. [Online]. Available: https://doi.org/10.1007/s10618-015-0422-1

[18] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009. [Online]. Available: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154

[19] V. Batagelj, "Protein-protein interaction network in budding yeast." [Online]. Available: http://vlado.fmf.uni-lj.si/pub/networks/data/bio/Yeast/Yeast.htm

[20] V. Batagelj and A. Mrvar, "Pajek— analysis and visualization of large networks," in *Graph Drawing*, P. Mutzel, M. Jünger, and S. Leipert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 477–478.

[21] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[22] ——, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[23] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017.

[24] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "Sisa:

Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 282–297. [Online]. Available: https://doi.org/10.1145/3466752.3480133

[25] 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

[26] 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

[27] 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

[28] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, p. 575–577, sep 1973. [Online]. Available: https://doi.org/10.1145/362342.362367

[29] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[30] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," 2011.

[31] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 471–480.

[32] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 581–594.

[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] 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

[35] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, no. 3.1, 2008.

[36] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 277–288. [Online]. Available: https://doi.org/10.1145/2463676.2463722

[37] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in

Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 991–1002. [Online]. Available: https://doi.org/10.1145/2588555.2612179

[38] B. DasGupta and D. Desai, "On the complexity of newman's community finding approach for biological and social networks," *Journal of Computer and System Sciences*, vol. 79, no. 1, pp. 50 – 67, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000012000931

[39] T. A. Davis, "Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

[40] I. Derényi, G. Palla, and T. Vicsek, "Clique percolation in random networks," *Phys. Rev. Lett.*, vol. 94, p. 160202, Apr 2005. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.94.160202

[41] L. Donetti and M. A. Muñoz, "Detecting network communities: a new systematic and efficient algorithm," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2004, no. 10, p. P10012, Oct 2004. [Online]. Available: http://dx.doi.org/10.1088/1742-5468/2004/10/P10012

[42] Z. Du, J. Patchett, O. A. Rodriguez, and D. A. Bader, in *The 9th Annual Chapel Implementers and Users Workshop (CHIUW)*.

[43] A. J. Enright, S. van Dongen, and C. A. Ouzounis, "An efficient algorithm for large-scale detection of protein families." *Nucleic acids research*, vol. 30 7, pp. 1575–84, 2002.

[44] M. S. Exchange, "Number of triangles in a graph based on number of edges." [Online]. Available: https://math.stackexchange.com/questions/823481/number-of-triangles-in-a-graph-based-on-number-of-edges

[45] K. Faust and S. Wasserman, "Blockmodels: Interpretation and evaluation," *Social Networks*, vol. 14, no. 1, pp. 5 – 61, 1992, special Issue on Blockmodels. [Online]. Available: http://www.sciencedirect.com/science/article/pii/037887339290013W

[46] 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.

[47] ——, "Distributed community detection in large networks using an information-theoretic approach," in *2019 IEEE International Conference on Big Data (Big Data)*, Dec 2019, pp. 4773–4782.

[48] ——, "Fast stochastic block partitioning using a single commodity machine," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3632–3639.

[49] M. A. M. Faysal, S. Arifuzzaman, C. Chan, M. Bremer, D. Popovici, and J. Shalf, "Hypc-map: A hybrid parallel community detection algorithm using information-theoretic approach," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–8.

[50] ——, "HyPC-Map: A hybrid parallel community detection algorithm using information-theoretic approach," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–8.

[51] M. A. M. Faysal, M. Bremer, S. Arifuzzaman, D. Popovici, J. Shalf, and C. Chan, "Fast community

detection in graphs with infomap method using accelerated sparse accumulation," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2023, pp. 601–610.

[52] M. A. M. Faysal, "Hypc-map: A hybrid parallel community detection algorithm using information-theoretic approach." [Online]. Available: https://github.com/mfaysal101/hymap-sc

[53] 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)*. IEEE, 2019, pp. 4773–4782.

[54] M. A. M. Faysal, M. Bremer, C. Chan, J. Shalf, and S. Arifuzzaman, "Fast parallel index construction for efficient k-truss-based local community detection in large graphs," ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 132–141. [Online]. Available: https://doi.org/10.1145/3605573.3605637

[55] S. Fortunato, "Community detection in graphs," *ArXiv*, vol. abs/0906.0612, 2010.

[56] ——, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75 – 174, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0370157309002841

[57] 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

[58] 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.

[59] 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

[60] P. D. Grünwald, I. J. Myung, and M. A. Pitt, *Advances in Minimum Description Length: Theory and Applications (Neural Information Processing)*. The MIT Press, 2005.

[61] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral, "Modularity from fluctuations in random graphs and complex networks," *Phys. Rev. E*, vol. 70, p. 025101, Aug 2004. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.70.025101

[62] W. M. A. Habib, H. M. O. Mokhtar, and M. E. El-Sharkawi, "Discovering top-weighted k-truss communities in large graphs," *Journal of Big Data*, vol. 9, no. 1, p. 36, Apr 2022. [Online]. Available: https://doi.org/10.1186/s40537-022-00588-1

[63] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo, "Scalable static and dynamic community detection using grappolo," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–6.

[64] Y. Han and R. A. Wagner, "An efficient and fast parallel-connected component algorithm," *J. ACM*, vol. 37, no. 3, p. 626–642, jul 1990. [Online]. Available: https://doi.org/10.1145/79147.214077

[65] J. M. Hofman and C. H. Wiggins, "Bayesian approach to network modularity," *Phys. Rev. Lett.*, vol. 100, p. 258701, Jun 2008. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.100.258701

[66] P. W. Holland, K. B. Laskey, and S. Leinhardt, "Stochastic blockmodels: First steps," *Social Networks*, vol. 5, no. 2, pp. 109 – 137, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0378873383900217

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

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

[69] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1311–1322. [Online]. Available: https://doi.org/10.1145/2588555.2610495

[70] ——, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1311–1322. [Online]. Available: https://doi.org/10.1145/2588555.2610495

[71] ——, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1311–1322. [Online]. Available: https://doi.org/10.1145/2588555.2610495

[72] H. Kabir and K. Madduri, "Parallel k-truss decomposition on multicore systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[73] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith, "Streaming graph challenge: Stochastic block partition," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–12.

[74] A. Karataş and S. Şahin, "Application areas of community detection: A review," in *2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*, Dec 2018, pp. 65–70.

[75] 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

[76] ——, "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

[77] 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

[78] ——, "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

[79] J. M. Kumpula, M. Kivelä, K. Kaski, and J. Saramäki, "Sequential algorithm for fast clique percolation," *Physical Review E*, vol. 78, no. 2, aug 2008. [Online]. Available: https://doi.org/10.1103%2Fphysreve.78.026109

[80] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Phys. Rev. E*, vol. 80, p. 016118, Jul 2009. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.80.016118

[81] ——, "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

[82] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E*, vol. 78, p. 046110, Oct 2008. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.78.046110

[83] ——, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, no. 4, oct 2008. [Online]. Available: https://doi.org/10.1103%2Fphysreve.78.046110

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

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

[86] W. Li and A. Godzik, "Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences," *Bioinformatics*, vol. 22, no. 13, pp. 1658–1659, May 2006.

[87] 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

[88] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new parallel framework for machine learning," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[89] R. D. Luce and A. D. Perry, "A method of matrix analysis of group structure," *Psychometrika*, vol. 14, no. 2, pp. 95–116, Jun 1949. [Online]. Available: https://doi.org/10.1007/BF02289146

[90] A. Madhavan, R. Sindhu, B. Parameswaran, R. K. Sukumaran, and A. Pandey, "Metagenome analysis: a powerful tool for enzyme bioprospecting," *Applied Biochemistry and Biotechnology*, vol. 183, no. 2, pp. 636–651, Oct 2017. [Online]. Available: https://doi.org/10.1007/s12010-017-2568-3

[91] S. Maity and S. Rath, "Extended clique percolation method to detect overlapping community structure," *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 31–37, 2014.

[92] C. P. Massen and J. P. K. Doye, "Identifying communities within energy landscapes," *Phys. Rev. E*, vol. 71, p. 046101, Apr 2005. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.71.

046101

[93] A. Medus, G. Acuña, and C. Dorso, "Detection of community structures in networks via global optimization," *Physica A: Statistical Mechanics and its Applications*, vol. 358, pp. 593–604, Dec 2005. [Online]. Available: https://doi.org/10.1016/j.physa.2005.04.022

[94] M. A. Motaleb Faysal and S. Arifuzzaman, "A comparative analysis of large-scale network visualization tools," in *2018 IEEE International Conference on Big Data (Big Data)*, Dec 2018, pp. 4837–4843.

[95] 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.

[96] M. Newman, "Communities, modules and large-scale structure in networks," *Nature Physics*, vol. 8, pp. 25–31, 12 2011.

[97] 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

[98] ——, "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

[99] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, p. 026113, Feb 2004. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.69.026113

[100] M. E. J. Newman and E. A. Leicht, "Mixture models and exploratory analysis in networks," *Proceedings of the National Academy of Sciences*, vol. 104, no. 23, pp. 9564–9569, 2007. [Online]. Available: https://www.pnas.org/content/104/23/9564

[101] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, pp. 814–818, Jun 2005. [Online]. Available: https://doi.org/10.1038/nature03607

[102] R. Pearce and G. Sanders, "K-truss decomposition for scale-free graphs at scale in distributed memory," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

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

[104] T. P. Peixoto, "Entropy of stochastic blockmodel ensembles," *Phys. Rev. E*, vol. 85, p. 056122, May 2012. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.85.056122

[105] ——, "Parsimonious module inference in large networks," *Phys. Rev. Lett.*, vol. 110, p. 148701, Apr 2013. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.110.148701

[106] ——, "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

[107] M. A. Porter, J.-P. Onnela, and P. J. Mucha, "Communities in networks," *ArXiv*, vol. abs/0902.3788, 2009.

[108] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities

in networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. 9, pp. 2658–2663, 2004. [Online]. Available: https://www.pnas.org/content/101/9/2658

[109] G. Rao, J. Chen, and X. Qian, "Intersectx: An accelerator for graph mining," *ArXiv*, vol. abs/2012.10848, 2020.

[110] G. Rao, J. Chen, J. Yik, and X. Qian, "Sparsecore: Stream isa and processor specialization for sparse computation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 186–199. [Online]. Available: https://doi.org/10.1145/3503222.3507705

[111] J. Reichardt and D. R. White, "Role models for complex networks," *The European Physical Journal B*, vol. 60, no. 2, pp. 217–224, Nov 2007. [Online]. Available: https://doi.org/10.1140/epjb/e2007-00340-y

[112] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 5, pp. 465–471, Sep. 1978. [Online]. Available: http://dx.doi.org/10.1016/0005-1098(78)90005-5

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

[114] ——, "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

[115] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 475–486, jun 2013. [Online]. Available: https://doi.org/10.1145/2508148.2485963

[116] P. Sao, O. Green, C. Jain, and R. Vuduc, "A self-correcting connected components algorithm," in *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, ser. FTXS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 9–16. [Online]. Available: https://doi.org/10.1145/2909428.2909435

[117] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek, "Incremental k-core decomposition: Algorithms and evaluation," *The VLDB Journal*, vol. 25, no. 3, p. 425–447, jun 2016. [Online]. Available: https://doi.org/10.1007/s00778-016-0423-8

[118] 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

[119] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Experimental and Efficient Algorithms*, S. E. Nikoletseas, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 606–609.

[120] 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

[121] H.-W. Shen, X.-Q. Cheng, and J.-F. Guo, "Quantifying and identifying the overlapping community structure in networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2009, no. 07, p. P07042, jul 2009. [Online]. Available: https://doi.org/10.1088%2F1742-5468%2F2009%2F07%2Fp07042

[122] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, p. 287–311, may 2006. [Online]. Available: https://doi.org/10.1177/1094342006064482

[123] Y. Shiloach and U. Vishkin, "An o(log n) parallel connectivity algorithm," *J. Algorithms*, vol. 3, pp. 57–67, 1982.

[124] 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.

[125] G. M. Slota, S. Rajamanickam, and K. Madduri, "Bfs and coloring-based parallel algorithms for strongly connected components and related problems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 550–559.

[126] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–6.

[127] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 939–948. [Online]. Available: https://doi.org/10.1145/1835804.1835923

[128] 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.

[129] ——, "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.

[130] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 12–21.

[131] S. Szabó, "Parallel algorithms for finding cliques in a graph," *Journal of Physics: Conference Series*, vol. 268, p. 012030, jan 2011. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F268%2F1%2F012030

[132] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli, "Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New

York, NY, USA: Association for Computing Machinery, 2013, p. 104–112. [Online]. Available: https://doi.org/10.1145/2487575.2487645

[133] 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.

[134] 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.

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

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

[137] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, p. 812–823, may 2012. [Online]. Available: https://doi.org/10.14778/2311906.2311909

[138] R. Wang, L. Yu, Q. Wang, J. Xin, and L. Zheng, "Productive high-performance k-truss decomposition on gpu using linear algebra," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

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

[140] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo, "K-truss decomposition of large networks on a single consumer-grade machine," in *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2018, pp. 873–880.

[141] P. Wu and L. Pan, "Detecting highly overlapping community structure based on maximal clique networks," in *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*, 2014, pp. 196–199.

[142] Y. Wu, R. Jin, J. Li, and X. Zhang, "Robust local community detection: On free rider effect and its elimination," *Proc. VLDB Endow.*, vol. 8, no. 7, p. 798–809, feb 2015. [Online]. Available: https://doi.org/10.14778/2752939.2752948

[143] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endow.*, vol. 7, no. 14, p. 1821–1832, oct 2014. [Online]. Available: https://doi.org/10.14778/2733085.2733089

[144] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10115-013-0693-z

[145] Y. Yang, S. R. Kuppannagari, A. Srivastava, R. Kannan, and V. K. Prasanna, "Fasthash: Fpga-based high throughput parallel hash table," *High Performance Computing*, vol. 12151, pp. 3 – 22, 2020.

[146] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 895–907.

[147] 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.

[148] ——, "A study of graph partitioning schemes for parallel graph community detection," *Parallel Computing*, vol. 58, no. C, pp. 131–139, Oct. 2016. [Online]. Available: https://doi.org/10.1016/j.parco.2016.05.008

[149] ——, "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

[150] ——, "Effectively unified optimization for large-scale graph community detection," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 475–482.

[151] C. Zhang, M. Bremer, C. Chan, J. Shalf, and X. Guo, "Asa: Accelerating sparse accumulation in column-wise spgemm," *ACM Trans. Archit. Code Optim.*, may 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3543068

[152] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 440–452. [Online]. Available: https://doi.org/10.1145/3352460.3358272

[153] S. Zhang, X. Ning, and X.-S. Zhang, "Identification of functional modules in a PPI network by clique percolation clustering," *Comput Biol Chem*, vol. 30, no. 6, pp. 445–451, Nov. 2006.

[154] 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.

[155] 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

Curriculum Vitae

Please see the next page.

Md Abdul Motaleb Faysal, Ph.D.

https://mfaysal101.github.io faysal.cse101@gmail.com linkedin.com/faysal101

RESEARCH INTEREST

Parallel and Distributed Computing, Machine Architecture, Techniques for HPC Performance Modeling and Simulation, Graph Algorithms, Community Discovery, Scalable Algorithm Design, Big Data Mining

EDUCATION

University of Nevada, Las Vegas (UNLV) *Fall 2023*
Ph.D. in Computer Science
Ph.D. Advisor: Dr. Shaikh Arifuzzaman
Dissertation: Scalable Algorithm Design and Performance Analysis for Graph Motifs Discovery

University of New Orleans (UNO) *Fall 2017 - Summer 2022*
Ph.D. Student in Computer Science
Transferred to UNLV (Fall' 22)

University of New Orleans (UNO) *Spring 2020*
M.S. in Computer Science
Thesis: Accelerating the Information-Theoretic Approach of Community Detection Using Distributed and Hybrid Memory Parallel Schemes

Bangladesh University of Engineering and Technology (BUET) *July 2014*
B.Sc. in Computer Science and Engineering
Thesis: Content-Based Image Retrieval using Relevance Feedback

WORK EXPERIENCE

Graduate Research Assistant, UNLV *Fall 2022 - Fall 2023*
Data-intensive Scalable Computing Group

- Designing parallel algorithm for k-triangle induced local community discovery delivering up to 55× speedup than the sequential approach
- Designing scalable algorithms for memory-bound applications capable of processing billion-size sparse network datasets

Graduate Summer Intern/Affiliate, Berkeley Lab (LBNL) *Summer '20, '21, '22, '23*

- Fast community detection in graphs with Infomap method using Accelerated Sparse Accumulation delivering 5.6× performance
- Improved 5× speedup of a billion-size graph clustering application
- Validation of performance portability of Kokkos framework in CPU/GPU
- Identified performance bottleneck of the SpGEMM approach
- Performance modeling of compute kernels in HPC platforms
- Software-hardware co-design in heterogeneous architecture

Graduate Research Assistant, UNO *Fall 2017 - Spring 2022*
Big Data and Scalable Computing Group

- Distributed-memory parallel community detection using an information-theoretic approach delivering up to 5× speedup
- Comparing network visualization tools and analytics

Software Engineer *August 2014 - July 2017*
ReliSource, Bangladesh
Role:

- Developed and maintained software solutions for health care management.
- Solved critical software issues hindering throughput in production line
- Developed IoT-based software solution for cold chain management.

TEACHING AND MENTORING

Guest Lecture, UNLV *Fall '23, Spring '23, Fall '22*

- Guest lectures on undergraduate course CS302 (Data Structure)
- Guest lecture on graduate course CS789 (Graph Data Mining)
- Conducting quiz and grading programming assignment

Course Instructor, UNO *Spring '22, Fall '20, Spring '20*
Courses taught:

- Introduction to Programming in C++
- Machine Structure and Assembly Language Programming
- Introduction to Computers

Teaching Assistant, UNO

- Course : Machine Structure and Assembly Language Programming

Mentoring at UNLV and UNO

- Mentored a UNLV undergrad CS student during summer internship '23 at Berkeley Lab
- Mentored 2 undergrad students in UNO on graph algorithms research

PUBLICATIONS

- **Md Abdul Motaleb Faysal**, Maximilian Bremer, Cy Chan, John Shalf, and Shaikh Arifuzzaman. 2023. "Fast Parallel Index Construction for Efficient K-truss-based Local Community Detection in Large Graphs." In Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23). Association for Computing Machinery, New York, NY, USA, 132–141.
- **M. A. M. Faysal**, M. Bremer, S. Arifuzzaman, D. Popovici, J. Shalf and C. Chan, "Fast Community Detection in Graphs with Infomap Method using Accelerated Sparse Accumulation," 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), St. Petersburg, FL, USA, 2023, pp. 601-610.
- **M. A. M. Faysal**, S. Arifuzzaman, C. Chan, M. Bremer, D. Popovici and J. Shalf, "HyPC-Map: A Hybrid Parallel Community Detection Algorithm Using Information-Theoretic Approach," 2021 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2021, pp. 1-8.

- **M. A. M. Faysal** and S. Arifuzzaman, "Distributed Community Detection in Large Networks using An Information-Theoretic Approach," 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 4773-4782.
- **M. A. M. Faysal** and S. Arifuzzaman, "Fast Stochastic Block Partitioning using a Single Commodity Machine," 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 3632-3639.
- **M. A. Motaleb Faysal** and S. Arifuzzaman, "A Comparative Analysis of Large-scale Network Visualization Tools," 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 2018, pp. 4837-4843.
- Sattar, Naw Safrin & Arifuzzaman, Shaikh & **Faysal, Md Abdul Motaleb**. (2018). "Parallel Algorithms for Mining Large-scale Time-varying (Dynamic) Graphs." In PDSW-DISCS Workshop in SC'18, Dallas, TX, USA, Nov 2018
- Naw Safrin Sattar, **Md A. M. Faysal**, Minhaz Zibran, Shaikh Arifuzzaman, Md Rakibul Islam, "Data Mining in-IDE Activities: Why Software Developers Fail", ISCA 27th International Conference on Software Engineering and Data Engineering, SEDE 2018.

TECHNICAL SKILLS

Language	C, C++, Java, C#, LaTeX, Assembly, Python, PHP, Prolog
HPC Frameworks	MPI, OpenMP, CUDA, TAU, Metis, ZSim, Hadoop
Other Frameworks	Ant, JavaFX, JUnit, OpenGL, .NET
RDBMS	MySQL, MSSQL, Oracle
Version Control	Git, SVN, TFS
Others	Intel Pin, Vtune, Valgrind, Amazon AWS, Matlab, Weka

RELEVANT GRADUATE COURSES

Applied Combinatorics & Graph Theory, Parallel & Sci Computing, Concurrent Programming, Cloud Computing, Machine Learning, Advanced Machine Learning, Big Data Analytics and Systems, Categorical Data Analysis, Network Penetration, Agile Software Engineering

AWARDS, GRANTS, HONORS

- Received travel award for International Conference on Parallel Processing (ICPP), 2023
- Research poster accepted in the Ph.D. forum in the International Parallel and Distributed Processing Symposium (IPDPS), 2023
- Research proposal accepted and grant awarded for Summer Research Program under Sustainable Horizon Pathways (SRP) program, 2023
- Contributed to research proposal for National Science Foundation (NSF) award (grant#2323533) to work on algorithms for dynamic graph
- Student Volunteer SC'21, and SC'20
- Secretary, Bangladesh Student Association (BSA), UNO, 2021-22

PROJECT HIGHLIGHTS

HyPC-Map: A Hybrid Memory Parallel Infomap

- Uses a random process to discover the communities by using a graph's regularity of information from an information-theoretic formulation.
- Ensures the scalability up to 1280 processing cores while maintaining the accuracy of the sequential approach
- Combines hybrid memory parallelism (MPI + OpenMP) to achieve 25× speedup

Fast Hash Accumulation: Accelerator Aided Community Discovery

- Hash accumulation is a major computation in community detection (Infomap, HipMCL, etc.)
- Accelerator aids faster hash accumulation for insertion and search
- Reduces branch misprediction and number of instructions in software hash
- Reduces performance gap in roofline modeling for hash-based graph kernels

Parallel EquiTruss: A k-truss-based Parallel Index Construction for Local Community Search

- The formulation breaks down the original graph into k-truss-based indexes
- The indexes are connected through k-triangle connectivity to build supergraph with supernodes and superedges.
- Parallel EquiTruss uses Shiloach-Vishkin and Afforest Connected Components (CC) kernels to construct indexes in parallel