On the Optimization of Methods for Establishing Well-Connected Communities

Mohammad Dindoost[1], Oliver Alvarado Rodriguez[1], Bartosz Bryg[1], Minhyuk Park[2], George Chacko[2], Tandy Warnow[2], and David A. Bader[1]

[1] New Jersey Institute of Technology, Newark, NJ, USA,
{md724,oaa9,bb474,bader}@njit.edu
[2] University of Illinois Urbana-Champaign Urbana, IL, USA,
{minhyuk2,chackoge,warnow}@illinois.edu

Abstract. Community detection plays a central role in uncovering meso scale structures in networks. However, existing methods often suffer from disconnected or weakly connected clusters, undermining interpretability and robustness. Well-Connected Clusters (WCC) and Connectivity Modifier (CM) algorithms are post-processing techniques that improve the accuracy of many clustering methods. However, they are computationally prohibitive on massive graphs. In this work, we present optimized parallel implementations of WCC and CM using the HPE Chapel programming language. First, we design fast and efficient parallel algorithms that leverage Chapel's parallel constructs to achieve substantial performance improvements and scalability on modern multicore architectures. Second, we integrate this software into Arkouda/Arachne, an open-source, high-performance framework for large-scale graph analytics. Our implementations uniquely enable well-connected community detection on massive graphs with more than 2 billion edges, providing a practical solution for connectivity-preserving clustering at web scale. For example, our implementations of WCC and CM enable community detection of the over 2-billion edge Open-Alex dataset in minutes using 128 cores, a result infeasible to compute previously.

Keywords: Community Detection, Complex Networks, High-Performance Computing, Parallel Algorithms

1 Introduction

Detecting community structure is a foundational problem with broad impact in science and engineering, with applications ranging from cybersecurity [2] to biology [9] and social network analysis [19]. Numerous approaches have been developed to tackle this challenge, including graph partitioning [11, 15], modularity maximization and its scalable heuristics such as Louvain and Leiden [18, 4, 29], probabilistic models such as the stochastic block model (SBM) [13, 1, 24], and flow- and motif-based techniques [28, 3, 30].

Although density is often used as the main criterion for communities, ensuring they remain well connected is essential for interpretability and robustness [14,

29, 21]. However, many popular techniques can produce disconnected or weakly linked clusters: modularity-based methods can fragment groups in sparse graphs, and SBM inference may cluster weakly linked vertices together [14, 10]. Such artifacts undermine interpretability and robustness. To address this, methods such as Well-Connected Clusters (WCC) [20] and the Connectivity Modifier (CM) [21, 26] explicitly enforce user-defined intra-community connectivity standards.

WCC and CM are post-processing techniques to improve the edge-connectivity of many clustering methods, including SBMs and modularity, and have been shown to produce more accurate community structures compared to traditional approaches [21, 20]. WCC and CM repeatedly refine and split clusters while checking connectivity, leading to significant memory and running-time costs that restrict their use to small- and medium-scale graphs but their use is computationally prohibitive for today's massive networks with billions of edges. High-performance frameworks such as Arkouda/Arachne [27] have demonstrated scalable solutions for other graph analytics [7, 8], but connectivity-preserving clustering has not yet been incorporated. We address this scalability gap by developing optimized parallel implementations of WCC and CM in the Chapel programming language [6], integrated into the open-source Arkouda/Arachne graph analytics framework [27]. Our contributions are twofold:

1. **Novel Parallel Algorithms:** Chapel-based WCC and CM redesigns that reduce redundant work, increase concurrency, and achieve substantial performance improvements.
2. **Integration into Arachne:** Arachne now supports community detection with well-connectedness guarantees for practical large-scale use.

By combining rigorous enforcement of well-connectedness with high-performance computing (HPC)-level scalability, our approach makes well-connected community detection feasible for networks at previously unattainable scale.

Our parallel implementations of WCC and CM are freely-available as open source from GitHub: at https://github.com/Bears-R-Us/arkouda-njit.

2 Methods for Establishing Well-Connected Communities

In this section, we introduce highly-scalable parallel algorithms and their implementations of the WCC and CM methods. The initial WCC algorithm, implemented in C++ with OpenMP parallelism, employs a shared work-queue model: the initial clusters populate a common queue accessible to all OpenMP threads, with each worker pulling the next cluster to process and pushing any new clusters generated back into the queue. For the CM algorithm, first implemented in Python and parallelized using the multiprocessing module, the initial set of clusters is evenly divided among worker processes, each handling its assigned clusters.

Our new approach in this paper extends the original methods by incorporating scalable parallel computing strategies while preserving the guarantees of WCC and CM that all returned clusters are well-connected. Whereas the initial

implementations relied on queue-based task management, our Chapel implementations generalize this to a recursive framework that operates over a large collection of initial clusters, treating them as independent subgraphs, and thereby enabling more flexible and scalable processing.

WCC and CM share a common structure: both begin with a phase – connected component refinement (`CCR`) – that ensures that all input clusters are internally connected, and both apply recursive refinement to evaluate and partition clusters based on global minimum cut criteria. In both algorithms, subgraphs that satisfy a user-defined metric, typically based on cut size relative to graph size, are accepted and stored, while those that fail are recursively subdivided. The key difference lies in how they handle subgraphs that do not meet the well-connectedness criterion. WCC always bisects such subgraphs using their minimum cut, applying further recursion to the resulting parts. In contrast, CM employs another approach by incorporating a user-selected community detection algorithm (referred to as CDA henceforth, where CDA can be Leiden or any other community detection method) as a refinement step. Following the established methodology, when a subgraph fails the well-connectedness criterion, after removing the min-cut to partition the cluster into two parts, CM applies the chosen community detection method to each resulting part to identify community structure. If multiple communities are found within a part, CM recurses on each community separately; otherwise, it processes the entire part as a single unit.

This approach allows CM to identify semantically meaningful substructures by first removing weak connections and then leveraging the user's preferred community detection method to find cohesive groups within the resulting components.

As mentioned previously, both WCC and CM rely on a user-defined criterion function to determine whether a given subgraph is sufficiently well-connected. This function typically takes the form $f(n)$, where n is the number of vertices in the subgraph under consideration. Common choices for f include logarithmic and sublinear functions such as $\log_{10}(n)$, $\log_2(n)$, $\sqrt{n}$, as well as linear functions such as kn, where k is a user-specified constant. The flexibility of this criterion allows users to tailor the sensitivity of the connectivity check to the size and structure of their input graphs.

2.1 Optimized Parallel Implementations

Our parallel implementations achieve performance improvements through several key optimizations tailored for large-scale graph processing in the Chapel programming language [6]. Chapel's high-level parallel constructs, including `forall` loops for parallel iteration, built-in parallel reductions, and domain-based parallelism, provide natural optimization opportunities for graph algorithms. The language's ability to handle large-scale data structures and its built-in support for parallel collections with thread-safe operations eliminate many low-level synchronization concerns while maintaining high performance. Inside the open-source Arkouda/Arachne framework graphs are stored in double-index (DI) format [27],

which extends compressed sparse row (CSR) representation with an edge-to-source array that allows $O(1)$ edge access. This optimization is critical for our algorithms, which require frequent edge lookups. We employ a highly-parallel connected components algorithm to process clusters after the CCR preprocessing step. Each input cluster is evaluated in parallel, and those passing the size threshold s_{pre} are distributed across processing queues for concurrent evaluation. This approach leverages Chapel's parallel constructs for efficient work distribution.

To minimize overhead, we prioritize parallel cluster-level processing over parallelism within individual minimum-cut computations. This design choice recognizes that most cluster subgraphs are relatively small, making fine-grained parallelization of min-cut algorithms counterproductive. Instead, we employ a sequential variant of the VieCut [12] algorithm for individual subgraphs, which suffices due to their modest size, while parallelizing across the large number of clusters that require processing.

Finally, our recursive design eliminates the queue management overhead present in the original implementations, reducing memory pressure, and improving cache locality. The recursive approach naturally maps to Chapel's parallel execution model, enabling automatic work distribution and efficient memory access patterns.

2.2 Connected Component Refinement

The first step of both WCC and CM is to ensure that the input clusters do not contain disconnected components. Algorithm 1 outlines the connected component refinement (`CCR`) routine, which addresses this issue.

Each cluster $c \in C$ is then converted into an induced subgraph $G_c = (V_c, E_c)$. If G_c contains any edges, the algorithm computes its connected components via `GetConnectedComponents`. Each connected component cc with a size greater than the threshold s_{pre} is added to the local output queue Q for further processing. This procedure ensures that only clusters composed of only one connected component are passed to the main WCC or CM routines.

2.3 Well-Connected Clusters

Algorithms 2 and 3 define the WCC procedure, which recursively evaluates whether clusters are internally well-connected using global minimum cuts and a user-defined criterion. The top-level routine (Algorithm 2) takes a graph $G = (V, E)$, a set of clusters C, and size thresholds s_{pre} and s_{post}. After applying the connected component refinement (`CCR`) to ensure that input clusters are connected, each resulting component q_i is converted into a subgraph G_{q_i} and passed to the recursive `isWCC` procedure.

The `isWCC` check (Algorithm 3) computes a global minimum cut and compares it to the threshold returned by `ComputeCriterion` (e.g., $\log_{10}(n)$ for $n = |V_c|$). A larger cut value indicates stronger connectivity, as more edges must

Algorithm 1 Connected Component Refinement

```
1: procedure CCR(G = (V, E), C, spre)
2:     Q ← ∅
3:     for all c ∈ C do
4:         V_c ← c
5:         E_c ← {(u, v) ∈ E : u ∈ V_c ∧ v ∈ V_c}                ▷ Induced edges
6:         G_c ← (V_c, E_c)
7:         if |E_c| > 0 then
8:             CC ← GetConnectedComponents(G_c)
9:             for all cc ∈ CC do
10:                if |cc| > spre then
11:                    Q ← Q ∪ {cc}
12:                end if
13:            end for
14:        end if
15:    end for
16:    return Q
17: end procedure
```

Algorithm 2 Well-Connected Clusters

```
1: procedure WCC(G = (V, E), C, spre, s_post)
2:     Q ← CCR(G, C, spre)
3:     W ← ∅                                  ▷ accepted well-connected clusters
4:     for all q_i ∈ Q do
5:         V_{q_i} ← q_i
6:         E_{q_i} ← {(u, v) ∈ E : u ∈ V_{q_i} ∧ v ∈ V_{q_i}}      ▷ induced edges
7:         G_{q_i} ← (V_{q_i}, E_{q_i})
8:         if IsWCC(G_{q_i}, s_post) then
9:             W ← W ∪ {V_{q_i}}
10:        end if
11:    end for
12:    return W
13: end procedure
```

be removed to disconnect the subgraph. If the cut size exceeds the threshold, the cluster is accepted. Otherwise, the subgraph is partitioned along the cut into G_{c_1} and G_{c_2}, which are recursively processed if larger than s_{post}. This hierarchical bisection continues until all accepted clusters satisfy the well-connectedness criterion. Unlike other clustering methods, no merging is performed, only recursive refinement of the input set C.

2.4 Connectivity Modifier

Algorithms 4 and 5 define the CM procedure, which refines clusters using global min-cut and user-selected community detection algorithm (CDA), such as Leiden. As in WCC, the `CM` algorithm first applies connected component refinement

Algorithm 3 The Well-Connectedness Check

```
1: procedure isWCC(G_c = (V_c, E_c), s_post)
2:     if |E_c| ≥ 1 then
3:         cut ← GetMinCut(G_c)
4:         criterion ← ComputeCriterion(G_c)
5:         if cut > criterion then
6:             Save V_c with a unique cluster identifier
7:         else
8:             (V_c1, V_c2) ← MinCutPartition(V_c, cut)
9:             E_c1 ← {(u, v) ∈ E_c : u, v ∈ V_c1}
10:            E_c2 ← {(u, v) ∈ E_c : u, v ∈ V_c2}
11:            if |V_c1| > s_post then
12:                isWCC(G_c1 = (V_c1, E_c1), s_post)
13:            end if
14:            if |V_c2| > s_post then
15:                isWCC(G_c2 = (V_c2, E_c2), s_post)
16:            end if
17:        end if
18:    end if
19: end procedure
```

Algorithm 4 Connectivity Modifier

```
1: procedure CM(G = (V, E), C, s_pre, s_post, CDA)
2:     Q ← CCR(G, C, s_pre)
3:     for all q_i ∈ Q do
4:         V_qi ← q_i
5:         E_qi ← {(u, v) ∈ E : u ∈ V_qi ∧ v ∈ V_qi}
6:         CMC(G_qi = (V_qi, E_qi), s_post, CDA)
7:     end for
8: end procedure
```

(CCR) to produce connected subgraphs G_{q_i}, which are then passed to the recursive CMC routine.

The CMC check computes the global minimum cut and compares it against a user-defined threshold. If the cut exceeds the criterion, the subgraph is accepted. Otherwise, it is partitioned along the cut into G_{c_1} and G_{c_2}. Then each part is processed with GetCommunities, based on user-selected Community Detection Algorithm(CDA). If multiple communities are found, they are recursively refined (subject to the s_{post} threshold); if not, the entire part is processed as a single unit. By applying community detection only after bottleneck removal, CM identifies cohesive substructures. As with WCC, refinement proceeds strictly through recursive subdivision, never merging clusters.

Algorithm 5 The Connectivity Modifier Check

procedure CMC($G_c = (V_c, E_c)$, s_{post}, CDA)
  if $|E_c| \geq 1$ **then**
    $cut \leftarrow$ GetMinCut(G_c)
    $criterion \leftarrow$ ComputeCriterion(G_c)
    if $cut > criterion$ **then**
      Save V_c with a unique cluster identifier
    else
      $(V_{c_1}, V_{c_2}) \leftarrow$ MinCutPartition(V_c, cut)
      $E_{c_1} \leftarrow \{(u, v) \in E_c : u, v \in V_{c_1}\}$
      $E_{c_2} \leftarrow \{(u, v) \in E_c : u, v \in V_{c_2}\}$
                                ▷ Process for each min-cut part
      for all $G_{part} \in \{G_{c_1}, G_{c_2}\}$ where $G_{part} = (V_{part}, E_{part})$ **do**
        if $|V_{part}| > s_{post}$ **then**
          $C \leftarrow$ GetCommunities(G_{part}, CDA)
          if $|C| > 1$ **then** ▷ Multiple communities found
            for all $V_i \in C$ **do**
              $E_i \leftarrow \{(u, v) \in E_{part} : u, v \in V_i\}$
              if $|V_i| > s_{post}$ **then**
                CMC($G_i = (V_i, E_i), s_{post}$)
              end if
            end for
          end if
        end if
      end for
    end if
  end if
end procedure

3 Experimental Evaluation

To evaluate the performance and efficacy of our optimized parallel implementations of the WCC and CM algorithms, we performed a series of experiments on real-world networks. The types of experiments include: (1) performance benchmarks to assess runtime and speedup compared to the original implementations; and (2) scalability tests, encompassing strong scaling (fixed graph size, varying processors).

All experiments were performed on dual 2.0GHz AMD EPYC 7713 processors (128 cores total) with 512GB RAM. The parallel implementations were executed using Arachne [27], while baseline comparisons utilized the original implementations. For all experiments, we used $\log_{10}(n)$ as a user-defined criterion function, and running times were measured in wall-clock seconds. The datasets consisted of real-world networks selected for diversity in size and structure, as detailed in Table 1. The graphs were pre-processed, and the isolated vertices were removed. We set pre- and post-size thresholds to $s_{pre} = s_{post} = 1$. We selected Leiden-CPM as the community detection algorithm (CDA) used recursively within the CM pipeline to identify communities after min-cut partitioning.

Table 1. Real-World Networks Used in Experiments

Small to Medium Networks			Large Networks		
Network	vertices	Edges	Network	vertices	Edges
Bitcoin [25]	6,336,770	16,057,711	Open-Alex [5]	256,997,006	2,148,871,058
Livejournal [25]	4,847,571	68,993,773	Open-Citations-v2 [17]	121,052,490	1,962,840,983
Cit-Patents [23]	3,774,768	16,518,947	Open-Citations [22]	75,025,194	1,363,605,603
Orkut [23]	3,072,441	117,185,083	CEN [23]	13,989,436	92,051,051
Hyves [25]	1,402,673	2,777,419	Wikipedia-Links [16]	13,593,032	437,217,424

3.1 Performance Benchmarks

We first compared the running time of the new WCC-Chapel and CM-Chapel implementations against the earlier WCC and CM baselines to demonstrate the efficiency gains from the novel Chapel parallelization. Experiments were conducted on networks with input clusters generated using the Leiden algorithm with Constant Potts Model using resolution parameter values of 0.001 and 0.01, reflecting two distinct modes of cluster initialization to assess performance across varying granularity levels.

Table 2. Runtime Comparison (in seconds) on Small to Medium Datasets

Leiden CPM 0.001				
Dataset	WCC-baseline	WCC-Chapel	CM-baseline	CM-Chapel
Bitcoin	805.9	111.5	65.8	78.8
Livejournal	916.2	87.6	58.4	56.1
Cit-Patents	372.3	59.3	43.6	37.8
Orkut	314.0	82.2	88.8	67.5
Hyves	74.8	78.1	18.7	16.8
Leiden CPM 0.01				
Dataset	WCC-baseline	WCC-Chapel	CM-baseline	CM-Chapel
Bitcoin	277.1	128.5	116.8	100.9
Livejournal	271.1	120.7	84.0	81.1
Cit-Patents	223.1	95.4	66.7	62.8
Orkut	211.9	95.5	74.4	56.3
Hyves	28.3	84.1	26.0	21.4

As shown in Tables 2 and 3, our Chapel-based implementations demonstrate strong performance advantages on both small-to-medium- and large-scale networks. In small-to-medium-sized networks, the new implementations show consistent and substantial improvements for WCC, achieving substantial speedups, with some networks reaching over 10x improvement. The CM results show more varied performance, with Chapel-based achieving competitive or better running time in most cases, though occasional instances favor the baseline implementation, particularly on Bitcoin where the baseline outperforms the CM-Chapel version.

Table 3. Runtime Comparison (in seconds) on Large Networks. All the dashes mean that the analysis failed due to OOM or Segmentation Faults.

Leiden CPM 0.001				
Dataset	WCC-baseline	WCC-Chapel	CM-baseline	CM-Chapel
Open-Alex	-	1306.4	-	1317.3
Open-Citations-v2	-	1343.9	-	1346.2
Open-Citations	-	1230.9	-	971.6
CEN	4330.0	307.4	152.3	196.2
Wikipedia-Links	-	-	238.7	298.9
Leiden CPM 0.01				
Dataset	WCC-baseline	WCC-Chapel	CM-baseline	CM-Chapel
Open-Alex	-	2144.7	-	2133.73
Open-Citations-v2	-	1891.6	-	1850.4
Open-Citations	-	1494.7	-	1518.3
CEN	1369.3	240.9	206.9	230.0
Wikipedia-Links	-	377.2	304.1	372.8

The performance advantage becomes even more pronounced on large-scale networks. On massive networks such as Open-Alex and Open-Citations with more than a billion edges, the baseline implementations consistently fail due to memory limitations, whereas our Chapel-based implementations complete successfully. When both implementations can handle the dataset, as with CEN, the improvement is dramatic: WCC-Chapel achieves up to 14x speedup, while baseline CM outperforms CM-Chapel in this particular case.

These results demonstrate that our optimizations provide substantial performance improvements across the full spectrum of graph sizes, with the added critical advantage of robust scalability to networks where existing methods fail entirely. The consistent WCC performance gains and competitive CM results, combined with the ability to process billion-edge networks, establish our Chapel-based implementations as both more efficient and more capable than existing approaches. The remaining performance variations on smaller CM instances likely reflect the overhead of external C library integration for community detection (Leiden [29]) and min-cut computation (VieCut [12]), direct Chapel implementations of these components would eliminate such foreign function call costs and provide even greater performance consistency.

3.2 Scalability Analysis

We evaluate the scalability of our implementations using strong scaling experiments on two representative networks: Bitcoin and Open-Alex. As shown in Fig. 1, both algorithms demonstrate clear scaling benefits up to moderate core counts before encountering performance degradation due to parallel overheads. On the Bitcoin dataset, WCC-Chapel exhibits strong scaling from single-core up to 32 cores, achieving approximately 3x speedup at the optimal point. Beyond 32 cores, performance degrades as parallelization overheads begin to dominate

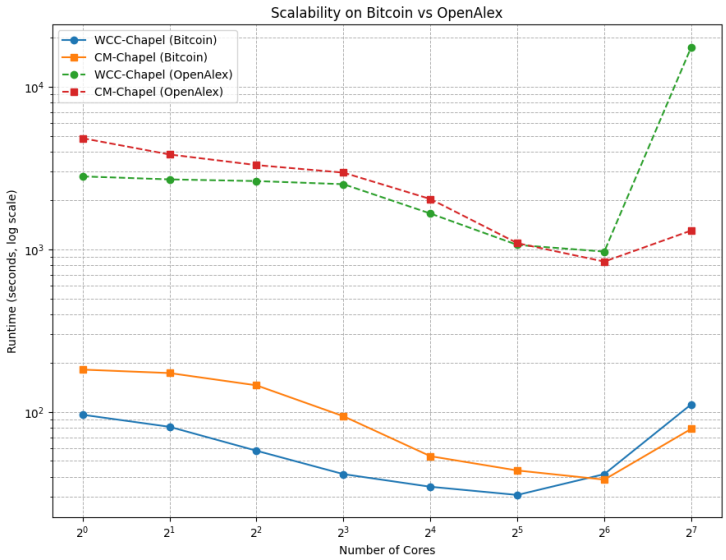

Fig. 1. Strong scaling of WCC-Chapel and CM-Chapel on the Bitcoin and OpenAlex networks.

the diminishing per-core workload. CM-Chapel shows similar scaling behavior but sustains improvement to 64 cores, achieving nearly 5x speedup before experiencing degradation at higher core counts. This difference suggests that the algorithmic structure of CM provides better load distribution characteristics on this particular network.

The larger Open-Alex dataset reveals more pronounced scaling differences between the algorithms. Both implementations benefit from the increased computation to communication ratio inherent in larger graphs, sustaining parallel efficiency to higher core counts. CM-Chapel demonstrates particularly strong scaling, achieving nearly 6x speedup at 64 cores and maintaining reasonable performance characteristics even at higher parallelization levels. However, WCC-Chapel exhibits a critical failure mode at maximum core count on the large network (OpenAlex), with runtime becoming dramatically worse than single-core performance. This catastrophic degradation suggests fundamental load-balancing issues or resource contention that emerge only under extreme parallelization on massive graphs. In contrast, CM-Chapel shows more graceful degradation, maintaining stability across the full range of tested core counts. These results highlight several key insights: optimal performance occurs at moderate core counts (typically 32-64), larger networks generally support higher degrees of parallelization, and algorithmic differences between WCC and CM lead to distinct scalability profiles. The severe collapse of WCC performance at high core counts on large graphs indicates that recursive refinement strategies require careful consideration of load balancing to avoid pathological behavior at scale.

4 Conclusion

In this work, we presented novel Chapel-based parallel implementations of community detection algorithms for WCC and CM, designed to operate efficiently on massive real-world networks. Our framework demonstrates that it is possible to combine strict connectivity guarantees with scalable performance by leveraging recursive refinement and Chapel's parallel tasking model. Looking ahead, two directions are promising. First, extending this framework to distributed-memory environments will allow scaling beyond shared-memory, enabling analysis of truly web-scale networks. Second, continued optimization of Chapel-native kernels, including direct implementations of Leiden and VieCut, will reduce runtime overhead and further improve scalability. Our new parallel implementations of WCC and CM are freely-available in the open-source Arachne framework on GitHub at https://github.com/Bears-R-Us/arkouda-njit.

5 Acknowledgments

This research was funded in part by NSF grant numbers CCF-2109988, OAC-2402560, and CCF-2453324 (Bader) and OAC-2402559 (Warnow and Chacko).

References

1. Abbe, E.: Community detection and stochastic block models: recent developments. Journal of Machine Learning Research **18**(177), 1–86 (2018)
2. Akoglu, L., Tong, H., Koutra, D.: Graph based anomaly detection and description: a survey. Data mining and knowledge discovery **29**, 626–688 (2015)
3. Benson, A.R., Gleich, D.F., Leskovec, J.: Higher-order organization of complex networks. Science **353**(6295), 163–166 (2016)
4. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment **2008**(10), P10,008 (2008)
5. Caetano Machado Lopes, L., Chacko, G.: A citation graph from OpenAlex (Works) (2024). DOI 10.13012/B2IDB-7362697_V1
6. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. The International Journal of High Performance Computing Applications **21**(3), 291–312 (2007)
7. Dindoost, M., Rodriguez, O.A., Bagchi, S., Pauliuchenka, P., Du, Z., Bader, D.A.: VF2-PS: Parallel and Scalable Subgraph Monomorphism in Arachne. In: 2024 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–9. IEEE (2024)
8. Dindoost, M., Rodriguez, O.A., Bryg, B., Koutis, I., Bader, D.A.: HiPerMotif: Novel Parallel Subgraph Isomorphism in Large-Scale Property Graphs. In: 2025 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2025)
9. Fortunato, S.: Community detection in graphs. Physics reports **486**(3-5), 75–174 (2010)

10. Fortunato, S., Newman, M.E.: 20 years of network community detection. Nature Physics **18**(8), 848–850 (2022)
11. Hagen, L., Kahng, A.B.: New spectral methods for ratio cut partitioning and clustering. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **11**(9), 1074–1085 (1992)
12. Henzinger, M., Noe, A., Schulz, C.: Shared-memory Exact Minimum Cuts. In: Proceedings of the 33rd International Parallel and Distributed Processing Symposium (IPDPS) (2019)
13. Holland, P.W., Laskey, K.B., Leinhardt, S.: Stochastic blockmodels: First steps. Social networks **5**(2), 109–137 (1983)
14. Kannan, R., Vempala, S., Vetta, A.: On clusterings: Good, bad and spectral. Journal of the ACM (JACM) **51**(3), 497–515 (2004)
15. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing **20**(1), 359–392 (1998)
16. Kunegis, J.: Konect: the Koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1343–1350 (2013)
17. Mohasel Arjomandi, H., Korobskiy, D., Chacko, G.: Parsed Open Citations and PubMed Data (2024). DOI 10.13012/B2IDB-5216575_V1
18. Newman, M.E.: Modularity and community structure in networks. Proceedings of the National Academy of Sciences **103**(23), 8577–8582 (2006)
19. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E **69**(2), 026,113 (2004)
20. Park, M., Feng, D.W., Digra, S., Vu-Le, T.A., Chacko, G., Warnow, T.: Improved community detection using stochastic block models. In: International Conference on Complex Networks and Their Applications, pp. 103–114. Springer (2024)
21. Park, M., Tabatabaee, Y., Ramavarapu, V., Liu, B., Pailodi, V.K., Ramachandran, R., Korobskiy, D., Ayres, F., Chacko, G., Warnow, T.: Well-connectedness and community detection. PLOS Complex Systems **1**(3), 1–25 (2024). DOI 10.1371/journal.pcsy.0000009
22. Park, M., Tabatabaee, Y., Warnow, T., Chacko, G.: Data for well-connected communities in real networks. (2023). DOI 10.13012/B2IDB-0908742_V1
23. Park, M., Tabatabaee, Y., Warnow, T., Chacko, G.: Data for well-connectedness and community detection (2024). DOI 10.13012/B2IDB-6271968_V1
24. Peixoto, T.P.: Bayesian stochastic blockmodeling. Advances in network clustering and blockmodeling pp. 289–332 (2019)
25. Peixoto, T.P.: The Netzschleuder network catalogue and repository (2020). DOI 10.5281/zenodo.7839981. Accessed: August 17, 2025
26. Ramavarapu, V., Ayres, F.J., Park, M., Pailodi, V.K., Lamy, J.A.C., Warnow, T., Chacko, G.: CM++-A Meta-method for Well-Connected Community Detection. Journal of Open Source Software **9**(93), 6073 (2024)
27. Rodriguez, O.A., Du, Z., Patchett, J., Li, F., Bader, D.A.: Arachne: An Arkouda package for large-scale graph analytics. In: 2022 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2022)
28. Rosvall, M., Bergstrom, C.T.: Maps of random walks on complex networks reveal community structure. Proceedings of the National Academy of Sciences **105**(4), 1118–1123 (2008)
29. Traag, V.A., Waltman, L., Van Eck, N.J.: From Louvain to Leiden: guaranteeing well-connected communities. Scientific reports **9**(1), 1–12 (2019)
30. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. In: Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, pp. 1–8 (2012)