SAND2020-7141C

A scalable graph generation algorithm to sample over a given shell distribution

M. Yusuf Özkaya*, M. Fatih Balın*, Ali Pınar†, Ümit V. Çatalyürek*
*Georgia Institute of Technology, School of Computational Science and Engineeering, Atlanta, GA, USA
†Sandia National Laboratories, Livermore, CA, USA
Email: {myozka, balin, umit}@gatech.edu, apinar@sandia.gov

***Abstract*—Graphs are commonly used to model the relationships between various entities. These graphs can be enormously large and thus, scalable graph analysis has been the subject of many research efforts. To enable scalable analytics, many researchers have focused on generating realistic graphs that support controlled experiments for understanding how algorithms perform under changing graph features. Significant progress has been made on scalable graph generation which preserve some important graph properties (e.g., degree distribution, clustering coefficients).**

In this paper, we study how to sample a graph from the space of graphs with a given shell distribution. Shell distribution is related to the k-core, which is the largest subgraph where each vertex is connected to at least k other vertices. A k-shell is the subset of vertices that are in k-core but not $(k+1)$-core, and the shell distribution comprises the sizes of these shells. Core decompositions are widely used to extract information from graphs and to assist other computations.

We present a scalable shared and distributed memory graph generator that, given a shell decomposition, generates a random graph that conforms to it.

Our extensive experimental results show the efficiency and scalability of our methods. Our algorithm generates 2^{33} vertices and 2^{37} edges in less than 50 seconds on 384 cores. [1]

***Index Terms*—graph generation, scalable graph algorithms, distributed algorithms, shared memory.**

I. Introduction

Graphs have emerged as the standard language to model interactions between entities in many applications including social sciences, biology, cyber security, and finance. Graphs derived from real-world datasets can scale up to billions of entities and relations. Many researchers focus their efforts on analyzing these networks to increase understanding of the inherent interaction properties between entities of these networks. Despite the availability of many public data sets, a lack of real-world data at larger scales continues to hinder computational efforts.

To mitigate this problem, many research efforts focus on synthetic graph generators which closely capture properties of real world graphs. There are two main challenges here: What are the critical properties that need to be preserved? And how can we generate such graphs efficiently while preserving the desired properties? Notable efforts in this area include preferential attachment models such as [5], [9], which are build on the "rich keep getting richer" principle, Kronecker graphs [22], which generate a recursive structure, degree and joint degree distribution-based methods [10], [36], and BTER [19], which preserves both the degree distribution and the clustering coefficient distribution. Some generators are application driven such as Internet-specific [12], road connectivity [6], and infrastructure [25]. Other efforts specifically focus on benchmarking such as the LFR [21] networks for community detection algorithms. Stochastic block models [16], while designed as a generative model, are also used to generate test cases. Additional efforts use a sample graph to generate a (scaled) replica [37].

Many works focus on delivering synthetic graphs at scale, as few real-world graphs are small, and subsequently, synthetic copies required for an accurate analysis and interpretation must be large. Some of the works focusing on scalable graph generators are [37], [35], [32], [30], [20], [14]. Recent work of Funke et al. [14] focuses on scalable generation of Erdos-Renyi [13], Random Geometric [29], Random Hyperbolic [38], and Random Delaunay Graphs [30].

In this work, we focus on a well-known, important property that indirectly affects density and structure of a graph: *core decomposition*. The k-core of a graph is the maximal connected subgraph in which each node is connected to at least k other nodes. Typical core decomposition algorithms break down the graph by incrementing k until the core is empty. It has been observed that real graphs have much larger cores than random graphs [33]. This is due to locally dense structures which reveal information about the underlying system. As such, the core decomposition of graphs have been used in many applications, such as network visualization [3], [39], characterizing the internet topology [4], [8], accelerating community detection [28], resilience of communities [15], identifying anomalous network nodes [34], finding influencers [1], [18], [23], predicting protein functions [2], explaining jamming transitions [26], and explaining structural collapse of ecosystems [27].

Baur et al. [7], proposed an algorithm that given a shell decomposition histogram N and a shell-connectivity matrix M that stores inter-shell and intra-shell edge counts, generates a graph from inside out. That is, it starts from the highest shell

[1]This work is funded by the Laboratory Directed Research and Development program of Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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-NA0003525.

value and progressively adds shells as layers to the graph. To further improve the quality of the resulting graphs, they propose methods of edge rewiring and swapping to adjust the degree distribution and connected components, although it is not always easy to find the correct ranges that make the graph realizable.

Here, we present S^{3}G^2, *Scalable Shell Sequence Graph Generator*, set of parallel graph generators that given a shell decomposition of a graph, generates a random graph that conforms to this decomposition. Our work is based on the sequential work of Karwa et al. [17]. These algorithms are equivalent to ALGORITHM 3 of Karwa et al. They generate a graph from the space of graphs with a given shell distribution histogram where any graph can be generated with a positive probability (see Theorem 9 of [17]). This does not imply a uniform sampling of possible graphs. Despite empirical evidence of near-uniform distributions over possible graphs, formally quantifying the bias of the distribution is an open question.

This work provides carefully designed shared-memory and distributed-memory parallel algorithms for this process. Our main contributions are:

- Design and analysis of shared and distributed memory algorithms for the graph generator,
- Analysis of best and worst cases,
- Extensive empirical evaluation of the proposed parallel algorithms, and
- A software package containing implementations of our parallel algorithms as well as the optimized sequential algorithm.[2]

The rest of the paper is organized as follows. First, the model introduction and formalization of the problem is in Section II. Section III introduces and explains major issues to tackle in parallel approaches to preserve the probabilistic properties of sequential algorithm. Next, the proposed shared memory/distributed algorithms are described in Sections IV and V, respectively, and they are evaluated through extensive simulations in Section VI. Finally, conclusion and directions for future work are given in Section VII.

II. Background

A. Definitions and Notations

An undirected graph $G = (V, E)$, contains a set of vertices V and a set of edges E of the form $e = \{u, v\}$, where the edge e is undirected. We limit our focus on simple undirected graphs where there are no self-loops ($e = \{u, u\}$). The *degree* of a vertex u is the number of vertices that are connected to u. A subgraph $G_s = (V_s, E_s)$, is a graph where $V_s \subseteq V$ and $E_s \subseteq E$. A *k-clique* is a complete graph on k vertices.

The *k-core* of a graph G, is the maximal connected subgraph of G in which all vertices have degree of at least k in the subgraph. A vertex u has a *coreness* (or also referred as *k-shell* value) k, if it is in some k-cores of the graph but no $(k+1)$-cores. The *degeneracy* (or k_{max}) of a graph G is the largest k such that there is a non-empty k-core. Algorithm 1 outlines a process known as *peeling*, which computes a k-shell sequence, s, i.e., k-shell values s_i, for all $v_i \in V$, by repeatedly removing all vertices with degree less than k. This algorithm also returns the degeneracy (k_{max}) of the graph.

[2]Available at: http://tda.gatech.edu/software/s3g2/

Algorithm 1: Compute Core Values

```
Data: a graph $G = (V, E)$
Result: $s_i, \forall v_i \in V$ and $k_{max}$
1 $s_i \leftarrow -1, \forall v_i \in V$
2 $k \leftarrow 0$
3 repeat
4     Remove vertices with degree at most $k$ (and their
      associated edges) in G until there are no changes
      to G (i.e., if at least one vertex is removed, restart
      on the remaining G).
5     Assign the value $k$ (i.e., $s_i \leftarrow k$) for all removed
      vertices in the previous step (i.e., line 4).
6     $k \leftarrow k + 1$
7 until $G = \emptyset$
8 return $s$, $k - 1$
```

We define k_{max}-shell histogram (array of $k_{max}+1$ integers), $(S_0, S_1, S_2, \ldots S_{k_{max}})$ to describe the ***sizes*** of a graph's shells. That is, the graph has $n = |V| = \sum_i S_i$ vertices in total, and it has S_i vertices with shell value i, for $i = 0, 1, 2, \ldots k_{max}$.

Theorem 1. *There exists a graph G with a given shell histogram, $(S_0, S_1, S_2, \ldots S_{k_{max}})$, if and only if $S_{k_{max}} > k_{max}$.*

Proof. The necessary condition follows from the definition of core decomposition. Each vertex in the k_{max}-core of a graph should have k_{max} neighbors and thus there should be at least $k_{max} + 1$ vertices in the core.

To prove sufficiency, we will use a constructive algorithm. Given the k-shell histogram, we can start by generating the k_{max}-core as a $(k_{max}+1)$-clique on the first $k_{max}+1$ nodes of the $S_{k_{max}}$ vertices. Then, add the remaining nodes one-by-one, connecting them to exactly k_{max} last elements (i.e., for vertex v_i, the vertices from $v_{i-k_{max}}$ to v_{i-1}). For each of the $S_{k_{max}-1}$ vertices in the $(k_{max}-1)$-shell, we connect them to $(k_{max}-1)$ arbitrary vertices in the k_{max}-shell. Observe that this will guarantee that all these vertices are in the $(k_{max}-1)$-shell and at the same time, the shell numbers of the vertices in the graph do not change.

We can continue this procedure for all the shells, processing the shells in decreasing order. $\square$

B. Sequential Algorithm

Karwa et al. [17] presented a sequential algorithm which forms the baseline of our parallel algorithms. Therefore, here we briefly present the algorithm, with only some small tweaks for performance.

The algorithm generates a graph where the core values of the vertices are non-decreasingly ordered, i.e., lower vertex IDs will have lower core values. Furthermore, the algorithm will generate edges in the vertex ID order, i.e., each vertex will generate edges only towards the higher ID vertices. For each vertex v_i, there must be at least s_i edges towards higher or equal core values as per the definition.

Since the definition of k-core involves vertices with higher or equal core values for any given vertex in the graph, we can safely separate any edges from lower shells to higher shells, as there is no dependency between them. For edges between pairs of vertices in the same shell, we need to keep a counter of the required edge count that has been realized before we reach the current vertex.

The sequential algorithm follows the reverse of the peeling approach (Alg. 1) to generate graphs. It starts from the lowest shell value, and generates edges towards vertices with same or higher shell value. Since the graph is generated in non-decreasing order of the shell values, it is easier to generate one where if $s_i < s_j$ then $i < j$. Other labeled combinations can be generated by permuting the vertex IDs.

Pseudo-code of the sequential algorithm is given in Alg. 2. It generates a graph with a shell sequence distribution that matches a given shell histogram $(S_0, S_1, S_2, \dots S_k)$. First, the algorithm divides the vertex set into two subsets: $v_1, ..., v_{n-k-1}$ and $v_{n-k}, ..., v_n$, which are processed in two separate *phases* of the algorithm. The first phase is the core of the algorithm where it uses the reverse of the peeling approach presented above. The second phase generates the last $k_{max}+1$ nodes of the last shell (i.e., k_{max}-shell). This phase is typically negligible compared to the first phase in terms of runtime (see Table I).

The main points from the sequential algorithm are:

- Any vertex may generate at most s_i edges towards higher index vertices
- The only dependency among vertices i and j is between vertices where $s_i = s_j$.
- The last $k_{max} + 1$ vertices require a special case (*Phase 2*).

In addition, for random number generators, any distribution that covers the entire space can be used to achieve a variation of this algorithm that can generate all possible graphs with a given shell sequence with a positive probability. Changing the probability of selecting a node with the same shell value, and skewing the probability distribution of random subset size towards either end would allow one to adjust the average total number of edges in the generated graphs. In our experiments, we applied the uniform distribution for all random number generation as in Karwa et al. [17].

Algorithm 2: Generate graph with given shell histogram

```
Data: a shell histogram $(S_0, S_1, S_2, \dots S_k)$.
Result: a graph $G$ with shell sequence matching the
        given histogram.
1  $G(V,E) \leftarrow G(\{v_1, v_2, ..., v_n\}, \emptyset)$
2  initialize shell values $s_1, s_2, ..., s_n$ using histogram $S$
3  $t_i \leftarrow 0, \forall i = 1, 2, \dots, n$
   // Phase 1
4  for $i \leftarrow 1$ to $n-k-1$ do
5      Choose a random subset R of $\{v_{i+1}, ...v_n\}$ where
         $max\{0, s_i - t_i\} \leq |R| \leq s_i$
6      for $v_j \in R$ do
7          Add edge $(v_i, v_j)$ to $G$
8          if $s_i = s_j$ then $t_j \leftarrow t_j + 1$
   // Phase 2
9  Rem $\leftarrow (v_i : n-k \leq i \leq n)$
10 Swap all $v_i$ where $t_i = 0$ to the end of Rem
11 while Rem $\neq \emptyset$ do
12     Assign $v_i$ = last element of Rem
13     Rem $\leftarrow$ Rem $\setminus \{v_i\}$
14     Choose a random subset $R$ of Rem with
         $|\text{Rem}| - t_i \leq |R|$
15     for $v_j \in R$ do
16         Add edge $(v_i, v_j)$ to $G$
17         if $s_i = s_j$ then $t_j \leftarrow t_j + 1$
18     for $v_j \in$ Rem do
19         $t_j \leftarrow t_j - 1$
20         if $t_j = 0$ then
21             Rem $\leftarrow$ Rem $\setminus \{v_j\}$
22             Add edges from $v_j$ to all vertices in Rem
```

III. S^{3}G^2: Scalable Shell Sequence Graph Generation

Since Phase 1 is the most time consuming part of the sequential algorithm (see Table I), we parallelize this phase by simply partitioning the work required for shells and/or vertices. If the workload, consisting of separate shells, can be divided in a properly balanced fashion to the processing elements (PEs), then, the shells can be processed in a *pleasingly parallel* fashion using the sequential algorithm, without requiring any communication. If this is not possible, we partition the vertices, in the shells, to different PEs but communicate the information required by the higher ID vertices.

The sequential algorithm (Alg. 2) generates a random number of edges (line 5) for each vertex in the range from its minimum number of edges ($s_i - t_i$) to the shell value itself (s_i), where t_i is the number of received edges from lower ID vertices with the same shell value (i.e., edges contribute to the k-core value of the vertex). If a vertex already received enough edges from the vertices in the same shell, then it does not need to generate any edges but may still choose to do so. In the parallel algorithms, we treat this operation (line 5) as a two-step procedure. The maximum number of edges each vertex may generate is bounded by its shell value. We can assume the worst and generate all those edges. Next is to decide on the random *number of* edges each vertex generates, which requires knowledge about the edges coming from the vertices with lower IDs within the same shell.

If the PEs do not communicate this information, the algorithm cannot generate all possible graphs: It forces the nodes that have dependencies to nodes in previous PEs to create edges towards higher indices. Our parallel algorithms tackle this problem by executing multiple *fix rounds* where all PEs relay the information of edge counts they generate to the PEs containing the corresponding higher-ID vertices. This information is used by the receiving PE to *re-select* a random subset size with the updated information. In other words, our algorithms first generate an array of possible edges from a node. Next, they generate a random subset size (which is equal to the size of this array at this moment). Then, the *fix rounds* only change how many elements of this array are being used, by updating the subset size. The shared memory algorithm asynchronously invalidates the affected nodes, and queues them to be processed again, achieving the same goal. Observe that this is not a re-run of the whole generation for a node but just a range of confirmed edges for it.

In distributed memory setting, this re-selection for a vertex v_i may cause the previous information sent about v_i's edges to be obsolete/incorrect, thus causing a cascading effect. Our distributed memory algorithm assigns the workload to PEs by increasing vertex ID, thus requiring sending information only towards higher rank PEs. Thus, in the worst case scenario the parallel algorithm running on p PEs will have up to $p-1$ cascading fix rounds.

By properly handling the fix operations in our parallel algorithms, the overall mechanics of the sequential algorithm are preserved. As a result, the parallel algorithms conform to all statistical findings reported in [17].

IV. S^{3}G^2: Shared Memory Algorithm

The shared-memory algorithm starts by computing the computational load of each shell, and schedules them with the largest-job-first scheduling policy, that is, it orders the shells in non-decreasing workload and starts assigning them in that order. However, the actual scheduling is completely dynamic after the p initial assignments, where p is number of PEs, or threads in shared-memory code. Computation of each shell can be done independently from each other (See Section II-B). Therefore p initial threads are assigned to shells in a round-robin manner. When a thread completes its work, it searches for new work. If all shells are being processed by other threads, it joins the shell with largest load, still following a round-robin fashion among least crowded shells and takes a task from that shell. This process continues until all shells and their tasks have executed.

There are three implementation choices in our shared memory algorithm:

a) Granularity: We assign tasks to threads. The finest granularity task is to process a single vertex, which we call Vertex. The coarsest granularity task is to process a complete shell, however, in real-world cases k_{max} values might be smaller than or close to the available number of PEs p. Hence, we decided to use medium granularity tasks, where processing a *chunk* of 64 vertices constitutes a task. We named this option as Chunk.

b) Memory: We implemented two options: i) Global: All the necessary data for the algorithm is stored in shared memory accessible by all threads, 2) Local: The processing of each shell is almost independent, therefore in this model we have local data for each shell, which is allocated by the first thread that takes a task from that shell.

c) Queue: Since a vertex v_i requires information of the edges from vertices with lower IDs (with the same shell value), to define the number of edges towards vertices with higher IDs that it needs to generate, we process the vertices in increasing index order as much as possible. To do so, we implemented two different queues: i) mutex-protected priority queue: where all the vertices with the same shell value are pushed to a priority queue, and access to the queue is protected via a mutex, which we call Mutex, ii) a lock-free version of the approximate priority queue proposed by Matias et al. [24]. We call this Trie, and it uses bitwise and atomic operations to keep track of the queue's state.

In the shared memory algorithm, there are no global fix rounds. Instead, nodes that have obsolete information are simply re-added to the task queue.

V. S^{3}G^2- Distributed Memory Algorithm

Alg. 3 outlines our high level parallel distributed algorithm, which follows the two phases of the sequential algorithm (Alg. 2) and in addition, includes a *fix phase* following Phase 1. The algorithm starts by distributing the shell histogram and work (load) to PEs according to chosen load balancing algorithm (loadBalAlg), and then each PE generates the shell sequence they are responsible for, using the input histogram. After that Phase 1 starts, where each PE first generates a random subset (of size s_i for vertex v_i) of edges (R_i) for each of their local vertices. Then, they pick a random subset of the edges in R_i. For each chosen edge $\{v_i, v_j\}$ $(i < j)$, t_j values need to be incremented, which may require communication if v_j is not a local vertex.

After Phase 1 is completed, the parallel algorithm requires a *fix phase* (Alg. 4). In this phase updates from other PEs regarding local nodes are accumulated, and if updates cause a change in the number of outgoing edges selected for a vertex, edges from that vertex are adjusted accordingly. Then, an update for the affected neighbors of the vertex is prepared. If those affected vertices do not reside in the same PE as the vertex, updates to the corresponding PEs are sent in the next round.

The performance of this distributed memory algorithm mainly depends on two choices: i) the load balancing algorithm (loadBalAlg) and ii) the communication scheme used for the updates. Below we present these choices in more detail.

A. Load Balancing

We start by defining the metric, and then discuss the algorithms used to partition the load.

Algorithm 3: Distributed Memory Algorithm

```
Data: $p$: number of PEs; $P_i$ is the processor executing
      this algorithm; and a histogram of number of
      vertices per shell sorted by shell value
      $hist$:$(S_0, S_1, ..., S_k)$
Result: Generated graph
1  distributeLoad(hist, loadBalAlg, $P_i$, $p$)
2  $s \leftarrow$ generateLocalShellVector($hist$)
   // Phase 1
3  for $i \leftarrow 0$ to $|s| - 1$ do
4      Choose a random subset $R_i$ of $\{v_{i+1}, ...v_n\}$ with
         $|R_i| = s_i$
5  for $i \leftarrow 0$ to $|s| - 1$ do
6      Pick a subset size $c$ where
         $max\{0, s_i - t_i\} \leq c \leq s_i$
7      for $v_j \in R_i(slice[0, c-1])$ do
8          if $s_i = s_j$ then
9              if $v_j$ is a local vertex then
10                 $t_j \leftarrow t_j + 1$
11             else
12                 $P_t \leftarrow$ Processor containing vertex $v_j$
13                 sendVal[$P_t$][$v_j$] $\leftarrow$ sendVal[$P_t$][$v_j$] +1
   // Fix the misinformation
14 DistributedFixPhase()
   // Phase 2
15 if $P_i = P_{p-1}$ then
16     Initialize Rem $\leftarrow (v_i : n - k \leq i \leq n)$
17     Swap all $v_i$ where $t_i = 0$ to the end of Rem
18     while Rem $\neq \emptyset$ do
19         Assign $v_i$ = last element of Rem
20         Rem $\leftarrow$ Rem $\setminus \{v_i\}$
21         Choose a random subset $R$ of Rem with
             $|\text{Rem}| - t_i \leq |R|$
22         for $v_j \in R$ do
23             Add edge $(v_i, v_j)$ to $G$
24             if $s_i = s_j$ then $t_j \leftarrow t_j + 1$
25         for $v_j \in$ Rem do
26             $t_j \leftarrow t_j - 1$
27             if $t_j = 0$ then
28                 Rem $\leftarrow$ Rem $\setminus \{v_j\}$
29                 Add edges from $v_j$ to all vertices in
                     Rem
```

Algorithm 4: DistributedFixPhase

```
1  while true do
2      Prepare $sendCnts, sendData$ using non-zero
         $sendVal$
3      globalSend $\leftarrow$ ALLREDUCE (sendCnts, MAX)
4      if $globalSend = 0$ then
5          break
6      Communicate the $sendData$s to their targets
7      recvData $\leftarrow$ Receive data from each process to
         receive from
8      for $< v, a >$ in recvData do
           // $< vertex, adjustment >$ pair
9          $oldval = t_v$
10         $t_v \leftarrow t_v + a$
11         if $oldval = t_v$ then continue;
12         oldc $\leftarrow R_v.size()$
13         Repick a subset size $c$ where
             $max\{0, s_v - t_v\} \leq c \leq s_v$
14         $s, l \leftarrow minmax(oldc, c)$
15         $d \leftarrow (oldc < c)\ ?\ 1 : -1$
16         for $v_j$ in $R_v[s, l-1]$ do
17             if $s_v = s_j$ then
18                 if $v_j$ is a local vertex then
19                     $t_j \leftarrow t_j - 1$
20                 else
21                     $P_t \leftarrow$ Processor containing $v_j$
22                     sendVal[$P_t$][j] $\leftarrow$ sendVal[$P_t$][j] + d
```

1) Metric: In a general sense, *load* is the work a computing resource is assigned. In particular for our algorithm, load is a function of $n = |V|$ and $m = |E|$ assigned to a PE, as for each vertex the process generates a number of edges and stores them locally. In each PE, the distributed algorithm generates s_i edges for each vertex v_i over all v_i it is assigned ($\approx O(m)$). While doing this, the algorithm generates and maintains several arrays proportional to n. For each edge that points out of the current process, it inserts/updates a value in a priority queue dedicated to the receiver process. Thus, we cannot simply say the load is proportional to number of vertices, or edges alone.

We use a linear expression of number of vertices and edges to define the workload due to its simplicity and effectiveness in practice. The workload is, then, characterized as $w = \alpha * n + m$, where α is a small constant (i.e., $\alpha < 10$).

2) Load Balancing Algorithm: Our algorithm utilizes continuous block partitioning, which is also known as *chains-on-chains* partitioning [31]. We have implemented two heuristics and one optimal algorithm.

a) Greedy Balancing (GB): First, the total and average load per PE is computed. Then, starting from the first PE, each PE takes vertices until it reaches the average load. The maximum load imbalance is bounded by $k_{max}(+\alpha)$, which is typically small in real world graphs.

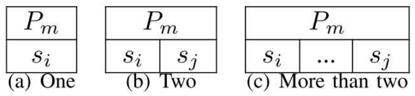
b) Relaxed Greedy Balancing (RB): The communication and fix phases are only required when the vertices of a shell value are divided into multiple PEs. Thus, a logical approach would be to try minimizing the spread of each shell value as much as possible with minimal damage to the load balance. We relaxed our greedy balancing approach to allow PEs to load approximately $x\%$ more if the remaining load for a given shell after the current PE takes its share is less than that amount. Similarly, we allow a PE to delegate its remaining load to the next PE if it is going to get the first $x\%$ of the average load from a shell. In the experiments, we tried x values ranging from 10 to 16, and results were similar.

c) Optimal Balancing (OB): The *Greedy Balancing* and *Relaxed Greedy Balancing* algorithms are heuristics. We also experimented with a variation of Nicol's 1D Optimal Partitioning algorithm, NicolPlus [31].

B. Communication Scheme Variations

1) ALL-TO-ALL *(A2A):* All PEs communicate the amount of data ($counts$) they need to send to other PEs, followed by an ALLREDUCE (AR) with the total amount of information ($nSent$) needing to be sent from each PE. If the maximum of these total amounts is zero, that means no process is sending information to anyone, which completes the *fix* phase. Otherwise, all PEs communicate the $data$ they have for other PEs.

2) ALL-TO-ALL-SPLIT *(Split after first fix round) (A2AS):* Due to the block distribution we used and nature of our algorithm, there actually are subsets of PEs that (i) may communicate, (ii) communicate only once, or (iii) never communicate. Let us start by reviewing the possible load of a PE P_m in terms of the shells it participates in. Figure 1 shows the three possible cases of shell distribution for any given PE. First, as Figure 1(a) shows, the PE may be processing a single shell value. Nodes of this shell may (or may not) have started in a lower rank PE and (may or may not) continue in a higher rank PE. Second, as Figure 1(b) shows, the PE might have two shell values. In this case, $s_i < s_j$ and the PE P_m is the first (smallest-ranked) PE that contains the shell s_j. Similarly, it is the last (highest-ranked) PE that contains the shell s_i. The third case (Figure 1(c)), is where the PE contains more than two shell values. Similar to second case, the PE P_m is the last PE that holds s_i and first that holds s_j. All the shell values $\{s_x | s_i < s_x < s_j\}$, are only contained by P_m.



(a) One (b) Two (c) More than two

Figure 1. Possible shell distributions for a given PE P_m.

Now, considering all three possibilities, and noting that communication is required only within the same shell (only between the PEs that hold vertices from the same shell), a PE P_m may be at most participating in the communication related to two shells: s_i and s_j. Furthermore, the PE P_m may only send data regarding s_j and receive data regarding s_i. Note that in Case 1(a), the PE may both receive and send data regarding s_i, and in the Case 1(c), the PE cannot receive any data related with shell values $\{s_x | s_i < s_x < s_j\}$, because there is no other PE that holds them.

Following this and the nature of the cascading updates explained earlier, we can deduce that the only time a PE P_m may communicate regarding two different shell values is the first round of communications where it may receive information for s_i and send information about s_j. Since P_m is the first PE that contains the shell s_j, it does not receive any information regarding this shell, and does not require any further updates or communications. (We will give more detail about this in the next section, Sec V-B3.)

Hence, after the first *fix round*, the PEs may send and receive information only about the first shell s_i they contain. This means, after the first *fix round*, we can split the PEs into disjoint subgroups according to the first shell value they have, and only communicate within their group instead of ALL-TO-ALL communication.

In summary, all PEs do the first *fix round* the same as explained in V-B1. For the following rounds, each PE only communicates within their own group. The performance of this variant is expected to beat ALL-TO-ALL as the number of PEs increases and the shells are distributed to many PEs.

3) POINT-TO-POINT *(P2P):* On average, realistic graphs contain more than a single shell value. Thus, they rarely require communicating with all other PEs. And, at each round, the number of PEs a PE need to communicate with decreases, rendering ALL-TO-ALL communication unnecessary. Instead, allowing PEs to communicate with each other POINT-TO-POINT, asynchronously, and synchronizing them once each round is better when there are not many PEs that require a fix round.

In this approach, each PE posts its non-blocking receives for the data they might receive from each PE. Then, each posts their non-blocking sends to each PE working on the same shell value. All PEs wait on any of the receives posted, then deserialize and insert all received information into a local priority queue of the vertex IDs, as vertices of the same shell value need to be processed in increasing order.

After transferring the information, all PEs synchronize with an ALLREDUCE to receive the maximum $globalSend$ value from any PE, learning if the fix phase is complete. Here, ALLREDUCE is also not required to be global after the first round of *fix phase*, it can be split into separate ALLREDUCES for each group with respect to the first shell value of each PE.

In general, the communication requirement comes from the updates received from another PE P_j that invalidates the current PE P_i's computation. If a PE P_i is the first PE that works on a shell value, then it cannot receive any updates invalidating its computation. Thus, it should not create a new fix communication packet.

All PEs, within the subset of PEs that work on the same shell may only send fixes if their place in the ascending PE-ID ordered list is greater than or equal to the fix round. For example, PE P_0 (first PE that processes a shell value), can only send fixes in the first round, P_1 may send in first and second rounds, etc.

VI. EXPERIMENTAL EVALUATION

A. Experimental Setup

Our algorithms take a shell distribution as input. Since our algorithm is fast, we use 6 real-world undirected unweighted

Table I
GRAPH INSTANCES.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	avg. deg.	k_{max}	Phase 1 %	Phase 2 %
`cit-Patents`	3,774,768	16,518,947	4.38	64	99.990	0.010
`friendster`	65,608,366	1,806,067,135	27.53	304	99.999	0.001
`soc-LiveJournal1`	4,847,571	42,851,237	8.84	372	99.853	0.147
`twitter`	61,578,414	1,202,513,046	19.53	2,488	99.749	0.251
`uk-2005`	39,459,925	783,027,125	19.84	588	99.984	0.016
`wb-edu`	9,845,725	46,236,105	4.70	448	99.860	0.140

simple graphs with more than 3M vertices, Table I presents number of nodes, edges, average degree, k_{max} value for each graph, and percentages of the computation time breakdown for Phase 1 and 2 of the sequential algorithm. Note that computation time of Phase 2 is negligible compared to that of Phase 1.

Figure 2 shows the normalized number of edges in each k-core value for the graphs in our dataset (cutoff at $k = 80$ for readability, the succeeding shells have near zero percent). This shows that the load of a single k-shell, in real-world graphs, do not comprise a large portion of the graph.

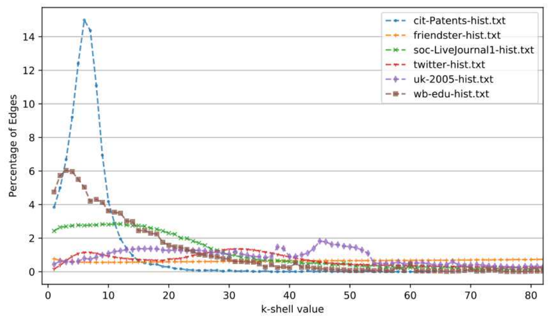


Figure 2. Normalized number of edges for the first 80 k-shells for the graphs in our dataset.

The experiments were conducted on three different microarchitectures: *Cascade Lake, Newell,* and *Haswell*. Table II details the properties of the computing platforms.

Table II
OVERVIEW OF THE ARCHITECTURES.

	Cascade Lake	**Newell**	**Haswell**
CPU	Intel, Xeon 6226	POWER9	Intel, E7-4850
Cores × Sockets	2 × 12	2 × 16	4 × 14
Host Memory	193 GB	320 GB	2 TB
L2-Cache	1 MB	512 KB	256 KB
L3-Cache	19 MB	10 MB	35 MB
Compute Nodes	16	1	1
Networking Technology	Mellanox CS7500 Switch EDR 100 Gb/s		

The experiments were conducted using the GNU G++-9.2.0 compiler (with -O3 optimization level) with C++17 standard libraries. The distributed algorithm uses the MPI implementation MVAPICH2 (v2.3.1) and is executed only on the Cascade Lake system. The shared memory implementation uses the C++11 threads library. All experiments are repeated 25 times and arithmetic means are reported unless otherwise noted.

B. Shared Memory Experiments

For the shared memory algorithm, we have 3 choices and 2 options per choice. Due to space limitations, we only present 5 different variants of the algorithm:

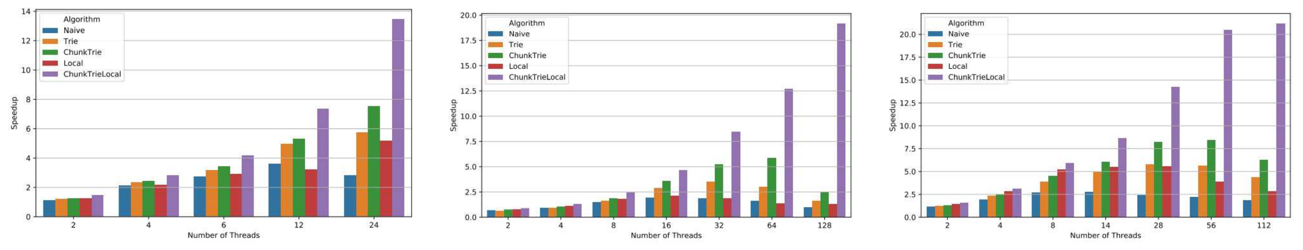
Algorithm	Memory	Granularity	Queue
Naive	GLOBAL	VERTEX	MUTEX
Trie	GLOBAL	VERTEX	TRIE
ChunkTrie	GLOBAL	CHUNK	TRIE
Local	LOCAL	VERTEX	MUTEX
ChunkTrieLocal	LOCAL	CHUNK	TRIE

Figure 3 shows the comparison of these variants on the three architectures in Table II. As expected, Naive performs worst. Our lock-free TRIE priority queue improves the performance significantly as the number of threads increases. Increasing granularity and using chunking further improves the performance. One unexpected, but welcomed result of this experiment was the performance of the Local algorithm. It performs significantly better than Naive, as one would expect, but is also comparable to other sophisticated variants. When combined with those sophisticated variants, the final algorithm, ChunkTrieLocal gave the best performance on all architectures.

C. Distributed Memory Experiments

1) Parameter Tuning for Load Balance: There are multiple parameters to tune. For the sake of simplicity in the presentation, we will test only one at a time and fix the rest. For this experiment, we set the communication scheme to ALL-TO-ALL and only present results for $p = 96$ PEs (4 nodes × 24 cores on Cascade Lake).

The first experiment is to select load balancing algorithm. Figure 4 shows average execution time of the three algorithms using histograms of the 6 real-world graphs as inputs. Just looking at the graph generation runtime, *OB* and *GB* have similar performance. The critical observation here is that *GB*, in general, gives a similar performance to *OB* since the imbalance is bounded by the maximum weight of a single work element (plus alpha). For our dataset, the largest is 2496 with a degeneracy of 2488 and $\alpha = 8$ (twitter). On the other hand, computation of *OB*, requires the computation of a prefix sum. In the case of sequential computation at each process, this takes about 0.2 seconds in `friendster` and `twitter` graphs. In case of a parallel prefix sum, we are at the mercy of network bandwidth and latency for this computation. Thus, the lead goes to *GB*. *OB* does the best job of minimizing the maximum load, however, the computation



(a) Cascade Lake (b) Newell (c) Haswell

Figure 3. Speedup Comparison of Shared Memory: geometric means of speedups for 6 graphs in our dataset for number of processes on: a) Cascade Lake, b) Newell, c) Haswell.

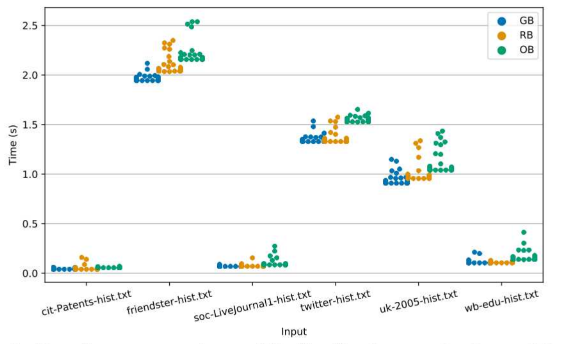


Figure 4. Runtime comparison of 3 distribution methods on 96 cores, $\alpha = 3$, communication scheme = ALL-TO-ALL.

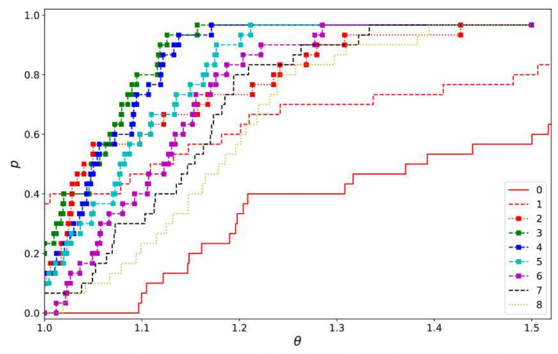


Figure 5. The performance profile showing the comparison of total runtimes with ALL-TO-ALL communication scheme with *GB* variant and 9 α values. For number of cores: (24, 48, 72, 96, 120).

of the optimal distribution itself is significantly slowing down overall runtime.

In the *fix period*, the total runtime is proportional to the highest number of cascading updates (maximum number of fix iterations). *RB* is supposed to help decrease this number by relaxing the load balance in favor of decreasing the communication. However, if the maximum number of iterations is not decreasing, the application will have to wait for the lagging PEs and the relaxation does not practically decrease the runtime. In addition, the increased load imbalance may even slow down the overall runtime.

For the subsequent experiments, we will use *GB*.

The second experiment is to understand the load balancing metric. The parameter α explained in V-A1 can affect the performance significantly. We compared all 3 load balancing algorithms with all $\alpha = [0, 9)$ values. Our results showed similar behavior for different algorithms. Here, due to space limitations we only show the results with *GB*. Figure 5, shows the performance profile comparing alpha values on 24, 48, 72, 96, and 120 cores (1, 2, 3, 4, and 5 nodes respectively) for *GB* decomposition. A *performance profile* shows the ratio of the problem instances in which a variation obtains a value (run time) on a problem instance that is no larger than θ times the best value reached by any variation for that instance [11]. The figure shows that *GB* with $\alpha = 3$ and $\alpha = 4$ give the best overall runtimes, and as α deviates from those values, the performance degrades. In addition, $\alpha = 3$ achieves runtimes within $\times 1.12$ the fastest instance 90% of the time. Thus, for the subsequent experiments, we will use $\alpha = 3$.

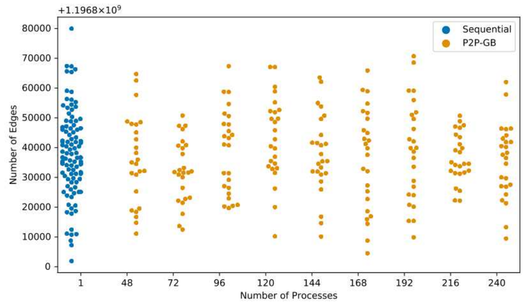
2) Selecting Communication Scheme: Figure 7 shows the comparison of geometric means of speedups for three communication schemes (see Sec. V-B): ALL-TO-ALL, ALL-TO-ALL-SPLIT, and POINT-TO-POINT. All schemes perform similarly within a single node (up to 24 processes). Starting from 24 processes, ALL-TO-ALL-SPLIT starts outperforming ALL-TO-ALL, as the number of processes increases. The performance of POINT-TO-POINT communication surpasses the other variations since it can reduce the number of pairs communicating at each *fix round*, and can better overlap communication and computation.

D. Comparing Average Degrees

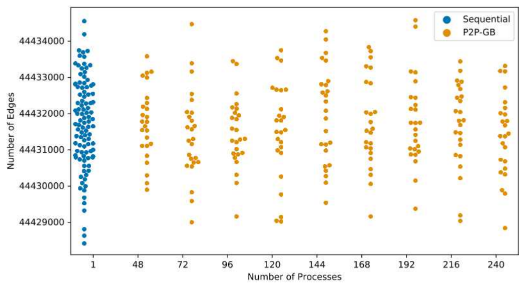
Figure 6 shows the number of edges for the generated graphs (for two sample cases) over a different number of cores (25 repetitions in each case), together with the runs for sequential algorithm (100 repetitions). As shown, the range for the number of edges generated is not vast. Compared to total number of edges, the variation is very small. For `friendster`, the sequential algorithm experiences a maximum of 0.004% variation for 100 repetitions, and it is very similar for the distributed algorithm.

E. Strong Scaling Experiments

Figure 8 shows the strong scaling of our algorithm on `Cascade Lake`, using 1 to 10 nodes, as the geometric mean of speedups for our dataset. Shared memory (ChunkTrieLocal) algorithm is run on single node. Our results on single node



(a) Twitter



(b) soc-LiveJournal1

Figure 6. The number of edges for the generated graphs. (Mind the notation for (a), where the range is 1 196 800 000 to 1 196 880 000).

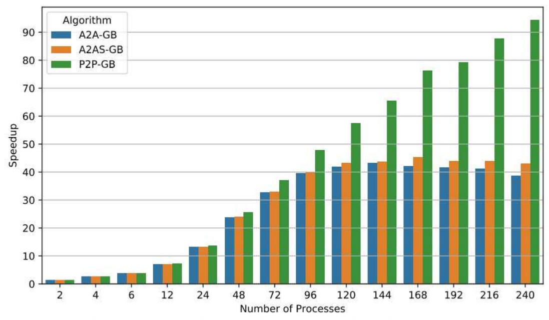


Figure 7. Speedup comparison of communication schemes: geometric means of speedups for 6 graphs in our dataset.

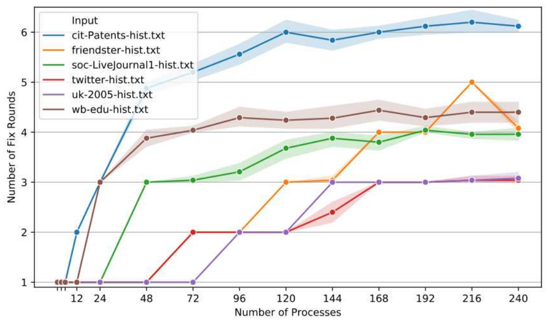


Figure 9. Maximum number of *fix* rounds for any process per number of processes.

comparison showed that shared memory implementations do not provide a significant speedup advantage over the distributed algorithm. As seen in the figure, our shared memory and distributed memory algorithms achieved similar performances on up to 24 PEs, hence we decided not to implement a hybrid code.

Our distributed memory algorithm achieves up to 90 speedup on 240 PEs. The slight decrease in the speedup as the number of PEs increases can be correlated with the increase in the number of *fix* rounds (as seen in Fig. 9).

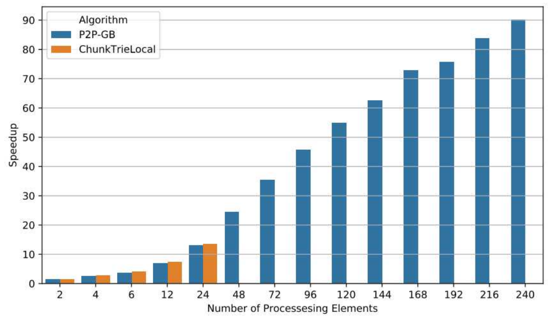


Figure 8. Strong scaling for shared and distributed memory algorithms.

F. Weak Scaling Experiments

Finally, we present the weak scaling experiments. We scale the graphs from our dataset by $\times 1$, $\times 2$, $\times 4$, $\times 8$, $\times 16$, $\times 32$, $\times 64$, and $\times 128$. (We just multiply the size of each shell by this amount to reach to a bigger graph. Curve fitting over the histograms yielded insignificant variations.) Figure 10(a) shows the weak scaling for our algorithm. The results show that scaling from $\times 1$ to $\times 128$, our runtime goes at most up to twice the initial the runtime. The worst result in this experiment is for the smallest graph `cit-Patents`, which also has the lowest k_{max} value, thus with scaling, communication requirements increase more for this particular graph. The increase in the runtime as the number of PEs and the input size increased proportionally can be further explained with the increase on the number of *fix* rounds, as displayed in Fig. 10(b).

Our biggest scale ($\times 128$), reaches about 2^{33} vertices and 2^{37} edges. The distributed algorithm can generate this graph in less than 50 seconds on 384 cores.

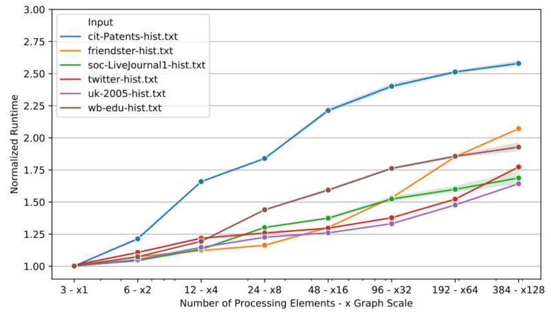
VII. Conclusion

Our algorithms focus on another well-known graph property that has not been explored for scalable graph generation: cores decomposition. We presented scalable shell sequence based graph generators for both shared memory and distributed architectures. We analyzed our algorithms' strength and weaknesses, and empirically showed their scalability.

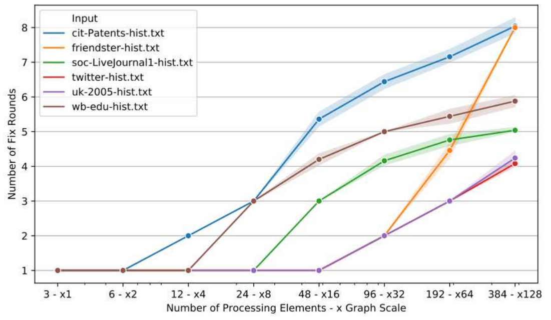
Our future work includes approaches to more realistic graph generation using shell histograms, extensive analysis of the quality of generated graphs compared to real-world datasets and conformity to well-known graph properties.

Acknowledgments

We would like to thank Kasimir Gabert for his careful review and editing of the paper.



(a) Normalized Running time for MPI on 6 graphs in our dataset. The runtimes for each repetition on each graph is divided by the shortest runtime achieved for that graph's scaled version.



(b) Maximum number of *fix* rounds for any process per number of processes.

Figure 10. Weak scaling.

References

[1] M. A. Al-garadi, K. D. Varathan, and S. D. Ravana, "Identification of influential spreaders in online social networks using interaction weighted k-core decomposition method," *Physica A*, vol. 468, 2017.

[2] M. Altaf-Ul-Amine, K. Nishikata, T. Korna *et al.*, "Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences," *Genome Informatics*, vol. 14, pp. 498–499, 2003.

[3] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *NIPS*, 2005.

[4] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases," *Networks and Heterogeneous Media*, vol. 3, no. 2, 2008.

[5] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[6] R. Bauer, M. Krug, S. Meinert, and D. Wagner, "Synthetic road networks," in *International Conference on Algorithmic Applications in Management*. Springer, 2010, pp. 46–57.

[7] M. Baur, M. Gaertler, R. Görke, M. Krug, and D. Wagner, "Generating graphs with predefined k-core structure," in *Proceedings of the European Conference of Complex Systems*, 2007.

[8] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "A model of internet topology using k-shell decomposition," *PNAS*, vol. 104, no. 27, pp. 11 150–11 154, 2007.

[9] 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.

[10] F. Chung and L. Lu, "The average distances in random graphs with given expected degrees," *Proceedings of the National Academy of Sciences*, vol. 99, no. 25, pp. 15 879–15 882, 2002.

[11] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical programming*, vol. 91, no. 2, pp. 201–213, 2002.

[12] A. Elmokashfi, A. Kvalbein, and C. Dovrolis, "On the scalability of bgp: The role of topology growth," *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 8, pp. 1250–1261, 2010.

[13] P. Erdos, "On the evolution of random graphs," *Publications of the mathematical institute of the Hungarian academy of sciences*, vol. 5, pp. 17–61, 1960.

[14] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz, "Communication-free massively distributed graph generation," in *IPDPS*, May 2018, pp. 336–347.

[15] D. Garcia, P. Mavrodiev, and F. Schweitzer, "Social resilience in online communities: The autopsy of friendster," in *COSN*, 2013.

[16] P. W. Holland, K. B. Laskey, and S. Leinhardt, "Stochastic blockmodels: First steps," *Social networks*, vol. 5, no. 2, pp. 109–137, 1983.

[17] V. Karwa, M. J. Pelsmajer, S. Petrović, D. Stasi, and D. Wilburne, "Statistical models for cores decomposition of an undirected random graph," *Electron. J. Statist.*, vol. 11, no. 1, pp. 1949–1982, 2017.

[18] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature physics*, vol. 6, no. 11, 2010.

[19] T. G. Kolda, A. Pınar, 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.

[20] S. Lamm, "Communication efficient algorithms for generating massive networks," 2017, master's thesis.

[21] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical review E*, vol. 78, no. 4, p. 046110, 2008.

[22] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

[23] Y. Liu, M. Tang, T. Zhou, and Y. Do, "Core-like groups result in invalidation of identifying super-spreader by k-shell decomposition," *Scientific reports*, vol. 5, 2015.

[24] Y. Matias, S. C. Sahinalp, and N. E. Young, "Performance evaluation of approximate priority queues," in *Proceedings of Fifth DIMACS Implementation Challenge*, 1996.

[25] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: An approach to universal topology generation," in *MASCOTS 2001*, 2001, pp. 346–353.

[26] F. Morone, K. Burleson-Lesser, H. Vinutha, S. Sastry, and H. A. Makse, "The jamming transition is a k-core percolation transition," *Physica A*, vol. 516, pp. 172–177, 2019.

[27] F. Morone, G. Del Ferraro, and H. A. Makse, "The k-core as a predictor of structural collapse in mutualistic ecosystems," *Nature Physics*, vol. 15, no. 1, p. 95, 2019.

[28] C. Peng, T. G. Kolda, and A. Pınar, "Accelerating community detection by using k-core subgraphs," *arXiv preprint arXiv:1403.2226*, 2014.

[29] M. Penrose *et al.*, *Random geometric graphs*. Oxford university press, 2003, vol. 5.

[30] M. Penschuck, "Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory," in *16th International Symposium on Experimental Algorithms (SEA 2017)*, 2017.

[31] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel Distrib. Comput.*, vol. 64, no. 8, pp. 974–996, aug 2004.

[32] P. Sanders and C. Schulz, "Scalable generation of scale-free graphs," *Information Processing Letters*, vol. 116, no. 7, pp. 489–491, 2016.

[33] C. Seshadhri, A. Pınar, and T. G. Kolda, "An in-depth analysis of stochastic kronecker graphs," *J. ACM*, vol. 60, no. 2, pp. 13:1–13:32, May 2013.

[34] K. Shin, T. Eliassi-Rad, and C. Faloutsos, "Corescope: Graph mining using k-core analysis - patterns, anomalies and algorithms," in *ICDM*, 2016.

[35] 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, pp. 1–14.

[36] I. Stanton and A. Pınar, "Constructing and sampling graphs with a prescribed joint degree distribution," *J. Exp. Algorithmics*, vol. 17, pp. 3.1–3.25, 2012.

[37] C. L. Staudt, M. Hamann, A. Gutfraind, I. Safro, and H. Meyerhenke, "Generating realistic scaled complex networks," *Applied Network Science*, vol. 2, no. 1, p. 36, Oct 2017.

[38] M. von Looz, M. S. Özdayi, S. Laue, and H. Meyerhenke, "Generating massive complex networks with hyperbolic geometry faster in practice," in *IEEE High Perf. Extreme Computing Conf. (HPEC)*, 2016.

[39] F. Zhao and A. K. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *PVLDB*, vol. 6, no. 2, 2012.