A Parallel LFR-like Benchmark for Evaluating Community Detection Algorithms

George M. Slota and Jack Garbus
Rensselaer Polytechnic Institute, Troy, NY
slotag@rpi.edu, garbuj@rpi.edu

***Abstract*—Community detection is one of the most widely-used graph analytics. As recent community detection algorithms have been targeting large-scale networks, an emerging problem is how best to evaluate the output of these algorithms. Common measures such as modularity have several well-known issues, so comparisons against a notion of a "ground truth" community structure, such as in the Lancicinetti-Fortunato-Radicchi (LFR) benchmark, is preferred. This current work targets the parallel generation of graphs matching the specifications of the LFR benchmark. We are able to generate such graphs at the billion-edge scale in seconds, giving orders-of-magnitude speedup relative to prior work.**

***Index Terms*—community detection; LFR benchmark; graph generation**

I. Introduction

Community detection (CD) is a common graph analytic covering a wide variety of applications [6]. Loosely defined, the problem of community detection is to decompose a graph into dense clusters, which can be possibly overlapping or hierarchical in structure. Evaluating the output quality of CD algorithms is deceptively non-trivial. It is common to measure the output of a CD algorithm relative to global measures such as modularity [18], though other measures such as conductance or edge cut are also used [11]. There are several known issues with global measures like modularity, notably the resolution limit [7] or the fact that it can be often incorrectly measured on skewed simple graphs. This motivates the preference for algorithm output comparisons to an existing *ground truth* community structure. When a ground truth does exist for a given dataset, the output of an algorithm can be directly related to this ground truth via a comparative measure like normalized mutual information (NMI).

As most real-world datasets lack a ground truth, synthetic benchmarks have been developed to incorporate an inherent community structure. The most commonly-used benchmark is that of Lancichinetti, Fortunato, and Radicchi [15], referred to as the "LFR Benchmark". This benchmark generates a graph or n vertices and m edges matching a degree distribution and community size distribution, which usually follow power-law curves. The benchmark utilizes a *mixing parameter* (μ), which controls the average ratio of inter- to intra-community edges. As the mixing parameter increase, the relative number of edges going between communities increases, and the output quality of a given CD algorithm is expected to decrease. Algorithms can then be compared on the basis of NMI as a function of this mixing parameter.

The primary issue with the use of the LFR is its inherent lack of scalability. Even the fastest implementation takes hours to generate a single instance of a multi-billion edge graph [12]. For comparison, modern graph generators can output graphs orders-of-magnitude larger in times that are orders-of-magnitude faster [8]. As efficient implementations of CD algorithms targeting massive graph datasets become common [10], there is a significant "gap" in our ability to solve the CD problem and properly evaluate our solutions. In our prior recent work [19], we modified the BTER generator [13] to output CD benchmark graphs up to a scale of trillions of edges. However, due to inherent properties of BTER, we were unable to output graphs containing communities matching a specified distribution.

a) Contributions: This paper can be considered a spiritual successor to our recent prior work. This work can generate HPC-scale networks that correspond to the baseline LFR specification, matching input degree and community size distributions. To do this, we implement a two-layered hierarchical strategy, where we first generate intra-community edges before overlaying a set of inter-community edges to create our final graph. In this work, we describe shared-memory and distributed-memory approaches for our method that both scale to generating a full benchmark suites of graphs at the billion-edge scale in mere minutes. Our code will soon be released pending internal copyright approvals[1].

II. Benchmark Graph Generation

We describe two hierarchical methods for generating LFR-benchmarks at HPC-scale in shared and distributed memory. Our methods are termed CMCL (Configuration Model Chung-Lu) for shared-memory and TLCL (Two Level Chung-Lu) for distributed-memory. These methods each have four distinct phases: 1) Initialization of input distributions, 2) Vertex to community assignment, 3) Internal edge generation, 4) External edge generation. We avoid self-loops as well as multi-edges and can make our code run deterministically if serialized with a given seed. The total work complexity for our methods are both $O(m)$, with parallel complexity approximately of $O(\frac{m}{t})$ on t threads or ranks.

III. Phase 1: Initialization of Input Distributions

The inputs to the primary CMCL and TLCL generation phases include target degree and community size distributions

[1]https://github.com/HPCGraphAnalysis/SAGE

and the mixing parameter μ. We generate degree distributions following a power-law or generalized log-normal distribution, but we can also parse them from real-world graphs. We generate (power-law) community-size distributions similarly to LFR.

IV. Phase 2: Community Assignment

A. CMCL: *Exact Assignment*

We use an exact assignment method with CMCL similar to LFR, creating a one-to-one mapping of vertex identifiers to community slots, such that each vertex v is in a community of size $d_{inter}(v) + 1$ or greater, where $d_{inter}(v) = d(v) \times (1 - \mu)$ is the *internal degree* of a vertex relative to its community. We assign vertices in parallel, using atomic read-modify-write updates to validate *remaining* vacancies in each community.

B. TLCL: *Probabilistic Assignment*

We develop a probabilistic interpretation of the CMCL approach for TLCL, where we can instead do lock-free assignments in distributed memory. Our approach is given in Algorithm 1. S and N are the community and degree distribution, c_{num} is the number of communities, d_{max} is the maximum degree, and I is the start vertex ids for each degree – we set global vids is increasing order of degree.

Algorithm 1 Parallel Probabilistic Assignment Method.

```
1:  procedure ProbAssign(S, c_num, N, I, d_max)
2:      R ← Copy(S)                                  ▷ Expected Remaining Space
3:      for d = d_max ... 1 do
4:          s ← FindSmallest(S, d)
5:          P ← PrefixSums(R_{s...c_num})
6:          P ← P/P(d_max)
7:          for v = I(d) ... I(d+1) do in parallel
8:              r ← Rand(0, 1)
9:              c ← BinarySearch(P)
10:             C(v) ← c                             ▷ Finalize assignment
11:         for c = s ... c_num do in parallel
12:             x ← N(d) * (P(c+1) − P(c))           ▷ Expected # to c
13:             R(c) ← R(c) − x
14:     return C
```

Similar to the shared-memory exact version, we have an array R that tracks remaining vacancies in each community. Instead of explicit decrements per assignment in C, we instead decrement each index in R after processing all vertices of degree d by the *expected* number of assignment that would have occurred.

V. Phase 3: Internal Edge Creation

A. CMCL: *Parallel Configuration Model*

For our shared-memory generator, we assign to each thread some subset of communities to generate. For edge generation, we use a configuration model generator based on a linear-complexity implementation of Havel-Hamiki. We use internal degrees $d_{inter}(v)$ for this process, but randomly round these degrees up or down to the nearest integer proportional to the fractional part of the number. We randomize edge connectivity via a single round of double edge swaps across the generated edge list [1].

B. TLCL: *Parallel Edge-skipping Chung-Lu*

To generate internal edges for TLCL, we use a shared-memory parallel edge-skipping [17] Chung-Lu generator using modified degree-pairwise probabilities from our related work [9]. Naïve Chung-Lu probabilities are based on loopy multi-graphs and result in significant error when applied to simple graph generation, particularly for the skewed and dense communities in this context. We use a hierarchical parallelization scheme, where community generation is partitioned among MPI ranks with load balance being based on the number of edges to generate. Within a rank, communities are either fully generated per-thread or parallelized across threads based off of a tunable size *cutoff*. For our results, we set the cutoff to be 10 million edges.

VI. Phase 4: External Edge Creation

For both methods we use a distributed and shared-memory edge-skipping Chung-Lu generator from our prior work [19]. In this instance, we can use the naïve Chung-Lu probabilities without significant error in the output degree distribution; we can also directly use fractional external degrees to compute these edge probabilities. To avoid multi-edge generation, we discard any internal community edges that get generated by chance. At the scale we consider, this has minimal effect on the output degree distribution.

VII. Implementation Details

All of our code is written in `C++` using `OpenMP` and `MPI` for parallelization. The code will be released into our *Scalable Graph Generation* repository on GitHub (https://github.com/HPCGraphAnalysis/SAGE) under a BSD 3-clause license, pending internal copyright approvals.

VIII. Experimental Setup

TABLE I
Test graphs. # Vertices (n), # Edges (m), average (d_{avg}) and max (d_{max}) vertex degrees, source. $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

Network	n	m	d_{avg}	d_{max}	Source
LiveJournal	2.1 M	25 M	24	2.0 K	[16]
Wikilinks	1.9 M	21 M	21	8.6 K	[14]
RMAT_26	63 M	1.1 B	33	6.7 K	[2], [5]
Twitter	39 M	1.4 B	73	56 K	[4]
Friendster	40 M	1.8 B	90	5.2 K	[16]
uk-2007	81 M	3.3 B	80	82 K	[3]

For our testing, we use several degree distributions parsed from real-world and generated datasets, listed in Table I. As is common with the LFR benchmark, we generate communities with a minimal size of 6 and vertices with a minimum degree of 5, which is accomplished by deleting lower-degree vertices from our distribution. Similarly, to enable vertex $\Longleftrightarrow$ community matching for all μ, we also delete the tail of the distributions with vertices of degree greater than $d(v) > \sqrt{n} \log n$. Our test system is the *Voltrino* testbed at Sandia National Labs. Each *Voltrino* node has 96 GB DDR, 16 GB MCDRAM high bandwidth memory, and a Knight's

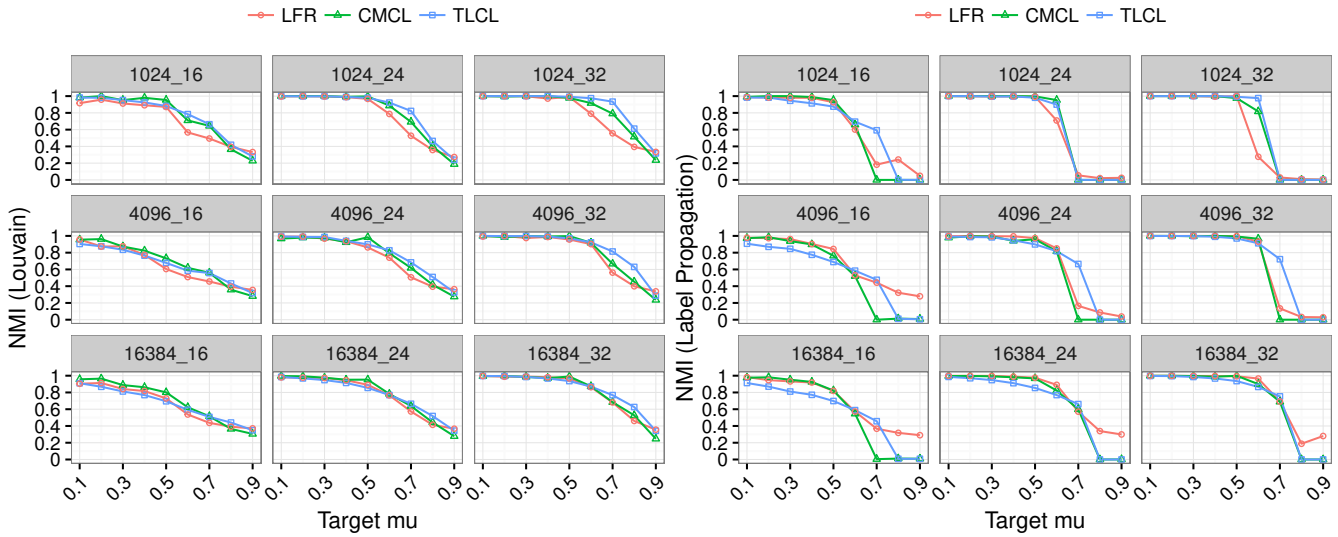


Fig. 1. NMI versus μ for Louvain (**left**) and Label Propagation (**right**). A plot title n_d_{avg} indicates a graph with n vertices and average degree d_{avg}.

Landing (KNL) processor with 68 cores. For validating on actual CD algorithms, we use efficient implementations of Label Propagation [20] and Louvain [10].

IX. Results

For our experiments, we compare outputs at the small scale against LFR, scaling performance of our implementations, and the applicability of our codes to large-scale CD benchmarking. We reproduce a number of experiments from our prior work [19] for the possibility of direct comparison.

A. LFR *Benchmark Comparisons*

We first compare the in-practice outputs of CMCL and TLCL relate to LFR. We generate LFR graphs with order $n = 1024, 4096, 16384$, $d_{avg} = 16, 24, 32$, $\mu = 0.1 \ldots 0.9$, degree distribution exponent of 2, and community size exponent of 1. We generate equivalent graphs with CMCL and TLCL, run Louvain and Label Propagation, and report the resultant NMI. We show these results in Figure 1.

We note a near close equivalence in CD algorithm output between these generators, modulo noise. This indicates the validity of CMCL and TLCL as benchmarks for this class of evaluation. The primary difference at even this relatively modest scale is generation speed. To generate all instances from Figure 1 on a Core i5 laptop in serial, LFR took 21 minutes, CMCL took 3.8 seconds, and TLCL took 6.5 seconds. This represents a 230× speedup relative to LFR for CMCL.

B. *Strong Scaling*

We next look at how CMCL strong scales on a single KNL node. We give results from 17 to 272 threads in Figure 2 (top). Times reported, in seconds, are the total time to generate all 9 graphs for $\mu = 0.1 \ldots 0.9$. We note strong scaling through all thread counts for the largest test instances, with an average speedup of 3.2× from 17 to 272 threads. The multiple phases of our algorithm limits scalability, as work imbalances through each phase accumulate through the entire program; addressing these would be a good avenue for future work. Overall, we

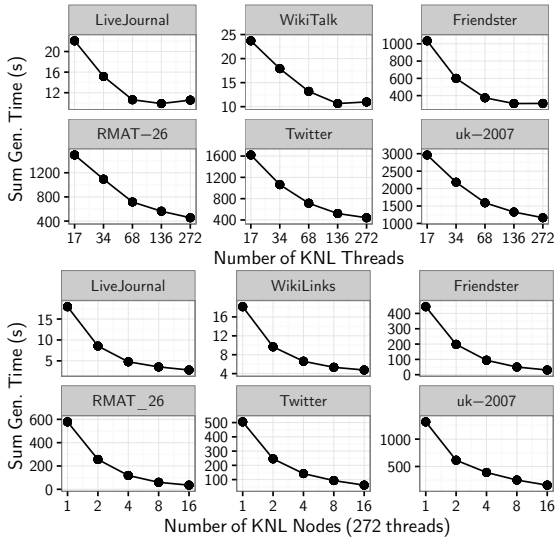


Fig. 2. **Top:** Shared memory strong scaling of CMCL on 17-272 threads of a single KNL node. **Bottom:** Distributed memory strong scaling of the TLCL generators on 1-16 KNL nodes.

note that we can generate a set of benchmark tests having over 1.8 billion edges each on a single node in under ten minutes. We measure that the LFR code outputs a single graph of $\frac{1}{1000}$th the scale in about an hour.

Figure 2 (bottom) also gives the distributed-memory strong scaling performance of TLCL from 1-16 KNL nodes on *Voltrino*. We note good strong scaling in all instances, with an average speedup of 8.6× across all test inputs. We note that the strong scaling is better in this instance due to our hierarchical approach to intra-community edge generation, which helps mitigate load imbalances in the worst cases.

C. *Large-Scale Benchmarking*

We demonstrate the efficacy of using TLCL and CMCL by running Label Propagation on all six test graph across the range of μ values. We give the resultant NMI values in Figure 3. Note that the serial LFR code does not output benchmark

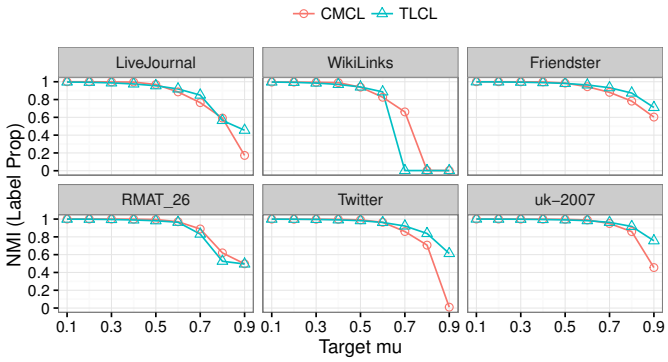


Fig. 3. Normalized mutual information output for all six test instances by running 10 iterations of Label Propagation (**Top**).

graphs at a scale within orders-of-magnitudes of our largest instances. Even the most recent scalable implementation of LFR by Hamann et al. requires 17 hours to generate a single test instance with 10 billion edges. We can generate a single test instance of 3.3 billion edges of the uk-2007 graph in 10 seconds on 16 nodes.

D. Community Size Generation Accuracy

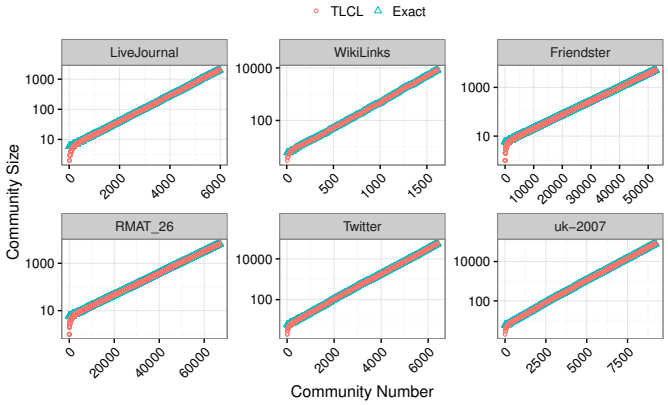


Fig. 4. Achieved versus exact community sizes for all six test distributions for the probabilistic method in TLCL.

We include one final experiment to validate our distributed probabilistic method for community assignment. In Figure 4, we overlay the community sizes generated by our TLCL method versus the exact method used by CMCL, which is similar to that of LFR. We note that the probabilistic method performs quite admirably, with only a slight hint of "noise" at the lower end of the distribution.

X. Discussion

The fastest prior work [12] generates multi-billion edge LFR instances on the order of hours. In this current work, we generate the same scale but on the order of seconds. In future work, we intend to modify our methods to allow for overlapping or hierarchical communities, exactly match the LFR specification for CMCL, and investigate the performance of CD algorithms at the large scale.

XI. Acknowledgements

This research was partially funded through the Laboratory Directed Research and Development (LDRD) program at Sandia. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] Y. Artzy-Randrup and L. Stone, "Generating uniformly distributed random networks," *Physical Review E*, vol. 72, no. 5, p. 056708, 2005.
[2] D. A. Bader and K. Madduri, "GTgraph: A synthetic graph generator suite," 2006.
[3] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, 2004, pp. 595–601.
[4] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2010.
[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.
[6] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.
[7] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.
[8] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. von Looz, "Communication-free massively distributed graph generation," *Journal of Parallel and Distributed Computing*, vol. 131, pp. 200–217, 2019.
[9] J. Garbus, C. Brissette, and G. M. Slota, "Parallel generation of simple null graph models," in *ParSocial Workshop*, 2020.
[10] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 885–895.
[11] D. F. Gleich and C. Seshadhri, "Vertex neighborhoods, low conductance cuts, and good seeds for local community methods," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 597–605.
[12] M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner, "I/o-efficient generation of massive graphs following the LFR benchmark," *J. Exp. Algorithmics*, vol. 23, no. 1, pp. 2.5:1–2.5:33, Aug. 2018. [Online]. Available: http://doi.acm.org/10.1145/3230743
[13] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C424–C452, 2014.
[14] J. Kunegis, "KONECT – the koblenz network collection," in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350.
[15] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, no. 4, pp. 1–5, Oct. 2008.
[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[17] J. C. Miller and A. Hagberg, "Efficient generation of networks with given expected degrees," in *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 2011, pp. 115–126.
[18] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.
[19] G. M. Slota, J. Berry, S. D. Hammond, S. Olivier, C. Phillips, and S. Rajamanickam, "Scalable generation of graphs for benchmarking hpc community-detection algorithms," in *SC*, 2019.
[20] G. M. Slota and S. Rajamanickam, "Experimental design of work chunking for graph algorithms on high bandwidth memory architectures," in *IPDPS*, 2018.