Scalable Generation of Graphs for Benchmarking HPC Community-Detection Algorithms

George M. Slota
slotag@rpi.edu
Rensselaer Polytechnic Institute

Jonathan W. Berry
jberry@sandia.gov
Sandia National Laboratories

Simon D. Hammond
sdhammo@sandia.gov
Sandia National Laboratories

Stephen L. Olivier
slolivi@sandia.gov
Sandia National Laboratories

Cynthia A. Phillips
caphill@sandia.gov
Sandia National Laboratories

Sivasankaran Rajamanickam
srajama@sandia.gov
Sandia National Laboratories

ABSTRACT

Community detection in graphs is a canonical social network analysis method. We consider the problem of generating suites of terascale synthetic social networks to compare the solution quality of parallel community-detection methods. The standard method, based on the graph generator of Lancichinetti, Fortunato, and Radicchi (LFR), has been used extensively for modest-scale graphs, but has inherent scalability limitations.

We provide an alternative, based on the scalable Block Two-Level Erdos-Renyi (BTER) graph generator, that enables HPC-scale evaluation of solution quality in the style of LFR. Our approach varies community coherence, and retains other important properties. Our methods can scale real-world networks, e.g., to create a version of the Friendster network that is 512 times larger. With BTER's inherent scalability, we can generate a 15-terabyte graph (4.6B vertices, 925B edges) in just over one minute. We demonstrate our capability by showing that label-propagation community-detection algorithm can be strong-scaled with negligible solution-quality loss.

ACM Reference Format:
George M. Slota, Jonathan W. Berry, Simon D. Hammond, Stephen L. Olivier, Cynthia A. Phillips, and Sivasankaran Rajamanickam. 2019. Scalable Generation of Graphs for Benchmarking HPC Community-Detection Algorithms. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356206

1 INTRODUCTION

A community detection (CD) algorithm typically partitions the vertex set of a graph into subsets of vertices called communities, where nodes in the same community are more closely connected to each other than to the rest of the graph. There are many variants allowing community overlap, hierarchical communities, the local community of a node, alternative definitions of communities, etc. However, we focus on the basic problem of partitioning the vertex set into subsets. Finding communities is a useful data mining step in social networks. For example, Papadopoulos et. al. survey applications of CD in social media including recommendation systems and event detection [22]. With the increasing scale of social network datasets, algorithm researchers are developing parallel CD algorithms with an emphasis on scalability. Our work will enable them to better assess changes to solution quality as they scale.

There are two primary methods for measuring solution quality in community detection. The first method, to date the only alternative for HPC-level scalability studies, is to use optimization objective functions such as modularity [21] or conductance[1]. When comparing CD algorithms, the one that produces a community assignment with the highest objective value wins. This approach has drawbacks. As the qualitative definition of communities implies, there is still no universally accepted notion of a correct community assignment based on an optimization objective function. The popular modularity objective is a statistical measure: how many more edges are inside communities than expected for a random graph with the same communities and vertex degrees. However, solutions that maximize modularity provably may not resolve communities smaller than $\sqrt{m/2}$, where m is the number of edges in the graph [11]. Real human communities are much smaller than that. Furthermore, an Erdős-Rényi (ER) random graph is conceptually a single community. However, with small expected variations in density, the maximum-modularity communities for an ER graph will have more than one community. While variants of modularity or measures related to conductance might improve upon straight modularity, squeezing the last bit of objective quality out of an assignment is not consistently correlated with "better" communities.

The second method, based on suites of graphs in the style of LFR, which we term VCCS-based (for Varying-Community-Coherence Suites"), compares algorithm solutions to an engineered ground truth. The seminal Lancicinetti, Fortunato, and Radicchi (LFR) graph generator [18] was the first to enable this approach. LFR graphs have a notional set of "ground truth" communities and a parameter that determines their coherence or tightness (the ratio of external [intra-community] edges to total edges). We note that these are not truly ground truth communities and use the term "Engineered Approximate Solution" (EAS) to describe them. When comparing CD algorithms, the one that produces a community assignment closest to the EAS over a range of tightness values wins. One measure of the similarity between two community assignments is normalized mutual information (NMI), where higher values are closer matches. A VCCS-based comparison between two CD algorithms plots NMI

SC '19, November 17–22, 2019, Denver, CO, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356206

[1]The conductance of a community that is small compared to the whole graph is the ratio: (number of edges with exactly one endpoint in the community)/(total degree of all nodes in the community).

as a function of the EAS tightness parameter. If one algorithm is consistently higher in NMI than the other, then it wins. Community comparison is an active research area and NMI has some known flaws [33]. Researchers can use other comparison methods, but NMI is still one of the most frequently-used.

To date, VCCS-based evaluation has been limited to suites of LFR graphs. In this paper, we demonstrate VCCS-based evalution at HPC scale (terascale) with a different and more scalable graph generation strategy.

The VCCS-based approach is much more resource-intensive than modularity maximization. A VCCS-based study involves generating many graphs, executing many CD algorithm runs, and comparing the results of each to the EAS. No current graph generator supports VCCS-based studies at the terabyte or greater scale. In Section 2, we acknowledge previous works that generate large LFR graphs in external memory, that generate truly huge graphs (lacking EAS) at HPC-scale, or that generate scaled-up versions of existing graphs (again lacking EAS). None of these options admit VCCS-based studies at scale.

We provide a scalable graph generator that enables VCCS-based studies at HPC-scale. Our approach is based upon the Block Two-Level Erdős-Rényi (BTER) generator [28]. This generator accepts two inputs: a degree distribution and a per-degree clustering coefficient distribution[2]. These distributions could be obtained from a real graph, or could be synthetic. The BTER generator groups vertices into "affinity blocks," forms an Erdős-Rényi graph [9] within each block, and connects these blocks using the degree-biased approach of Chung and Lu [8]. See Section 3 for a more detailed description. The BTER authors [28] also prove a theorem stating that a constant fraction of the edges in a real-life community are contained in a dense Erdős-Rényi graph. The BTER affinity blocks model these community kernels. The BTER generator is scalable to HPC sizes [16], but it lacks a tunable tightness parameter to support VCCS-based studies.

Our approach, A-BTER (for Adapted BTER), uses BTER as a black box, providing it with inputs that give us sets of graphs that enable VCCS-based studies at any scale fitting into a supercomputer's main memory. For example, we can generate A-BTER graphs with almost a trillion edges in just over one minute on 512 nodes (14,336 cores) of an ARM supercomputer. Thus we can generate a full test suite for a VCCS-based study in a reasonable amount of time, and regenerate if necessary without significant disruption to a CD algorithm study.

Contributions. Our main contributions are 1) The A-BTER generation strategy, enabling VCCS-based algorithm studies at HPC-scale, 2) Methods for scaling the degree and clustering-coefficient distributions of real graph to much larger sizes, 3) Demonstration of a VCCS-based study on two different CD algorithms, and 4) A parallel $O(n)$-time algorithm to compute NMI to compare two community assignments on an n-vertex graph. Previous algorithms might require $\Omega(n^2)$ time.

[2]A vertex v of degree d_v could potentially be part of $\binom{d_v}{2}$ triangles. The clustering coefficient of v is the fraction of possible triangles on v that are in the graph. The per-degree clustering coefficient distribution of a graph gives the average clustering coefficient over all nodes of degree d, for all d. The clustering coefficient of a graph is the fraction of possible triangles in the whole graph, given the degree distribution.

2 RELATED WORK

Several graph generators are capable of producing graphs at HPC scale. The Graph-500 competition is based on R-MAT graphs [7], but these were shown in [28] to have flat per-degree clustering coefficient distributions, i.e., they have no communities. The award-winning generator of Funke *et. al.* [12] can generate enormous graphs based on random graph models like random geometric or hyperbolic graphs. Such graphs may have communities, but there is no engineered approximate solution. Sanders *et. al.* give Kronecker-based generators that produce instances of arbitrary size and known triangle (3-cycle) count [27]. This is useful for evaluating triangle counting and enumeration algorithms, but does not admit VCCS-based CD algorithm comparisons.

Significantly departing from these approaches in spirit, the Evo-Graph generator of Park and Kim [23] directly produces scaled-up versions of a given graph. Suppose that the original graph has an engineered approximate solution (for example, it could be an LFR or BTER instance). EvoGraph can produce a version of that graph at huge scale while approximately preserving clustering-coefficient distributions. However, we have noted a property of EvoGraph that is undesireable for our application: each triangle in the scaled graph has a representative triangle in the original graph and connects to it in a structured way. This structure tends to produce sparser communities at scale. There is no clear way to control the tightness of communities, so we cannot directly use EvoGraph for VCCS-based studies.

Hamann *et. al.* have the most efficient published methods to exactly generate LFR graphs [14]. These are external-memory algorithms. They can generate LFR graphs with 37 billion edges in 17 hours on a single multicore with 1TB of SSD. This is orders of magnitude better than the original LFR generator. Their contribution is the ability to generate huge graphs on workstations with SSD drives via external-memory algorithms. This is modest hardware many researchers own. However, we require a different graph-generation method to support HPC-scale VCCS-based studies.

3 BACKGROUND

The LFR benchmark accepts the number of nodes n, a reference degree distribution, a reference community-size distribution and a desired community "tightness" parameter μ. The goal is to generate a graph with the appropriate degree and community size distributions (assumed to be power law) such that the average over all vertices of the fraction of inter-community edges is approximately μ. The LFR generator draws degrees for each node and creates a graph using the configuration model. It draws community sizes from the distribution, reducing the size of the last community if necessary so the sum of the community sizes equals n. It assigns nodes randomly to communities with the constraint that the community is large enough to contain the desired number of intra-community edges. There is some iterating and shuffling to maintain exact community sizes. Finally, LFR performs rewiring to approximately reach the desired community density μ. Rewiring swaps endpoints between a pair of edges to preserve degrees while improving the average μ. This rewiring requires many iterations, is the most expensive step, and is not guaranteed to finish. The open-source LFR code performs rewiring for some suitable time given a community assignment, then gives up and tries another community assignment. Rewiring swaps one pair of edge endpoints at a time, which can limit parallel scalability. We are not aware of any LFR algorithms that rewire multiple pairs simultaneously.

The typical way to use LFR to compare two community-detection (CD) algorithms is to generate a set of graphs with varying values of μ. Communities become quite ill-defined by $\mu = 0.7$. Each CD algorithm computes communities for these graphs and compares its solution to the EAS. The algorithm with solutions closer to LFR as μ grows is considered superior.

The comparison step is itself an active and unresolved research area. Vinh, et al. survey several information-theoretic measures for doing this comparison and assess their metric and normalization properties, as well as corrections for chance [33]. It is not our goal here to endorse any particular comparison method. We choose to present our results using normalized mutual information (NMI) as a representative method and scalable alternative, and we note that other methods could be substituted.

The BTER graph generator accepts a degree distribution and a per-degree clustering coefficient distribution. The clustering coefficient (CC) for a vertex is the fraction of its pairs of neighbors that close into triangles. The per-degree CC, c_d, for degree d is the average CC over all nodes of degree d. This input to BTER can come from goal distributions or from real graphs. It tends to be high among low-degree vertices for graphs with communities. Since BTER does not try to meet a goal μ, its edge-generation process is more direct, simpler, and faster than LFR's. BTER creates *affinity blocks*, which are groupings of vertices with similar degrees, usually $d+1$ vertices of degree d. For an affinity block with minimum degree d (with goal CC c_d), BTER adds each possible edge with probability $p = \sqrt[3]{c_d}$. It then joins blocks with a Chung-Lu-style [8] process. The desired degree of a node in this step is its remaining degree (original minus its generated within-block edges). We treat affinity blocks as our EAS.

BTER was developed to generate many random graphs that approximately match a real graph's degree and clustering-coefficient distributions. We wish to alter these distributions to approximate the LFR μ parameter or to vary the graph's size. Given the degree distribution n_d, for n total nodes, and the per-degree clustering coefficients, c_d, we can compute a "native" μ for the graph BTER will generate. Assume the affinity blocks all have $d + 1$ nodes of degree d, even though this is not generally true for the few larger communities. The average fraction of inter-community edges is:

$$\mu_{\text{BTER}} = \frac{\sum_{d \in D} n_d (1 - \sqrt[3]{c_d})}{\sum_{d \in D} n_d} = 1 - \frac{\sum_{d \in D} n_d (\sqrt[3]{c_d})}{n}$$

where $D = \{d \mid n_d > 0\}$. By adjusting the clustering-coefficient distribution, we can alter the effective μ for the BTER graph.

We consider two CD algorithms. Louvain [4] is the most popular CD algorithm. It starts with all nodes in independent communities and iteratively moves each vertex into the community of a neighbor that maximizes modularity. However, it attempts to address the resolution limit by using a hierarchy of nested communities. Label propagation is a similar algorithm except that each node moves to the community that the majority of its neighbors belong to. It is easily parallelizable on our target architecture [29].

4 A-BTER FOR HPC-SCALE BENCHMARK GRAPH GENERATION

We present a distributed lock-free approach called A-BTER (Adapted BTER) that uses a distributed-memory version of the BTER generator as a black box. Parallel edge generation separates the degree of a vertex into its *internal* and *external* degree and produces inter-community and intra-community graphs independently. We break the process down into four stages: 1) Initialization of input distributions, 2) Vertex-to-community assignment, 3) Internal-edge generation, 4) External-edge generation. We can avoid self-loops and multi-edges without having to hold *any* edges in memory. Our process also runs deterministically when serialized with a given seed.

Each of t compute nodes is an MPI rank. There is a thread per core within the nodes. Each compute node has a copy of the input degree and CC distributions. For generating community assignments, we partition the graph vertices evenly among the compute nodes based on vertex ID.

4.1 Phase 1: Initializing Input Distributions

The two inputs to BTER are a degree distribution and a clustering-coefficient (CC) distribution. By modifying the CC distribution input to BTER, we can control the relative number of intra-block to inter-block edges, mimicking the LFR mixing parameter (μ) when considering affinity blocks as communities. We use a linear program (LP) to do this. The LP minimizes the change in the input clustering coefficient distribution such that the output graph has a desired goal μ. Recall that μ is the proportion of inter-community edges for a vertex, averaged over all vertices. This definition is given as the primary constraint for our LP:

$$\mu = \frac{1}{n} \sum_{d} \left(n_d * \frac{i_d}{d} \right)$$

Here, n is the number of vertices, n_d is the number of vertices of degree d, and i_d is the expected number of inter-community edges for vertices of degree d as determined by the clustering coefficient distribution:

$$\begin{aligned} p_d &= \sqrt[3]{c_d} \\ i_d &= d * (1 - p_d) \end{aligned}$$

The Erdős-Rényi (ER) probabilities p_d for degree d for BTER's first stage are derived from c_d, the clustering coefficient for vertices of degree d. Due to the cubic relationship between clustering coefficients and BTER's ER probabilities (and our primary constraints), we cannot create an LP that directly minimizes the shift in clustering coefficients. Instead, we minimize the absolute shift in the probabilities. Let μ_g be our goal community coherence. The full LP to compute the new ER probabilities $\hat{p}_d$ is:

$$\begin{aligned}
\text{minimize} \quad & \sum_d |\hat{p}_d - p_d| \\
\text{subject to} \quad & \sum_d n_d \hat{p}_d = n(1 - \mu_g) \\
& 0 \le \hat{p}_d \le 1 \\
& |\hat{p}_d - p_d| \ge |\hat{p}_{d+1} - p_{d+1}| \\
& |\hat{p}_d - \hat{p}_{d+1}| \le 0.01 \\
\text{output} \quad & \hat{c}_d = \hat{p}_d^3
\end{aligned}$$

We include two additional *smoothing constraints*. One restricts the change in $\hat{p}_d$ from p_d to decrease with increasing degree. The other restricts the absolute magnitude of change in output probabilities from $\hat{p}_d$ to $\hat{p}_{d+1}$. This is designed to eliminate large jumps in the resulting CC distribution, especially for low μ, occurring at lower degrees (large CC values aren't typically observed at high degrees). Large shifts away from a graph's native μ can still drastically change the shape of the distribution.

The number of variables and constraints grows linearly with length of the distributions, i.e., the maximum degree d_{max}. However, real-world distributions are usually quite sparse, and we only need to solve for degree values with a nonzero number of vertices. We also use a binning-based degree-averaging approach that can reduce variables and constraints by a log factor. Our code will be available in the github *HPCGraph*[3] codebase in order to expose such details. See Section 6 for our software dependencies. All of these are open-source, and they proved adequate for the studies in this paper. As we show in Section 8, the LP solves were not a bottleneck, taking between 5% and 25% of the total generation time. This translates to no more than a minute in the terascale studies.

4.1.1 Scaling Distributions. Static datasets such as SNAP [19] have proved extremely useful in non-HPC settings. The largest instances, such as Friendster [24] with its 1.8 billion edges, fit in RAM on large workstations. We just showed how to generate clustering-coefficient distributions to enable VCCS-based studies based on any graph. We use the graph's degree distribution for all values of μ_g. In this section, we extend this result by showing how to generate such suites at much greater (or smaller) scale. Now we must also alter the degree distribution. Section 8 includes results from graphs mimicking Friendster's degree and clustering-coefficient properties, but 512× its size (roughly a trillion edges).

We describe an empirical method for scaling degree and clustering-coefficient distributions. Other work [31] has considered scaling real graphs by using their input characteristics to adjust input parameters of random graph generators such as LFR and BTER. Our approach is similar, but generally more lightweight, as our goal is to replicate distribution properties rather than more complex graph properties. We do not attempt to infer and extend an explicit growth process.

We take original degree and clustering-coefficient distributions of a graph with n vertices, m edges, maximum degree d_{max}, average and maximum CC of c, c_{max}, and generate new distributions representing a graph of n' vertices, m' edges, d'_{max} max degree, and average and maximum CC c' and c'_{max}. As the distributions by themselves capture limited structural characteristics of the underlying graph, the metric we consider for evaluation is the inherent shape of each distribution. Our current method is heuristic. It could be replaced by more statistically rigorous methods in the future if necessary. However, empirical results suggest that our heuristic is effective (see Figure 9 in Section 8.7). Algorithm 1 gives pseudocode for scaling degree distributions. Scaling CC distributions is similar.

Algorithm 1 Scaling method for degree distributions.

```
1:  procedure ScaleDist($D, n', d'_{max}, \alpha$)
2:  ▷ Degree distribution D is a dense array where $D(d) = n_d$ and
    $\alpha$ is a small bias factor
3:      $d_{max} \leftarrow |D|$
4:      $n \leftarrow \sum_{i=1}^{d_{max}} D(i)$
5:      $P \leftarrow$ PrefixSums($D$)/$n$        ▷ Initial probability distribution
6:      $l \leftarrow (d_{max} - 1)/(d'_{max} - 1)$  ▷ Step length for interpolation
7:      $s \leftarrow 1$                      ▷ Current step for new distribution
8:      for $d = 1 \ldots d'_{max}$ do
9:          $s \leftarrow s + l$
10:         $x \leftarrow$ Floor($s$)
11:         $y \leftarrow$ Ceil($s$)
12:         $P_i(d) \leftarrow P(x) + (P(y) - P(x)) * (s - x)$
13:         $P_i(d) \leftarrow P_i(d) \times d^{\alpha}$
14:     for $d = 2 \ldots d'_{max} - 1$ do
15:         $P'(d) \leftarrow (P_i(d-1) + P_i(d) + P_i(d+1))/3$     ▷ Smooth
16:     $P' \leftarrow$ PrefixSums($P'$)/ $\sum_{i=1}^{d'_{max}} P'(i)$
17:     $D' \leftarrow \{0, 0, \ldots, 0\}$              ▷ New degree distribution
18:     for $i = 1 \ldots n'$ do
19:         $r \leftarrow$ Rand(0, 1)
20:         $d \leftarrow$ BinarySearch($r, P'$) + 1
21:         $D'(d) \leftarrow D'(d) + 1$
22:     return $D'$
```

We first transform a given degree or clustering-coefficient curve into a probability distribution P. We then interpolate points along the curve P to translate $d_{max} \rightarrow d'_{max}$ and create a new curve P_i. To change the expected average degree, or $\frac{2m'}{n'}$, we can introduce a very small positive or negative exponential bias (α) versus degree for the interpolation, such that the distance of the interpolated points can increase or decrease with degree along the curve. To hit a desired target average degree, we can use gradient descent or simply a brute force search iteratively calling the scaling algorithm. In our current code, we perform a binary search of potential α values. We perform a final smoothing step before scaling P' to create a new scaled probability distribution. We perform n' samples on P', which gives degrees for all new n' vertices for our new distribution. We use a similar approach for generating a new clustering coefficient distribution curve, with an additional final scaling step to some target c'_{max}. To modify the expected average clustering coefficient, we can again bias the interpolation on the original. We have also considered methods for fitting power-law or generalized log-normal parameters and generating synthetic instances; while such methods are good for matching e.g. a distribution's Gini Coefficient, unique characteristics of the distribution are lost.

[3]https://github.com/HPCGraphAnalysis/

4.2 Phase 2: Community Assignment

We use the native degree-ordered assignment method of BTER. Recall that vertex identifiers are ordered by degree. Neighbors within the same affinity block have consecutive vertex IDs. Thus the vertices for each affinity block are assigned to the same node, or are split across at most two nodes. During pre-processing, BTER creates and tracks the boundaries for each *group* of affinity blocks (affinity blocks with the same number of vertices). The boundary is the vertex ID where the affinity block type changes. Since each compute node knows the degree distribution, it knows the total number of groups, affinity blocks per group, and vertices per block. So each node can assign its vertices to communities. Nodes can output this ($v_{id} \rightarrow comm_{id}$) information in parallel to create a single file if desired. We can also use binary search on the group boundaries to get some vertex's block assignment. We do this to avoid explicit multi-edge generation in Phase 4.

4.3 Phase 3: Internal Edge Creation

Due to the *Coupon Collector's Problem*, the original BTER implementation as described by Kolda et al. [28] requires generation of multi-edges to fill each Erdős-Rényi affinity block to the desired amount. While this is not problematic for many native graph distributions, for CC distributions generated for low values of μ the factor of erroneous edges could be in the dozens of m. Additionally, removing multi-edges in distributed memory is non-trivial.

Therefore, we implement the *edge-skipping* technique, which can efficiently generate Erdős-Rényi [3] and Chung-Lu [20] graphs. There is a recent parallel version [1] for these random graph models that applies directly to BTER. This allows us to implement a parallel work- and memory-efficient Erdős-Rényi internal edge generation phase (described below). Per-node work complexity is $O(\frac{m_{internal}}{t})$, or simply linear in the number of edges generated. With edge-skipping, we also do not need to hold edges in memory to avoid multi-edges, so we could immediately write out generated edges to achieve a $O(d_{max})$ memory complexity. However, in our experiments, we generally retain the edge list in memory.

The baseline methods for generating an Erdős-Rényi graph perform some number of random samples over the entire space of possible edges. We'll consider this entire space as X, where $X \leftarrow \{(u,v) \in V \times V \mid u \neq v\}$ – i.e., all possible unique vertex pairs. Now consider generating m edges. Baseline methods will generate these edges by performing m random draws from all of X. With edge-skipping, we create an explicit ordering for X and sample approximately m *skip lengths*, with the average skip length about $\frac{|X|}{m}$. We start at the ordered beginning of X, move through the space of X by our sampled skip length, and select where we land as our next sampled edge. This gives us approximately m unique draws from X. Note that we don't need to explicitly hold X itself in memory. We can compute a current (u,v) pair using the current offset within X.

For parallelization, we can partition the total space of X and do independent skip-length generation on each subspace to perform edge-skipping in parallel. This is the general parallel approach from [1] we use for Erdős-Rényi edge generation within a BTER framework. Because we're generating graphs orders of magnitude larger than [1] and other prior edge-skipping implementations, we had to handle certain numerical conditions. For example, if probabilities where small enough that the expected number of edges was sufficiently close to zero, naïve skip length calculations would return infinity. For more detail into the specifics of how the ordering, skip length, offset calculations, and parallelizations are performed, see [1, 3, 20].

4.4 Phase 4: External Edge Creation

As with Phase 3, we use an edge-skipping Chung-Lu generator based on the parallelization scheme as described in [1] to create external edges. For edge-skipping with Chung-Lu graphs specifically, the key consideration in contrast with Phase 3 Erdős-Rényi generation is that edge probabilities are non-uniform with respect to vertices of differing degree. Therefore, we can define our space X of possible edges as a superset of Erdős-Rényi graph spaces for vertices of the same degree and bipartite Erdős-Rényi graph spaces for vertices of differing degrees. E.g., we define X as

$$X = \{X_{(1,1)}, X_{(1,2)}, \ldots, X_{(d_{max}-1, d_{max})}, X_{(d_{max}, d_{max})}\}$$

where $X_{(d_i,d_i)}$ is an Erdős-Rényi graph generated with edge skipping among all vertices of degree d_i. $X_{(d_i,d_j)}$ is a bipartite Erdős-Rényi graph generated among all vertices of degree d_i and d_j, where one bipartite set contains all vertices of degree d_i and the other bipartite set contains all vertices of degree d_j. The skip lengths for $X_{(d_i,d_j)}$ are based on the specific Chung-Lu edge probability calculated between degrees d_i and d_j and our total desired external edges $m_{external}$.

To avoid multi-edge generation between Phase 3 and Phase 4, we compute the community assignments of u and v from generated edge (u,v) and discard edges internal to the same block; this can be done exactly for one edge in time $O(\log(d_{max}))$ without increasing memory complexity by using BTER block offset arrays. We however use an approximate $O(1)$ per-edge scheme, where we discard edges if their vertex identifiers are closer in value than the minimum degree of the pair plus one – this is the maximum difference within BTER such that two vertices could be in the same block. In practice, we observe few erroneous edge discards. Our total Phase 4 generation time complexity is therefore linear in external edges as $O(\frac{m_{external}}{t})$. As with Phase 3, it is not necessary to hold the edge list in memory, giving us a memory complexity for Phase 4 of $O(d_{max})$.

5 EVALUATION AND PARALLEL NMI

In Section 8, we evaluate our method's scalability and accuracy in achieving a target μ and matching input distributions. We also compare them to LFR for evaluating community detection algorithms. We compare to EAS using the standard metric normalized mutual information (NMI). Consider some set of community assignments U and V each of length n and having r and c unique cluster labels, respectively. In practice, U might be the assignments output from a community detection algorithm and V the assignments given by an EAS. Additionally, consider $|U_i|$ and $|V_j|$ as the number of vertices assigned to cluster i within U and cluster j within V; $|U_i \cap V_j|$ is the number of vertices assigned to both i in U and j in V. We define entropies H_U and H_V, joint entropy $H_{U,V}$, and our normalized mutual information NMI as

$$
\begin{aligned}
H_U &= -\sum_{i=1}^{r} \frac{|U_i|}{n} \log \frac{|U_i|}{n} \\
H_V &= -\sum_{j=1}^{c} \frac{|V_j|}{n} \log \frac{|V_j|}{n} \\
H_{U,V} &= -\sum_{i=1}^{r} \sum_{j=1}^{c} \frac{|U_i \cap V_j|}{n} \log \frac{|U_i \cap V_j|}{n} \\
NMI &= \frac{H_U + H_V - H_{U,V}}{\max(H_U, H_V)}
\end{aligned}
$$

As recently as 2018 [26], researchers have claimed that computing NMI has complexity quadratic in the number of communities. While we are surprised that this technique is apparently not well known (or at least not published), we have exploited sparsity to implement an embarrasingly-parallel calculation with complexity $O(n)$, linear in the number of vertices. We validate our parallel method against the calculations given by [18][4] and obtain equivalent outputs. Our method experimentally scales linearly as expected. We can compute NMI for tens of millions of vertices and hundreds of thousands of unique clusters in seconds on an Intel Knights Landing (KNL) node.

Algorithm 2 Parallel Calculation of NMI.

```
1:  procedure CalcNMI($n, U, V$)
2:      $r \leftarrow$ NumUniqueValues($U$)
3:      $c \leftarrow$ NumUniqueValues($V$)
4:      $T \leftarrow \emptyset$                              ▷ Thread-safe hash table
5:      $A(1 \ldots r) \leftarrow (0 \ldots 0)$               ▷ Community sizes in $U$
6:      $B(1 \ldots c) \leftarrow (0 \ldots 0)$               ▷ Community sizes in $V$
7:      for $i = 1 \ldots n$ do in parallel
8:          $x \leftarrow U(i)$                           ▷ Vertex $i$'s community in $U$
9:          $y \leftarrow V(i)$                           ▷ Vertex $i$'s community in $V$
10:         $T \leftarrow$ AtomicIncrement($T, x, y$)
11:         $A(x) \leftarrow A(x) + 1$
12:         $B(y) \leftarrow B(y) + 1$
13:     $H_{U,V} \leftarrow 0.0$
14:     for $i = 1 \ldots n$ do in parallel
15:         $x \leftarrow U(i)$
16:         $y \leftarrow V(i)$
17:         $n_{xy}$ = AtomicFetchReset($T, x, y$)
18:         if $n_{xy} > 0$ then
19:             $H_{U,V} = H_{U,V} - \frac{n_{xy}}{n} \log \frac{n_{xy}}{n}$
20:     $H_U \leftarrow 0.0$
21:     $H_V \leftarrow 0.0$
22:     for $i = 1 \ldots r$ do in parallel
23:         $H_U \leftarrow H_U - \frac{A(i)}{n} \log \frac{A(i)}{n}$
24:     for $i = 1 \ldots c$ do in parallel
25:         $H_V \leftarrow H_V - \frac{B(i)}{n} \log \frac{B(i)}{n}$
26:     $NMI = \frac{H_U + H_V - H_{U,V}}{\max(H_U, H_V)}$
27:     return $NMI$
```

Given community assignments U and V as defined before, Algorithm 2 determines the marginal entropies H_U and H_V as well as the joint entropy $H_{U,V}$ to compute the final *NMI* value. In the confusion matrix/contingency table used for calculating joint entropy (tabulating which clusterings overlap and with what frequency) there can be at most n nonzeros. We replace an explicit matrix or equivalent data structure with an efficient thread-safe hash table T, which has inserted into it as a key the cluster assignments x and y for vertex i. We use an unsigned 64-bit integer to store the key, where we can pack both x and y to create a unique value ($key = (x << 32 \mid y)$), assuming $x, y < 2^{32}$. These x, y values would refer to row, column indices in the contingency table. Upon initial insertion into T, the key sets a value of 1. If the key already exists in the table, the stored value is incremented. For calculating the joint entropy, we effectively go through all unique keys in the hash table and retrieve their stored value. In lieu of explicitly tracking a list of unique x, y keys, which would require a more complex insertion procedure, we atomically reset to 0 the value of a key on the first encounter. The marginal entropies are trivial to calculate and parallelize.

6 IMPLEMENTATION DETAILS

We use C++. For A-BTER, we directly follow Kolda et. al.'s descriptions [16, 28] for creating block and group data, and generate edges with edge skipping. For Erdős-Rényi and Chung-Lu edge skipping, we follow the implementation details in [1]. We perform thread-based parallelism using OpenMP and distributed parallelism with MPI as opposed to the MapReduce approach taken by [16]. Our focus is HPC performance. Utilizing modern HPC hardware and not bounded by MapReduce overhead, our generation times are roughly 100 times faster than those presented in [16] (e.g. ~8s for 7B edges on 32 ARM nodes vs. ~900s for 4B edges on 32 Intel i7 930 nodes). We positively validate our parallel outputs against the open-source serial MATLAB code[5] for BTER, and our own serial code for our other methods. We implement the LP for A-BTER using Pyomo [15] and solve it with CBC [10], both open source. We intend to make all codes publicly available at https://github.com/HPCGraphAnalysis/.

7 EXPERIMENTAL SETUP

Table 1: Test graph characteristics. # Vertices (n), # Edges (m), average (d_{avg}) and max (d_{max}) vertex degrees, average (c_{avg}) and maximum (c_{max}) clustering coefficient, and source. $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

Network	n	m	d_{avg}	d_{max}	c_{avg}	c_{max}	Source
LiveJournal	2.1 M	25 M	24	2.0 K	0.27	0.39	[19]
Wikilinks	1.9 M	21 M	21	8.6 K	0.12	0.18	[17]
R-MAT$_{26}$	63 M	1.1 B	33	6.7 K	0.00	0.00	[2, 7]
Friendster	40 M	1.8 B	90	5.2 K	0.13	0.33	[19]
Twitter	39 M	1.4 B	73	56 K	0.07	0.49	[6]
uk-2007	81 M	3.3 B	80	82 K	0.78	0.99	[5]

Two of our experimental systems are Cray XC machines with KNL processors and Aries interconnects: a 96-node testbed cluster for small-scale experiments (*Mutrino*) and a large scale system of over 9800 nodes (*Trinity*). Each node of *Trinity* and *Mutrino* has

[4] https://sites.google.com/site/andrealancichinetti/files

[5] https://www.sandia.gov/~tgkolda/bter_supplement/

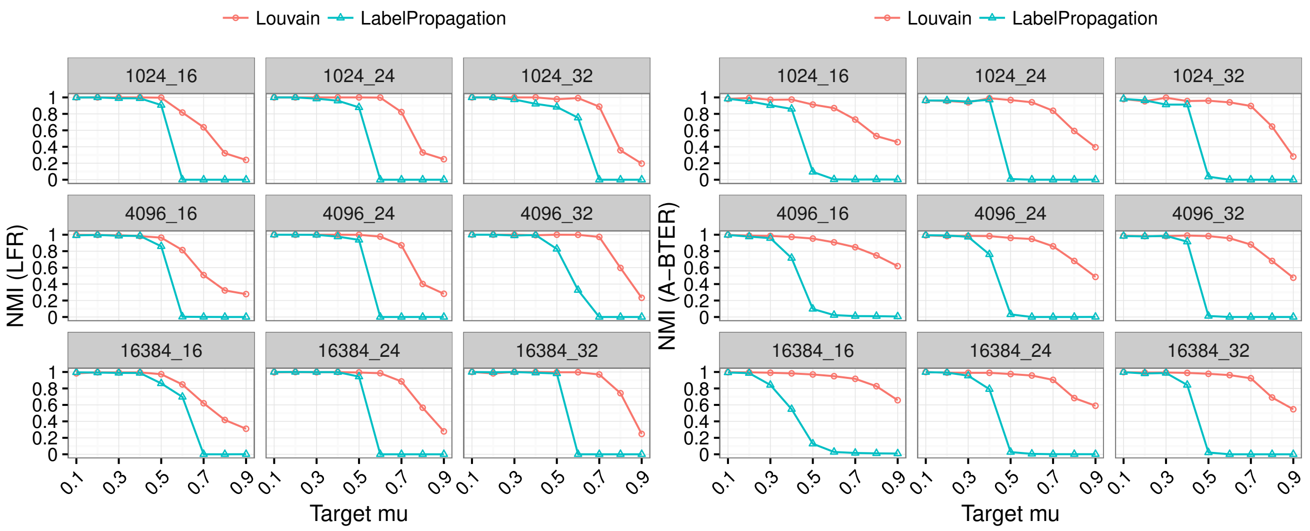

Figure 1: Comparisons of the solution quality of two algorithms, based on LFR instances (left), and A-BTER instances (right). Each pane compares runs of both Louvain and Label Propgation on a varying-density suite of graph instances. We plot NMI (higher is better), and for each pane on the left, the corresponding pane on the right consistently shows a similar trend as μ (the community coherence) changes. These results suggest that A-BTER is usable as a replacement for LFR when comparing CD algorithms. A plot title n_d_{avg} indicates a graph with n vertices and average degree d_{avg}.

96 GB DDR and 16 GB MCDRAM HBM and a KNL processor with 68 cores. On these systems, we used Cray MPICH 7.7.4 and the Intel 18.0.5 compiler with '-fopenmp -xMIC-AVX512 -O3 -std=c++11' flags. The third system is a cluster comprising over 2500 nodes with ARM processors and an HDR InfiniBand interconnect (*Astra*). Each *Astra* node has 128 GB DDR and two Marvel ThunderX2 ARM processors with 28 cores per processor. On this system we used OpenMPI 3.1.3 and the ARM HPC 19.1 compiler with '-fopenmp -mcpu=thunderx2t99 -mtune=thunderx2t99 -O3 -std=c++11' flags.

We selected well-known large-scale graph data from a number of sources to generate degree and clustering coefficient distributions. The graph properties and sources are listed in Table 1. To demonstrate the efficacy of our methods for any arbitrary distribution, we also generate a scale-26 R-MAT graph with GTGraph. We made directed graphs undirected before computing distribution properties. To make our distributions less noisy and more amenable to defined community generation, we considered only degrees $d = 5 \ldots \sqrt{n} \log n$. This pre-processing is not necessary for our methods to work. The properties after processing are listed in Table 1. To validate our generator performance in terms of community detection algorithm output, we use scalable implementations of Label Propagation [25, 30] and Louvain [4, 13].

8 RESULTS

8.1 Comparison to LFR Benchmark

For our first set of experiments, we demonstrate the close equivalence of a community detection benchmarking comparison between A-BTER and LFR. Figure 1 shows such an LFR-style comparison in terms of NMI on output by the Louvain and Label Propagation community detection algorithms. We show outputs from LFR graphs (left) and A-BTER (right). We run label propagation to convergence and a single level of Louvain to convergence. The title for each graph should be read as {number of vertices}_{average degree}.

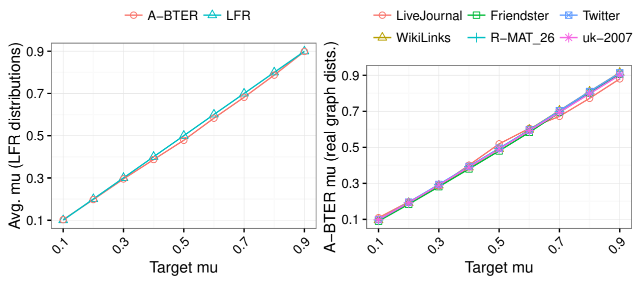

Figure 2: Average achieved μ versus target μ. We show achieved μ for LFR and A-BTER averaged over all LFR-style graphs (left) and achieved μ for A-BTER for generating graphs from real distributions (right).

To generate the LFR graphs for these experiments, we vary the number of vertices as $n = 1024, 4096, 16384$, the average degree as $k = 16, 24, 32$, the mixing parameter as $\mu = 0.1, 0.2, \ldots, 0.9$; we set the maximum degree as $maxk = \sqrt{n} \log n$, the maximum community size as $maxc = maxk + 1$, the minimum community size as $minc = 6$, the degree distribution exponent as $t1 = 2$, and the community size distribution exponent as $t2 = 1$. To generate A-BTER, we use as input the degree and clustering-coefficient distributions of LFR graphs. Note that the qualitative assessment of the two CD algorithms is very similar: Louvain is the prefered algorithm, and a distinct drop-off in label propagation is exposed. NMI-scored detection performance thus correlates well (modulo randomness in generation) between LFR and A-BTER. The drop-off in detection quality for label propagation is the result of a single label dominating the graph, a well known occurrence when label propagation is run on dense and small-diameter graphs with an ill-defined community structure.

Benchmark Generation Time. Though we closely match benchmark outputs from LFR at the small scale, the greatest benefit of our method is speed and scalability. To generate all 81 graphs represented in Figure 1 on a Core i5 laptop with one thread, LFR took 21 minutes and A-BTER took 60.5 seconds (54s for LP solutions, 6.5s for generation). This represents a 21× speedup relative to LFR for A-BTER, and this relative speedup quickly increases with graph scale.

8.2 Generation Accuracy

We examine our accuracy in hitting the target mixing parameter and target degree distribution. Figure 2 plots the average target versus achieved mixing parameter μ for the LFR graphs and LFR and A-BTER generator (left) and our six test distributions (right) for A-BTER. Overall, we hit our target mixing parameters accurately. Random deviations are a natural consequence of our BTER-style distributed edge generation, and the fact that Chung-Lu probabilities are inexact for simple graph generation [34]. We also validate our A-BTER outputs in terms of the resulting degree and clustering coefficient distributions, and we note our quality is equivalent to the performance of the baseline BTER generator; i.e., quite good.

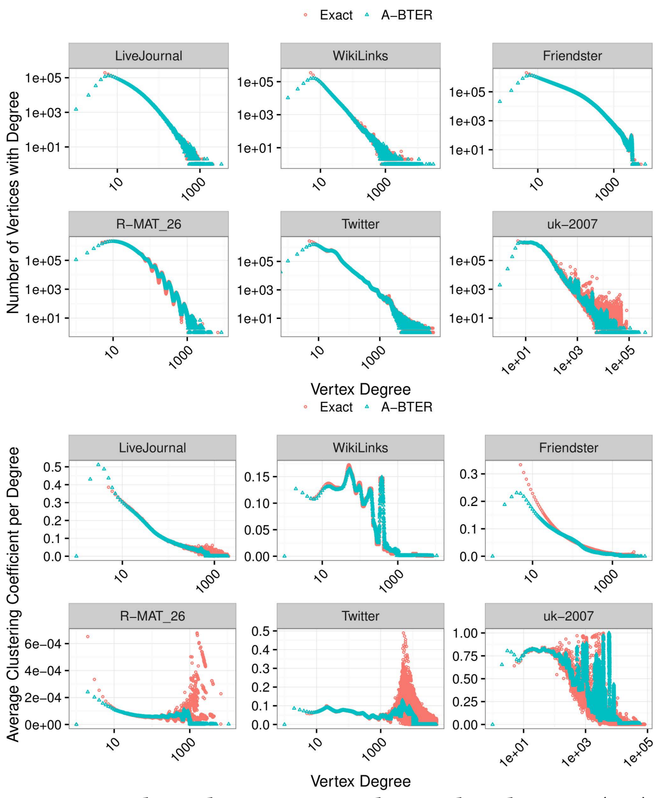

Figure 3: Achieved versus exact degree distributions (top) and clustering coefficient (bottom) for all six test distributions and the A-BTER generator.

Figure 3 shows how well our A-BTER matches a target degree and clustering coefficient distribution. Due to probabilistic generation, there is a slight dropoff below the minimum-degree threshold. However, the number of such vertices is small relative to the total number, and the rest of the distributions generally align well. We also note that in the clustering coefficient plots for e.g., Twitter and R-MAT_26 we don't perfectly achieve the extremes in the tail of the distributions; again, this is due to probabilistic generation, where a general smoothing results as vertices don't fall directly into their target degree bucket.

8.3 Scaling of A-BTER

In Figure 4, we show strong-scaling performance for A-BTER when processing all six test distributions on 1 to 16 nodes of *Mutrino*. Again, we report the total time to generate all instances varying the mixing parameter. Our reported times include initial I/O, preprocessing, and the explicit handling of community assignments, not just the edge generation phase. However, we don't include output I/O.

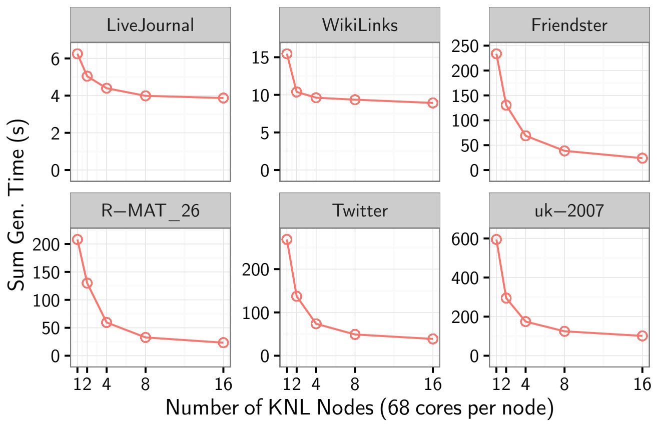

Figure 4: Distributed memory strong scaling of the A-BTER generator on 1-16 KNL nodes using all six test distributions. Times given are the sum to generate all nine benchmark graphs for $\mu = 0.1, 0.2, \ldots, 0.9$.

Overall, the strong scaling behavior is relatively good, with an average speedup of 5.8× for A-BTER when scaling from 1 to 16 nodes. We note better speedup numbers as the graphs increase in scale. There is almost a 100× difference in edges generated between LiveJournal and uk-2007, and their speedups (1.6× vs. 6×) reflect that difference.

Table 2: Test graph characteristics for distribution scaling experiments given as the minimum (2×) and maximum (16×) # edges (m), # vertices (n), max degree (d_{max}), and per-node edge generation rate (R) in millions of edges per second for 2× to 16× scale.

Network	$n_{2\times}$	$n_{16\times}$	$m_{2\times}$	$m_{16\times}$	$d_{max_{2\times}}$	$d_{max_{16\times}}$	$R_{2\times}$	$R_{16\times}$
LiveJournal	3.6 M	16 M	50 M	405 M	3.0 K	6.3 K	27	21
Wikilinks	3.5 M	17 M	40 M	325 M	45 K	106 K	11	3
R-MAT$_{26}$	101 M	437 M	2.1 B	16 B	9.8 K	20 K	37	30
Friendster	108 M	260 M	3.6 B	29 B	7.8 K	15 K	63	53
Twitter	74 M	315 M	2.9 B	22 B	96 K	150 K	51	26
uk-2007	145 M	620 M	6.6 B	52 B	98 K	220 K	50	23

Figure 5 shows additional scaling performance from 2 to 16 *Mutrino* KNL nodes for A-BTER. We use our distribution scaling

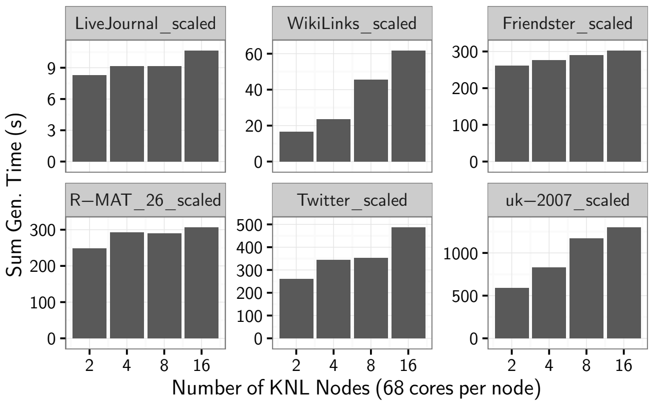

Figure 5: Distributed memory scaling of the A-BTER generator on 2-16 KNL nodes using all six test distributions with our distribution scaling methods. Each point is the total time to generate all nine benchmark graphs for $\mu = 0.1, 0.2, \ldots, 0.9$. The amount of work per node increases super-linearly as the distributions scale, thus the plots are not perfectly horizontal. With x nodes we generate scaled versions of the real world graphs that are x-times larger.

method to generate the distributions for these tests, and we approximately hold constant the ratio of generated edges to nodes. Note that the ratio of vertices to node decreases due to our deletion of vertices with degree less than 5. However, this has minimal impact on the total number of edges, as $n << m$ in general and the vast majority of edges are attached to vertices of degree five and greater.

The plot does not present pure weak scaling since the work more than doubles when we double the graph size, as our distribution scaling method also intrinsically increases the maximum degree by an approximately logarithmic factor. This contributes to the observed logarithmic increase in timing for successive runs, as our generation methods are dependent both on the number of edges as well as the graph's maximum degree. Our method has minimal work dependence on the number of vertices. Our performance is quite good considering this, as the ratio of our performance difference between 2× and 16× is consistently less than the ratio of increase in d_{max}. As with prior results, we show the aggregate time to generate all nine graphs with varying μ. Table 2 summarizes graph characteristics of 2× and 16× scaled graphs for the six distributions along with a performance rate of edges generated per second per node as we scale (in millions of edges).

8.4 A-BTER Terascale Runs

To further stress the scaling of our generation method, we ran additional scaling experiments on *Astra* and *Trinity*. We used the baseline Friendster distribution to generate tests for up to 512 compute nodes, representing a graph with close to 1 trillion edges and an edge list consuming 15 terabytes of memory. This represents a graph approximately 20× larger than the largest LFR-style graphs generated to date [14], and an increase of *several orders-of-magnitude* for the largest instances generated in memory. We create a single set of degree distributions and clustering-coefficient distributions for each node count, with a μ held equal to Friendster's native μ (~0.67) as we scale. Table 3 shows the graph characteristics and scaling of graph generation time.

Table 3: Test graph characteristics and results for terascale graph generation experiments given as # edges (m), # vertices (n), and max degree (d_{max}) for Friendster at 1× to 512× scale, run time (T) in seconds and per-node edge generation rate (R) in millions of edges per second for the KNL (*Trinity*) and ARM (*Astra*) systems. Results that are x-times larger are generated with x nodes.

Scale	m	n	d_{max}	Memory	T_{KNL}	T_{ARM}	R_{KNL}	R_{ARM}
1×	1.8 B	40 M	5.2 K	29 GB	33	22	55	82
4×	7.2 B	93 M	10 K	115 GB	35	28	52	64
16×	29 B	260 M	15 K	459 GB	35	29	50	61
64×	115 B	786 M	20 K	1.8 TB	55	32	33	56
256×	464 B	2.5 B	26 K	7.4 TB	102	69	18	26
512×	925 B	4.6 B	30 K	15 TB	134	76	14	24

We observe no crippling bottlenecks in our largest tests, with a difference in timing of only about 4× from 1 to 512 nodes, or from 1.8 billion to 925 billion edges. Our largest tests give a total edge generation rate of close to 7 billion edges per second on KNL and 12 billion edges per second on ARM.

8.5 Timing Breakdown

In Figure 6 we demonstrate a breakdown of timing for each phase of graph generation for A-BTER on 8 nodes. "CommAssign" refers to all pre-processing associated with vertex ID to group and block mapping for A-BTER. The portion for "Dist" refers to the parallel overheads associated with work partitioning and distribution of edge skipping. "IntGen" and "ExtGen" refer to the intra-community and inter-community edge generation, respectively. These proportions are averaged over all ranks. We show the results for all six graphs with a mixing parameter of $\mu = 0.5$. This parameter value is selected as it will result in approximately the same number of edges being generated for the internal and external stages. A larger μ would increase the relative portion in external edge generation, and conversely, a smaller μ would increase the relative portion of internal edge generation. μ has little impact on community assignment times and the relative parallel overheads.

We note that external edge generation on average requires the largest proportion of time. This is expected, as while on average the number of internal and external edges generated is equivalent, edge-skipping for Chung-Lu graphs can be more expensive. This is because inter-degree probabilities vary for each degree pair while there's only a single edge probability among all vertices in an Erdős-Rényi graph. Parallel overheads are proportional to the maximum degree and general skew of the degree distribution. As prior work alluded [1], perfectly balancing parallel edge-skipping for edge generation is a challenge, requiring a fine-grained partitioning of expected work. We observe similar challenges, particularly as we scale to an order-of-magnitude more edges on almost two orders-of-magnitude more threads. Improving on these parallelization schemes is a promising future direction.

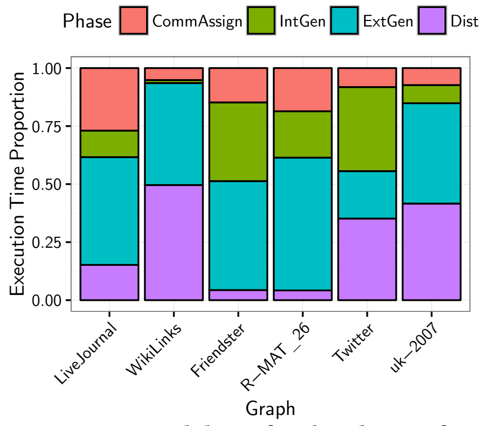

Figure 6: Timing Breakdown for the Phases of A-BTER.

8.6 Large Scale Community Detection Benchmarking

To demonstrate the efficacy of using our methods for large-scale community detection benchmarking, we ran Label Propagation [30] and Louvain [13] on all six test distributions and calculated the resultant NMI across a range of μ. Due to the large scale of these tests, we restricted Louvain to one level of 5 iterations and Label Propagation to 10 iterations[6]. We ran these set of tests on a single KNL node. We plot these outputs in Figure 7.

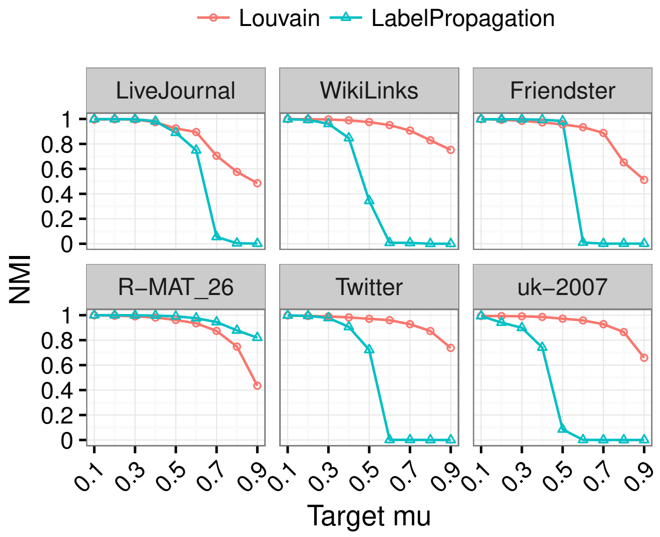

Figure 7: NMI comparisons on a varying-density suite of graph instances generated from the six test distributions. We compare Louvain and Label Propagation.

At this scale and for the iteration count, Louvain appears more capable at resolving communities across a wide range of mixing parameters. In particular, Label Propagation fails on several graphs to resolve communities for $\mu \geq 0.5$. We noted this similar observation at the smaller scale with baseline LFR generator. The one exception is the R-MAT graph, which is likely due to its relatively skewed power-law distribution, which results in a correspondingly large number of small communities that label propagation can readily resolve.

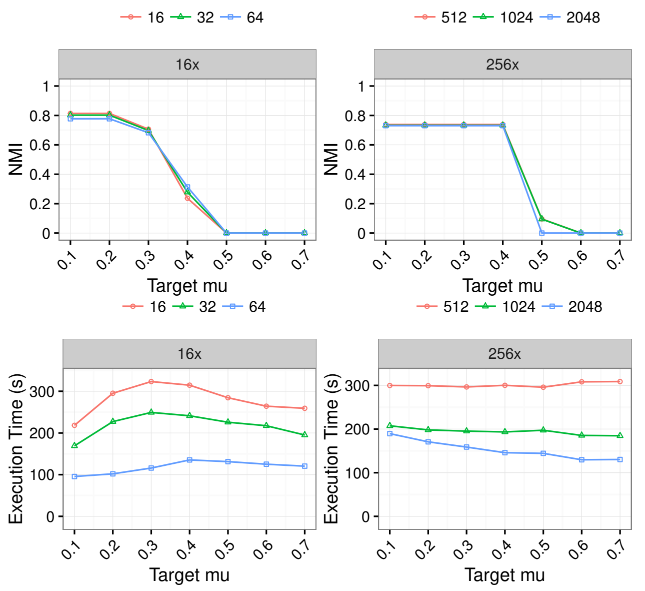

Figure 8: VCCS-based study on the 16× (0.5 TB) and 256× Friendster (7.4 TB) graphs (top) comparing Label Propagation at different node counts. Execution time for each node count vs. μ is shown on the bottom. We run on both KNL (16×) and ARM (256×) systems. Our capability provides evidence that we can strong scale this distributed computation without a penalty in solution quality.

We also used our generation methods to test the impact of strong scaling on community-detection algorithms in terms of the resultant output quality. We show in Figure 8 an LFR-style study on the 16× and 256× scaled Friendster distributions, with A-BTER generating graphs at $\mu = 0.1, \ldots, 0.7$. At this scale, the only community detection algorithm we have been able to successfully run is Label Propagation. So instead of comparing the impact on quality in terms of NMI between differing algorithms, we instead examine the impact on quality of decreasing the time to solution by increasing the node count. We run the 16× tests on *Mutrino* and the 256× tests on *Astra*.

We note two primary results. First, there is a large hit to solution quality in terms of NMI when moving from shared memory to distributed memory. We observe a maximum NMI of only about 0.8 on these tests, and in other similar tests we have noted similar observations about such a decrease when running label propagation in distributed versus shared memory. Implementations of label propagation or similar algorithms should seek to mitigate these effects. More interestingly, we observe minimal further impact on solution quality in terms of NMI as we increase the node count. This suggests that further increasing the number of processors in distributed memory might be a viable way to improve time to solution without severely impacting solution quality.

[6]The authors of [29] observed that on graphs of this scale Label Propagation can compute for 1000s of iterations over many hours

We note that these preliminary observations warrant further study, and A-BTER is a viable tool to enable such studies and future insights.

8.7 Distribution Scaling

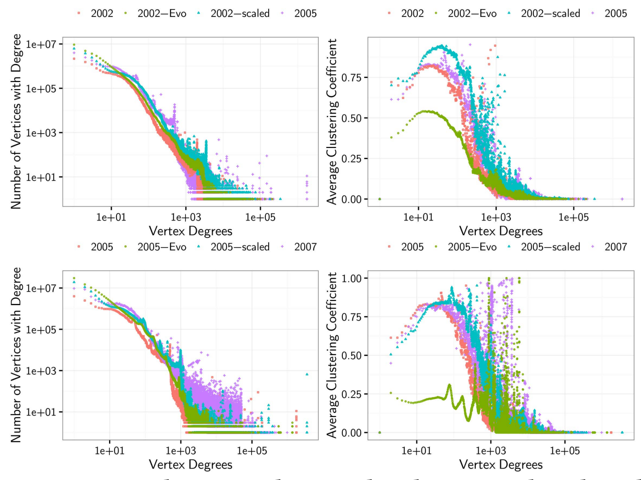

Figure 9: Distribution scaling on the uk test graphs. Plotted are the degree distributions (left) and clustering coefficient distributions (right) of uk-2002 being scaled to match the parameters of the uk-2005 (top) and uk-2005 being scaled to match the parameters of uk-2007 (bottom). Note that as the real UK graphs evolve, their clustering coefficients tend to increase. We match this trend; the EvoGraph distributions tend to decrease.

We empirically demonstrate our distribution scaling method in Figure 9. Here, our comparison is to the natural growth of the .uk domain web graph from 2002 to 2005 and from 2005 to 2007. Due to the limited availability of large-scale temporal networks, we are restricted in our testing to these specific crawls.

We generate our distributions using the described method with the observed scaling factors for the number of vertices and edges for both growth periods. While we acknowledge that accurately modeling future growth of a specific network distribution would be impossible without knowing the growth factors a priori, the purpose of these tests is to evaluate how well, given these parameters, our scaling method compares to the real growth. We also include the distributions resulting from growth simulated by EvoGraph [23] for comparison, the current state-of-the-art. We select as a scaling factor for EvoGraph the nearest integer to the edge ratio. We emphasize that we don't explicitly grow the graph topology itself like EvoGraph, as our method is tailored for creating input distributions for our graph generators.

Figure 9 demonstrates a comparison in scaling the uk-2002 crawl to the uk-2005 crawl and from the uk-2005 crawl to the uk-2007 crawl. The degree distributions are shown on the left and the clustering coefficient distributions are shown on the right. We note that both our method and EvoGraph overlap quite well with observed growth, modulo random noise, in terms of the degree distribution. In these instances, we observe that the average clustering coefficient of the .uk domain increases over time. However, it appears that EvoGraph understates this clustering while we slightly overstate it; we observe similar results when calculating the changes to the distributions' *native* μ.

Obvious future work would be to better understand network growth, specifically in terms of how the average clustering coefficient and clustering coefficient distribution evolve.

8.8 Performance Comparisons with Other Methods

We know of two other works that generate LFR or LFR-like graphs. NetworKit [32] implements an LFR generator as one of their many graph generation routines. Their generator closely follows the original LFR code in output and performance. Recently, scalable parallel LFR generator variants have been proposed [14]. They show scalability to over 50 billion edges and can generate a 10 billion edge graph in 17 hours[7]. Our 512× scale Friendster graph has ~920 billion edges, and we can generate it in 134 seconds with A-BTER on 512 KNL nodes. However, we don't claim to exactly match the input distribution or target mixing parameter, and we don't include I/O times for outputting the edge list. We do claim that A-BTER is a similarly useful but much more scalable means to benchmark HPC-scale community detection algorithms.

There has been recent work on massive-scale graph generation of Erdős-Rényi, random geometric, random hyperbolic, random Delaunay graphs [12]. They report for Erdős-Rényi generation on 2^{15} Sandy Bridge cores a time of about 25 seconds to generate about 275 billion edges. In our terascale experiments, we generate a A-BTER instance of 460 billion edges graph on about 2^{14} KNL cores in 100 seconds, giving an approximately equivalent edges-per-core generation rate (~300 K edges per core per second). Other recent work exploring the parallelization of edge-skipping methods [1] implements the Erdős-Rényi,Chung-Lu, and BTER generation and gives a generation rate of approximately 12 billion edges per second on 1024 Sandy Bridge cores for a web crawl similar to but slightly larger than uk-2007 with a variant of BTER. Our BTER generator rates are not directly comparable due to various design decisions targeting terascale and involving duplicate edge removal (we do not allow duplicates), but we match the edge generation rate of 12 billion edges per second on 15TB variants of the Friendster network (see Table 3). Furthermore, since we treat BTER as a black box our A-BTER process could absorb and benefit from advances in BTER generation such as those in [1].

9 DISCUSSION

We briefly discuss two related topics that represent interesting future work. First, consider an adversarial situation in which an algorithm designer wishes to "game" the VCCS-based method to give an unfair advantage to his/her algorithm. Explicitly addressing this situation is beyond the scope of this paper. However, we note that the engineered algorithmic solutions produced by both A-BTER and LFR are based upon random graphs. Furthermore, LFR performs a rewiring step which would perhaps make formulating arguments

[7]The authors of [14] expect improvements from applying the Global Undirected Curveball methods presented at ESA 2018.

about robustness against attack more difficult. The affinity blocks with BTER are simply Erdös-Rényi random graphs, which have been analyzed for decades.

Secondly, we consider other ways one might support VCCS-based experiments at varying scale. We are developing generators of graphs more similar to LFR instances. However, while these generators enable VCCS-based studies and outperform published LFR variants, they are not yet as scalable as A-BTER.

EvoGraph approximately preserves per-degree clustering coefficient distributions at scale. This is a valuable property, and it might be possible to support VCCS-based studies as follows: use EvoGraph to scale up a graph with engineered approximate solution, obtain the scaled graph's degree and clustering coefficient distributions, use them as BTER inputs, and use our A-BTER process to produce a suite of graphs with varying community tightness. While such an approach might work, our approach is more direct and more scalable in test runs.

10 CONCLUSIONS

We have introduced a capability for evaluating and comparing the solution quality of community detection algorithms at HPC-scale via the generation of suites of graphs with varying community coherence. Our method, termed A-BTER, uses a specially-designed BTER graph generator along with pre-processing methods for input degree and clustering coefficient distributions. Our pre-processing can both modify the community coherence output by BTER as well as generate graphs at a scale multiple times larger than the baseline inputs. We note that our methods are quite efficient, generating graphs with almost a trillion edges in minutes on HPC systems. Future users of our software will be able to base algorithm comparison studies on scaled-up models of real graphs, enabling research into the effects of large-scale processing on community detection performance.

Acknowledgements:
We thank the ASC Advanced Architectures test-bed team at Sandia National Laboratories for supplying and supporting the systems used in this paper. The Trinity platform is operated by Los Alamos National Laboratory on behalf of the Advanced Simulation and Computing (ASC) Stockpile Stewardship program. Access to the Astra system is provided by Sandia National Laboratories for the ASC Vanguard program, a sequence of prototype hardware deployments to evaluate technology for future NNSA and DOE missions. 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. This research was funded through the Laboratory Directed Research and Development (LDRD) program at Sandia. We thank Tamara Kolda for discussions helpful in refining the abstract.

REFERENCES

[1] Maksudul Alam, Maleq Khan, Anil Vullikanti, and Madhav Marathe. 2016. An efficient and scalable algorithmic method for generating large: scale random graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 32.
[2] David A Bader and Kamesh Madduri. 2006. GTgraph: A synthetic graph generator suite. (2006).
[3] Vladimir Batagelj and Ulrik Brandes. 2005. Efficient generation of large random networks. *Physical Review E* 71, 3 (2005), 036113.
[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008. http://stacks.iop.org/1742-5468/2008/i=10/a=P10008
[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.
[6] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*.
[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.
[8] Fan Chung and Linyuan Lu. 2002. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics* 6, 2 (2002), 125–145.
[9] Paul Erdős and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5, 1 (1960), 17–60.
[10] John Forrest, Ted Ralphs, Stefan Vigerske, LouHafer, Bjarni Kristjansson, jpfasano, EdwinStraver, Miles Lubin, Haroldo Gambini Santos, rlougee, and Matthew Saltzman. 2018. (COIN-OR/Cbc): Version 2.9.9. https://doi.org/10.5281/zenodo.1317566
[11] S. Fortunato and M. Barthélemy. 2007. Resolution Limit in Community Detection. *PNAS* 104, 1 (2007), 36–41.
[12] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schultz, Darren Strash, and Mortiz von Looz. 2018. Communication-Free Massively Distributed Graph Generation. In *International Parallel & Distributed Processing Symposium (IPDPS)*.
[13] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 885–895.
[14] Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. 2018. I/O-Efficient Generation of Massive Graphs Following the LFR Benchmark. *J. Exp. Algorithmics* 23, 1, Article 2.5 (Aug. 2018), 33 pages. https://doi.org/10.1145/3230743
[15] William E Hart, Jean-Paul Watson, and David L Woodruff. 2011. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation* 3, 3 (2011), 219–260.
[16] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C Seshadhri. 2014. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing* 36, 5 (2014), C424–C452.
[17] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350.
[18] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical Review E* 78, 4 (Oct. 2008), 1–5. https://doi.org/10.1103/PhysRevE.78.046110
[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[20] Joel C Miller and Aric Hagberg. 2011. Efficient generation of networks with given expected degrees. In *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 115–126.
[21] M. E. J. Newman and M. Girvan. 2004. Finding and evaluating community structure in networks. *Phys. Rev. E* 69 (Feb 2004), 026113. Issue 2. https://doi.org/10.1103/PhysRevE.69.026113
[22] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in Social Media: Performance and application considerations. *Data Mining and Knowledge Discovery* 24, 3 (2012), 515–554.
[23] Himchan Park and Min-Soo Kim. 2018. EvoGraph: an effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2051–2059.
[24] Web Archive Project. [n. d.]. Friendster social network dataset: friends. https://archive.org/details/friendster-dataset-201107.
[25] U. N. Raghavan, R. Albert, and S. Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E* 76, 3 (2007), 036106.
[26] Giulio Rossetti and Rémy Cazabet. 2018. Community Discovery in Dynamic Networks: A Survey. *ACM Comput. Surv.* 51, 2, Article 35 (Feb. 2018), 37 pages. https://doi.org/10.1145/3172867
[27] Geoffrey Sanders, Roger Pearce, Timothy La Fond, and Jeremy Kepner. 2018. On large-scale graph generation with validation of diverse triangle statistics at edges and vertices. *arXiv preprint arXiv:1803.09021* (2018).
[28] Comandur Seshadhri, Tamara G Kolda, and Ali Pinar. 2012. Community structure and scale-free collections of Erdős-Rényi graphs. *Physical Review E* 85, 5 (2012), 056109.
[29] G. M. Slota and S. Rajamanickam. 2018. Experimental Design of Work Chunking for Graph Algorithms on High Bandwidth Memory Architectures. In *International Parallel & Distributed Processing Symposium (IPDPS)*.
[30] G. M. Slota, S. Rajamanickam, and K. Madduri. 2016. A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization. In *International Parallel & Distributed Processing Symposium (IPDPS)*.

[31] Christian L Staudt, Michael Hamann, Ilya Safro, Alexander Gutfraind, and Henning Meyerhenke. 2016. Generating scaled replicas of real-world complex networks. In *International Workshop on Complex Networks and their Applications*. Springer, 17–28.

[32] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530. https://doi.org/10.1017/nws.2016.20

[33] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research* 11, Oct (2010), 2837–2854.

[34] M Winlaw, H DeSterck, and G Sanders. 2015. *An In-Depth Analysis of the Chung-Lu Model*. Technical Report LLNL-TR-678729. Lawrence Livermore National Laboratory.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

KNL Machine:

We ran our largest KNL experiments on the Trinity Platform at Los Alamos National Laboratory, USA. Cray XC40 with Aries interconnect, 9408 Intel Haswell nodes (not used in this paper), and 9,984 Intel KNL nodes. Each KNL node has one Intel Xeon Phi Knights Landing Processor (Model 7250, 68 cores, 272 hardware threads, 16GB MCDRAM) and 96GB DDR. Lustre parallel file system.

We used the Cray Linux Environment (CLE) 6.0.UP05 based on SUSE Linux Enterprise Server (SLES) 12.3, kernel 4.4.103 (on KNL compute node).

We used the Intel 18.0.5 compiler and that Cray MPICH 7.7.4 MPI library. Compiler flags: -fopenmp -xMIC-AVX512 -O3 -std=c++11

We ran all KNL experiments involving graphs smaller than 64X Friendster on Mutrino, a smaller system at Sandia National Laboratories with the same architecture as Trinity, 96 KNL nodes and 96 Haswell nodes available to general users.

Arm Machine:

We ran our Arm experiments on the Astra Platform at Sandia National Laboratories, USA. Astra has 2592 dual-socket Marvel ThunderX2 compute nodes in which each socket has 28 cores and 64GB (for a 128GB node total) of memory capacity. Each ThunderX2 socket provides 8 channels of DDR4 memory with 8GB provided per channel. The nodes are interconnected with Mellanox HDR InfiniBand (100Gb/s) arranged as a fat-tree topology with 2:1 bandwidth tapering to the top level of the tree.

The operating system on Astra is the TOSS 3.3 kernel developed by Lawrence Livermore National Laboratory, which is based on RedHat Enterprise Linux 7.6.

We used the Arm HPC 19.1 compiler and OpenMPI 3.1.3. Compiler flags: -fopenmp -mcpu=thunderx2t99 -mtune=thunderx2t99 -O3 -std=c++11

Software:

Our software is not yet available, but we plan to release it on GitHub.com. We have two primary software dependencies: Pyomo (version 5.3 using CPython 2.7.12), an open-source Python-based modeling language, and CBC (version 2.8.12), an open-source linear and integer program solver.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

```
The software has not been through our internal release
↪  process, so there is no URL or DOI yet.
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Trinity/Mutrino - KNL nodes with 96GB DDR; Astra - 2x ARM nodes with 128GB DDR

Compilers and versions: KNL: Intel 18.0.5 compiler; ARM: Arm HPC 19.1

Libraries and versions: Pyomo 5.3 and CBC 2.8.12; KNL: Cray MPICH 7.7.4; ARM: OpenMPI 3.1.3

Input datasets and versions: Stanford SNAP database; Koblenz Network Collection; Laboratory for Web Algorithmics; R-MAT