Applied Network Science

https://doi.org/10.1007/s41109-026-00818-y

Article in Press

Well-connected community detection at extreme scale: shared- and distributed-memory parallel algorithms

Received: 27 February 2026

Accepted: 27 July 2026

Published online: 28 August 2026

Cite this article as: Dindoost M., Alvarado Rodriguez O., Uddin A. *et al.* Well-connected community detection at extreme scale: shared- and distributed-memory parallel algorithms. *Appl Netw Sci* (2026). https://doi.org/10.1007/s41109-026-00818-y

Mohammed Dindoost, Oliver Alvarado Rodriguez, Asif Uddin, Bartosz Bryg, Haotian Yi, Minhyuk Park, George Chacko, Tandy Warnow & David A. Bader

Well-Connected Community Detection at Extreme Scale: Shared- and Distributed-Memory Parallel Algorithms

Mohammed Dindoost[1*], Oliver Alvarado Rodriguez[1], Asif Uddin[1], Bartosz Bryg[1], Haotian Yi[2], Minhyuk Park[2], George Chacko[2], Tandy Warnow[2], David A. Bader[1]

[1*]Department of Data Science in the Ying Wu College of Computing, New Jersey Institute of Technology, 323 Dr Martin Luther King Jr Blvd, Newark, 07102, New Jersey, United States.
[2]Siebel School of Computing and Data Science, University of Illinois Urbana-Champaign, 601 E. John Street, Champaign, 61820, Illinois, United States.

*Corresponding author(s). E-mail(s): md724@njit.edu;
Contributing authors: oaa9@njit.edu; au249@njit.edu; bb474@njit.edu; yi54@illinois.edu; minhyuk2@illinois.edu; chackoge@illinois.edu; warnow@illinois.edu; bader@njit.edu;

Abstract

Community detection algorithms such as Louvain frequently produce clusters that are internally disconnected or poorly connected, limiting their utility in downstream network analysis. The Well-Connected Clusters (WCC) and Connectivity Modifier (CM) algorithms address this by post-processing any input clustering to enforce a user-defined edge connectivity criterion through recursive minimum cut bisection. While prior work demonstrated shared-memory parallel implementations of WCC and CM in Chapel on graphs with up to two billion edges, scalability remains constrained by single-node memory capacity and by the separate subgraph-construction preprocessing pass used in the original pipeline. This paper presents distributed-memory parallel implementations of WCC and CM in both C++ with MPI and Chapel with multi-locale execution. The central contribution is an architectural redesign that integrates subgraph generation into the Leiden clustering step, eliminating the separate WCC/CM subgraph preprocessing pass. Each compute node receives only its assigned subgraph files and

executes a fully independent pipeline without ever loading the full graph. Connected component computation is parallelized within each node and distributed across nodes via round-robin assignment, and memory-mapped I/O accelerates file loading throughout. Experiments on ten real-world networks spanning up to 2.1 billion edges show that the C++ distributed implementation achieves up to 65× speedup over the original baseline on graphs where both complete successfully. The Chapel distributed implementation is integrated into Arachne, an open-source graph analytics framework built on the Arkouda platform, available at https://github.com/Bears-R-Us/arkouda-njit. It achieves broader graph coverage than the C++ distributed implementations, successfully processing billion-edge configurations including Open-Alex and Open-Citations on which all C++ distributed implementations fail, while Wikipedia-Links remains unsuccessful for both implementations. On successful configurations, Chapel distributed delivers speedups up to 19.7× at CPM 0.001 and up to 55.8× at CPM 0.01 over the Chapel shared-memory reference, with one reported slowdown on Livejournal WCC at CPM 0.001. Failures on a subset of large graphs are associated with memory leaks and data races in VieCut.

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

1 Introduction

Graph-based community detection is a foundational problem with broad impact across science and engineering, with applications spanning cybersecurity [1], computational biology [2], and social network analysis [3]. A wide range of methodologies have been proposed to address this challenge, including spectral graph partitioning [4, 5], modularity maximization and its efficient heuristics such as Louvain and Leiden [6–8], probabilistic generative models such as the stochastic block model (SBM) [9–11], and flow- and motif-based approaches [12–14].

Despite the diversity of available methods, a shared limitation persists: many widely-used algorithms produce clusters that are internally disconnected or only weakly linked, undermining both the interpretability and the robustness of the resulting communities [8, 15, 16]. Modularity-based approaches such as Louvain do not guarantee internal connectivity of output communities, and can produce clusters with low edge connectivity, while SBM inference can group vertices that share little internal cohesion [15, 17]. This is not merely a computational artifact: disconnected or weakly-connected communities can compromise the reliability of downstream analyses that depend on community structure, such as functional module identification, influence analysis, and research field classification. To enforce principled intra-community connectivity, post-processing frameworks such as Well-Connected Clusters (WCC) [18, 19] and the Connectivity Modifier (CM) [16, 20] have been developed. Both methods operate by recursively evaluating and refining clusters using global minimum cut criteria against a user-defined connectivity threshold, and have been shown to improve

the accuracy of community structures produced by both modularity- and SBM-based methods [16, 18, 19].

The recursive refinement strategies employed by WCC and CM, however, carry substantial computational cost, historically confining their applicability to small- and medium-scale networks. Yet the networks where these analyses matter most, large-scale citation graphs, web-scale social networks, and biological interaction networks, routinely contain billions of edges and grow larger every year. For such networks, high-performance computing is not merely an implementation convenience but a scientific prerequisite: without scalable implementations, well-connected community detection cannot be performed on the datasets where it is most scientifically relevant, regardless of algorithmic advances. The Open-Alex citation network [21], for example, covers over 256 million publications with 2.1 billion citation edges; running WCC or CM on such a graph to obtain scientifically meaningful, well-connected research communities was previously infeasible. In prior work [22], we addressed this limitation by developing optimized shared-memory parallel implementations of WCC and CM in the Chapel programming language [23], integrated into the open-source Arachne graph analytics framework [24]. Those implementations demonstrated that well-connected community detection is feasible on graphs with over two billion edges, previously unattainable, processing the Open-Alex citation network in under 35 minutes on a 128-core shared-memory machine. Nonetheless, shared-memory architectures impose a hard ceiling: as graph sizes continue to grow toward the tens of billions of edges and beyond, memory capacity on a single node becomes the binding constraint, not compute.

Distributed-memory computing offers a principled path beyond this ceiling, but distributing the WCC and CM workflows introduces non-trivial challenges. Both algorithms are inherently recursive, with data-dependent branching that creates irregular workloads poorly suited to naive task decomposition. The per-cluster minimum cut computations are sequential bottlenecks that must be carefully assigned across nodes, and inter-node communication costs can erode the benefits of parallelism if not managed deliberately. In this work, we address these challenges through two independent contributions: a suite of distributed C++ implementations of WCC and CM built on MPI that integrate subgraph file generation into the Leiden clustering step and distribute the resulting subgraph files across nodes for fully independent per-node minimum cut execution; and distributed Chapel implementations of both WCC and CM that leverage Chapel's native locale model for multi-node execution within the Arachne framework. Together, these contributions extend the reach of well-connected community detection from shared-memory HPC machines to true multi-node cluster environments, enabling network scientists to apply connectivity-preserving community refinement to datasets at previously inaccessible scales. The Chapel distributed implementation achieves broader coverage than the C++ distributed implementations, completing several Open-Alex and Open-Citations configurations on which the C++ distributed implementations fail, as discussed in detail in Section 5.8, while the C++ distributed variants achieve up to 65× speedup over the original baseline on graphs where both complete successfully. The primary contributions of this paper are as follows:

1. **Shared-memory Chapel implementations of WCC and CM (prior work [22]):** Optimized parallel Chapel redesigns integrated into Arachne, enabling well-connected community detection on graphs with over two billion edges on a single shared-memory node.
2. **Distributed C++ implementations of WCC and CM:** A family of distributed implementations, distributed recursive and distributed queue, built on MPI, incorporating memory-mapped I/O, parallel and distributed connected component computation, and subgraph file generation integrated into the Leiden clustering step. By eliminating the subgraph construction preprocessing from the WCC and CM pipeline entirely, these implementations achieve up to 65× speedup over the original baseline on graphs where both complete successfully, while maintaining aggregate output consistency.
3. **Distributed Chapel implementations of WCC and CM:** Multi-locale extensions of the Arachne-integrated Chapel implementations using Chapel's native locale model, enabling distributed-memory execution of both WCC and CM. The Chapel distributed implementation successfully processes more large-graph configurations than the C++ distributed implementation, including several billion-edge configurations; the differential behavior relative to the C++ distributed implementation is characterized empirically and discussed in Section 5.8.
4. **Load-balanced cluster distribution:** A sorted round-robin assignment strategy (`rr_npm`) that accounts for the heavy-tailed cluster size distributions produced by Leiden CPM partitioning, improving runtime on most successful configurations with gains up to 2.7× over plain round-robin assignment.

This paper extends [22] in four substantive ways. First, the conference paper presented only shared-memory Chapel implementations; this paper introduces entirely new distributed-memory implementations in both C++ with MPI and Chapel with multi-locale execution, constituting the primary technical contribution. Second, the architectural redesign, which integrates subgraph generation into the Leiden clustering step and eliminates the separate WCC/CM subgraph construction preprocessing pass, was developed for this journal version and does not appear in the conference paper. Third, the experimental evaluation is substantially expanded: the conference paper reported shared-memory performance results, while this paper adds distributed C++ and Chapel experiments over a ten-network benchmark suite, reports both successful configurations and observed failures, and updates the shared-memory results with revised implementations averaged over at least three runs. Fourth, a load-balanced cluster distribution strategy (`rr_npm`) is introduced and evaluated, improving runtime on most successful configurations over plain round-robin assignment and addressing the load imbalance inherent in heavy-tailed cluster size distributions.

The remainder of this paper is organized as follows. Section 2 reviews related work on community detection, parallel graph algorithms, and HPC graph frameworks. Section 3 presents the theoretical foundations of WCC and CM. Section 4 summarizes the shared-memory Chapel implementations. Section 5 describes the distributed architecture, engineering optimizations, and language-specific implementations in both C++ and Chapel. Section 6 reports experimental results and scalability analysis. Section 7 concludes with directions for future work.

2 Related Work

2.1 Community Detection

Community detection has been studied extensively across a wide range of methodological paradigms. Modularity-based methods, beginning with Newman and Girvan [3] and extended by Newman [6], define communities as subgraphs with higher internal than external edge density. The Louvain algorithm [7] made modularity maximization tractable at scale through a greedy agglomerative heuristic, but its output communities are not guaranteed to be internally connected. The Leiden algorithm [8] subsequently addressed this by refining the community movement phase to guarantee well-connectedness within a modularity or Constant Potts Model (CPM) objective. Probabilistic approaches such as the stochastic block model [9–11] offer a generative framework for community inference, but SBM-based methods can similarly produce poorly-connected groupings when the model does not explicitly enforce internal cohesion [15, 17]. Flow-based methods, including the map equation approach of Rosvall and Bergstrom [12], and motif-based techniques [13, 14] provide complementary perspectives, but none of these methods natively guarantee a user-specified minimum connectivity criterion. WCC [18, 19] and CM [16, 20] address this gap as post-processing refinement frameworks applicable to the output of any community detection method. These methods have been shown to improve community accuracy in large-scale bibliometric networks, biological interaction networks, and social networks, where poorly-connected clusters can lead to incorrect conclusions in downstream analyses such as research field delineation and functional module identification [16, 18, 19]. Enabling WCC and CM at scale is therefore directly motivated by the needs of network science practitioners working with modern billion-edge datasets.

2.2 Parallel and Distributed Community Detection

Scaling community detection to large graphs has motivated a substantial body of parallel work. For shared-memory architectures, Staudt and Meyerhenke [25] demonstrated that standard community detection algorithms, including Louvain, can be engineered to achieve practical speedups on multicore machines with careful attention to synchronization. Halappanavar et al. [26] developed Grappolo, a shared-memory parallel Louvain implementation that achieves strong scaling on moderately large graphs. More recently, Sahu et al. [27] introduced GVE-Leiden, a shared-memory parallel implementation of the Leiden algorithm achieving substantial speedups through optimized data structures.

Distributed-memory community detection has been pursued primarily through distributed variants of the Louvain method. Ghosh et al. [28] developed an MPI and OpenMP implementation that distributes cluster assignments across nodes using a round-robin strategy. Que et al. [29] demonstrated Louvain scaling to 138 billion edges on thousands of Blue Gene/Q nodes. Sattar and Arifuzzaman [30] more recently proposed a distributed Louvain variant with improved load balancing for skewed degree distributions. Critically, none of the above distributed community detection methods provide any guarantee on the connectivity of their output communities.

To our knowledge, the present work is the first to enforce user-defined connectivity standards in a distributed-memory community detection pipeline, bridging the gap between HPC-scale graph processing and principled connectivity-preserving community refinement.

2.3 High-Performance Graph Analytics Frameworks

Several distributed graph processing frameworks have been developed to support general graph algorithms at scale. Pregel [31] introduced the vertex-centric BSP model for distributed graph computation. GraphX [32] integrated graph processing into Apache Spark's dataflow model. PowerGraph [33] addressed the challenge of high-degree vertices in natural graphs through a GAS (Gather-Apply-Scatter) decomposition. Gluon [34], the communication substrate underlying distributed Galois, introduced optimized bulk synchronous communication for heterogeneous graph workloads. Gemini [35] demonstrated that careful cache- and communication-aware graph layout can substantially outperform prior systems on commodity clusters.

Within the Chapel ecosystem, Arkouda [36] provides a NumPy-like interactive analytics interface backed by Chapel's distributed memory model. The Arachne package [24] extends Arkouda with graph-specific data structures and algorithms, including subgraph isomorphism [37, 38], property graph support [39] and other large scale graph analysis [40]. Jenkins et al. [41] evaluated Chapel against UPC++ for graph algorithms in the PGAS setting, finding competitive performance with substantially reduced programming effort in Chapel. The Chapel distributed implementations in this paper extend the Arachne framework to multi-locale execution, adding connectivity-preserving community detection to its growing set of scalable graph analytics capabilities.

3 Background and Preliminaries

This section establishes the graph-theoretic foundations and algorithmic framework underlying the WCC and CM methods. We introduce the core definitions, describe the well-connectedness criterion, and present the algorithmic structure shared by both methods, which the implementations in subsequent sections extend to parallel and distributed settings.

3.1 Graph Notation

Let $G = (V, E)$ denote an undirected, unweighted graph with vertex set V and edge set $E \subseteq \binom{V}{2}$. We write $n = |V|$ and $m = |E|$. For a subset $S \subseteq V$, the induced subgraph $G[S] = (S, E_S)$ has edge set $E_S = \{(u, v) \in E : u \in S \text{ and } v \in S\}$.

A clustering $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ is a partition of a subset of V into disjoint groups, where each $C_i \subseteq V$ is called a cluster or community. A clustering is produced by an upstream community detection algorithm such as Leiden or a stochastic block model (SBM) method; WCC and CM operate as post-processing refinement steps applied to such an input.

3.2 Edge Connectivity and Well-Connectedness

The edge connectivity $\lambda(G)$ of a connected graph G is the minimum number of edges whose removal disconnects G, equivalently the size of a global minimum cut of G:

$$\lambda(G) = \min_{\emptyset \neq S \subsetneq V} |\delta(S)|,$$

where $\delta(S) = \{(u, v) \in E : u \in S, v \notin S\}$ is the cut induced by S. A graph with higher edge connectivity requires the removal of more edges to become disconnected, making it more robust to perturbation.

A cluster C with induced subgraph $G[C]$ is said to be well-connected with respect to a non-decreasing criterion function $f : \mathbb{Z}_{>0} \to \mathbb{R}_{>0}$ if and only if:

$$\lambda(G[C]) > f(|C|).$$

The criterion function f can be any monotone non-decreasing function of the cluster size; examples include $\log_{10}(|C|)$, $\log_2(|C|)$, $\sqrt{|C|}$, and linear functions $k|C|$ for a user-specified constant k. The choice of f governs the sensitivity of the connectivity requirement to cluster size: sublinear functions such as $\log_{10}(|C|)$ impose a mild requirement that grows slowly with cluster size, whereas linear functions impose a strong requirement. Throughout our experiments we use $f(|C|) = \log_{10}(|C|)$, the criterion used in the original WCC and CM studies [16, 18, 19].

3.3 Connected Component Refinement

Both WCC and CM begin with a preprocessing step called connected component refinement (CCR), which ensures that all input clusters passed to the main refinement routine are themselves connected. Given an input clustering $\mathcal{C}$, CCR processes each cluster $C \in \mathcal{C}$ by constructing its induced subgraph $G[C]$ and computing its connected components. Each connected component exceeding s_{pre} vertices is forwarded as an independent subgraph for further processing; smaller components are discarded. The output of CCR is thus a collection of connected induced subgraphs, each ready for the well-connectedness check. The parameter s_{pre} allows practitioners to filter out negligibly small components that would not yield meaningful communities.

3.4 Well-Connected Clusters (WCC)

The WCC algorithm [18, 19] recursively evaluates each connected input subgraph against the well-connectedness criterion and bisects those that fail. Given a connected subgraph $G[C]$, WCC computes its global minimum cut $\lambda(G[C])$ and evaluates the criterion $f(|C|)$. If $\lambda(G[C]) > f(|C|)$, the cluster is accepted and stored as output. Otherwise, the minimum cut partitions C into two parts C_1 and C_2, and WCC recurses independently on $G[C_1]$ and $G[C_2]$, provided each part exceeds the post-processing size threshold s_{post}. Recursion terminates when all surviving subgraphs either satisfy the criterion or fall below s_{post}. Crucially, WCC performs only subdivision: it never merges clusters, so the output is a subset of recursive bisections of the input communities.

3.5 Connectivity Modifier (CM)

The CM algorithm [16, 20] shares the same recursive bisection structure as WCC but incorporates an additional community detection step after each minimum cut partition. When a subgraph $G[C]$ fails the well-connectedness criterion and is split into $G[C_1]$ and $G[C_2]$ along the minimum cut, CM applies a user-selected community detection algorithm (CDA), such as Leiden with a CPM objective, to each part independently. If the CDA finds multiple communities within a part, CM recurses on each community separately; if only one community is found, the entire part is treated as a unit for the next recursion step. This two-phase structure, minimum cut removal followed by community re-detection, allows CM to uncover semantically cohesive substructures that may be obscured in the original clustering by weak inter-component connections. The choice of CDA is left to the user, providing flexibility to match the clustering objective to the application domain.

3.6 Shared Algorithmic Structure and Parallelism Opportunities

Both WCC and CM exhibit a recursive tree structure over the space of cluster subgraphs. At the top level, CCR produces a collection of independent connected subgraphs that can be processed entirely in parallel, since there are no data dependencies between distinct input clusters. Within each subgraph, the recursion proceeds depth-first, with each bisection generating two independent subtrees. This structure offers two natural levels of parallelism: cluster-level parallelism, exploiting independence across the input collection, and within-cluster parallelism, exploiting independence across the two subtrees generated by each bisection. In practice, cluster-level parallelism dominates the available concurrency in typical workloads, since the number of input clusters is large (often hundreds of thousands) while individual cluster subgraphs are typically small. This observation guides the implementation strategy in both the shared-memory Chapel implementations reviewed in Section 4 and the distributed implementations introduced in Sections 5.5 and 5.6.

The minimum cut computation via VieCut [42] is the dominant per-cluster cost and is applied sequentially to individual subgraphs. The aggregate cost scales with the number of clusters requiring refinement and the size distribution of those clusters, both of which depend on the upstream clustering quality and the choice of resolution parameter. Graphs with many small, poorly-connected clusters, typical of fine-grained Leiden CPM partitions, exhibit high cluster-level parallelism and relatively low per-cluster cost, while coarser partitions produce fewer but larger subgraphs that demand more work per cluster. This variability makes load balancing a central concern in all parallel and distributed implementations, as discussed in subsequent sections.

4 Shared-Memory Parallel Implementations

This section summarizes the Chapel-based shared-memory parallel implementations of WCC and CM first introduced in [22], which serve as the foundation for the distributed

extensions presented in Sections 5.5 and 5.6. We describe the key design decisions, Chapel-specific optimizations, and integration with the Arachne framework.

4.1 Graph Representation in Arachne

Within Arachne [24], graphs are stored using a double-index (DI) format that extends the standard compressed sparse row (CSR) representation with an auxiliary edge-to-source array. This augmentation enables $O(1)$ lookup of the source vertex for any given edge index, a capability that is exploited heavily in the induced subgraph construction step of both WCC and CM. When constructing the induced subgraph $G[C]$ for a cluster C, all edges incident to vertices in C must be filtered to retain only those with both endpoints in C. The DI format allows this filtering to proceed without a full adjacency traversal, reducing the constant factors in subgraph extraction substantially on large inputs.

4.2 Design Principles

Three design principles govern the shared-memory implementations. First, parallelism is applied at the cluster level rather than within individual minimum cut computations. Since most cluster subgraphs are small relative to the full graph, the overhead of launching parallel work within a single min-cut computation outweighs the benefit; instead, large numbers of clusters are evaluated concurrently across Chapel tasks. Second, the queue-based task management of the original C++ is replaced by a recursive framework that maps naturally onto Chapel's parallel tasking model, eliminating explicit queue synchronization overhead and improving cache locality. Third, a preprocessing size threshold s_{pre} and post-processing threshold s_{post} gate entry into and continuation of the recursion, avoiding wasted work on trivially small subgraphs.

4.3 Parallel Connected Component Refinement

Algorithm 1 presents the connected component refinement procedure. Each input cluster $C \in \mathcal{C}$ is converted to its induced subgraph $G[C]$, connected components are computed, and any component exceeding s_{pre} vertices is enqueued for further processing. The outer loop over clusters is parallelized using Chapel's `forall` construct, allowing all clusters to be processed concurrently. The connected components computation within each cluster uses a parallel union-find implementation from the Arachne framework.

4.4 Parallel WCC

The top-level WCC procedure (Algorithm 2) applies CCR to obtain a collection of connected subgraphs and then evaluates each subgraph independently via the recursive IsWCC check (Algorithm 3). The `forall` loop over the CCR output queue distributes subgraphs across Chapel tasks. Each task evaluates its assigned subgraph independently with no shared mutable state between tasks, avoiding the need for locks or atomic operations at the cluster level.

Algorithm 1 Connected Component Refinement

procedure CCR($G = (V, E)$, $\mathcal{C}$, s_{pre})
  $Q \leftarrow \emptyset$
  for all $c \in \mathcal{C}$ **do** ▷ parallel `forall`
    $V_c \leftarrow c$
    $E_c \leftarrow \{(u, v) \in E : u \in V_c \wedge v \in V_c\}$
    $G_c \leftarrow (V_c, E_c)$
    if $|E_c| > 0$ **then**
      $CC \leftarrow$ ConnectedComponents(G_c)
      for all $cc \in CC$ **do**
        if $|cc| > s_{\text{pre}}$ **then**
          $Q \leftarrow Q \cup \{cc\}$
        end if
      end for
    end if
  end for
  return Q
end procedure

Algorithm 2 Well-Connected Clusters

procedure WCC($G = (V, E)$, $\mathcal{C}$, s_{pre}, s_{post})
  $Q \leftarrow \text{CCR}(G, \mathcal{C}, s_{\text{pre}})$
  $W \leftarrow \emptyset$
  for all $q_i \in Q$ **do** ▷ parallel `forall`
    $G_{q_i} \leftarrow (q_i,\ \{(u, v) \in E : u, v \in q_i\})$
    IsWCC(G_{q_i}, s_{post})
  end for
  return W
end procedure

The recursive bisection in IsWCC generates two independent subtrees at each step. In the Chapel implementation, each recursive call is issued as a Chapel task via `begin` or `cobegin`, allowing both subtrees to proceed in parallel when sufficient task resources are available. In practice, task creation overhead limits the benefit of within-cluster parallelism for small subgraphs, so a depth cutoff is applied beyond which recursion proceeds sequentially. The minimum cut computation within each call uses a sequential variant of VieCut [42], which provides near-optimal practical performance on the small subgraphs that arise after the first few levels of bisection.

4.5 Parallel CM

The CM procedure (Algorithms 4 and 5) follows the same top-level structure as WCC. The key distinction is in the CMC check: after partitioning a failing subgraph along its minimum cut, CM applies the user-selected community detection algorithm (CDA)

Algorithm 3 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 ← f(|V_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_ci ← {(u, v) ∈ E_c : u, v ∈ V_ci} for i ∈ {1, 2}
10:            if |V_c1| > s_post then
11:                IsWCC((V_c1, E_c1), s_post)
12:            end if
13:            if |V_c2| > s_post then
14:                IsWCC((V_c2, E_c2), s_post)
15:            end if
16:        end if
17:    end if
18: end procedure
```

to each resulting part before recursing. If the CDA identifies multiple communities within a part, each community is recursed upon independently; if only one community is found, the part is processed as a unit. This design allows CM to recover semantically meaningful structure within weakly connected regions, at the cost of invoking the CDA repeatedly during refinement.

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                          ▷ parallel forall
4:         G_qi ← (q_i, {(u, v) ∈ E : u, v ∈ q_i})
5:         CMC(G_qi, s_post, CDA)
6:     end for
7: end procedure
```

In the Chapel implementation, the outer `forall` in CM distributes the initial cluster workload across tasks. The CDA invocation within CMC calls Leiden via a Chapel foreign function interface to the C++ Leiden library. The overhead of this foreign function call contributes to the performance variability observed in CM relative to WCC, particularly on smaller graphs where the per-cluster cost is low and the call overhead is relatively significant. A Chapel-native Leiden implementation would eliminate this overhead and is identified as a direction for future work in Section 7.

Algorithm 5 Connectivity Modifier Check

```
1:  procedure CMC(G_c = (V_c, E_c), s_post, CDA)
2:      if |E_c| ≥ 1 then
3:          cut ← GetMinCut(G_c)
4:          criterion ← f(|V_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:              for all G_part ∈ {G[V_c1], G[V_c2]} do
10:                 if |V_part| > s_post then
11:                     C' ← GetCommunities(G_part, CDA)
12:                     if |C'| > 1 then
13:                         for all V_i ∈ C' do
14:                             if |V_i| > s_post then
15:                                 CMC(G[V_i], s_post, CDA)
16:                             end if
17:                         end for
18:                     else
19:                         CMC(G_part, s_post, CDA)
20:                     end if
21:                 end if
22:             end for
23:         end if
24:     end if
25: end procedure
```


4.6 Load Balancing and Scalability Characteristics

The shared-memory implementations distribute work using Chapel's task pool, which dynamically assigns clusters to available processor cores. Because cluster sizes follow a heavy-tailed distribution in most real-world graphs, the workload is inherently imbalanced: a small number of large clusters dominate the runtime while the majority of clusters complete quickly. The dynamic task scheduling in Chapel's `forall` loops partially mitigates this imbalance, but large outlier clusters can still cause processor stalls when they occupy a thread for an extended period without subdivision.

Strong scaling experiments on the Bitcoin (medium-scale) and Open-Alex (billion-edge) networks (from 1 to 128 cores) demonstrate that WCC-Chapel scales well up to 32 cores on Bitcoin and up to 64 cores on Open-Alex, while CM-Chapel sustains improvement to 64 cores on both networks before encountering diminishing returns from parallelization overhead. These trends are illustrated in Figure 1, where the scaling curves reveal distinct divergence at higher core counts. At 128 cores on the Open-Alex network, WCC-Chapel exhibits a severe performance collapse attributable to load imbalance in the recursive refinement tree, while CM-Chapel degrades more gracefully due to the additional community detection step producing finer-grained

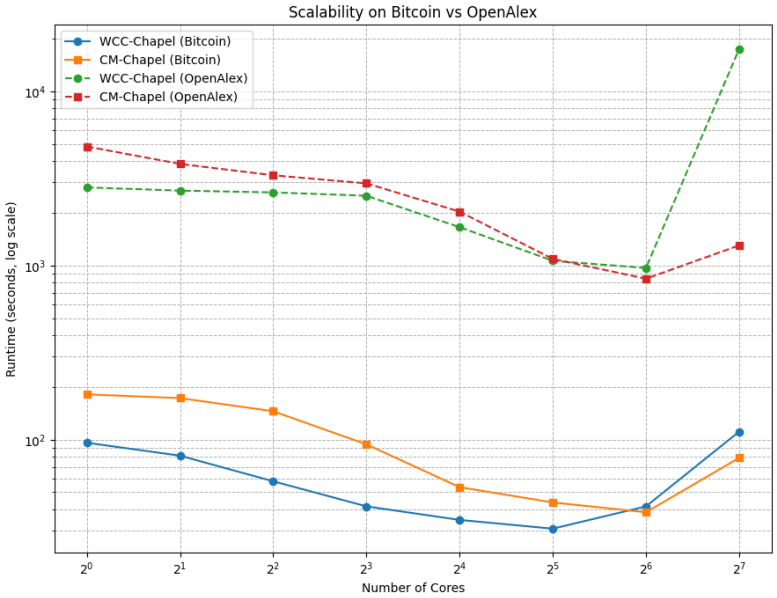

Fig. 1 Strong scaling of WCC-Chapel and CM-Chapel on the Bitcoin and OpenAlex networks.

work units. The sharper inflection in the WCC curve compared to the smoother tapering of CM highlights the different parallel bottlenecks in the two algorithms. These scalability characteristics motivate the distributed extensions in subsequent sections, which address both the memory capacity constraint and the load balancing challenge through explicit work distribution across compute nodes.

5 Distributed Implementations of WCC and CM

This section presents the distributed implementations of WCC and CM in both C++ with MPI and Chapel with multi-locale execution. Both implementations share the same fundamental architecture and engineering optimizations; the distinction lies in the communication and task management layer used to realize that architecture. We describe the shared design once and then note the language-specific details for each.

5.1 Limitations of the Original Pipeline

The original C++ baselines are shared-memory parallel implementations using standard thread management. The full graph, the input clustering, and all intermediate cluster subgraphs must reside concurrently in the memory of a single machine. The pipeline loads the full graph, constructs disjoint cluster subgraphs, identifies their connected components, and passes components to threads for minimum cut evaluation, all within a single shared address space. For billion-edge graphs such as Open-Alex (∼2.1B edges) and Open-Citations-v2 (∼2.0B edges), this imposes peak memory

requirements exceeding 250 GB on a single machine, making execution infeasible without access to a large-memory server. The shared-memory Chapel implementations from Section 4 alleviate the memory constraint for graphs that fit within a single large-memory node, but share the same fundamental limitation: the entire pipeline is confined to one machine.

It is worth noting that the architectural redesign introduced in this paper (Section 5.3) makes single-node execution of billion-edge graphs feasible in principle, since no node ever loads the full graph. However, the distributed implementation remains necessary to achieve practical performance: distributing subgraph files across nodes provides both memory capacity scaling and parallel throughput that a single-node serialization of the same pipeline could not match. The following subsection establishes the motivation for this architectural redesign.

A second limitation, independent of memory capacity, affects some large-graph configurations. Memory leaks and data races within VieCut [42] are associated with the crashes and segmentation faults observed during experimentation. These cases are reported as dashes in the experimental tables. The C++ distributed implementation is consistently affected on the impacted graphs, while the Chapel distributed implementation shows a different failure profile, as discussed in Section 5.8.

5.2 Motivation for the Architectural Redesign

The original WCC/CM pipeline requires constructing all cluster induced subgraphs from the full in-memory graph at runtime as a separate preprocessing pass. This is redundant: the same subgraphs are a natural byproduct of the Leiden clustering step, which must run regardless. By integrating subgraph generation into Leiden, the distributed pipeline eliminates this preprocessing pass entirely. Each compute node receives only its assigned subgraph files and begins the minimum cut stage immediately upon clustering completion, without ever loading the full graph.

5.3 Architectural Overview

Both the C++ and Chapel distributed implementations are built around a key architectural insight: the subgraph files needed for distributed WCC and CM execution can be produced during the Leiden clustering step rather than in a separate WCC/CM preprocessing pass. When Leiden produces its output clustering, our modified pipeline simultaneously writes each cluster's induced subgraph, and its connected components, to disk as independent subgraph files. By the time clustering is complete, the distributed WCC and CM pipeline can begin with no separate subgraph-construction pass required.

5.3.1 Subgraph File Generation (integrated with Leiden).

During Leiden execution, each cluster's induced subgraph is constructed and its connected components are identified in parallel across all available cores. Each connected component is written to disk as a standalone subgraph file. The result is a collection of subgraph files, one per connected component, that fully encode the input to the distributed minimum cut stage. Because this happens inside Leiden rather than as

a separate pass over the data, the full graph never needs to be loaded again by any WCC or CM process.

5.3.2 Distributed Minimum Cut Stage.

The subgraph files are distributed across compute nodes using a round-robin assignment strategy. Each node loads only its assigned subgraph files and executes a fully independent WCC or CM pipeline, connected component refinement, well-connectedness evaluation, and recursive bisection, without any inter-node communication during refinement. The full graph is never loaded on any node during this stage. Final cluster assignments are aggregated via a collective operation after all nodes complete. Because each node's memory footprint is proportional only to its assigned subgraph files rather than to the full input graph, this design substantially extends the graph sizes that can be processed on commodity cluster hardware. Round-robin assignment provides a simple and effective initial load distribution; a load-aware sorted round-robin strategy that improves on this baseline is presented in Section 5.7.

5.4 Engineering Optimizations

Three optimizations are applied consistently across both the C++ and Chapel distributed implementations, and to both WCC and CM.

5.4.1 Memory-Mapped I/O

Both implementations replace standard file I/O with memory-mapped I/O via the `mmap` system call. The C++ baseline uses buffered sequential reads, which incurs significant system call overhead for large files. Memory mapping exposes file contents directly in the process address space, allowing the operating system to handle paging on demand and enabling concurrent reads across cores for different file regions. This yields 3–5× speedup in file loading across all datasets and configurations.

5.4.2 Parallel and Distributed Connected Component Computation

The baseline performs connected component discovery on a single node over the full in-memory graph. Our design replaces this with a two-level approach that is both parallel within each node and distributed across nodes. Within a node, connected components are computed in parallel across all available cores using a parallel union-find algorithm. Across nodes, the subgraph files produced during Leiden execution are distributed round-robin so that each node owns a disjoint set of components. Every node then executes a fully self-contained WCC or CM pipeline on its local components with no inter-node communication during the recursive refinement phase. This eliminates the serial single-node bottleneck of the baseline entirely, yielding 6–10× speedup in the minimum cut phase on multi-node configurations.

5.4.3 Decoupled Subgraph Loading

By materializing each connected component as a standalone subgraph file during Leiden execution, the minimum cut stage can load and process components independently

and on demand, without reconstructing them from the original graph at runtime. This eliminates the subgraph construction step from the hot path of the distributed pipeline, a step that in the baseline costs up to 60 minutes for billion-edge graphs at 8 cores. With memory-mapped loading of pre-built files, this overhead is reduced to seconds per node.

5.5 C++ Implementation

The C++ distributed implementation uses MPI for inter-node coordination. Subgraph files produced during Leiden execution are written to a shared filesystem accessible by all worker nodes. The minimum cut stage launches an MPI job in which each rank loads its round-robin-assigned subgraph files independently and executes the WCC or CM pipeline locally. After all ranks complete, a single `MPI_Gather` collective operation aggregates the cluster assignments. We provide two distributed variants for both WCC and CM:

- **Distributed Recursive (WCC-DR, CM-DR):** Multi-node MPI with recursive task management; subgraph files distributed round-robin across ranks.
- **Distributed Queue (WCC-DQ, CM-DQ):** Multi-node MPI with an explicit distributed work queue for task management.

As shown in Section 6, neither the recursive nor queue variant dominates uniformly across all graphs; the optimal choice depends on the cluster size distribution induced by the upstream clustering method and resolution parameter. On a subset of large graphs, the C++ distributed implementation encounters consistent failures associated with memory leaks and data races within VieCut, as discussed in Sections 5.1, and 5.8.

5.6 Chapel Implementation

The Chapel distributed implementation realizes the same architecture using Chapel's native multi-locale execution model [23]. Chapel's locale abstraction maps directly onto physical compute nodes, and its distributed domain model allows subgraph file assignments to be expressed as locale-indexed data structures without explicit message passing. Subgraph files produced during Leiden execution are written to shared storage accessible by all locales. In the minimum cut stage, Chapel tasks are spawned on each locale via `on loc do` blocks, with each locale loading its round-robin subgraph assignment and executing the WCC or CM pipeline independently. Result aggregation uses Chapel's built-in parallel reduction across locales.

In contrast to the C++ implementation where both recursive and queue variants remain competitive depending on graph topology, empirical evaluation on our benchmark graphs showed that the recursive variant consistently outperforms the queue-based variant in Chapel. This difference arises from Chapel's tasking model, in which recursive task spawning maps efficiently onto Chapel's work-stealing runtime, while explicit queue management introduces synchronization overhead that offsets its load-balancing benefit. We therefore adopt the recursive variant as the Chapel distributed implementation for both WCC and CM.

The Chapel implementation is integrated into the Arachne framework [24], making distributed well-connected community detection directly accessible through Arachne's Python-facing interface. The Chapel distributed implementation also exhibits a broader coverage of large graphs compared to the C++ distributed implementation, succeeding on Open-Alex and Open-Citations configurations where the C++ implementation encounters failures, as discussed in Section 5.8

5.7 Load-Balanced Cluster Distribution

The round-robin assignment described in Section 5 distributes equal numbers of subgraph files across locales but ignores cluster size, producing load imbalance when cluster sizes follow a heavy-tailed distribution, a property common to Leiden CPM partitions of real-world networks. This section introduces a load-aware distribution strategy that improves total runtime on most successful configurations.

There are two levels at which distribution decisions are made. **Inter-locale:** deciding which clusters go to which of the 4 locales. **Intra-locale:** deciding the order in which clusters are listed inside each locale's folder. Chapel splits the file into 32 equal-sized chunks, one per core, in file order. If the heaviest clusters all end up in the first chunk, core 0 does all the hard work while the others finish quickly and sit idle. To address this, we apply **interleaved intra-locale ordering**: sort clusters by $n + m$ descending, then assign positions 1, 33, 65, ... to core 0; positions 2, 34, 66, ... to core 1, and so on. This ensures every core receives a proportional mix of large and small clusters.

For inter-locale assignment, we evaluated five variants of Longest Processing Time First (LPT), using vertex count, edge count, $n + m$, $m \cdot \log(n)$, and m^2/n as weights, alongside plain round-robin and sorted round-robin. Every LPT strategy, regardless of weight function, consistently produces the same failure pattern: one locale finishes 3–6× faster than the others and sits idle. The weight function overestimates how long large clusters actually take to process. LPT assigns the single heaviest cluster to Locale 0. Locale 0 finishes its giant cluster fast, picks up a few tiny ones, and goes idle. Meanwhile, the other locales have hundreds of medium-sized clusters that require more work and take much longer collectively.

The strategy that consistently performs best is **sorted round-robin** (`rr_npm`): sort clusters by $n + m$ from largest to smallest, then assign them like a deck of cards, Locale 0 gets ranks 1, 5, 9, 13, ...; Locale 1 gets ranks 2, 6, 10, 14, ...; and so on. Instead of predicting per-cluster runtime, this approach simply guarantees that every locale receives the same diversity of cluster sizes. It makes no assumption about per-cluster cost and is therefore robust to VieCut's early-exit behavior on large graphs.

An important source of run-to-run variance in all configurations is VieCut's non-deterministic early-exit behavior: VieCut can terminate early once it determines a cluster is well-connected, but the exact point of early exit varies across runs due to randomized internal decisions. A large cluster that exits in 2 seconds on one run may take 8 seconds on the next. Two consecutive CM runs on Cit-Patents at CPM 0.001 illustrate this: node finish times of 3.37s, 6.21s, 6.25s, 6.44s (total 6.61s) versus 3.23s, 3.43s, 5.86s, 5.90s (total 6.08s), an 8% difference from identical inputs. Since large clusters are the dominant cost drivers, a single cluster flipping between fast and slow exit

can shift a locale's total time by 10–20 seconds. Despite this variance, `rr_npm` usually reduced total runtime across repeated runs by eliminating the accidental concentration of large clusters on a single locale that plain round-robin produces.

5.8 Output Consistency and Failure Behavior

Both the C++ and Chapel distributed implementations produce cluster assignments consistent with the C++ baseline. For every dataset and configuration where the distributed implementations complete successfully, output cluster counts match the baseline with at most minor differences. These differences are attributable to non-determinism in the pipeline, specifically, the non-deterministic behavior of Leiden used as the community detection step within CM, and variations in task scheduling across distributed nodes, rather than to algorithmic incorrectness. All distributed variants, including the recursive and queue C++ implementations and the recursive Chapel implementation, exhibit this correctness property across all tested datasets and node configurations.

Both implementations invoke the same VieCut library [42] for minimum cut computation; however, they exhibit different failure profiles on a subset of large graphs. The C++ distributed implementation failed consistently on every run of Open-Alex, Open-Citations, and Wikipedia-Links at CPM 0.001. The Chapel distributed implementation succeeded on several Open-Alex and Open-Citations configurations across repeated runs (each experiment run at least three times), graphs on which C++ distributed fails consistently. However, Chapel distributed does not complete every large-graph configuration: Open-Alex CM at CPM 0.001 fails. Wikipedia-Links fails for both C++ and Chapel distributed implementations; the VieCut-related issues manifest on this graph regardless of implementation. The C++ and Chapel implementations produce subgraph representations with different internal memory layouts. We attribute the different failure profiles to these layout differences: the C++ representation consistently triggers the problematic behavior, while the Chapel representation does so only rarely.

6 Experimental Evaluation

We evaluate all implementations across two axes: performance benchmarks comparing runtime against the C++ baseline, and scalability analysis measuring strong scaling behavior with respect to core count. Experiments cover both WCC and CM on a diverse set of real-world networks spanning five orders of magnitude in edge count.

6.1 Experimental Setup

All shared-memory experiments were conducted on a dual-socket AMD EPYC 7713 system with 128 cores total and 512 GB RAM. Distributed experiments used a cluster of nodes with the same per-node specification, connected via high-speed interconnect, with 4 nodes (128 cores total across nodes) for the distributed configurations reported unless otherwise noted.

Table 1 lists the real-world networks used in all experiments. Graphs are grouped into small-to-medium and large categories. Isolated vertices were removed in preprocessing; all graphs are treated as undirected.

Table 1 Real-World Networks Used in Experiments

Category	Network	Vertices†	Edges
Small to Medium	Bitcoin [43]	6,336,770	16,057,711
	Livejournal [43]	4,847,571	68,993,773
	Cit-Patents [44]	3,774,768	16,518,947
	Orkut [44]	3,072,441	117,185,083
	Hyves [43]	1,402,673	2,777,419
Large	Open-Alex [21]	256,997,006	2,148,871,058
	Open-Citations-v2 [45]	121,052,490	1,962,840,983
	Open-Citations [46]	75,025,194	1,363,605,603
	CEN [44]	13,989,436	92,051,051
	Wikipedia-Links [47]	13,593,032	437,217,424

† Vertex counts reflect total vertices in the source dataset prior to preprocessing. Isolated vertices were removed before WCC/CM processing.

Input clusterings were generated using the Leiden algorithm with the Constant Potts Model (CPM) at resolution parameters $\gamma \in \{0.001, 0.01\}$, reflecting fine-grained and coarser partitions respectively. We use $f(n) = \log_{10}(n)$ as the well-connectedness criterion and set $s_{\text{pre}} = s_{\text{post}} = 1$ throughout, which is the most conservative setting: no cluster is pruned before or after refinement regardless of size. This maximizes the computational workload passed to WCC and CM and therefore represents the hardest case for our implementations; it ensures that reported runtimes are not artifacts of aggressive early pruning. For CM experiments, Leiden-CPM serves as the internal community detection algorithm (CDA). All runtimes are wall-clock seconds averaged over at least three successful runs for both distributed and shared-memory experiments. Shared-memory results are consistent with those reported in [22].

6.2 Shared-Memory Performance

Table 2 and Table 3 report the runtime of the Chapel-based shared-memory implementations of WCC and CM against the baseline on all datasets, using 128 cores on a single node. These results extend and consolidate the findings first reported in [22].

WCC-ChSM (Chapel Shared Memory) achieves consistent and substantial speedups over the baseline on small-to-medium networks, reaching up to 12× improvement. CM-ChSM is competitive or faster in most cases, with occasional instances where the baseline is marginally faster due to the overhead of Chapel's foreign function interface for the Leiden CDA call. On large networks, the C++ baseline implementations fail entirely due to memory exhaustion, while the Chapel implementations complete successfully on most large networks. On CEN, the only large network where both can run, WCC-Chapel achieves up to 15.5× speedup.

Table 2 Shared-Memory Runtime (seconds) on Small to Medium Datasets, 128 cores.

Leiden CPM 0.001				
Dataset	WCC-base	WCC-ChSM	CM-base	CM-ChSM
Bitcoin	805.9	100.2	72.3	62.1
Livejournal	916.2	78.4	58.4	40.3
Cit-Patents	372.3	48.3	43.6	22.1
Orkut	314.0	72.3	88.8	44.2
Hyves	74.8	70.8	18.7	9.1
Leiden CPM 0.01				
Dataset	WCC-base	WCC-ChSM	CM-base	CM-ChSM
Bitcoin	277.1	105.4	116.8	90.1
Livejournal	271.1	108.9	84.0	72.3
Cit-Patents	223.1	82.1	66.7	55.3
Orkut	211.9	85.3	74.4	47.1
Hyves	28.3	78.1	26.0	12.3

Table 3 Shared-Memory Runtime (seconds) on Large Datasets, 128 cores. Dashes indicate failure due to OOM or segmentation fault.

Leiden CPM 0.001				
Dataset	WCC-base	WCC-ChSM	CM-base	CM-ChSM
Open-Alex	—	1130.4	—	1117.3
Open-Cit.-v2	—	1120.8	—	1146.2
Open-Citations	—	1030.1	—	871.6
CEN	4330.0	278.4	152.3	156.2
Wikipedia-Links	—	—	238.7	278.9
Leiden CPM 0.01				
Dataset	WCC-base	WCC-ChSM	CM-base	CM-ChSM
Open-Alex	—	1962.5	—	1912.7
Open-Cit.-v2	—	1632.1	—	1721.4
Open-Citations	—	1270.7	—	1346.3
CEN	1369.3	229.3	206.9	199.1
Wikipedia-Links	—	304.6	304.1	372.8

6.3 Distributed Performance

Tables 4 and 5 report the runtime of the distributed C++ implementations (WCC-DR, WCC-DQ, CM-DR, and CM-DQ). Dashes indicate failures in the C++ implementations inherited from the VieCut limitation described in Sections 5.1 and 5.8.

The distributed C++ implementations deliver substantial speedups over the C++ baseline on graphs where both complete successfully. On CEN, the largest graph on which all implementations run, WCC-DQ achieves up to 65× speedup over the C++ baseline at CPM 0.001, reducing a computation of over an hour to under 70 seconds. On Open-Citations-v2, the only billion-edge graph where C++ distributed execution succeeds, WCC-DQ completes in 72 seconds at CPM 0.01, a graph on which the C++ baseline and shared-memory Chapel both fail entirely. On small-to-medium networks, where the baseline runs successfully, the distributed variants consistently outperform

Table 4 C++ Distributed Runtime (seconds) on Small to Medium Datasets. 32 cores × 4 nodes.

Leiden CPM 0.001				
Dataset	WCC-DR	WCC-DQ	CM-DR	CM-DQ
Bitcoin	69	33	122	96
Livejournal	119	56	58	144
Cit-Patents	56	27	37	74
Orkut	85	35	37	74
Hyves	12	4	10	9
Leiden CPM 0.01				
Dataset	WCC-DR	WCC-DQ	CM-DR	CM-DQ
Bitcoin	9	11	14	23
Livejournal	9	12	12	13
Cit-Patents	8	11	11	12
Orkut	9	10	11	14
Hyves	7	5	8	7

Table 5 C++ Distributed Runtime (seconds) on Large Datasets. 32 cores × 4 nodes. Dashes indicate failure due to a known VieCut limitation (see Section 5.8).

Leiden CPM 0.001				
Dataset	WCC-DR	WCC-DQ	CM-DR	CM-DQ
Open-Alex	—	—	—	—
Open-Cit.-v2	340	259	—	—
Open-Citations	—	—	—	—
CEN	97	67	31	51
Wikipedia-Links	—	—	—	—
Leiden CPM 0.01				
Dataset	WCC-DR	WCC-DQ	CM-DR	CM-DQ
Open-Alex	—	—	—	—
Open-Cit.-v2	100	72	95	136
Open-Citations	—	—	—	—
CEN	17	14	13	14
Wikipedia-Links	—	—	—	—

it across all datasets and both resolution parameters, with WCC-DQ reaching up to 8× speedup on Bitcoin at CPM 0.001.

The recursive (DR) and queue (DQ) variants exhibit graph-dependent performance profiles, with neither uniformly dominating across all datasets. On networks with relatively uniform cluster size distributions, DR tends to perform comparably to or better than DQ. On graphs with more skewed distributions, DQ can outperform DR due to its more flexible dynamic task assignment. The optimal choice depends on the cluster size distribution induced by the upstream clustering method and resolution parameter, as discussed in Section 5.

Wikipedia-Links, Open-Alex, and Open-Citations fail across all C++ configurations (shared-memory baseline and C++ distributed) in both resolution parameter settings. As established in Sections 5.1 and 5.8, these failures are associated with memory leaks and data races within VieCut that manifest on specific cluster subgraph

structures. They are not caused by the distributed architecture itself, and the Chapel distributed implementation handles several of these cases successfully, as reported in Section 6.4. Additionally, the round-robin assignment strategy can concentrate memory pressure on individual nodes when subgraph size distributions are highly skewed, which may compound failures on graphs with extreme size imbalance.

6.4 Chapel Distributed Performance

Tables 6 and 7 report the runtime of the Chapel distributed implementation on all datasets at both resolution parameters, with the Chapel shared-memory reference (WCC-ChSM, CM-ChSM) included for direct comparison.

Table 6 Chapel Distributed Runtime (seconds) on Small to Medium Datasets. 32 cores × 4 nodes.

Leiden CPM 0.001						
Dataset	WCC-ChSM	WCC-ChDis	Speedup	CM-ChSM	CM-ChDis	Speedup
Bitcoin	100.2	36.2	2.77×	62.1	56.7	1.10×
Livejournal	78.4	103.9	0.75×	40.3	21.9	1.84×
Cit-Patents	48.3	46.3	1.04×	22.1	8.9	2.48×
Orkut	72.3	58.6	1.24×	44.2	30.6	1.44×
Hyves	70.8	3.6	19.7×	9.1	8.8	1.03×
Leiden CPM 0.01						
Dataset	WCC-ChSM	WCC-ChDis	Speedup	CM-ChSM	CM-ChDis	Speedup
Bitcoin	105.4	6.3	16.73×	90.1	7.3	12.34×
Livejournal	108.9	6.7	16.25×	72.3	8.8	8.22×
Cit-Patents	82.1	3.2	25.66×	55.3	4.6	12.02×
Orkut	85.3	6.2	13.76×	47.1	7.2	6.54×
Hyves	78.1	1.4	55.79×	12.3	4.1	3.00×

Table 7 Chapel Distributed Runtime (seconds) on Large Datasets. 32 cores × 4 nodes. Dashes indicate failure (see Section 5.8).

Leiden CPM 0.001						
Dataset	WCC-ChSM	WCC-ChDis	Speedup	CM-ChSM	CM-ChDis	Speedup
Open-Alex	1130.4	894.1	1.26×	1117.3	—	—
Open-Cit.-v2	1120.8	322.7	3.47×	1146.2	277.2	4.13×
Open-Citations	1030.1	360.5	2.86×	871.6	188.8	4.62×
CEN	278.4	50.8	5.48×	156.2	74.7	2.09×
Wikipedia-Links	—	—	—	278.9	—	—
Leiden CPM 0.01						
Dataset	WCC-ChSM	WCC-ChDis	Speedup	CM-ChSM	CM-ChDis	Speedup
Open-Alex	1962.5	180.5	10.87×	1912.7	164.9	11.60×
Open-Cit.-v2	1632.1	245.9	6.64×	1721.4	259.9	6.62×
Open-Citations	1270.7	68.5	18.55×	1346.3	80.3	16.77×
CEN	229.3	9.2	24.92×	199.1	23.4	8.51×
Wikipedia-Links	304.6	—	—	372.8	—	—

The Chapel distributed implementation delivers speedups over the Chapel shared-memory reference on most successful configurations, with Livejournal WCC at CPM 0.001 as the sole small-to-medium slowdown. On small-to-medium networks, successful CPM 0.001 speedups range up to 19.7×, while CPM 0.01 speedups range from 3.0× to 55.8×. On large networks, successful CPM 0.001 speedups reach up to 5.5× on CEN for WCC and 4.6× on Open-Citations for CM, with CPM 0.01 producing gains up to 24.9× and 16.8×, respectively.

The Chapel distributed implementation also demonstrates broader graph coverage than the C++ distributed implementation. As established in Section 5.8, the C++ distributed implementation encounters consistent VieCut-related failures on Open-Alex, Open-Citations, and Wikipedia-Links, while Chapel distributed completes several Open-Alex and Open-Citations configurations that fail in C++. Wikipedia-Links remains a failure in the Chapel distributed results reported here.

At CPM 0.001, speedups on large datasets remain in the 1.2×–5.5× range, reflecting two factors. First, the Chapel shared-memory baseline is already highly optimized, leaving less room for improvement through distribution alone. Second, the round-robin workload distribution does not account for cluster size imbalance, so some locales complete early while others remain occupied with large outlier clusters. The load-balanced `rr_npm` strategy presented in Section 6.5 addresses this and yields further improvements.

6.5 Load-balanced Chapel Distributed Performance

Tables 8 and 9 compare the `rr_npm` load-balanced strategy against the round-robin Chapel distributed baseline from Tables 6 and 7.

Table 8 Chapel Distributed Runtime (seconds) with `rr_npm` load-balanced distribution on Small to Medium Datasets. 32 cores × 4 nodes.

Leiden CPM 0.001						
Dataset	WCC-ChDis	WCC-LB	Speedup	CM-ChDis	CM-LB	Speedup
Bitcoin	36.2	29.6	1.22×	56.7	53.6	1.06×
Livejournal	103.9	57.5	1.81×	21.9	13.1	1.67×
Cit-Patents	46.3	25.9	1.79×	8.9	5.9	1.51×
Orkut	58.6	51.5	1.14×	30.6	22.3	1.37×
Hyves	3.6	2.5	1.44×	8.8	7.6	1.16×
Leiden CPM 0.01						
Dataset	WCC-ChDis	WCC-LB	Speedup	CM-ChDis	CM-LB	Speedup
Bitcoin	6.3	3.4	1.85×	7.3	4.7	1.55×
Livejournal	6.7	2.8	2.39×	8.8	4.0	2.20×
Cit-Patents	3.2	1.9	1.68×	4.6	2.2	2.09×
Orkut	6.2	5.1	1.22×	7.2	5.6	1.29×
Hyves	1.4	0.9	1.56×	4.1	3.7	1.11×

The `rr_npm` strategy reduces total runtime on most datasets by 1.1× –2.4× on small-to-medium graphs and up to 2.7× on large ones. Gains are larger at CPM 0.01 than at 0.001. This is because finer-grained clustering produces more clusters with

Table 9 Chapel Distributed Runtime (seconds) with `rr_npm` load-balanced distribution on Large Datasets. 32 cores × 4 nodes. Dashes indicate failure (see Section 5).

Leiden CPM 0.001						
Dataset	WCC-ChDis	WCC-LB	Speedup	CM-ChDis	CM-LB	Speedup
Open-Alex	894.1	823.6	1.09×	—	—	—
Open-Cit.-v2	322.7	293.4	1.10×	277.2	266.9	1.04×
Open-Citations	360.5	246.5	1.46×	188.8	68.8	2.74×
CEN	50.8	32.8	1.55×	74.7	86.9	0.86×
Wikipedia-Links	—	—	—	—	—	—
Leiden CPM 0.01						
Dataset	WCC-ChDis	WCC-LB	Speedup	CM-ChDis	CM-LB	Speedup
Open-Alex	180.5	184.7	0.98×	164.9	114.3	1.46×
Open-Cit.-v2	245.9	183.8	1.34×	259.9	217.2	1.20×
Open-Citations	68.5	35.1	1.95×	80.3	39.7	2.02×
CEN	9.2	4.1	2.24×	23.4	18.8	1.24×
Wikipedia-Links	—	—	—	—	—	—

greater size variations, which is exactly what sorted round-robin is designed to exploit. Livejournal and Open-Citations show the largest improvements at both resolutions; Orkut and Hyves, where clusters are more uniform in size, show the smallest.

Two results fall outside the general trend. On Open-Alex WCC at CPM 0.01, `rr_npm` finishes in 184.7s against a 180.5s baseline, indicating that one very large cluster can still dominate the critical path regardless of the assignment strategy. On CEN CM at CPM 0.001, `rr_npm` takes 86.9s against a 74.7s baseline, a 16% slowdown. We therefore report these cases as exceptions to the general trend rather than claiming uniform improvement. Wikipedia-Links has no results in either table. It crashed with SIGSEGV, consistent with the VieCut-related failure behavior discussed in Section 5.8.

7 Conclusion

This paper presented distributed-memory parallel implementations of the Well-Connected Clusters (WCC) and Connectivity Modifier (CM) algorithms, extending our prior shared-memory Chapel implementations [22] to multi-node execution in both C++ with MPI and Chapel with multi-locale support. The central architectural contribution is the integration of cluster subgraph generation into the Leiden clustering step, eliminating the separate subgraph-construction preprocessing pass used in the original baseline. By the time clustering is complete, the distributed WCC and CM pipeline can begin with subgraph files already materialized on disk, requiring no separate preprocessing pass and no node ever loading the full input graph. A load-balanced cluster distribution strategy `rr_npm` further improves runtime on most successful configurations, with gains up to 2.7× over plain round-robin assignment.

The C++ distributed implementations achieve substantial speedups over the C++ baseline on graphs where both complete successfully. In particular, the distributed queue variant (WCC-DQ) completes Open-Citations-v2 in 72 seconds at CPM 0.01, a graph on which the C++ baseline fails while the Chapel shared-memory implementation completes but is substantially slower. The Chapel distributed implementation

delivers speedups over the Chapel shared-memory reference on most successful configurations, reaching up to 19.7× at CPM 0.001 and 55.8× at CPM 0.01. It also demonstrates broader graph coverage than the C++ distributed implementation, completing several Open-Alex and Open-Citations configurations on which the C++ distributed implementation fails. Wikipedia-Links remains a failure in the Chapel distributed results reported here.

These results show that the distributed architecture provides meaningful scalability gains for connectivity-preserving community refinement on large-scale network data. By eliminating the separate subgraph-construction pass and distributing the minimum-cut workload across nodes, the proposed pipeline extends WCC and CM beyond the limitations of the original single-node baseline. The evaluation uses a fixed four-node distributed configuration; a systematic study of scaling with node count remains an important direction for future work.

A subset of large-graph configurations, including configurations involving Open-Alex, Open-Citations, and Wikipedia-Links, exhibit failures associated with VieCut. These failures are associated with memory leaks and data races in VieCut, arising on specific subgraph structures and memory layouts. The C++ distributed implementation encounters these failures consistently on the impacted configurations, while the Chapel distributed implementation is less affected in several cases because it materializes subgraphs with a different internal memory layout, as discussed in Section 5.8. These failures limit empirical coverage, but they do not invalidate the timing results reported for configurations that complete successfully. Our findings are being shared with the VieCut developers.

Several directions remain open for future work. First, dynamic work-stealing could further reduce load imbalance on graphs with extreme cluster size skew, particularly for the CPM 0.001 setting where gains from `rr_npm` are more modest. Second, evaluating alternative minimum-cut backends would address the current reliability limitation and may extend successful execution to the remaining benchmark configurations. Third, a native Chapel implementation of the Leiden algorithm would eliminate the foreign function interface overhead currently incurred in CM, potentially improving CM performance on small graphs where that overhead is relatively significant. Finally, extending the distributed architecture to support dynamic graph updates, relevant for temporal citation networks and evolving social networks where community structure changes over time, represents a natural continuation of this line of work.

The Chapel implementation is integrated into Arachne [24], an open-source graph analytics framework available at https://github.com/Bears-R-Us/arkouda-njit, making distributed well-connected community detection directly accessible through Arachne's Python-facing interface for practitioners working with large-scale network data.

8 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] Akoglu, L., Tong, H., Koutra, D.: Graph-Based anomaly detection and description: A survey. Data Mining and Knowledge Discovery **29**, 626–688 (2015)

[2] Fortunato, S.: Community detection in graphs. Physics Reports **486**(3–5), 75–174 (2010)

[3] Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E **69**(2), 026113 (2004)

[4] 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)

[5] 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)

[6] Newman, M.E.J.: Modularity and community structure in networks. Proceedings of the National Academy of Sciences **103**(23), 8577–8582 (2006)

[7] 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), 10008 (2008)

[8] Traag, V.A., Waltman, L., Eck, N.J.: From Louvain to Leiden: Guaranteeing well-connected communities. Scientific Reports **9**(1), 1–12 (2019)

[9] Holland, P.W., Laskey, K.B., Leinhardt, S.: Stochastic blockmodels: First steps. Social Networks **5**(2), 109–137 (1983)

[10] Abbe, E.: Community detection and stochastic block models: Recent developments. Journal of Machine Learning Research **18**(177), 1–86 (2018)

[11] Peixoto, T.P.: Bayesian Stochastic Blockmodeling. Advances in Network Clustering and Blockmodeling, 289–332 (2019)

[12] 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)

[13] Benson, A.R., Gleich, D.F., Leskovec, J.: Higher-order organization of complex networks. Science **353**(6295), 163–166 (2016)

[14] 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)

[15] Kannan, R., Vempala, S., Vetta, A.: On clusterings: Good, bad and spectral. Journal of the ACM (JACM) **51**(3), 497–515 (2004)

[16] 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) https://doi.org/10.1371/journal.pcsy.0000009

[17] Fortunato, S., Newman, M.E.J.: 20 years of network community detection. Nature Physics **18**(8), 848–850 (2022)

[18] 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 (2024). Springer

[19] Vu-Le, T.-A., Park, M., Chen, I., Warnow, T.: Using stochastic block models for community detection. Applied Network Science **11**(1), 2 (2026)

[20] 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)

[21] Caetano Machado Lopes, L., Chacko, G.: A citation graph from OpenAlex (Works). University of Illinois Urbana-Champaign (2024). https://doi.org/10.13012/B2IDB-7362697_V1

[22] Dindoost, M., Rodriguez, O.A., Bryg, B., Park, M., Chacko, G., Warnow, T., Bader, D.A.: On the optimization of methods for establishing well-connected communities. In: Proceedings of the 14th International Conference on Complex Networks and Their Applications (Complex Networks 2025) (2025). arXiv preprint arXiv:2509.02590

[23] 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)

[24] 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 (2022). IEEE

[25] Staudt, C.L., Meyerhenke, H.: Engineering parallel algorithms for community detection in massive networks. IEEE Transactions on Parallel and Distributed Systems **27**(1), 171–184 (2015)

[26] Halappanavar, M., Lu, H., Kalyanaraman, A., Tumeo, A.: Scalable static and dynamic community detection using Grappolo. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2017). IEEE

[27] Sahu, S., Kothapalli, K., Banerjee, D.S.: Fast Leiden algorithm for community detection in shared memory setting. In: Proceedings of the 53rd International Conference on Parallel Processing, pp. 11–20 (2024)

[28] Ghosh, S., Halappanavar, M., Tumeo, A., Kalyanaraman, A., Lu, H., Chavarria-Miranda, D., Khan, A., Gebremedhin, A.: Distributed Louvain algorithm for graph community detection. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 885–895 (2018). IEEE

[29] Que, X., Checconi, F., Petrini, F., Gunnels, J.A.: Scalable community detection with the Louvain algorithm. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 28–37 (2015). IEEE

[30] Sattar, N.S., Arifuzzaman, S.: Scalable distributed Louvain algorithm for community detection in large graphs. The Journal of Supercomputing **78**(7), 10275–10309 (2022)

[31] Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135–146 (2010)

[32] Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 599–613 (2014)

[33] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed Graph-Parallel computation on natural graphs. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 17–30 (2012)

[34] Dathathri, R., Gill, G., Hoang, L., Dang, H.-V., Brooks, A., Dryden, N., Snir, M., Pingali, K.: Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 752–768 (2018)

[35] Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: A computation-centric distributed graph processing system. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 301–316 (2016)

[36] Merrill, M., Reus, W., Neumann, T.: Arkouda: Interactive data exploration backed by chapel. In: Proceedings of the ACM SIGPLAN 6th Chapel Implementers and Users Workshop, pp. 28–28 (2019)

[37] 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 (2024). IEEE

[38] 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 (2025). IEEE

[39] Rodriguez, O.A., Buschmann, F.V., Du, Z., Bader, D.A.: Property graphs in Arachne. In: 2023 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2023). IEEE

[40] Rodriguez, O.A.: MoMo: Combining neuron morphology and connectivity for interactive motif analysis in connectomes. IEEE Transactions on Visualization and Computer Graphics (2025)

[41] Jenkins, L., Firoz, J.S., Zalewski, M., Joslyn, C., Raugas, M.: Graph algorithms in PGAS: Chapel and UPC++. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2019). IEEE

[42] Henzinger, M., Noe, A., Schulz, C.: Shared-memory exact minimum cuts. In: Proceedings of the 33rd International Parallel and Distributed Processing Symposium (IPDPS) (2019)

[43] Peixoto, T.P.: The Netzschleuder network catalogue and repository. Accessed: August 17, 2025 (2020). https://doi.org/10.5281/zenodo.7839981

[44] Park, M., Tabatabaee, Y., Warnow, T., Chacko, G.: Data for well-connectedness and community detection. University of Illinois Urbana-Champaign (2024). https://doi.org/10.13012/B2IDB-6271968_V1

[45] Mohasel Arjomandi, H., Korobskiy, D., Chacko, G.: Parsed Open Citations and PubMed Data. University of Illinois at Urbana-Champaign (2024). https://doi.org/10.13012/B2IDB-5216575_V1

[46] Park, M., Tabatabaee, Y., Warnow, T., Chacko, G.: Data for well-connected communities in real networks. University of Illinois at Urbana-Champaign (2023). https://doi.org/10.13012/B2IDB-0908742_V1

[47] Kunegis, J.: KONeCT: The Koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1343–1350 (2013)