Communication-free Generation of Graphs with Planted Communities

Master Thesis of

Adrian Feilhauer

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Michael Hamann
	Sebastian Lamm

Time Period: 1st June 2020 – 30th November 2020

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, November 30, 2020

Abstract

Graphs representing social settings often contain structures where vertices in communities are closely linked. Algorithms analyzing these networks need to be evaluated on graphs whose structure is known. We present several models with generators to produce graphs with specific community structures. The generators developed in this thesis work in a distributed communication-free way. This allows generating graphs without being restricted by the rate of communication between used processing units. The scaling of the simpler models is near perfect, and those models can generate 10^8 edges per processor and second. This results in graphs with 2^{31} vertices and more than 2^{42} edges being generated in less than ten minutes on 128 cores. The more complex models featuring power law distributed community sizes and membership counts scale reasonably well. Even with sub-optimal load balancing during the tests, CKB graphs with 2^{25} vertices can be generated on 128 cores with an average of 10^6 edges per processor and second.

Deutsche Zusammenfassung

Graphen, die soziale Strukturen widerspiegeln, beinhalten oft sogenannte Communities von Knoten, deren Mitglieder untereinander häufig verbunden sind. Algorithmen, die derartige Netzwerke analysieren, sind zur Qualitätskontrolle auf Graphen mit bekannter Communitystruktur angewiesen. Hier werden einige Modelle mit zugehörigen Generatoren vorgestellt, die Graphen mit spezieller Communitystruktur erzeugen. Die Generatoren arbeiten in verteilten Systemen ohne Kommunikation. Dadurch können Graphen generiert werden, ohne von Beschränkungen der Kommunikation zwischen Prozessoren beeinträchtigt zu werden.
Die einfacheren Modelle skalieren nahezu perfekt. Mit 128 Kernen können dort Graphen mit 2^{31} Knoten und über 2^{42} Kanten in weniger als zehn Minuten erzeugt werden. Das entspricht ca. 10^8 Kanten pro Sekunde und Prozessor. Komplexere Modelle mit nach Potenzgesetz verteilten Communitygrößen und Knotenmitgliedschaften skalieren einigermaßen gut. Obwohl die Verteilung der Last zwischen den verwendeten 128 Kernen suboptimal war, konnten für Graphen mit 2^{25} Knoten durchschnittlich 10^6 Kanten pro Sekunde und Prozessor generiert werden.

Contents

1. Introduction

This thesis covers algorithms to generate graphs with a community structure in a distributed and communication-free way. In this context a community is a set of vertices that are more closely connected to each other than to vertices outside of the community. Generating graphs with communities allows verifying community detection algorithms and analysis of how communities affect a graphs properties. The framework is based on the KaGen library [FLS$^+$18] and this work expands the library by adding five community based models. They are build on each other, gradually increasing the complexity up to the CKB model [CKB$^+$14]. The intermediate models also aim at representing useful settings.

1.1 State of the Art

The KaGen [FLS$^+$18] library provides algorithms to generate community free graphs of several models using parallel computation without communication between processing entities (PEs). The models featured there are Erdős Rényi graphs, Delaunay graphs, geometric graphs and hyperbolic graphs. Erdős Rényi graphs can be generated both with directed or undirected edges and in $G(n, p)$ or $G(n, m)$ modality.
For the parallelization of the Erdős Rényi model, the adjacency matrix gets split into chunks that are distributed among the PEs. In the case of directed graphs successive rows form a chunk and each PE generates edges for the same number of rows. Undirected graphs only get edges generated in the lower triangle of the adjacency matrix. The rows are formed as in the directed case, but the segmentation process is also done for columns. This results in rectangular chunks below the main diagonal and triangular chunks on the main diagonal. A PE generates edges for all rectangle chunks in its rows and the triangle chunk at the end of those rows. It also generates the edges for all rectangle chunks in the same column of chunks as its triangle chunk. Both chunk structures are visualised in Figure 1.1. This ensures that each PE generates all edges incident to the vertices corresponding to the rows assigned to it. Seeding ensures that rectangle chunks whose edges are generated by two different PEs contain the same edges on both PEs.

Lancichinetti et al. introduced the LFR model [LF09] featuring overlapping communities. Community sizes are power law distributed in the LFR model. The distribution of membership counts can be specified. Vertices are assigned to communities at random using a configuration model. For edge generation, vertices have a degree that is drawn from a power law distribution. A mixing parameter indicates to how large the percentage of edges to vertices with shared communities is. The remaining edges to reach the intended

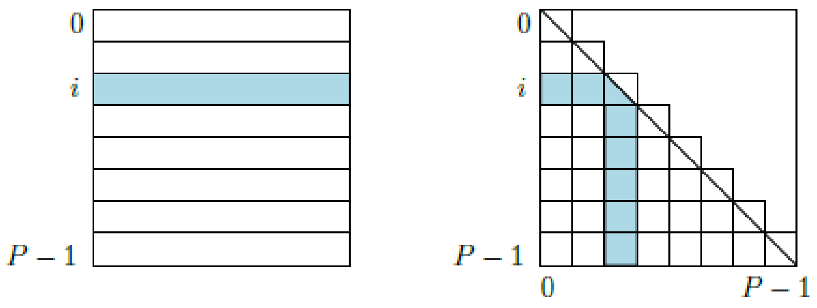
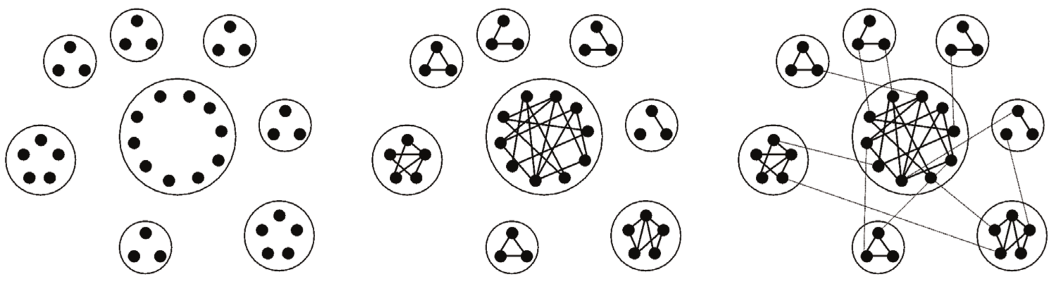


Figure 1.1: Possible chunk structures in KaGen.[FLS+18]

vertex degree are edges to vertices without shared communities. If the generating of inter-community edges has accidentally created an edge between vertices sharing a community, a rewiring process is employed to move the edges enforcing their inter-community nature.

The variation of the stochastic block model used in [Pei15] is another generative model for graphs with overlapping communities. Edges here are generated based on an edge-count matrix specifying how many edges between any pair of edges should be generated. Intra-community edges are edges between a community and itself and have to be passed with twice the intended edge count because the edge-count matrix stores vertex degrees for edges of the specified type. This edge generation allows multi-edges, but their occurrence is negligible for large graphs.

The BTER model [KPPS14] generates graphs with dense Erdős Rényi blocks as communities. But the generator can match an arbitrary degree distribution passed as a parameter for the vertices. The vertices are grouped into affinity blocks ideally filled with vertices of the same degree that could form a clique. This results in many small communities and few large communities. The density of each community is based on clustering coefficients passed as parameters for the generator. The inter-community edges are generated using a Chung-Lu model with the remaining degree sequence. A visualization over the process to generate a BTER graph is given in Figure 1.2. Because the community structure of BTER graphs is hard to influence, BTER graphs are not usually used as benchmarks for community detection.



(a) Preprocessing: Distribution of nodes into affinity blocks.

(b) Phase 1: Local links within each affinity block.

(c) Phase 2: Global links across affinity blocks.

Figure 1.2: Visualization of the phases used to generate BTER graphs.[KPPS14]

Community sizes and vertices' membership counts are both power law distributed in graphs generated by the CKB model [CKB+14]. Communities are Erdős Rényi blocks but there are no predefined degree sequences for its members. Instead, edges are generated uniformly at random based on a probability p_i depending on the community's size $|c_i|$. Since $p_i = \frac{\alpha}{|c_i|^\gamma}$ for parameters $\alpha > 0$ and $\gamma \in (0, 1)$, the edge count in communities increases superlinearly with community size, but the density decreases with increasing community size. Edges between different communities are generated by considering an ϵ-community that contains all vertices and generates edges with a probability p_o passed as a parameter.

1.2 Contribution

This thesis expands the KaGen library [FLS+18] by adding generators for graphs with Erdős Rényi communities. For this, the approach for the $G(n, p)$ model to generate graphs in a communication-free setting is generalized. The generalization covers overlapping and not overlapping communities. Community sizes and membership counts are taken from different distributions. All generators can create directed and undirected graphs. The first generators start by introducing communities and more and more features are added up to a communication-free generator for CKB graphs [CKB+14].

For generating basic models the generalization consists of splitting the chunks of the adjacency matrix in homogeneous blocks containing only edges within a community or only edges between different communities. For these blocks the edge generation used in the $G(n, p)$ model [FLS+18] can be used.

To be able to distribute a fixed number n of vertices among the communities, the samples drawn from the power law distribution have to add up to n. To achieve this, we consider two different sampling routines. Linear sampling can be done in-place, but scales poorly. Sqrt-based sampling stores results in a compressed way resulting in its space and time demands scaling in $\mathcal{O}(\sqrt{n})$.

To realize overlapping communities the assignment of vertices to communities has to be addressed. We discuss the problems and challenges of doing this assignment in a distributed communication-free manner and present two possible solutions. Group based assignment guarantees exact membership counts and community sizes as sampled with acceptable runtime scaling. But the overlap between communities is not sufficiently random and there are unwanted artifacts in the overlap structure.
These issues do not exist for the probabilistic assignment since it assigns vertices to communities completely at random, producing perfectly random overlap between communities. The tradeoff is that probabilistic assignment only produces the intended membership count as expectation with a non-zero variance. So membership counts in the generated graph will vary even if the specified membership counts are identical for all vertices. Community sizes are guaranteed to be exactly as sampled and runtime scaling is acceptable.

For the communication-free generating of CKB graphs we develop a generalised version of group based assignment, but it still suffers from the issue of incorrect community overlap. To mitigate some distortion to the distribution of membership counts caused by the probabilistic assignment, we introduce pre-assignment as an optional routine. If used, it guarantees that each vertex is a member of at least one community and the number of vertices with exactly one community has the expectation as sampled from the power law distribution.

We test the generators by evaluating several metrics on the generated graphs, and the produced graphs show the intended properties. Scaling for the basic models is practically perfect. For models with power law distributed community sizes the workload scales very good, but the PEs handle uneven amounts of the work. This can in theory be remedied. The CKB generator's work is dominated by the process of assigning members to communities for settings with small communities. As such the weak scaling is imperfect and the workload per PE increases roughly by $\mathcal{O}(\sqrt{n})$.

1.3 Outline

The basic models covered in Chapter 2 introduce communities. For the equal-communities model in Section 2.1 all communities have the same size. In graphs generated using the unique-communities model from Section 2.2 vertices pick their community uniformly at random.
Both models guarantee that every vertex is member of exactly one community and can generate directed or undirected graphs.

To come closer to the CKB model while producing useful graphs, the next logical step is to use power law distributed community sizes. The PL-communities model discussed in Chapter 3 still has each vertex being part of exactly one community, but the sizes of the communities are drawn from a power law distribution. For drawing random community sizes linear sampling is considered in Subsection 3.1.2 and discarded due to bad scaling. Instead, sqrt-based sampling described in Subsection 3.1.3 is used to sample community sizes. Like in the CKB model the probability $p_i = \frac{\alpha}{|c_i|^\gamma}$ for edges within a community c_i depends on the communities size.

Having incorporated power law distributions into the models, what is missing are overlapping communities. The PL-multicommunity model discussed in Chapter 4 has similarities to the LFR model [LF09] where all vertices have the same membership count and community sizes are power law distributed. However, here vertex degrees within one community are not drawn from a power law distribution, but are based on a fixed probability for all members of the community to keep the communities Erdős Rényi blocks. To determine the members of communities, group based assignment in Subsection 4.1.2 and probabilistic assignment in Subsection 4.1.3 are discussed.

The final generalization to be able to generate CKB graphs is to use power law distributed membership counts. The group based assignment is generalised for this use case in Subsection 5.1.1. Pre-assignment to improve the distribution created by probabilistic assignment is explained in Subsection 5.1.2. Combining all techniques developed in the earlier models allows generating CKB graphs in Chapter 5. These graphs have overlapping Erdős Rényi communities whose sizes are power law distributed and whose density depends on the community size. The vertices' membership counts are also power law distributed.

Results from testing metrics of the generated graphs and scaling of an implementation of these generators are presented in Chapter 6.

1.4 Preliminaries

In this work, a graph G is defined as a pair (V, E). V is the set of vertices and $E \subset V \times V$ the set of edges. A graph is said to be undirected if $(u, v) \in E \implies (v, u) \in E$, otherwise it is called directed. An edge $(v, v) \in E$ is called a loop. If the edge (u, v) is contained in the graph, it is said that u and are v adjacent and (u, v) is incident to u and v.
A community is a set of vertices that are – for usual parametrizations – more likely to be connected by edges than vertices not sharing a community. For community $c_i = \{v_1, v_2, v_3, v_4\}$ the vertices contained are called its members. The community size $|c_i| = 4$ is the number of members the community c_i contains. The internal degree of vertex v_1 in c_i is the number members of c_i adjacent to v_1.
For sampling random elements mainly the binomial and the hypergeometric distribution will be used. Let $\text{Bin}(n, p)$ be the binomial distribution, drawing n times with the probability of a success in each step being p. Let $\text{Hyp}(n, K, N)$ be the hypergeometric distribution, drawing n times from a set initially containing N elements of which K are considered successes.

2. Basic Models

This chapter describes generators for two basic models. Both models have similarly sized communities and fixed edge probabilities.

2.1 Equal Communities Model

The equal communities model features communities that are all of the same size.

2.1.1 Description and Parameterization

The first model considered here is very basic. All communities have nearly the same size. Sizes may differ by up to one to fit all vertices. Communities do not overlap and every vertex is member of one community. The number of communities is set as a parameter. Edges can appear either within a community or linking two different communities. Each pair of vertices gets connected by an edge with a specified probability depending on whether the vertices are in the same community or not. So there is a chance p_i for a possible intra-community edge to be created and a different, usually notably smaller, chance p_o for a possible inter-community edge to be created.
The parameters for the equal communities model are listed in Table 2.1. An example graph generated by this model can be seen in Figure 2.1.

Parameter	Description	Usual Value
n	number of vertices	-
c	number of communities	$n/2000$
p_i	probability for each edge within a community to exist	0.1
p_o	probability for each edge between communities to exist	$2/n$
self_loops	should loops be possible edges	false

Table 2.1: List of parameters for the equal communities model.

2.1.2 Algorithm

The simplicity of the community structure allows using a similar approach to the $G(n, p)$ model of KaGen [FLS$^+$18]. So the adjacency matrix is split into chunks, whose borders are only depended on their id. For each chunk the edges can be generated without knowledge about other chunks.

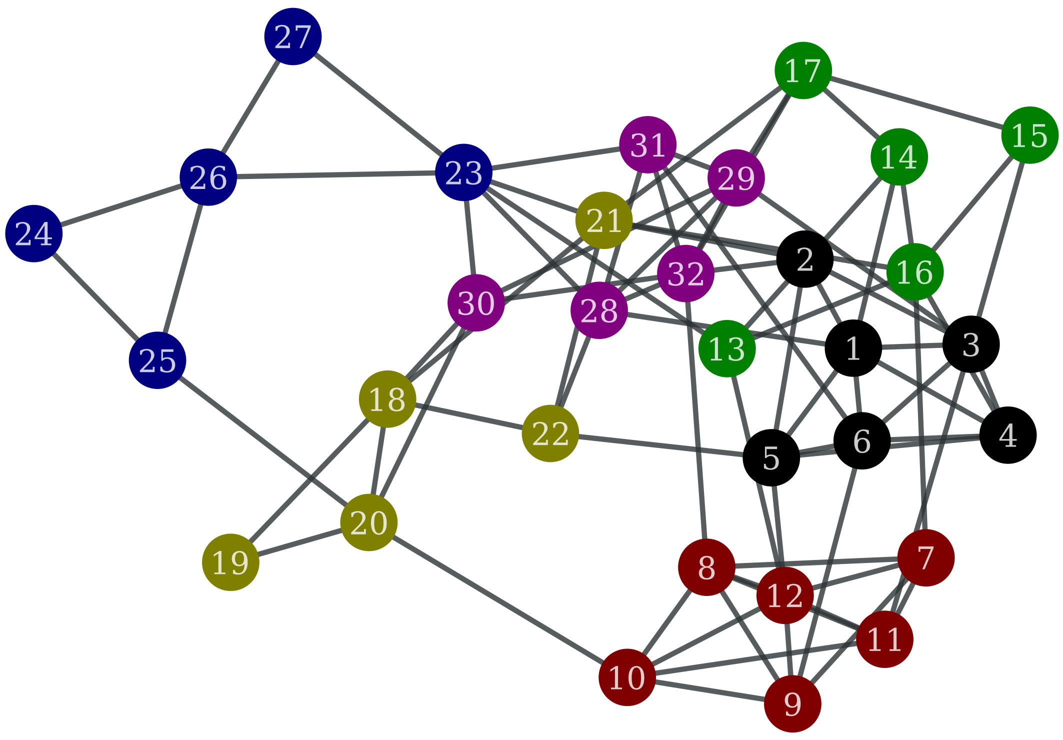


Figure 2.1: An undirected graph generated with the equal-communities model ($n = 32$, $c = 6$, $p_i = 0.7$, $p_o = 0.08$). Colours indicate community membership.

To allow large homogeneous blocks in the adjacency matrix, vertices are assigned to communities block-wise. So if the first community has size $|c_1|$, then the first $|c_1|$ vertices are its members. This results in intra-community edges being in square blocks on the main diagonal of the adjacency matrix. All graphs generated by this model will be isomorphic to a graph where the communities form this kind of block structure on the main diagonal. To scramble this structural information and prevent community detection algorithms from simply searching for the boundaries of these blocks, the order of vertices could be randomly permuted before writing edges to the result file.

With the structure of the adjacency matrix set, it has to be split into chunks and those chunks have to be split into homogeneous blocks where edge generation will occur. Chunks are formed exactly as in the $G(n, p)$ model of the KaGen library [FLS$^+$18], so for directed graphs consecutive rows of the adjacency matrix form a chunk. For undirected graphs chunks to the left of the main diagonal are rectangles while chunks on the main diagonal are triangular. A PE processes a line of such chunks that is reflected down at the main diagonal. This allows chunks to generate all edges for each source vertex it handles while maintaining consistency across all PEs without requiring communication.
Within each chunk blocks are split along community boundaries, so that a block either generates edges between communities, or within one community. This is visualized in Figure 2.2 and described in further detail in Subsection 2.1.3.

The number of edges to be generated in one such block is sampled from a binomial distribution. Hence for an inter-community edge block with n possible edges, $\mathrm{Bin}(n, p_i)$ determines how many edges are generated. A block with n possible inter-community edges determines its number of edges by drawing from $\mathrm{Bin}(n, p_o)$. These edges are then sampled uniformly at random from the set of possible edges.

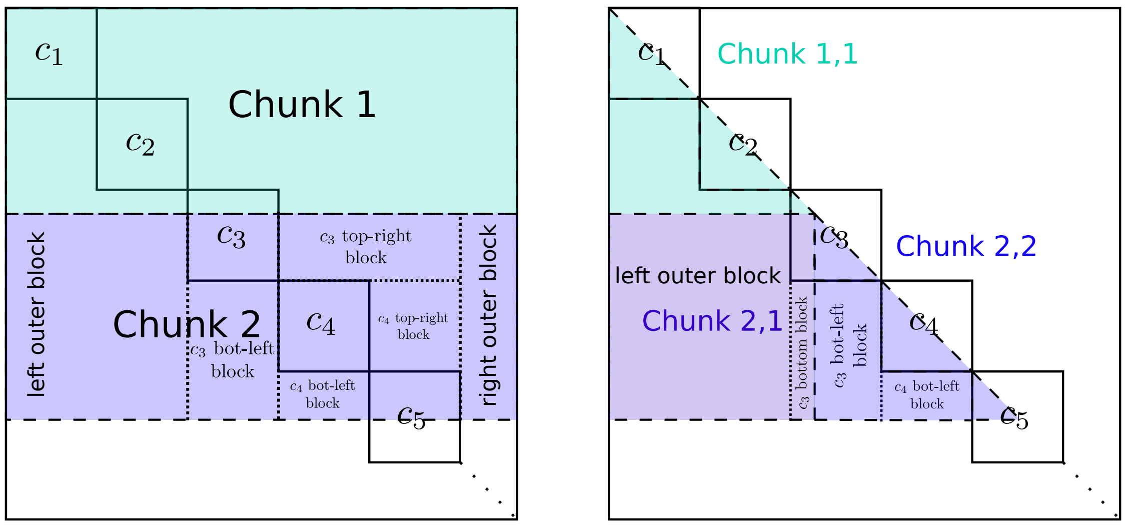


Figure 2.2: Example of how chunks are split into homogeneous blocks. The communities are blocks on the main diagonal of the adjacency matrix. The coulored areas are the chunks. The left graphic depicts the directed case and the right graphic visualizes the undirected case where only the lower triangle matrix is generated.

2.1.3 Implementation Details

The splitting of chunks into homogeneous blocks works differently for the directed and the undirected case due to their different chunk structure. In the directed case chunks are consecutive lines. The communities lie on the main diagonal of the adjacency matrix. Therefore there can be a block of inter-community edges left and right of the block handling the inner columns that also contain intra-community edges. The first and last chunk only contain a right outer respectively left outer block because the central mixed block is extending up to one side of the matrix. These left and right outer blocks are split of and passed to edge generation using p_o as edge probability. Figure 2.2 visualizes this process on the left for Chunk 2.
The central mixed block is successively split into homogeneous blocks by creating one block containing the topmost community part handled by the remaining central block. This block is part of a community, so its edge generation is called with edge probability p_i. The cells below this split off community block are inter-community edges, so they are a homogeneous block of outer edges to be created using probability p_o. The same holds for the part of the adjacency matrix that is to the right of the split off community. So it too is an outer block whose edges are created using p_o as edge probability. In Figure 2.2 on the left these are called c_i bot-left block and c_i top-right block. Thus, the topmost community part with all of its lines and columns of the mixed inner block has been dealt with and the remaining part has the same structure. Hence, it is processed in the same way until the last community part of the chunk has had edges generated.

The undirected model differentiates two different sub-chunks. There are triangle chunks on the main diagonal and rectangle chunks to the left of it. Handling of rectangle chunks is straight forward. Most rectangle chunks only contain outer edges and therefore are already homogeneous blocks ready for edge generation. But rectangle chunks near the main diagonal may still contain parts of up to one community in their upper right because chunk and community structure do not necessarily line up. However, they can not contain parts of multiple communities, because new communities always begin with their leftmost column on the main diagonal of the adjacency matrix. But chunks on the main diagonal are triangle chunks not rectangle chunks, so a rectangle chunk only needs to split off parts of at most one community.

This splitting off is done similarly to the directed case. First the left block of outer edges is split off and then the remaining mixed block is sliced horizontally, separating the upper block with intra-community edges from the lower block containing inter-community edges. In the example in Figure 2.2 on the right this is shown for Chunk 2,1, which contains a small part of c_3.
Triangle chunks on the main diagonal are similar to the mixed block in the directed case, but they are triangular, so the right upper part does not get generated. Thus, during successive splitting only the left lower outer block and the inner block have to be created. In Figure 2.2 on the right this is shown for Chunk 2,2. The right upper outer block does not have to be dealt with. The left lower outer block is rectangular, so it is processed the same as any outer block from a rectangle chunk. The edges of the intra-community block are arranged in a triangle and are generated like the triangle chunks in [FLS+18]. Whether the edges directly on the main diagonal are part of the chunk depends on whether loops are permitted as edges.

While it would be easily possible to reorder vertex ids to break the pattern that all members of a community have consecutive ids, this is forgone in the implementation. As a result, community membership is trivially organised in this model, which may aid in visualisation or debugging scenarios.

2.2 Unique Communities Model

The unique communities model has a fixed number of communities. Vertices pick their community uniformly at random.

2.2.1 Description and Parametrization

This model is fairly similar to the equal communities model. However, in this model, vertices pick a community uniformly at random. So while the communities in themselves have the same structure, their size can differ based on how many vertices ended up picking them. The number of communities is set as a parameter. Edges are generated using an inter-community edge probability p_o and an intra-community edge probability p_i as in the equal communities model.
The parameters for the unique-communities model are listed in 2.2. A graph generated with this model is shown in Figure 2.3. It features communities both smaller and larger than those generated by the equal communities model.

Parameter	Description	Usual Value
n	number of vertices	-
c	number of communities	$n/2000$
p_i	probability for each edge within a community to exist	0.1
p_o	probability for each edge between communities to exist	$2/n$
self_loops	should loops be possible edges	false

Table 2.2: List of parameters for the unique communities model.

2.2.2 Algorithm

Similar to the approach employed in the equal communities model, the adjacency matrix is split into homogeneous blocks. These blocks either only contain intra-community edges or exclusively handle inter-community edges. The overlaying chunk structure used to assign parts of the adjacency matrix to different PEs is slightly more complex, because the id of a chunk no longer directly specifies the size and count of all communities within the chunk.

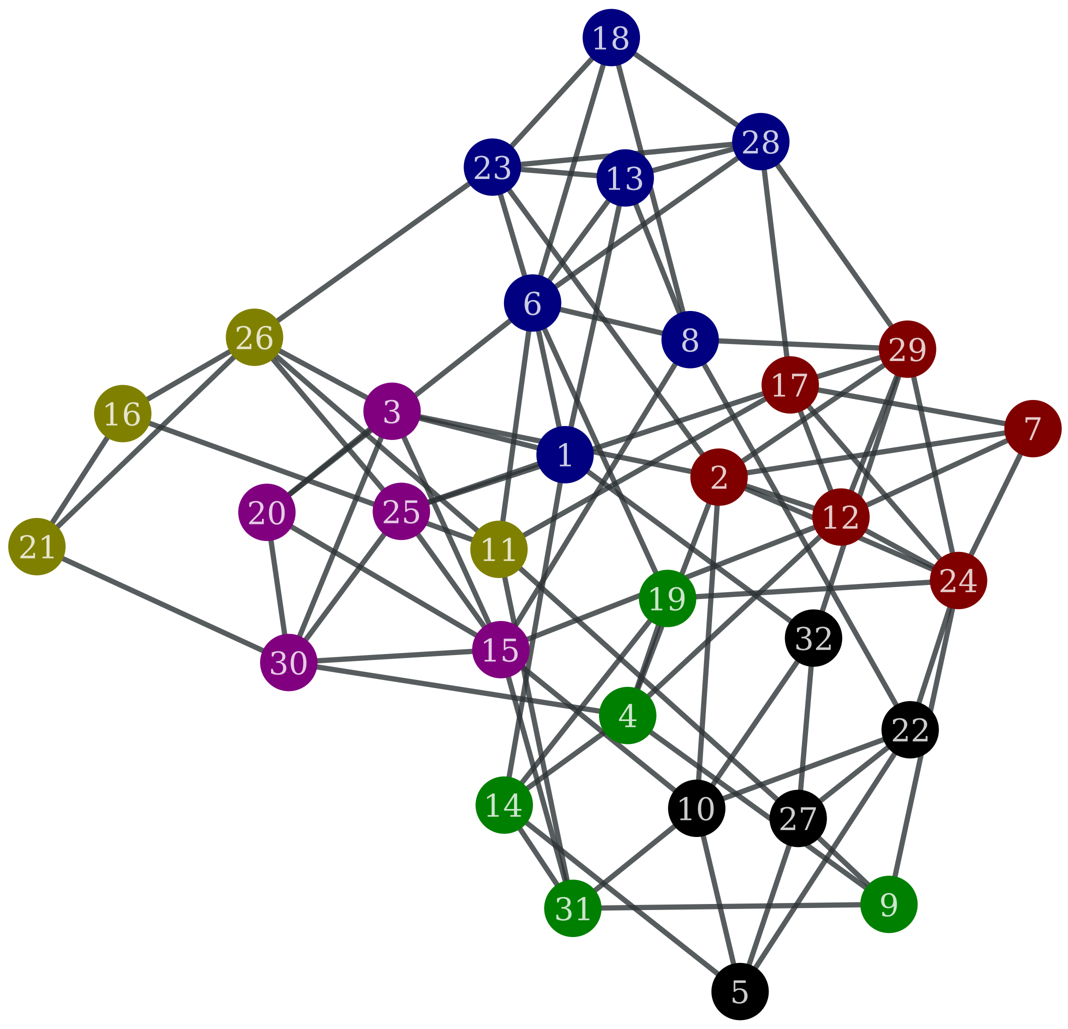


Figure 2.3: An undirected graph generated with the unique communities model ($n = 32$, $c = 6$, $p_i = 0.7$, $p_o = 0.08$). Colours indicate community membership.

To keep homogeneous blocks as large as possible within each chunk, vertices are still assigned to chunks based on which community they belong to. So apart from those communities that lie at the borders between chunks, all members of a community are handled by the same chunk. Hence, the picking of the community by each vertex has to be simulated based on the communities. This assignment of members to communities and therefore also the determining of the community sizes is done with a recursive process.
It starts with all communities and all vertices being available. In each step of the recursion, the number of available communities is split into two halves. A binomial random variable determines how many of the available vertices pick communities in each of the halves. Assume there are $|c_{\text{left}}|$ communities in the left split, $|c_{\text{total}}|$ communities still available at this recursion step, and they have a combined size of n. Then the combined size of all communities of the left split is drawn from $\text{Bin}(n, |c_{\text{left}}|/|c_{\text{total}}|)$.

Upon this structure, the edge generation works as it did for equal communities. For the undirected case, chunks are successive rows that get split into inter- and intra-community edge blocks for generating edges with probabilities p_o and p_i. In the directed case there are triangular chunks on the main diagonal and rectangular chunks left of it. A PE deals

with all chunks in its row up to the main diagonal and then with the chunks of the column where its row reached the main diagonal of the adjacency matrix. This allows each PE to generate all edges of each vertex it is handling.
These rectangle and triangle chunks are again split into homogeneous blocks. Blocks containing only intra-community edges generate edges with probability p_i while those containing inter-community edges use p_o. This model does randomly reorder the vertices' ids before edge generation, thus scrambling structural information about vertex community pairings.

2.2.3 Implementation Details

The recursion simulating the assignment of vertices to communities can be cropped whenever only communities handled by other chunks are left. The full sizes of all communities to be processed by the calling chunk are returned, as well as information about how many vertices of the first incomplete community of the chunk were already processed by a previous chunk and the id of the first community of the chunk.
For the splitting of chunks into homogeneous blocks, the same technique used for equal communities is utilised. To find borders between communities the information about already processed vertices along with the full size of all partially contained communities and the number of vertices of the first community left to process in this chunk is used. The size of previous chunks is given by the current chunk's id and the fact that all chunks have the same size, except for the initial chunks that may process one more vertex if the vertices could not be distributed evenly among all chunks. This covers the different communities' sizes and allows the splitting off of homogeneous blocks to be done exactly as described for equal communities in Section 2.1.

The implementation of this model uses a hash function $h(x)$ – a linear congruential generator to be exact – to change the order of the vertices before pushing the generated edges to the IO. If an edge from vertex v_i to v_j is internally generated, the edge $(v_{h(i)}, v_{h(j)})$ is pushed to the generated graph. As such the entire algorithm works on an adjacency matrix where the communities are blocks on the main diagonal, which allows it to profit from larger homogeneous blocks for edge generation. At the same time the generated graph does no longer have this artificial blocklike structure.

3. Power Law Communities

Power Laws are observed in many contexts. Many papers have discussed real world graphs and models where vertex degrees are power law distributed e.g. [ALPH01][XYLZ14]. There are also some models that contain communities whose sizes are power law distributed [LF09][CKB+14]. The CKB model in particular reports that it achieves power law distributed degrees with community sizes and the number of memberships per vertex also following a power law[CKB+14].

3.1 Sampling from a Power Law Distribution

Definition 3.1 (Power Law Distribution)**.** *Power law describes a situation, where relative changes to one variable x result in proportional relative changes to another variable y, independent of absolute values. The exact change is governed by two parameters, a constant factor a and an exponent $-k$, resulting in the relation $y = a \cdot x^{-k}$.*
In this context a power law distribution $PL_{v_{\min},v_{\max},k}$ is defined by the smallest and largest values $v_{\min}$ and $v_{\max}$ that may be sampled, replacing the factor a, as well as the exponent $-k$, denoting how much more likely small samples are than large ones.

To allow easy aggregation of probabilities and access to an efficiently calculated expected value, sampling will be based on a continuous power law function, even though only discrete values are sampled. The probability with which a given value v is sampled is the integral over the continuous distribution from $v - 0.5$ to $v + 0.5$.
To obtain samples, inverse transform sampling will be used. This requires the inverse of the cumulative density function, which is then used to map randomly drawn numbers in $[0, 1]$ to the value of the CDF.
For convenience the following constants will be used:
$s_{\min} := v_{\min} - 0.5$
$s_{\max} := v_{\max} + 0.5$
$a := \int_{s_{\min}}^{s_{\max}} x^{-k} dx = \begin{cases} \log s_{\max} - \log s_{\min}, & \text{if } k = 1 \\ \frac{s_{\max}^{1-k}}{1-k} - \frac{s_{\min}^{1-k}}{1-k}, & \text{if } k \neq 1 \end{cases}$
With that the probability density of $PL_{v_{\min},v_{\max},k}$ is:

$$f_{PL}(x) = \begin{cases} \frac{x^{-k}}{a}, & \text{if } x \in [s_{\min}, s_{\max}] \\ 0, & \text{if } x \notin [s_{\min}, s_{\max}] \end{cases} \tag{3.1}$$

Lemma 3.2. *The inverse CDF of $f_{PL}(x)$ is:*

$$F_{PL}^{-1}(x) = \begin{cases} s_{\min} \cdot e^{a \cdot x}, & \text{if } k = 1 \\ (s \cdot a \cdot (1-k) + s_{\min}^{1-k})^{\frac{1}{1-k}}, & \text{if } k \neq 1 \end{cases} \tag{3.2}$$

Proof. The CDF is the integral of f_{PL}. Since $f_{PL}(x) = 0$ for $x \notin [s_{\min}, s_{\max}]$, it is clear that the CDF has to be 0 for $x < s_{\min}$ and 1 for $x > s_{\max}$. In the relevant interval $[s_{\min}, s_{\max}]$ this integral evaluates to:

$$F_{PL}(x) = \frac{\int_{s_{\min}}^{x} x^{-k} dx}{a}$$

$F_{PL}(s_{\min}) = 0$ and $F_{PL}(s_{\max}) = 1$, validating the choice of the normalizing divisor a. Inverting this function is done by solving the following equation:

$$F_{PL}(F_{PL}^{-1}(x)) = x \iff \frac{\int_{s_{\min}}^{F_{PL}^{-1}(x)} x^{-k} dx}{a} = x$$

Due to the different integral of x^{-1} this has to be done separately for $k = 1$ and $k \neq 1$. For $k = 1$:

$$\iff \frac{\log(F_{PL}^{-1}(x)) - \log s_{\min}}{a} = x \iff F_{PL}^{-1} = e^{a \cdot x + \log s_{\min}} = s_{\min} \cdot e^{a \cdot x}$$

For $k \neq 1$:

$$\iff \frac{\frac{F_{PL}^{-1}(x)^{1-k}}{1-k} - \frac{s_{\min}^{1-k}}{1-k}}{a} = x \iff F_{PL}^{-1}(x) = (s \cdot a \cdot (1-k) + s_{\min}^{1-k})^{\frac{1}{1-k}}$$

□

Lemma 3.3. *The expectation of $PL_{v_{\min}, v_{\max}, k}$ is:*

$$\mathbb{E}(PL_{v_{\min}, v_{\max}, k}) = \begin{cases} \frac{s_{\max} - s_{\min}}{a}, & \text{if } k = 1 \\ \frac{\log s_{\max} - \log s_{\min}}{a}, & \text{if } k = 2 \\ \frac{s_{\max}^{2-k} - s_{\min}^{2-k}}{a \cdot (2-k)}, & \text{otherwise} \end{cases} \tag{3.3}$$

Proof. To calculate the expectation, there are two options. Either looking at the sampling process

$$\mathbb{E}(PL_{v_{\min}, v_{\max}, k}) = \int_0^1 F_{PL}^{-1}(x) dx$$

or looking at the probability density

$$\mathbb{E}(PL_{v_{\min}, v_{\max}, k}) = \int_{s_{\min}}^{s_{\max}} x \cdot f_{PL}(x) dx$$

The first case to be considered is $k = 1$:

$$\mathbb{E}(PL_{v_{\min}, v_{\max}, k}) = \int_0^1 s_{\min} \cdot e^{a \cdot x} dx = \frac{s_{\min}}{a} \int_0^a e^x dx = \frac{s_{\min}}{a}(e^a - 1) = \frac{s_{\max} - s_{\min}}{a}$$

This result can be confirmed with the alternative approach:

$$\mathbb{E}(PL_{v_{\min}, v_{\max}, k}) = \int_{s_{\min}}^{s_{\max}} x \cdot \frac{x^{-1}}{a} dx = \frac{s_{\max} - s_{\min}}{a}$$

For $k \neq 1$ the expectation is:

$$\mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k}) = \int_0^1 (x \cdot a \cdot (1-k) + s_{\text{min}}^{1-k})^{\frac{1}{1-k}} dx = \frac{1}{a \cdot (1-k)} \int_{s_{\text{min}}^{1-k}}^{a\cdot(1-k)+s_{\text{min}}^{1-k}} x^{\frac{1}{1-k}} dx$$

Which for $k = 2$ evaluates to $\mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k}) =$

$$\frac{1}{-a}(\log(a \cdot (1-k) + s_{\text{min}}^{-1}) - \log(s_{\text{min}}^{-1})) = \frac{-1}{a}(\log(s_{\text{max}}^{-1}) - \log(s_{\text{min}}^{-1})) = \frac{\log s_{\text{max}} - \log s_{\text{min}}}{a}$$

And for $k \notin \{1, 2\}$:

$$\mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k}) = \frac{1}{a \cdot (2-k)}((a \cdot (1-k) + s_{\text{min}}^{1-k})^{\frac{2-k}{1-k}} - (s_{\text{min}}^{1-k})^{\frac{2-k}{1-k}})$$

$$= \frac{1}{a \cdot (2-k)}((s_{\text{max}}^{1-k})^{\frac{2-k}{1-k}} - s_{\text{min}}^{2-k}) = \frac{s_{\text{max}}^{2-k} - s_{\text{min}}^{2-k}}{a \cdot (2-k)}$$

These cases can also be confirmed by the alternative approach, so for $k \neq 1$:

$$\mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k}) = \int_{s_{\text{min}}}^{s_{\text{max}}} x \cdot \frac{x^{-k}}{a} dx$$

Which for $k = 2$ evaluates to:

$$\mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k}) = \frac{\log s_{\text{max}} - \log s_{\text{min}}}{a}$$

And for $k \notin \{1, 2\}$:

$$\mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k}) = \frac{s_{\text{max}}^{2-k} - s_{\text{min}}^{2-k}}{a \cdot (2-k)}$$

□

Lemma 3.4. *The variance of $PL_{v_{\text{min}},v_{\text{max}},k}$ is:*

$$\mathbb{V}(PL_{v_{\text{min}},v_{\text{max}},k}) = \begin{cases} \frac{s_{\text{max}} - s_{\text{min}}}{a} - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2, & \text{if } k = 2 \\ \frac{\log s_{\text{max}} - \log s_{\text{min}}}{a} - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2, & \text{if } k = 3 \\ \frac{s_{\text{max}}^{3-k} - s_{\text{min}}^{3-k}}{a \cdot (3-k)} - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2, & \text{otherwise} \end{cases} \tag{3.4}$$

Proof.

$$\mathbb{V}(PL_{v_{\text{min}},v_{\text{max}},k}) = \int_{s_{\text{min}}}^{s_{\text{max}}} x^2 f_{PL}(x) dx - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2$$

$$= \int_{s_{\text{min}}}^{s_{\text{max}}} \frac{x^{2-k}}{a} dx - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2 = \begin{cases} \frac{s_{\text{max}} - s_{\text{min}}}{a} - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2, & \text{if } k = 2 \\ \frac{\log s_{\text{max}} - \log s_{\text{min}}}{a} - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2, & \text{if } k = 3 \\ \frac{s_{\text{max}}^{3-k} - s_{\text{min}}^{3-k}}{a \cdot (3-k)} - \mathbb{E}(PL_{v_{\text{min}},v_{\text{max}},k})^2, & \text{otherwise} \end{cases}$$

□

3.1.1 Problems when Sampling from Power Law

Because edge generation works based on the communities, each PE needs to know the size of all communities that affect edges generated by it. Also edge probabilities for power law based models will depend on community sizes. However, there cannot be any communication between different PEs and the generated sizes as well as the assignment of vertices to communities have to be identical for all PEs to allow a cohesive result.

Unlike with previous models, the distributed sampling technique of Sanders [SLHS$^+$18] cannot be used here. It requires having a distribution that describes how much of the mass to distribute lands on each half of the value range. It also requires a way to distribute any mass in any interval of values that may result from the successive splitting of the value space. To our knowledge this does not exist for the power law distribution. Creating such a recursive sampling method would also have to overcome the issue that it is unclear how many communities have to be sampled to cover all vertices. The fact that values sampled from the power law distribution may vary massively in size makes it hard to split the sampling of community sizes between PEs. Each PE would still have to evaluate all samples to find out by how much the distributed mass differs from the intended total mass to enable any sort of correction mechanism.
Since it is possible to distribute too much mass (sampling communities that are in total larger than the number of vertices), correction would have to include the option of dropping already sampled communities, so all community sizes would have to be stored. Not allowing dropping of single communities in the case of over-sampling but instead re-sampling everything in such a case would result in large communities being less likely than they should be in a power law distribution.

With this in mind it seems that every PE has to create all samples, exactly distributing all mass. This has to be achieved without the costs of this sampling process dominating the total work of the PE, even when a lot of PEs are used to generate massive graphs.

3.1.2 Linear Sampling

Since creating all communities seems necessary, investigating linearly sampling until all mass has been distributed is the straight forward approach. This approach draws a sample from the given power law distribution and adds it to the results, if the sample is smaller or equal in size than the remaining mass to be distributed. When a sample that is larger than the remaining mass is drawn, a sample of the size covering the remaining mass is added to the results. If this added sample is smaller than permitted, it and the previously drawn sample are removed from the results. Then a sample of the smallest permitted size and a sample covering the rest of the distributable mass are added to the results instead. This assumes that $v_{\min} \leq 2 \cdot v_{\max}$, or the last added sample may still not be in the permitted range.

An implementation of the linear sampling approach is given in Algorithm 3.1. Most of the code there describes special handling of the final communities to ensure that the intended cumulative size of all communities is reached exactly without violating the size constraints for individual communities.
This is achieved by checking whether the latest sample would induce a situation where fulfilling the constraints is impossible. If that is the case, that community is ignored and replaced by a community using up all remaining mass, or – if all remaining mass is too large for a single community – by one community of minimal size and one community taking the remaining mass. In this case the mass could theoretically also be distributed differently between two samples, but this way creates the largest possible community to replace the discarded community. Since the original community was discarded on account

of being too large for completing the sampling process, replacing it by a community of maximum size, considering the situation, seems like the approach most true to the power law distribution.
The used function SampleLP returns a single sample drawn from the specified power law distribution.

Algorithm 3.1: Linear PL-Sampling

```
Input: Parameters for PL distribution $v_{\min}, v_{\max}, k$, intended cumulative size of
       all samples total_mass
Data: List of samples S
Output: Samples S drawn from the specified PL distribution with total
        cumulative size total_mass
1  distributed_mass = 0
2  while distributed_mass < total_mass do
3      sample = SamplePL ($v_{\min}$,$v_{\max}$,$k$)
4      distributed_mass += sample
5      S.PushBack (sample)
6      if total_mass- distributed_mass < $v_{\min}$ then
7          S.PopBack ()
8          if distributed_mass ≥ total_mass then
9              distributed_mass-= sample
10         sample = total_mass- distributed_mass
11         S.PushBack (sample)
12         distributed_mass = total_mass
13         if sample > $v_{\max}$ then
14             S.PopBack ()
15             sample = distributed_mass- $v_{\min}$
16             S.PushBack (sample)
17             sample = $v_{\min}$
18             S.PushBack (sample)
```

Note that while Algorithm 3.1 saves all sampled communities in a list, a real implementation only has to keep those that are relevant for the PE executing it. Storing which cases of the special case handling at the end were used and what the last two added samples are, allows fast calculation of the size of any community without storing all results. It is possible to get the size of any sampled community in constant time by providing its index to offset the seed for sampling a single value and potentially considering the special cases. So assuming that the relevant communities' indices can be determined at runtime, e.g. they lie in an interval where the index can be incremented until the last relevant community is reached, this approach works with constant space.

However, the initial runtime of Algorithm 3.1 is linear in the number of communities created. The expected number of communities is $n/\mathbb{E}(PL_{v_{\min},v_{\max},k})$ for n being the number of vertices. Even with $v_{\max}$ scaling with n and $v_{\min}$ remaining constant, this may become too large for massive graphs, since it has to be executed by every PE, while the workload for generating edges can be evenly distributed among all PEs.

3.1.3 Square Root Based Sampling

As the linear approach is not fast enough for large settings and sampling only those communities that are relevant for each PE is impractical, it is necessary to look for a different sampling technique. Following the sampling approach based on [SLHS+18] is not possible directly, but it can still be used here. For a fixed number n_c of communities to be sampled, the number of communities whose sample $s = F_{PL}(\text{community\_size})$ is in a part $[s_{\text{low\_sep}}, s_{\text{up\_sep}}]$ of the sample space $[0, 1]$ can be determined using $\text{Bin}(n_c, p)$ where $p = s_{\text{up\_sep}} - s_{\text{low\_sep}}$. For values $v_{\text{low\_sep}}$ and $v_{\text{up\_sep}}$ bounding the size of permitted community sizes in this part the sample space is bounded by $s_{\text{low\_sep}} = F_{PL}(v_{\text{low\_sep}} - 0.5)$ and $s_{\text{up\_sep}} = F_{PL}(v_{\text{up\_sep}} + 0.5)$. With this, a number of communities that is expected to roughly reach the desired mass can be sampled by recursively splitting the space of potential samples in each step and determining how many samples fall into each of the splits.

For an arbitrary distribution this would not offer a notable advantage over the linear approach, since the recursive splitting still has to be followed through until everything has been sampled. Also, this result will have to be modified to account for the difference in distributed mass sampled compared to the desired total mass. But the nature of the power law distribution means that many small communities of the same size will be sampled. The recursion can stop when all possible samples in the sampled interval will result in the same value for the community's size. So one of these recursive sampling runs requires time and space proportional to the number of differently sized communities sampled. To allow easier handling of the over- or under-sampling, only communities smaller than $\sqrt{n}$ – where n is the intended combined size of all communities – will be sampled this way. For them, the accumulated number of communities of sizes smaller or equal to each size is stored in a vector. Larger communities are sampled linearly. But since there can be at most $\sqrt{n}$ communities of size larger than $\sqrt{n}$, asymptotically this does not increase the required runtime or space.
If this recursion has distributed too few vertices, it can be repeated for fewer communities. If the recursion has over-sampled, some sampled communities have to be discarded. Discarding works in the same way, first deciding how many communities should be discarded to probably achieve the desired total mass. But when breaking the intervals in half, the hypergeometric distribution is used instead of the binomial distribution to discard each community with the same probability. So assume there are n communities to be discarded, the left split contains $|c_{\text{left}}|$ communities and there are $|c_{\text{total}}|$ communities in both splits. Then drawing from $\text{Hyp}(n, |c_{\text{left}}|, |c_{\text{total}}|)$ determines how many communities of the lower split are discarded. Discarding a fixed number of communities also costs $\sqrt{n}$ time.
The process of repeatedly adding or removing communities ends when the deviation from the intended mass is less than $\sqrt{n}$. Removing a single community has cost $\log \sqrt{n}$ because the logarithmic descent has to be completed to find a random community, while adding a single community is possible in constant time. So when the process would stop adapting the communities block-wise at a point where the distributed mass is larger than the intended mass, instead another discard step is issued, where twice the usual number of communities is discarded. This should usually lead to a mass of less than $\sqrt{n}$ missing. If that is not the case, the process of adding or removing communities to get closer to the intended distributed mass is continued.

The final mass is added by sampling communities in an iterative manner until either the remaining mass to be distributed is too small to fit another community, or the distributed mass becomes larger than the desired total mass. These cases are then handled as described in Algorithm 3.1. If the last sampled community exactly claims the remaining space the sampling stops without having to handle special cases.

The coarse structure of this approach is given by Algorithm 3.2.
The used function ExpectedSampleSize returns $\mathbb{E}(PL_{v_{\min}, v_{\max}, k})$ as given in Equation 3.3. RecSampleSmallPL recursively draws a number of samples specified by its first parameter of size at most $\sqrt{\text{total\_mass}}$ from the power law distribution specified by the three other

parameters. LinSampleLargePL does the same, but for samples larger than $\sqrt{\text{total\_mass}}$ and the technique used to draw the samples is linear, not recursive. DrawBinomial and DrawHypergeometric draw a random value of the corresponding distribution, where the size of the universe is given by the first parameter and the probability is specified by the second parameter (for the hypergeometric distribution it is not a probability, but the number of success instances in the universe). AccSize returns the accumulated size of all samples passed and AccCount gives the number of samples passed. The functions RecRemoveSmallSpl and LinRemoveLargeSpl randomly remove a number of samples specified by the second parameter from the sample set passed as the first parameter. LinAddFinalSamples linearly samples the remaining samples to cover mass specified by the first parameter in basically the same way as Algorithm 3.1, but respecting the two different result sets for samples.

Algorithm 3.2: Sqrt PL-Sampling

```
Input: Parameters for PL distribution $v_{\min}, v_{\max}, k$, intended cumulative size of
       all samples total_mass
Data: List of samples small_samples and large_samples
Output: Samples small_samples and large_samples drawn from the specified PL
        distribution with total cumulative size total_mass
1  small_sample_perc = $F_{PL}(\sqrt{\text{total\_mass}})$
2  distributed_mass = 0
3  remaining_mass = total_mass
4  while total_mass − distributed_mass > $\sqrt{\text{total\_mass}}$ do
5      remaining_mass = total_mass − distributed_mass
6      exp_sample_count = remaining_mass/ ExpectedSampleSize ($v_{\min}$, $v_{\max}$, $k$)
7      needed_ss = DrawBinomial (exp_sample_count, small_sample_perc)
8      needed_ls = exp_sample_count − needed_ss
9      small_samples.PushBack (RecSampleSmallPL (needed_ss, $v_{\min}$, $v_{\max}$, $k$))
10     large_samples.PushBack (LinSampleLargePL (needed_ls, $v_{\min}$, $v_{\max}$, $k$))
11     distributed_mass = AccSize (small_samples) + AccSize (large_samples)
12     while total_mass < distributed_mass do
13         mass_excess = distributed_mass − total_mass
14         if mass_excess < $\sqrt{\text{total\_mass}}$ then
15             mass_excess *= 2
16         small_excess = DrawHypergeometric (AccCount (small_samples) +
               AccCount (small_samples), AccCount (small_samples))
17         large_excess = mass_excess − small_excess
18         RecRemoveSmallSpl (small_samples, small_excess)
19         LinRemoveLargeSpl (large_samples, large_excess)
20 LinAddFinalSamples (remaining_mass, small_samples, large_samples)
```

The runtime of this algorithm is in part governed by how often a batch of samples has to be added or removed. Due to the very high variance for low exponents k shown in Equation 3.4, it is hard to get a definitive bound on how likely oversampling is. For X being the random variable describing the accumulated mass of all samples drawn during on batch and μ_x and σ_X^2 being the expectation and variance of X, Chebyshev's inequality yields:

$$P(|X - \mu_X| \geq \alpha\mu_X) \leq \frac{\sigma_X^2}{\alpha^2\mu_X^2}$$

Since the samples drawn during one batch are independent and identically distributed, μ_X and σ_X^2 can be expressed in terms of the expectation and variance of a single draw. For drawing c samples in the batch this means $\mu_X = c \cdot \mathbb{E}(PL_{v_{\min},v_{\max},k})$ and $\sigma_X^2 = c \cdot V(PL_{v_{\min},v_{\max},k})$. So

$$P(|X - \mu_X| \geq \alpha\mu_X) \leq \frac{1}{c\alpha^2} \cdot \frac{\sigma_X^2}{\mu_X^2}$$

However, for large values of $s_{\max}$ and reasonably small exponents k, this does not guarantee small chances for over-sampling by notable margins. For exponents $k > 3$, where the unbounded power law distribution starts to have finite variance [New05] and $s_{\max}$ is notably larger than $s_{\min}$ the right side of this inequality is dominated by

$$\frac{1}{c\alpha^2} \cdot \left(\frac{-a(2-k)^2 s_{\min}^{k-1}}{(3-k)} - 1 \right)$$

This dominating factor becomes small even for later batches, where only as few as $c = \frac{\sqrt{n}}{E(PL_{v_{\min},v_{\max},k})}$ samples are drawn.
For lower exponents k, the central limit theorem still provides some use. It states that when drawing many samples, the total distributed mass across all drawn samples tends towards a normal distribution, which would also be sufficient for fast termination of the recursion, since large deviations from the expected mean are sufficiently unlikely. However, the central limit theorem does require a finite variance, which is technically given due to capping potential samples at $s_{\max}$, but if $s_{\max}$ scales with the size of the graph, this also increases the variance. Also, the central limit theorem does not provide how many samples have to be drawn to be sufficiently close to a normal distribution. Especially for settings where few small and many large samples are needed and only comparably few samples are drawn, this may be problematic.
Practical testing in Subsection 6.2.4 shows that the implementation of this algorithm takes very few sampling batches to be close enough to the intended mass to allow linear sampling to get the remaining samples.

3.2 PL-Communities Model

The community sizes for the PL-communities model are power law distributed. Each vertex is a member of exactly one community, so communities do not overlap.

3.2.1 Description and Parameterization

The PL-communities model has the communities' sizes drawn from a power law distribution. There is no overlap between different communities and each vertex is a member of exactly one community. Unlike with the previously discussed models, the number of communities is not a parameter. Instead, the parameters of the power law distribution are specified. Drawing samples from that power law distribution until the combined size of all communities equals the number of vertices determines both the community sizes and the number of communities. So for this model the number of communities is a random variable.
Edges are still generated based on whether they are within a community or connect vertices of different communities. For inter-community edges the probability of an edge existing is specified by the parameter p_o. Since community sizes can vary drastically using a single probability for edges within a community is not constructive. Therefore, the probability of an edge existing within a community depends on the size of the community in the same way as it does in the CKB model [CKB+14]. So the chance for an edge (v_i, v_j) in community c_l with size $|c_l|$ is given by $p_i = \frac{\alpha}{|c_l|^\gamma}$, where $\alpha > 0$ and $\gamma \in (0, 1)$ are parameters. This ensures that the probability of each individual edge is smaller in a larger community, but members of larger communities have a larger expected degree.
The parameters are listed in Table 3.1. A graph generated with the PL-communities model can be seen in Figure 3.1. The distribution of community sizes produces mostly small communities with few very large communities.

Parameter	**Description**	**Usual Value**
n	number of vertices	-
$c_{\min}$	smallest permitted community size	6
$c_{\max}$	largest permitted community size	$n/10$
k_{comm}	exponent of the PL-distribution for community sizes	2.5
α	numerator for calculation of p_i	4
γ	exponent for community size for p_i calculation	0.5
p_o	probability for each edge between communities to exist	$2/n$
self_loops	should loops be possible edges	false

Table 3.1: List of parameters for the PL-communities model.

3.2.2 Algorithm

Since this model has only one community per vertex, most of the methods used in the previous models are applicable here as well. Initially the sizes of the communities have to be sampled with the sqrt-based approach described in Subsection 3.1.3. These communities are again thought of as containing subsequent vertices, so the edges within communities lie in blocks around the main diagonal of the adjacency matrix. As with the unique communities model in Section 2.2 this block structure is scrambled by reordering the vertices before pushing generated edges to the graph.

Each PE is assigned an equal number of consecutive vertices to generated edges for. A logarithmic descent into the data structure storing the sampled community sizes can determine how many communities of which size are assigned to the current chunk and how many were already processed by previous chunks. Chunks are consecutive rows in the directed case and rectangles and triangles as described in Section 2.1 for generating undirected graphs. So this model has every PE generate all edges incident to each of its handled vertices.

For edge generation a chunk splits up its part of the adjacency matrix into homogeneous blocks using the same technique as the previous models. Generating edges in blocks containing only inter-community edges works by determining how many of the e_{pos} possible edges in this block are generated. The probability of an inter-community edge is p_o, so $\text{Bin}(e_{\text{pos}}, p_o)$ edges are generated. These edges are then randomly distributed among the potential edges of the chunk. For intra-community edges the process is the same, but the probability $p_i = \frac{\alpha}{|c_l|^\gamma}$ for the binomial draw first has to be calculated considering the size of the community that the current block is a part of.
Note that communities form blocks on the main diagonal of the adjacency matrix as discussed in Subsection 2.1.3. Blocks are rectangular sections of the adjacency matrix containing only intra- or only inter-community edges, so no block can contain intra-community edges from different communities. Thus, only one probability p_i has to be used for each block of intra-community edges.

3.2.3 Implementation Details

The sampling of community sizes has to be done for each PE. Even though only a limited range of communities contains the vertices whose edges are generated by any PE, the nature of the sampling algorithm in Section 3.1.3 does not allow selectively calculating the size of these communities. It is necessary to complete the whole process of adding and removing community sizes until the desired combined size is reached. Any early cropping could not detect whether a sampled community would be discarded in a later step. Sampling the community sizes of each chunk individually without considering other chunks with this algorithm would not allow communities split into multiple chunks, which would distort the

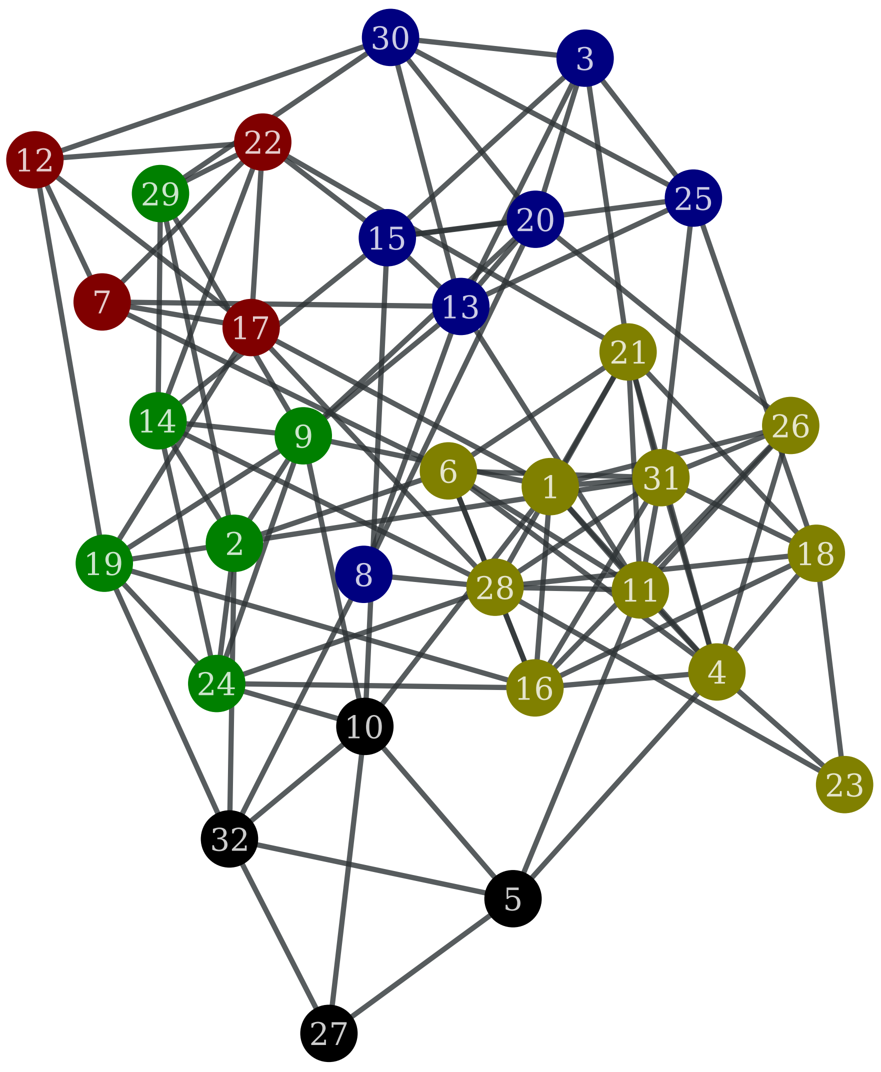


Figure 3.1: An undirected graph generated with the PL-communities model ($n = 32$, $c_{\min} = 4$, $c_{\max} = 20$, $\alpha = 2$, $\gamma = 0.5$, $p_o = 0.08$). Colours indicate community membership.

distribution. It would also prevent very large communities that due to their size have to be handled by multiple PEs.
However, the sqrt based approach is reasonably fast and its result can be stored in accumulated form. This means that instead of storing how many communities of a certain small size exist, or how large a given large community is, it is stored how many vertices are in communities of the given size or smaller for small communities, respectively how many vertices are in communities of the same or smaller id for large communities. For getting the communities relevant to the current chunk, a logarithmic search over this data structure allows returning the full size of all partially contained communities of the chunk as well as id and contained size of the first community of the chunk.

Note that this search potentially has to consider both the small communities, which are stored by how often their size was sampled and large communities for which the actual size is stored. This differentiation between small and large communities can be masked by the function calculating which communities are part of the current chunk. Thus, the

rest of the algorithm does not need to deal with the split data structure. With this information, borders between communities can be determined and the chunks can be cut into homogeneous blocks the same way as in Section 2.1.
Reordering of vertices to break up the block structure in the adjacency matrix is done with a hash function as described in Section 2.2.

4. Overlapping Communities

In this chapter overlapping communities are introduced. Vertices can be a member of multiple communities, causing those communities to overlap in the shared vertices. Techniques for assigning vertices to communities in a communication-free manner are discussed. For the first model utilizing overlapping communities all vertices have the same intended membership count.

4.1 Membership Assignment for Overlapping Communities

In many settings vertices can be members of multiple communities [WCL$^+$16][RG12]. So providing models that allow generating graphs with overlapping communities like the LFR-model [LF09] and the CKB-model [CKB$^+$14] is useful to explore these kinds of environments and allow comparing community detection algorithms with a base truth. Initially all vertices should be in the same number of communities, while for the CKB model vertices' membership counts should also be power law distributed.

4.1.1 Problems when Working with Overlapping Communities

Creating a random bigraph from vertices to communities with specified degrees in a communication-free setup is hard. Models like the one developed by Chojnacki et al. [CK10] work by sequentially expanding the graph and are inherently not suited for use in distributed calculations.
Other ways of generating random bigraphs like the curveball algorithm [Car15] work by starting with a nonrandom assignment and then repeatedly swapping random edges. In a communication free setting the result of these swaps cannot be broadcasted to other PEs, so this kind of technique can not utilize parallelization here.

The LFR model generates its vertex community pairings by connecting communities and vertices randomly while respecting their degree with the configuration model [LF09]. This process is also sequential, but could be modified to run in parallel by assigning each PE random subsets of vertices and communities to connect. But the configuration model can lead to a vertex being assigned to the same community multiple times. While these double assignments could be detected, fixing them by swapping a duplicate edge with another edge would have to be done by every PE to ensure consistency. This is not possible without communication unless every PE generates and checks all assignments, in which case there would no longer be any benefit to the parallelization.

The community assignment of the CKB model is done in a probabilistic manner [CKB+14]. They have each PE draw a set number of random vertex community pairs and draw more pairs than needed for the model to be able to remove duplicate draws without reducing the expected number of members and communities in the generated graph. For the elimination of duplicates a merge of the results of all PEs is required, which is not possible in a communication free environment.

So for the communication free approach, an algorithm has to be employed that can not produce the same pairing multiple times. Correcting these collisions where a vertex is assigned membership to one community multiple times consistently seems impossible while utilizing parallelization. Also, the algorithm has to be able to generate consistent assignments across all PEs, while each PE should only have to calculate assignments relevant for the edges it has to generate. A PE processes edges for members of consecutive communities. Running assignments on each PE independently of other PEs would alter the distribution of community sizes by preventing the option of splitting communities and making communities with more members than are processed by each PE impossible. Furthermore, an approach focusing on isolating PEs this way would require complicated splitting of the available vertices to ensure that no PE gets assigned vertices in a way that can not be distributed among its communities without double assignments.
This means that a PE has to assign members to a certain set of communities, keeping the content of split communities consistent across the different PEs processing it. Also, the vertices should be assigned to the correct number of communities each and as mentioned above, any double assignments can not be corrected without communication.

4.1.2 Group Based Assignment

The assignment of vertices to communities in the PL-communities model of Section 3.2 without overlapping communities works based on a random permutation of the vertices. The position of an element in a community – its position in that community plus the combined size of previous communities – is hashed with a randomly chosen bijective function to get the vertex's id. When increasing the size of the domain of such a hash function to include all positions in communities, multiple positions correspond to the same vertex. So the hash function can no longer be bijective and may map two positions in one community to the same vertex. As the communication free setup has no easy way of fixing these collisions, this is not a viable option.

When all vertices should have the same membership count, a way to keep being able to use bijective hash functions is by grouping communities into groups that in total have fewer members than there are vertices. Hashing from each of the group spaces into the vertex space using a function $h(x)$ for all groups does guarantee that all members of each community are unique. Since vertex sizes are used to create groupings each community also has the correct number of members. The interaction between the hash function and different groups has to ensure that each vertex is in the same number of communities.
To keep track of what vertices were assigned how often, each group uses the same hash function to assign vertices. So the first group g_1 containing communities of combined size $|g_1|$ hashes values from 0 to $|g_1| - 1$ to get the corresponding vertices. This leaves $n - |g_1|$ values, that were not hashed and whose corresponding vertices were not assigned to any community. They can be assigned to communities of the second group g_2. To get the members of the communities in g_2 the remaining values from $|g_1|$ to $n-1$ and values from 0 to $|g_1|+|g_2|-n-1$ are hashed. In this manner each group takes values left by the previous group in the upper range of the hash functions domain and – if necessary – the beginning of a next range of numbers and hashes them to receive the members of its communities.
The vertices obtained in this way can not be assigned to communities in this order, because this would result in values whose preimages are successive to share communities way more

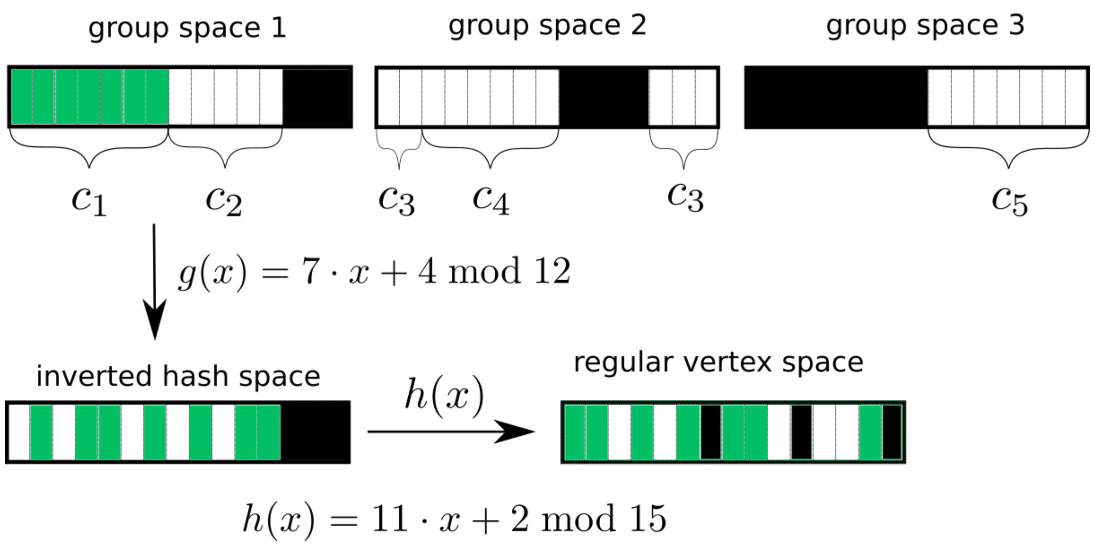


Figure 4.1: Example of the assignments of members (in green) to community c_1 with group based assignment.

often than they should. So the order within each group is scrambled by applying another bijective hash function $g(x)$. This function $g(x)$ reorders only the values within that group, so it is unique for every group.
Figure 4.1 visualizes this process with five communities and a membership count of 2 for all vertices. The members of community c_1 are highlighted in green and their position after each hashing is visualized. Black spaces are positions that represent vertices outside the current group and white spaces belong to other communities. Community c_3 in the second group fills both the remaining three slots not used by group one and the first two slots of the next iteration of vertices.

This approach is reasonably fast and can easily run on multiple PEs. Getting the members of a given community requires only knowing its group and the position of that group in the domain of $h(x)$. These pieces of information can be obtained by iterating over the sampled community sizes once in $\mathcal{O}(\sqrt{n})$ because of the structure in which power law sampled communities are stored. For community sizes given by different means, iterating over all communities would take linear time. So each PE only has to assign members to the communities for which it generates edges, which is efficient.

Unfortunately the assignments created by this approach do not work independently for different communities. Communities that are in the same group can not overlap at all. For small communities where chances of shared overlap are not very high this may be an acceptable flaw. But two large communities that contain ten percent of all vertices would be expected to share one percent of all vertices or ten percent of their members. Preventing these overlaps and in return forcing more shared vertices with communities of other groups, which is especially relevant when few groups exist, is a major distortion of the intended distribution. Even though this approach may be viable in some settings where guaranteeing exact community sizes and membership counts is important and implicitly adding a second layer of structure to the graph is not considered too harmful, it will not be considered further in this thesis.

4.1.3 Probabilistic Assignment

Because of the shortcomings of the group based approach and the general difficulty of calculating a consistent assignment of vertices to communities in a distributed communication free setting, a probabilistic approach will be used. As the name implies, this approach does not provide exact assignments from a given vertex set to given communities. Instead, the idea is to have each community pick its members randomly, probabilities of the pick being proportional to the membership count of the vertex.

This keeps the distribution of community sizes as specified, so for power law distributed community sizes the sizes of the communities in the generated graph are the exact sizes returned by the internal power law sampling algorithm used to draw community sizes. However, the membership counts of the vertices are not achieved precisely. The expectation of each vertex's membership count is the intended membership count, but there can be deviation from that expectation. So if each vertex should be in the same number of communities, this approach does produce vertices of both larger and smaller membership count around that specified membership count. The exact distribution of membership counts is hard to specify, because it is a result of independent draws with different probabilities depending on the size distribution of the communities and respecting that communities sample vertices without duplicates.
Because of the independence of the member drawing routines of different communities, overlap between communities is completely random, as it should be.

4.2 PL-Multicommunity Model

Vertices in the PL-multicommunity model have a fixed intended membership count that is the same for ever vertex. This causes overlap between the communities whose sizes are power law distributed.

4.2.1 Description and Parameterization

For this model the communities' sizes are drawn from a power law distribution. Each vertex is on average in a set number of communities. Since probabilistic assignment from Subsection 4.1.3 is used the membership count is not the same for every vertex, but they all have the same expected membership count. Having vertices being in multiple communities is equivalent to having overlapping communities.

Since this model is similar to the CKB model in that it has overlapping communities whose sizes are power law distributed, inter-community edge generation is done with an ϵ-community [CKB$^+$14]. So for this model p_o does specify the probability of an edge occurring between any two vertices, regardless of whether they share a community. This does not change the inter-community edge probability compared to previously discussed models, but it increases the probability of an edge between two vertices sharing at least one community by p_o. Additional to edges from this ϵ-community, intra-community edges are generated similar to the not overlapping PL-communities model in Section 3.2. This means that each community c_i calculates its own probability $p_i = \frac{\alpha}{|c_i|^\gamma}$ for intra-community edges. Different communities generate their intra-community edges independent of each other. Because communities can overlap in this model and all communities overlap with the ϵ-community, it is possible, that an edge is generated multiple times by different communities. In this case the duplicate edges are removed in post-processing if the graph is returned in a single file. If each PE returns its own result file and the graph is used in its distributed form, any application working with it has to deal with potential duplicate edges.
All parameters of this model are listed in Table 4.1. A graph generated by the PL-multicommunity model is shown in Figure 4.2. Because of overlapping communities, large parts of the graph are a connected a lot stronger than in the previous models. Communities are connected to other communities with inter-community edges.

Parameter	Description	Usual Value
n	number of vertices	-
$c_{\min}$	smallest permitted community size	6
$c_{\max}$	largest permitted community size	$n/10$
vertex_mc	expected membership count for each vertex	4
k_{comm}	exponent of the PL-distribution for community sizes	2.5
α	numerator for calculation of p_i	4
γ	exponent for community size for p_i calculation	0.5
p_o	probability for each edge in the ϵ-community to exist	$2/n$
self_loops	should loops be possible edges	false

Table 4.1: List of parameters for the PL-multicommunity model.

4.2.2 Algorithm

As this model features overlapping communities, splitting a permuted adjacency matrix between the different PEs is no longer efficient. There is no permutation for which all communities form blocks, so the blocklike structure that allows efficient processing can not be achieved this way. What is still possible is to divide continuous parts of communities among the PEs. This means that one PE no longer necessarily generates all edges for all the vertices for which it generates some edges.

Each PE is assigned an equal number of instances of vertex community pairs from continuous communities. So the PE handles all intra-community edges for a community except when it is the first or last community of this PE in which case that community may also be partially handled by the previous or next PE. To get the communities' sizes before distributing the instances between PEs, the community sizes are sampled from a power law distribution with sqrt-based sampling as described in Subsection 3.1.3.
The communities get assigned members using probabilistic assignment from Subsection 4.1.3. For each of the communities partially handled, the PE randomly uniformly distributed draws members of the community, seeded with the community's id to ensure that communities split between different PEs have the same members on each PE.

Intra community edges for those communities are generated similarly to the not overlapping power law community model from Section 3.2. The probability $p_i = \frac{\alpha}{|c_i|^\gamma}$ for any edge within c_i to occur is calculated considering its size $|c_i|$. For the directed case, each community part can be processed as a rectangle chunk. For the generating of undirected graphs it is sometimes necessary to split the community part. If the community was already started by a previous chunk, the part of the community of which both source and target vertex are handled by the current PE is split off as a triangle block. The remaining possible edges are processed as a rectangle block. For a permutation that keeps the current community as a block in the adjacency matrix, this process is similar to that of the previously discussed models with the difference that no outer chunks are considered here.
Also since PEs work based on assigned communities, it is no longer guaranteed that a PE can generate all edges for a handled vertex. So in the undirected case this model does not generate something corresponding to the vertical column of rectangle chunks to generate all intra-community edges of a handled instance.
The ϵ-community is divided between all PEs. Each PE gets the same number of rows of the ϵ-community and processes them the same way it would process parts of a regular community assigned to it but the edge generation uses the probability p_o rather than a probability calculated based on the community's size.

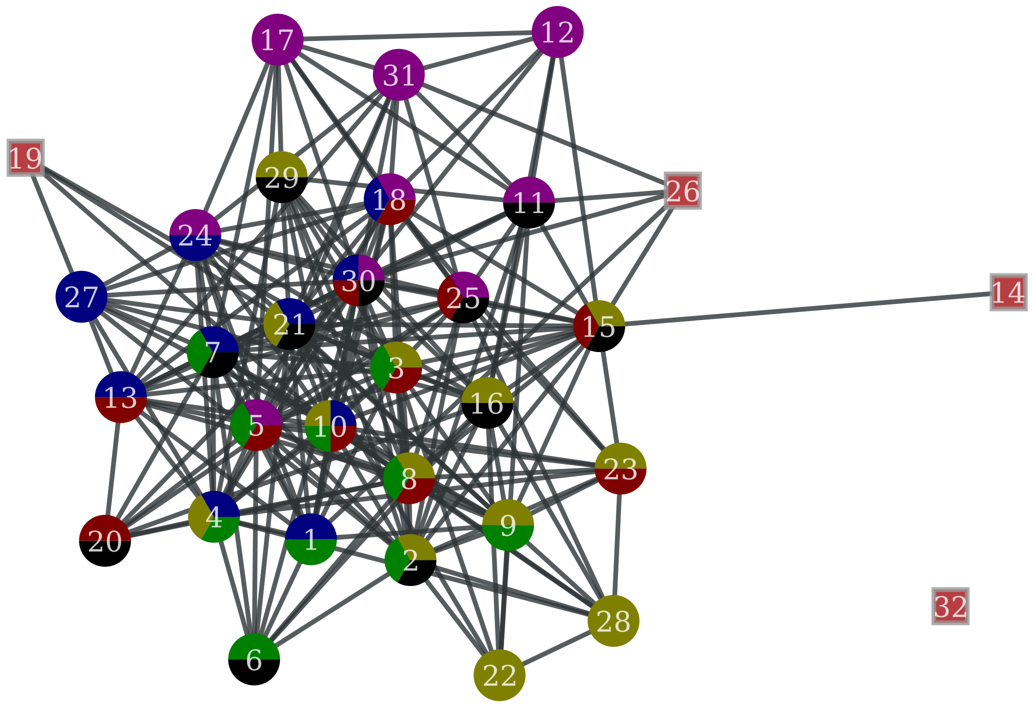


Figure 4.2: An undirected graph generated with the PL-multicommunity model ($n = 32$, $c_{\text{min}} = 4$, $c_{\text{max}} = 20$, vertex_mc = 2, $k_{\text{comm}} = 2.5$, $\alpha = 2$, $\gamma = 0.5$, $p_o = 0.08$). Colours indicate community membership, red squares are vertices with no community.

4.2.3 Implementation Details

For the reasons explained in Subsection 3.2.3, the sampling of community sizes has to be done by every PE. The communities relevant to each PE can be determined by a logarithmic search over the sampled community sizes. This will also return the id and the number of source vertices of that community handled by the current PE. The number of source vertices handled also provides information, whether the community has to be split into a rectangle and a triangle chunk in the undirected case.

The splitting of communities into multiple PEs works by letting each PE generate edges based on the source vertices assigned to it as instances. In the undirected case this does not mean that all intra-community edges of those vertices are generated by the current PE. The triangle chunk structure means, that only edges to vertices of equal or lower id – in the hypothetical permutation of vertices in which the community forms a block in the adjacency matrix – are generated by the PE, but those edges are added to the graph in both directions.

Due to the way in which vertices are assigned to communities, it is not necessary to reorder the vertices to break up any block structure. The block structure in this model is only ever local in a hypothetical permutation of the vertices, which allows working within communities in the same chunk structures that were utilised in previously discussed models. This allows profiting from large homogeneous blocks for which only a single binomial sampling has to be calculated to get the edge count, but has no practical impact on the graph structure. Instead of thinking about reordering the vertices to achieve a block for the edges in the community, it is also possible to imagine deleting all rows and columns of the adjacency matrix not belonging to members of the currently processed community. This produces the same block of possible edges in an adjacency matrix upon which the splitting between PEs and the edge generation are executed.

5. The CKB Model

This chapter describes a communication-free generator for CKB graphs. For this, vertex assignment to communities is generalized to be able to process vertices with power law distributed membership counts.

5.1 Membership Assignment for the CKB Model

The CKB model features overlapping communities like the PL-multicommunity model. The general challenges of assigning members to overlapping communities have already been described in Section 4.1. But unlike the PL multi-communities model of Section 4.2, the CKB model has the membership counts of vertices also sampled from a power law distribution.

From a theoretical standpoint this does not affect the probabilistic assignment from Subsection 4.1.3. It is still possible to have each community randomly pick its members with probabilities proportional to the membership counts of the vertices. The implementation of such a picking process becomes slightly more complex than the uniformly distributed sampling utilised for the PL multi-communities model.

5.1.1 Generalizing Group Based Assignment

The group based assignment has to be expanded to work with vertices whose membership counts are power law distributed. The structure in which both power law sampled membership counts and community sizes are stored is a result of the sampling algorithm in Subsection 3.1.3. For samples smaller than a threshold of $\sqrt{n}$ the number of how many samples of that size were drawn is stored. For larger samples the sizes are stored individually.

So for assigned members with a small membership count below the threshold of the data structure the group based assignment routine of Subsection 4.1.2 can be used to assign exact members to the community. But to decide how many members of each membership count are assigned to the community another process has to be used.

The number of available vertices has to be recursively split up into splits of communities. This starts with all vertices being available as often as their membership count indicates for all communities together. Then the communities are divided into different splits and each split is randomly hypergeometrically distributed assigned a subset of the available vertices. This is done recursively for the membership counts as well. Assume that there are n positions to be filled in the left split of communities. Also assume there are $|m_{\text{low}}|$

vertices in the lower split of membership counts still available at this recursion step and $|m_{\text{total}}|$ vertices left for assignment in the current recursion step. Then drawing from $\text{Hyp}(n, |m_{\text{low}}|, |m_{\text{total}}|)$ describes how many vertices of the lower membership-count-split are assigned to the left community-split. For each recursion step of the community splitting, all not cropped steps of the vertices' splits have to be done. Continuing down this process of recursive splits, the membership counts of members assigned to a community will be returned.

Note that this recursive process of hypergeometrically picking members for splits of communities can produce situations where a community will be forced to contain a vertex multiple times, which is not intended. These collisions can not be corrected after they occurred by swapping the over-sampled member with another community due to the communication free setting. So they must be prevented at the moment where a split would cause such an unsatisfiable situation.
To detect an unsatisfiable situation the Gale-Ryser Theorem [Kra96] can be used. For vertices stored by how often their membership count was sampled, it can be assumed that each individual vertex of that membership count was sampled as few times as possible or as often as possible. Later steps of the recursion can produce both of these extremes and if other assignments would force an unsatisfiable situation, vertices of the same membership count will be divided up among the available communities in any way that prevents the unsatisfiable situation. When checking for unsatisfiable situations with the Gale-Ryser Theorem, picks of vertices stored by membership count will always be considered to be distributed as evenly as possible between as many vertices as possible. This is the easiest way to assign them to communities. So if an assignment will be possible, it will be possible with this pick of vertices of the same membership count. Due to the compressed data structure of the power law sampled membership counts and community sizes this check can be done in $\mathcal{O}(\sqrt{n})$ time per split.
If an unsatisfiable situation would be caused by the split, vertices causing it can be swapped with vertices assigned to the other split until both splits can be realized as bipartite graphs. No communication is required for this correction because every PE will follow the same recursion tree to assign its communities' members and is therefore able to make this correction at the split where it is necessary.

To get the actual members of that community for members, which are stored by how often their membership count was sampled, the group based assignment has to be used. The elements grouped together are now not the complete communities, but only the members of the currently processed membership count assigned to communities. Offsets within the group space as well as the id of the group space the current community is grouped into can be calculated and carried down through the recursion. Members stored by their large membership count are already uniquely identified.
This generalization works for any algorithm that accurately assigns vertices of the same membership count to differently sized communities. So if an algorithm solving this issue in this communication free setting without the drawback of majorly distorted overlap structure between the communities exists, it could be used for generating CKB graphs.

5.1.2 Pre-Assignment

Because of the restrictions of the group based assignment, generating CKB graphs will use the probabilistic assignment from Subsection 4.1.3. It does not need to be adapted notably. Community sizes are kept as sampled and each community draws random vertices as members with probabilities proportional to the membership counts of each vertex.
This can lead to vertices without a community. While this is also possible for the PL-multicommunity model with fixed expected membership count from Section 4.2, it can

be more impactful for the CKB model. Here membership counts are also power law distributed and a usual value for the smallest permitted membership count is 1. So intended membership counts of 1 can be very frequent and missing that intended frequency of being sampled by 1 already results in a vertex without a community.
To prevent the unintended existence of vertices that are member of no community, pre-assignment can be used. This means assigning one community to each vertex before the probabilistic assignment. During the probabilistic assignment communities then only the remaining number of members to reach their intended size is drawn. Probabilities for each vertex being picked are then proportional to their intended membership count reduced by one, so vertices with intended membership count of 1 are no valid targets for the probabilistic assignment step when pre-assignment is used.
The probabilistic assignment step and the pre-assignment need to work together in a way that guarantees that a pre-assigned vertex is not assigned to the community again during the probabilistic assignment.

Initially it has to be decided how often each community is selected by vertices for pre-assignment. This is done using recursive hypergeometric picks. Initially all vertices have to be pre-assigned and all communities are available. Then the first community draws how many vertices are pre-assigned to it. Let there be n vertices left to be pre-assigned, the remaining communities – including the community c_i handled in this step – have a combined size of $|\hat{c_i}|$ and the currently handled community has size $|c_i|$. Then the number of vertices pre-assigned to c_i is determined by drawing from $\text{Hyp}(n, |c_i|, |\hat{c_i}|)$. This is repeated for different communities, until all communities have been given a number of pre-assigned vertices.
For the implementation in combination with sqrt based power law sampling described in Subsection 3.1.3, communities stored by how often they were sampled, only draw once how many vertices are pre-assigned to all communities of that size. Specific assignments are only decided upon when the members of the communities of that size are needed for edge generation. Since the membership counts of the vertices are also power law distributed, they are also stored in this compressed format. For easier access to the relevant data, there are four arrays of pre-assignment counts calculated and stored. The first two arrays store how often communities of each small community size get vertices pre-assigned and the last pair of arrays stores how often communities of large size get vertices pre-assigned. Each pair of arrays consists of one array indicating how many vertices of small membership count were pre-assigned to the corresponding communities and another array storing that information for vertices of large membership count. To get how many vertices with small membership count are pre-assigned to all the small communities combined, a draw from $\text{Hyp}(n_{\text{small}}, N, |\hat{c_{\text{small}}}|)$ is made, where n_{small} is the total number of vertices with small membership count, N is the combined size of all communities and $|\hat{c_{\text{small}}}|$ is the combined size of all small communities. The remaining vertices with small membership count are pre-assigned to large communities. For vertices with large membership count the process is analogue. The successive hypergeometric picking of how many vertices were pre-assigned to a specific community size or community is then done for these two arrays separately.
During edge generation it is necessary to get the actual vertices pre-assigned to a given community. If the community in question is small, the previous steps only fixed how many vertices were pre-assigned to all communities of this size together. So for a small community c_i it is first calculated how many vertices of each membership count are pre-assigned to all communities of size $|c_i|$. Then the pre-assigned vertices are divided up among all the communities of size $|c_i|$ to get the members. For large communities only the first step of this is necessary. Calculating how many vertices of each size are pre-assigned to communities of the correct id are already the members pre-assigned to the large community.

To find out how many vertices of each membership count are pre-assigned to communities of a given size, the space of available communities is recursively split until only one community size remains. For each split there is a total of available space s_{all} in all available communities and the available space s_{small} in the split of communities of smaller size. For each membership count it is determined how many vertices end up in which split. So for membership count j with m_j vertices available for this split, drawing from $\text{Hyp}(m_j, s_{\text{all}}, s_{\text{small}})$ determines how many vertices are in the split of communities of smaller size. There does not need to be a check whether vertices may have been pre-assigned twice to the same community causing a collision, because each vertex is only pre-assigned once. With each distribution step of vertices of a given membership count, the available space has to be adapted to respect the just pre-assigned vertices.
For small communities it is necessary to determine, which vertices were pre-assigned to the specific community, not just to all communities of its size. The process of this is similar to the distribution of pre-assigned vertices among communities of different sizes. Recursively the amount of communities looked at is split in half. In each split the vertices pre-assigned to each half of the split are calculated using a hypergeometric pick. Here s_{all} is the space available in all communities currently considered and s_{small} is the space in all communities of the split with smaller ids. Again for membership count j with m_j vertices available for this split, drawing from $\text{Hyp}(m_j, s_{\text{all}}, s_{\text{small}})$ returns how many vertices are in the split of communities with smaller ids. The available space also has to be updated after each distribution step for vertices of a given membership count.

These steps have returned how many vertices of what membership count are pre-assigned to any community. For vertices with small membership count l it is still necessary to determine the ids of vertices that were pre-assigned. To keep track of which vertices were already pre-assigned, vertices of the same membership count are pre-assigned in order and then their order is scrambled with a hash function $g_l(x)$ unique to the membership count l. This hashing removes the correlation between small communities being pre-assigned the lower id vertices of the same membership count. The relative position of the pre-assigned vertices compared to other vertices of the same membership count can be tracked during the recursion determining how many vertices of each membership count were pre-assigned to each community.
It needs to be ensured that the probabilistic assignment that fills up the remaining space of each community does not assign an already pre-assigned vertex. For this, the probabilistic assignment works with the information about which vertices were pre-assigned to the currently processed community c_i. Before the hashing, the pre-assigned vertices of the membership count l are consecutive. Let there be m_l vertices of membership count l and let $m^{\text{pre}}_{c_i,l}$ vertices of membership count l be pre-assigned to the currently handled community c_i. Then the probabilistic assignment has $m_l - m^{\text{pre}}_{c_i,l}$ possible vertices of membership count l to assign. Assume there are $m^{\text{prob}}_{c_i,l}$ vertices of membership count l probabilistically assigned to c_i. Those members are picked by hashing the values 0 to $m^{\text{prob}}_{c_i,l} - 1$ with a hash function $f_{c_i,l}(x)$ specific to this community and membership count. If such a picked number x is in or after the interval containing the already pre-assigned vertices, it is increased to $x + m^{\text{pre}}_{c_i,l}$. Using $f_{c_i,l}(x)$ for these specific assignments rather than a hypergeometric distribution allows keeping the members stored in a compressed format, where the specific member can be determined with $f_{c_i,l}(x)$ when needed, but keeping taps on all specific members of c_i is never necessary. This assignment is occurring before the hashing with $g_l(x)$. So both pre-assigned vertices and members obtained by the probabilistic assignment are hashed with $g_l(x)$ to get sufficiently random members assigned to the community.
This process is visualized with an example in Figure 5.1. There are 15 vertices of membership count l, two were already pre-assigned to c_1. So the three vertices of membership count l pre-assigned to community c_2 occupy cells 2 to 4 before any hashes were applied. The

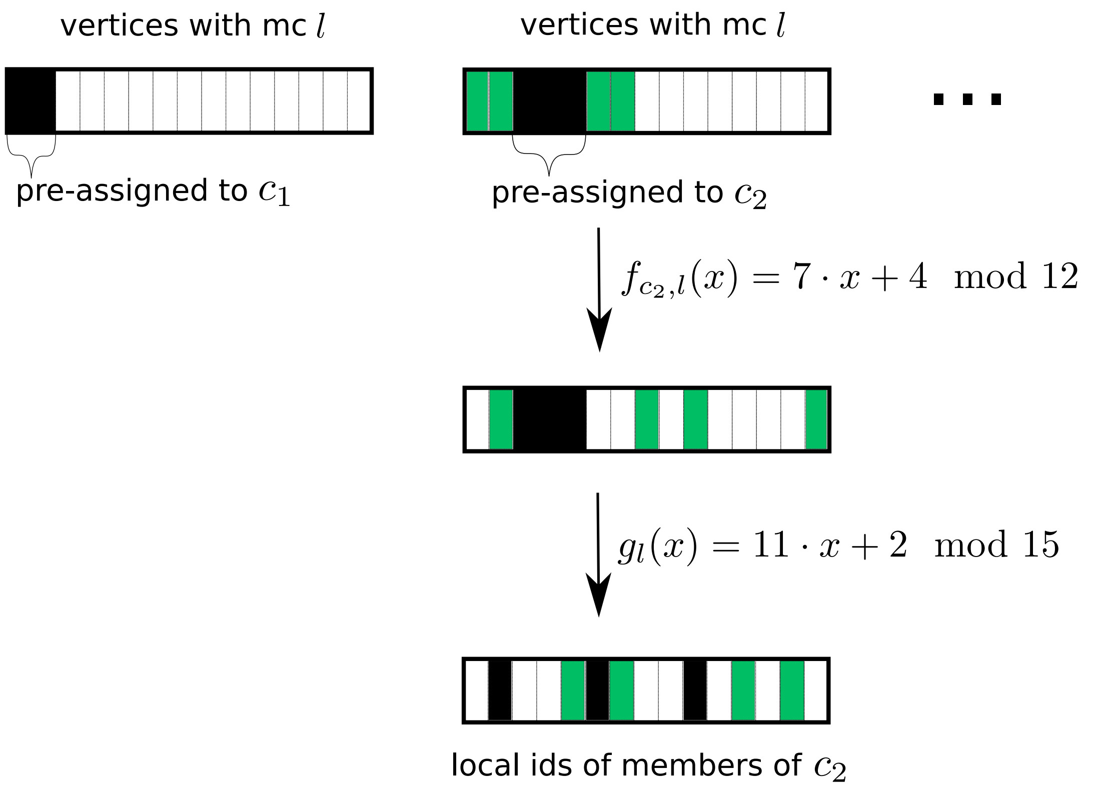

vertices with mc l
vertices with mc l
pre-assigned to c_1
pre-assigned to c_2
$f_{c_2,l}(x) = 7 \cdot x + 4 \mod 12$
$g_l(x) = 11 \cdot x + 2 \mod 15$
local ids of members of c_2

Figure 5.1: Visualization of the different hash functions to determine exact members after membership counts of members are known during pre-assignment. Cells marked in black were pre-assigned. Cells marked in green were probabilistically assigned.

probabilistic assignment now assigns four vertices of membership count l to c_2. The two larger elements are increased by 3 because they lie in or after the space reserved for pre-assigned members. For the space excluding the pre-assigned vertices the vertices positions are scrambled with $f_{c_i,l}(x)$ to have the probabilistically assigned members be random within vertices of the same membership count. To break up correlation between local member ids of pre-assigned vertices and the community's id, all ids are hashed with $g_l(x)$.
Vertices of large membership count do not need to determine specific vertices, because the pre-assignment recursion already returns their id. So for those vertices there is no hash function like $g_l(x)$. If the vertex was pre-assigned, it can not be assigned again during probabilistic assignment.

This way of pre-assigning each vertex exactly once to a random community does not reduce the distortion to the membership counts caused by the probabilistic assignment. In a way it makes it less natural, because vertices of sampled membership count 1 always have membership count 1 in the graph, but there are vertices of higher sampled membership count that do not get assigned during probabilistic assignment and therefore also have membership count 1 in the graph.
To counteract this increased number of vertices with membership count 1, the initial sampling of membership counts from the power law distribution should return fewer vertices of membership count 1. For this it is calculated how many vertices of membership count one are expected to be created beyond the intended number.

Lemma 5.1. *When approximating membership count probabilities by assuming that communities pick vertices with permitted duplicates, the expected number of unwanted vertices of membership count* 1 *is*

$$\mathbb{E}(U_1) = \sum_{\tilde{m}_i, i>1} \tilde{m}_i \cdot \left(\frac{N-n-i+1}{N-n}\right)^{N-n}$$

Here $\tilde{m}_i$ is the expected number of vertices with a membership count of i if no correction to the power law sampling is done.

$$\tilde{m}_i = \begin{cases} n \cdot \frac{\log(i-0.5)-\log(i+0.5)}{\log(s_{\max})-\log(s_{\min})}, & \text{if } k=1 \\ n \cdot \frac{(i-0.5)^{1-k}-(i+0.5)^{1-k}}{(s_{\max})^{1-k}-(s_{\min})^{1-k}}, & \text{if } k \neq 1 \end{cases}$$

The correction reducing the amount of sampled vertices of membership count 1 *has to reduce that count by*

$$\bar{\mathbb{E}}(U_1) = \frac{\mathbb{E}(U_1)\cdot n}{n-\mathbb{E}(U_1)}$$

Proof. An unwanted vertex of membership count 1 is equivalent to a vertex with higher sampled membership count not getting assigned to any communities during probabilistic assignment. During probabilistic assignment the chances of a vertex v_i with membership count $|v_i|$ to get assigned to x communities is

$$\tilde{f}_{v_i}(x) = \binom{N-n}{x} \cdot \left(\frac{|v_i|-1}{N-n}\right)^x \cdot \left(\frac{N-n-|v_i|+1}{N-n}\right)^{N-n-x}$$

So the chance to get an unwanted membership count of 1 from v_i is

$$\tilde{f}_{v_i}(0) = 1 \cdot 1 \cdot \left(\frac{N-n-|v_i|+1}{N-n}\right)^{N-n}$$

Since expectation is linear, the expected number of unwanted vertices with membership count 1 is

$$\mathbb{E}(U_1) = \sum_{\tilde{m}_i, i>1} \tilde{m}_i \cdot \left(\frac{N-n-i+1}{N-n}\right)^{N-n}$$

The value of $\tilde{m}_i$ is calculated easily by integrating over the probability density of the power law distribution introduced in Equation 3.1:

$$\frac{\tilde{m}_i}{n} = \int_{i-0.5}^{i+0.5} f_{LP}(x)dx \iff \tilde{m}_i = \begin{cases} n \cdot \frac{\log(i-0.5)-\log(i+0.5)}{\log(s_{\max})-\log(s_{\min})}, & \text{if } k=1 \\ n \cdot \frac{(i-0.5)^{1-k}-(i+0.5)^{1-k}}{(s_{\max})^{1-k}-(s_{\min})^{1-k}}, & \text{if } k \neq 1 \end{cases}$$

This is not the number by which the sampling of membership count 1 vertices should be reduced, because the vertices whose membership count is sampled as a result of this correction can also produce unwanted vertices of membership count 1. After the initial correction step the probability density of the membership counts is

$$f_{1,PL}(x) = \begin{cases} f_{PL}(1) - \frac{\mathbb{E}(U_1)}{n} + \frac{\mathbb{E}(U_1)}{n} \cdot f_{PL}(1), & \text{if } i=1 \\ f_{PL}(i) + \frac{\mathbb{E}(U_1)}{n} \cdot f_{PL}(i), & \text{if } i \neq 1 \end{cases}$$

Let $\tilde{m}_{i,1}$ be the expected number of vertices with a sampled membership count of i created by the re-sampling of the initial correction step. Because the re-sampling of the vertices that used to have unwanted membership count 1 before the correction is the same routine as the initial power law sampling, $\tilde{m}_{i,1}$ can be expressed based on $\tilde{m}_i$:

$$\tilde{m}_{i,1} = \frac{\mathbb{E}(U_1)}{n} \cdot \tilde{m}_i$$

Let $\mathbb{E}_1(U_1)$ be the number of unwanted vertices of membership count created by the correction based on $\mathbb{E}(U_1)$:

$$\mathbb{E}_1(U_1) = \sum_{\tilde{m}_{i,1}, i>1} \tilde{m}_{i,1} \cdot \left(\frac{N-n-i+1}{N-n}\right)^{N-n}$$

Let $\tilde{m}_{i,k}$ be defined recursively as the expected number of vertices with a sampled membership count of i created by the re-sampling of the k-th correction step. Similarly, let $\mathbb{E}_k(U_1)$ be the number of unwanted vertices of membership count created by the correction based on $\mathbb{E}_{k-1}(U_1)$.
The total correction should consider all membership counts created by the corrections and correct them as well. So the total correction necessary is:

$$\bar{\mathbb{E}}(U_1) = \mathbb{E}(U_1) + \sum_{k=1}^{\infty} \mathbb{E}_k(U_1)$$

The way that $\tilde{m}_{i,1}$ can be expressed based on $\tilde{m}_i$, $\mathbb{E}_k(U_1)$ can be generalised for the recursive definitions:

$$\tilde{m}_{i,k} = \frac{\mathbb{E}_{k-1}(U_1)}{n} \cdot \tilde{m}_i$$

Applying this to $\mathbb{E}_k(U_1)$ provides this recursive formula:

$$\mathbb{E}_k(U_1) = \frac{\mathbb{E}_{k-1}(U_1)}{n} \cdot E(U_1) = \left(\frac{\mathbb{E}(U_1)}{n}\right)^k \cdot \mathbb{E}(U_1)$$

This is sufficient to rewrite $\bar{\mathbb{E}}(U_1)$:

$$\bar{\mathbb{E}}(U_1) = \mathbb{E}(U_1) + \sum_{k=1}^{\infty} \mathbb{E}_k(U_1) = \mathbb{E}(U_1) \cdot \sum_{k=0}^{\infty} \left(\frac{\mathbb{E}(U_1)}{n}\right)^k = \mathbb{E}(U_1) \cdot \frac{1}{1 - \frac{\mathbb{E}(U_1)}{n}} = \frac{\mathbb{E}(U_1) \cdot n}{n - \mathbb{E}(U_1)}$$

□

In practice the calculation of $\mathbb{E}(U_1)$ can be simplified. The larger i gets, the smaller $\tilde{m}_i$ and the other factor becomes. So at some point all larger summands are negligible. If the value of the last added summand is smaller than a user specified value, all further summands are ignored.
The percentage of samples picked corresponding to these unintended vertices of $\bar{\mathbb{E}}(U_1)$ during power law sampling of membership counts is passed to the sampling of membership counts. For the calculation of how many membership counts are in the larger or smaller split during the recursion, the probability is adjusted by this percentage. This ensures that the expected number of vertices with membership count 1 is the same as the expectation from the unaltered power law distribution. This only guarantees correct expectation for membership count 1. Expectations of other membership counts are still distorted by the probabilistic assignment.

Using pre-assignment for the CKB model is optional. While we believe that there are situations where guaranteeing that each vertex is in at least one community is useful, it may also be desired that the distortion of the probabilistic assignment is not altered. In cases where the correction of the distribution provided by pre-assignment is not necessary it is preferable to deactivate it. Tests in Subsection 6.2.6 show that while scaling with pre-assignment is comparable to scaling without it, the absolute runtimes without pre-assignment tend to be notably faster. In particular Figure 6.23 contains a plot with absolute runtimes, where parameter-set 1 with pre-assignment takes about ten times longer than the corresponding parameter-set 2 without pre-assignment.

5.2 Communication-free generation of CKB Graphs

The CKB model has overlapping communities. The vertices' membership counts and the communities' sizes are power law distributed. This section describes a communication-free generator for CKB graphs.

5.2.1 Description and Parameterization

The CKB model was introduced by Chykhradze et al. [CKB+14]. It features overlapping communities whose sizes are drawn from a power law distribution. The membership counts of the vertices are also power law distributed.
Intra community edges for community c_i are generated with probability $p_i = \frac{\alpha}{|c_i|^\gamma}$ like for the power law model described in Section 3.2. Edges between vertices that do not necessarily have to share a community are generated with the ϵ-community. So any edge can be generated with probability p_o. If an edge is generated in multiple communities, it will only be added to the final graph once. If the output of the generated edges is not gathered into a single file, but done separately for each PE, it is possible that multiple PEs' files contain the same edge.
A list of all parameters for the CKB model can be found in Table 5.1. An example of a graph generated by this generator is shown in Figure 5.2. It has a dense core of vertices with high membership count and high degree. The periphery consists of vertices with a membership count of 1 that are predominantly connected to members of the same community.

Parameter	Description	Usual Value
n	number of vertices	-
c_{min}	smallest permitted community size	6
c_{max}	largest permitted community size	$n/10$
v_{min}	smallest permitted membership count	1
v_{max}	largest permitted membership count	$n/10$
k_{comm}	exponent of the PL-distribution for community sizes	2.5
k_{vertex}	exponent of the PL-distribution for membership counts	2.5
α	numerator for calculation of p_i	4
γ	exponent for community size for p_i calculation	0.5
p_o	probability for each edge in the ϵ-community to exist	$2/n$
pre_assign	should pre-assignment be used	false
self_loops	should loops be possible edges	false

Table 5.1: List of the parameters for the CKB model.

5.2.2 Algorithm

The workload distribution works similar to the PL-multicommunity model from Section 4.2. So communities or continuous part of communities are assigned to a PE that then generates all edges for that part of the community. The ϵ-community is also split up among the PEs.

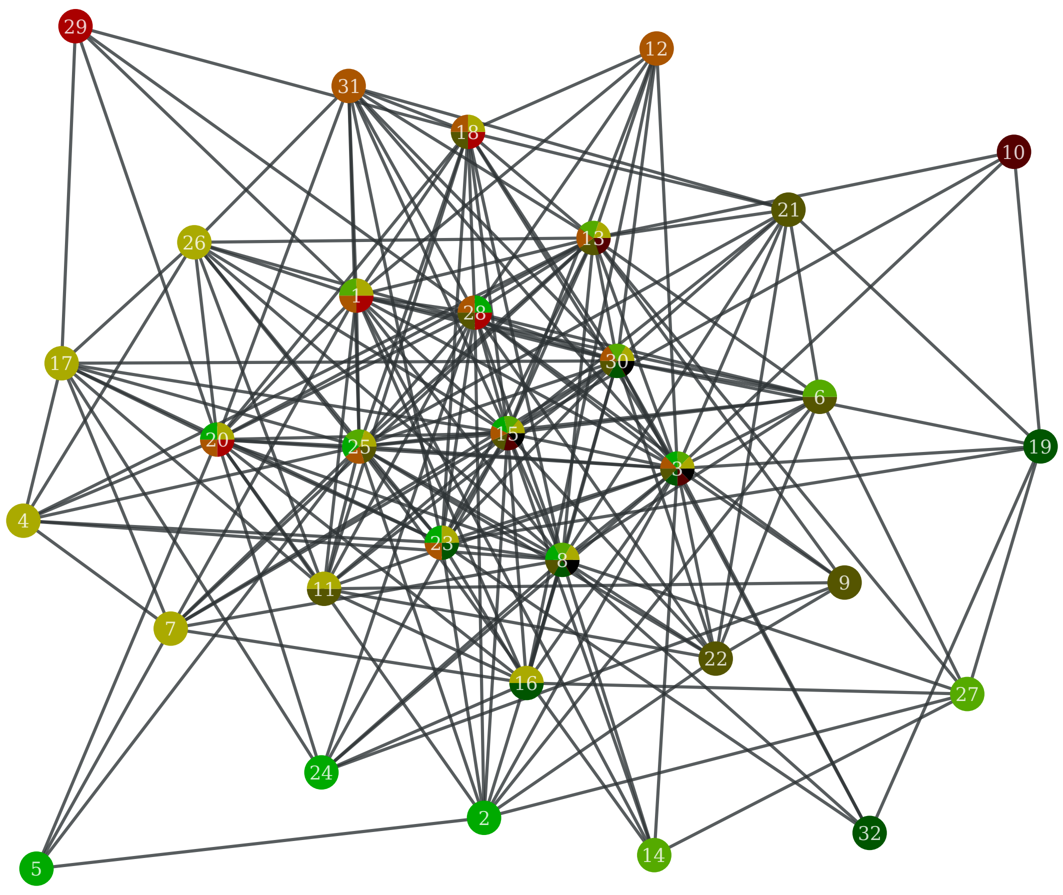


Figure 5.2: An undirected graph generated with the CKB model ($n = 32$, $c_{\text{min}} = 4$, $c_{\text{max}} = 20$, $v_{\text{min}} = 1$, $v_{\text{max}} = 20$, $k_{\text{comm}} = 2.5$, $k_{\text{vertex}} = 2$, $\alpha = 2$, $\gamma = 0.5$, $p_o = 0.08$). Colours indicate community membership.

Initially the membership counts are sampled from the power law distribution. Since this sampling does not need to reach a specified combined mass, the process is simpler than for community sizes. The structure of storing the samples is the same as for sqrt-based sampling in Subsection 3.1.3. So small sampled membership counts are stored by how often they were sampled and large membership counts are stored by what their membership count is.This results in space demands and generation time only scaling with the square root of the vertex count.
Filling these structures is done with a single recursive run. If the largest small membership count is v_{mid}, the number of vertices with small membership count is drawn from

$$\text{Bin}(n, \frac{F_{PL}(v_{\text{mid}} + 0.5) - F_{PL}(v_{\text{min}} - 0.5)}{F_{PL}(v_{\text{max}} + 0.5) - F_{PL}(v_{\text{min}} - 0.5)})$$

For small membership counts the space of possible membership counts is successively split. Assume the split starts with membership counts between i and j being possible and n membership counts to draw. The split occurs at $l = \lfloor \frac{i+j}{2} \rfloor$, so the lower split samples membership counts between i and l. The number of membership counts to be drawn by the lower split is determined by drawing from

$$\text{Bin}(n, \frac{F_{PL}(l + 0.5) - F_{PL}(i - 0.5)}{F_{PL}(j + 0.5) - F_{PL}(i - 0.5)})$$

For large membership counts samples are drawn one after another until enough samples were generated.
If pre-assignment is used, the correction for vertices of membership count 1 is utilized as described in Subsection 5.1.2. For a split as described above, where the lower split contains the vertices of membership count 1, the number of membership counts in the lower split is drawn from

$$\text{Bin}(n, \frac{F_{PL}(l+0.5) - F_{PL}(i-0.5)}{F_{PL}(j+0.5) - F_{PL}(i-0.5) - P_{U_1}})$$

instead. Here n is the number of vertices and $P_{U_1} = \frac{\bar{\mathbb{E}}(U_1)}{n}$ is the percentage of expected unwanted vertices of membership count 1 also considering the samples that are drawn as a result of this correction. The calculation of $\bar{\mathbb{E}}(U_1)$ is described in Lemma 5.1.
After the sampling of membership counts, the combined size of all communities, which is the combined size of all membership counts, is fixed. So the community sizes can be sampled using sqrt-based sampling as described in Subsection 3.1.3.
Members are assigned to each community by probabilistic assignment as described in Subsection 4.1.3. Because vertices have different membership counts, members are not picked uniformly at random, but with probabilities proportional to their membership count. If pre-assignment is used, vertices are probabilistically assigned with probabilities proportional to one less than their membership count, so vertices with a membership count of 1 can not be assigned to a second community.
Edge generation works as it did in the overlapping power law community model in Section 4.2. For intra-community edges the probability $p_i = \frac{\alpha}{|c_i|^\gamma}$ for each edge to be generated is calculated based on the size $|c_i|$ of the community. For edges belonging to the ϵ-community, p_o describes the chance of each edge to be created.

5.2.3 Implementation Details

The initial sampling of membership counts and community sizes has to be done by every PE as explained in Subsection 3.2.3. The communities relevant to the active PE as well as the number of vertices from which the PE has to generate edges in those communities can be determined by a logarithmic search over the community size data structures. The splitting of community parts among the PEs is described in Subsection 4.2.3.

While the different ways of storing small and large communities can be masked by the function returning which communities are relevant to the current PE and what parts of these communities should be handled, this is not easily possible for vertices. Members are assigned to communities by being picked by them. Each community can have small, large and pre-assigned members. The structure of how these vertices are assigned to communities without collisions is described in Subsection 5.1.2.
It is important to note that the probabilistic assignment has to sequentially assign vertices based on membership count. Assume that all vertices of membership count larger than m_i were already considered. Also assume there are n vertices still to be assigned to the current community and there are $\hat{m}_i$ vertices of membership count m_i or smaller that could be probabilistically assigned. Then the number of vertices of membership count m_i is determined by drawing from $\text{Hyp}(n, |m_i|, |\hat{m}_i|)$, where $|m_i|$ is the number of vertices with membership count m_i.
Especially for small communities it is tempting to use a recursive splitting process for probabilistic assignment, cutting the membership counts of potential members in half each step. This would allow cropping whenever no more assignments are left to do for vertices in a split of membership counts. But this distorts the distribution with which the assignment is done. Vertices sharing splits with vertices of very large membership count will get assigned more often than they should. This is because in a recursive splitting process the probability of being picked depends on the combined mass of all objects in

that split. So very large membership counts will contribute probability mass at a level that would cause that vertex to be assigned multiple times. Every splitting step can prevent any created split from having to assign more vertices than there are in that split. But it can not detect easily how unevenly the probability mass is distributed within that split. So especially for larger communities in later recursion steps there will be cases where all vertices in a split have to be assigned to the community, even those vertices who would – when individually considered – have a notably lower chance of being assigned.
For small members and small pre-assigned members, the vertices are hashed with a membership count specific hash function $g_l(x)$. In the case of small members assigned by the probabilistic assignment, the hash function $f_{c_i,l}(x)$ and a potential shift out of the part of the domain of $g_l(x)$ reserved for pre-assigned vertices is used to obtain the specific members before applying $g_l(x)$. To avoid having to store all specific members, these hash functions are set up once for each community and stored in a map, ensuring that only those hash functions belonging to membership counts of members of the current community are created. Seeding ensures, that the hash function for a given membership count is the same across all PEs. Because pre-assigned small members are in a specifically offset interval in the domain of $g_l(x)$ and their assignment process is different to that of not pre-assigned small members, small pre-assigned members are stored in a separate data structure during edge generation.
For large members and large pre-assigned members there is no need for membership count specific hashing. So the large members can be used directly for edge generation. This also means that there is no need for distinction between pre-assigned members and members assigned with probabilistic assignment. All members with large membership count are stored in a single data structure.
All members are hashed with a universal hash function $h(x)$ to break up the correlation between small vertex ids and small membership counts. So in this setting it would be possible to remove the secondary hashing step with $f_{c_i,l}(x)$ during membership internal assignment, because all structures removed by that hash would also be removed when only applying $h(x)$.

The theory of the intra-community edge generation is the same as in the previously discussed models. But there are three different data structures storing the members, that all have subtle differences in how the actual vertices are determined. So there are nine functions dealing with intra-community edge generation, one for each pair of source and target vertices. So e.g. there is a function generating edges from small members obtained from probabilistic assignment to small pre-assigned vertices. Small membership count based functions also dedicate the edge generation to functions only working with one membership count. This ensures, that within each of these functions all edges have incident vertices in a specific data structure. So in a block with m possible edges, a initial binomial pick of how many edges are to be generated from $\mathrm{Bin}(m, p_i)$. Those edges can then be uniformly random distributed among the homogeneous available space.

6. Experimental Evaluation

This chapter covers the experimental evaluation of the generators described in previous chapters. Several metrics are analysed for the generated graphs to check their properties. To evaluate weak and strong scaling, test with different parameter-sets are conducted.

6.1 Model Evaluation

Here the properties of the graphs produced by the generators are analysed. This both verifies the functionality of the generators and gives insight in how the probabilistic routines affect properties of the generated graphs.

6.1.1 Basic Models

To evaluate the graphs produced by the basic models, parameters creating a normal setting were used. They are listed in Table 6.1. The graphs were generated using 4 PEs and the seeds were set to 1.

Model	n	c	p_i	p_o
equal communities	2^{20}	500	0.1	$2 \cdot 10^{-6}$
unique communities	2^{20}	500	0.1	$2 \cdot 10^{-6}$

Table 6.1: Parameters used to produce the plots of graph metrics of basic models.

The communities in the equal-communities model have only two different sizes. The density of those communities seems normally distributed around the specified p_i of 0.1. The degrees of vertices are also normally distributed around the expected value. The plots can be seen in Figure 6.1.
Community size and density are both normally distributed around the expected values for the unique-communities model. This is shown in Figure 6.2. The vertex degrees are distributed almost exactly the same as in the equal-communities model.

6.1.2 PL Models

The PL-communities model and the similar PL-multicommunity model with overlapping communities have many similarities, so their evaluation is handled together here. The parameters used to generate the graphs can be seen in Table 6.2. The graphs were generated using 4 PEs and the seeds were set to 1.

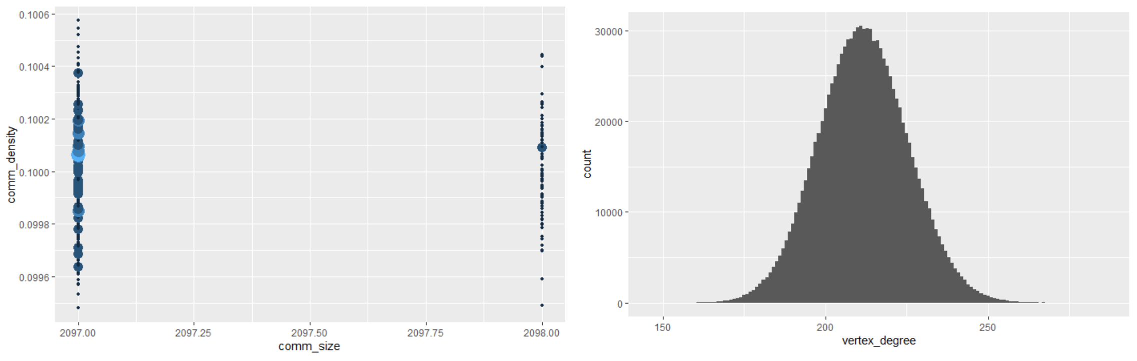


Figure 6.1: Community size vs density of the equal-communities model are plotted on the left and a histogram of the vertex degrees produced by that run is on the right.

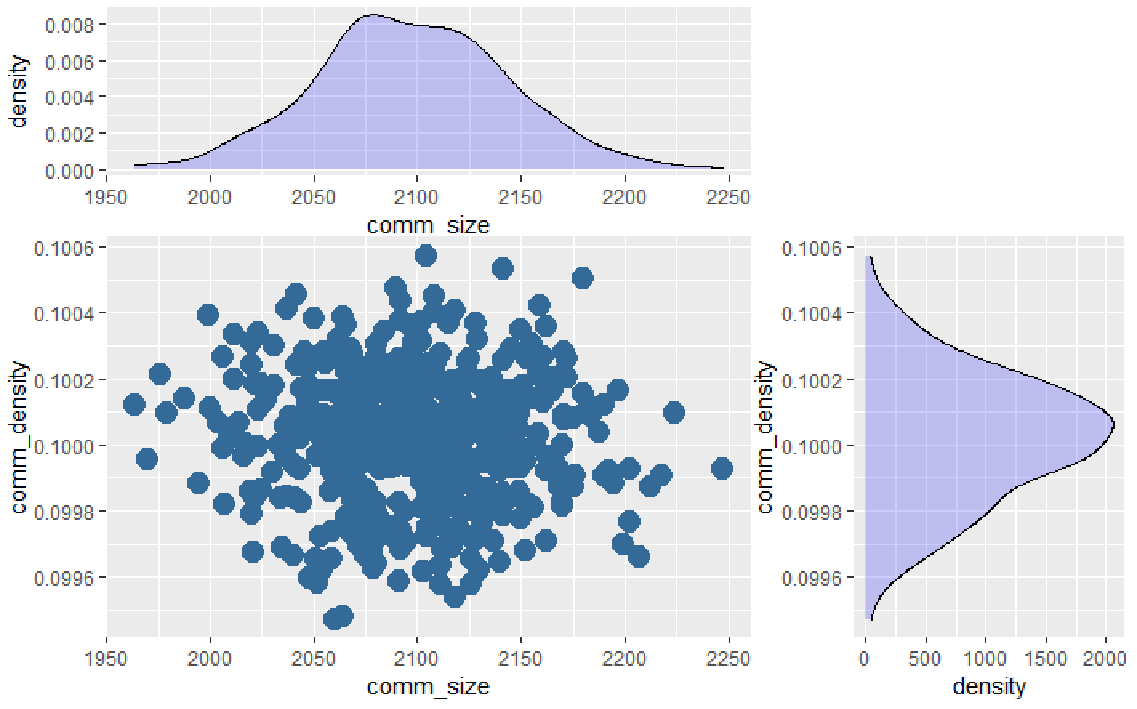


Figure 6.2: Plot showing the community sizes and densities produced by the equal-communities model.

Model	n	$c_{\min}$	$c_{\max}$	vertex_mc	k_{comm}	α	γ	p_o
PL-communities	2^{20}	6	$1 \cdot 10^5$	–	2.5	4	0.5	$2 \cdot 10^{-6}$
PL-multicommunity	2^{20}	6	$1 \cdot 10^5$	4	2.5	4	0.5	$2 \cdot 10^{-6}$

Table 6.2: Parameters used to produce the plots of graph metrics of the PL-community models.

The community sizes and their density of the PL-multicommunity model are shown in Figure 6.3. As intended, the density decreases gradually with increasing community size. Even though the power law distribution allows community sizes up to 100000, the largest community in this run contains just above 40000 members. The test run of the PL-communities model without overlapping communities has a combined community size that is four times lower because each vertex is only in a single community. There the largest community has only just above 6000 members. Apart from the reduced number of members, communities behave the same for the PL-communities model.

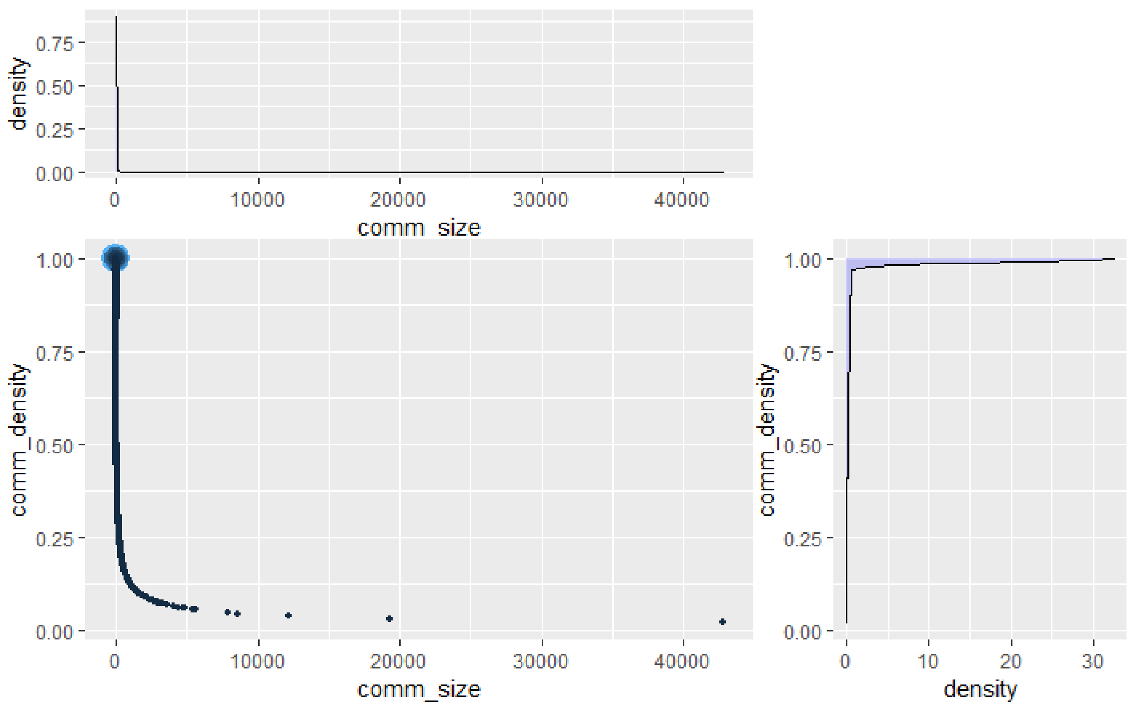


Figure 6.3: Plot showing the community sizes and densities produced by the PL-multicommunity model.

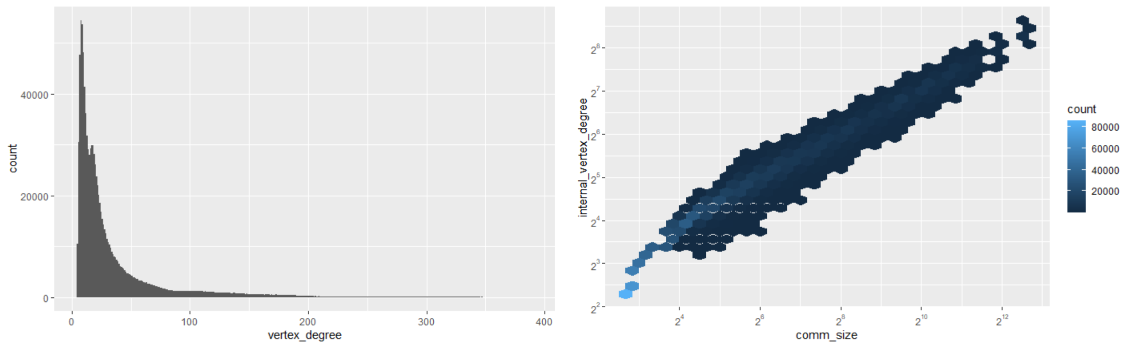


Figure 6.4: Histograms of vertex degrees (left) and plot of internal degrees against community sizes (right) of the PL-communities model.

The vertex degrees produced by those runs can be seen in Figure 6.4. On the left is the plot for the model without overlapping communities, which produces lower degrees overall. The degrees seem to be roughly power law distributed when very small degrees are ignored. Because every vertex is in at least one community, there are no vertices with extremely low degree. For this run communities up to size 16 are complete and each vertex is in a community of size at least 6. So unless a vertex is in a larger community and in spite of higher expected degree gets assigned very few vertices, each vertex is guaranteed to be incident to at least 5 edges.
The right plot of Figure 6.4 shows the internal degree of each community member plotted against the community's size. The guaranteed completeness of small communities can be seen in the lower left, where communities of small sizes all have the members of the same internal degree. For larger communities the range of values the internal degree takes grows slower than the number of members in that community, producing the intended sub-linear scaling. Members of larger communities have a wider range of internal degrees than those of smaller communities. The plot only becomes narrower due to the logarithmic scaling of

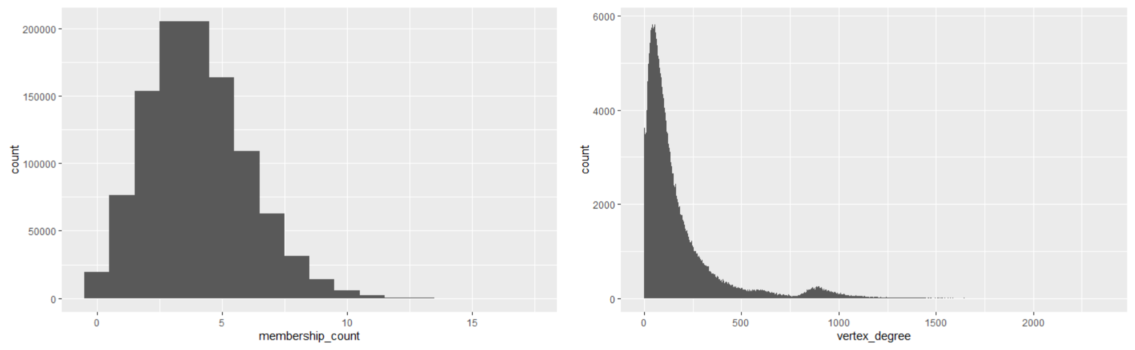


Figure 6.5: Histograms of membership counts (left) and vertex degrees (right) of the PL-multicommunity-model.

the axes. For overlapping communities in the PL-multicommunity model this behaviour can be observed as well.

The overlapping communities of the PL-multicommunity model do not guarantee membership in a community for each vertex. In Figure 6.5 the distribution of actual membership counts for an intended membership count of 4 can be seen. There are vertices without any communities, these vertices tend to have very low degree. In the right histogram of Figure 6.5 these low degree vertices can be seen. There are 2347 vertices without any incident edges in this graph.
The power law like distribution of degrees is broken by a increased number of degrees around 900. These degrees correspond to members of the largest community with roughly 42000 members. The expected internal degree in a community of that size with the given parameters is around 820. The edges missing to a degree of 900 come from the intra-community edges of other communities those vertices are a member of.

6.1.3 CKB Model

To evaluate the CKB model a graph was generated with the parameters in Table 6.3. Those parameters create a normal setting where vertices are pre-assigned to one community each. The seed used was 1 and the generator was run with 4 PEs.

n	c_{min}	c_{max}	v_{min}	v_{max}	k_{comm}	k_{vertex}	α	γ	p_o	pre-assign
2^{20}	6	$1 \cdot 10^5$	1	$1 \cdot 10^5$	2.5	2.5	4	0.5	$2 \cdot 10^{-6}$	true

Table 6.3: Parameters used to produce the plots of graph metrics of the CKB model.

The community sizes and their density is distributed similar to the PL-community models. They are plotted in Figure 6.6. The largest sampled community size is less than half the maximum permitted size and most communities are small and dense.
Because of the use of pre-assignment, there are no vertices with membership count 0. The low minimum membership count of 1 combined with the relatively high power law exponent of 2.5 result in a membership count of 1 for most vertices. Also, the maximum reached membership count is notably lower than the size of the largest community that was sampled with the same maximum permitted sample.
The degrees of the vertices are roughly power law distributed when very small degrees are ignored. The degree distribution of the generated graph is visualised in Figure 6.8. Because all vertices are in at least one community, there are no isolated vertices. The smallest degree of any vertex is 5, which corresponds to belonging to exactly one community of the smallest size and having no inter-community edges. Since both community sizes and

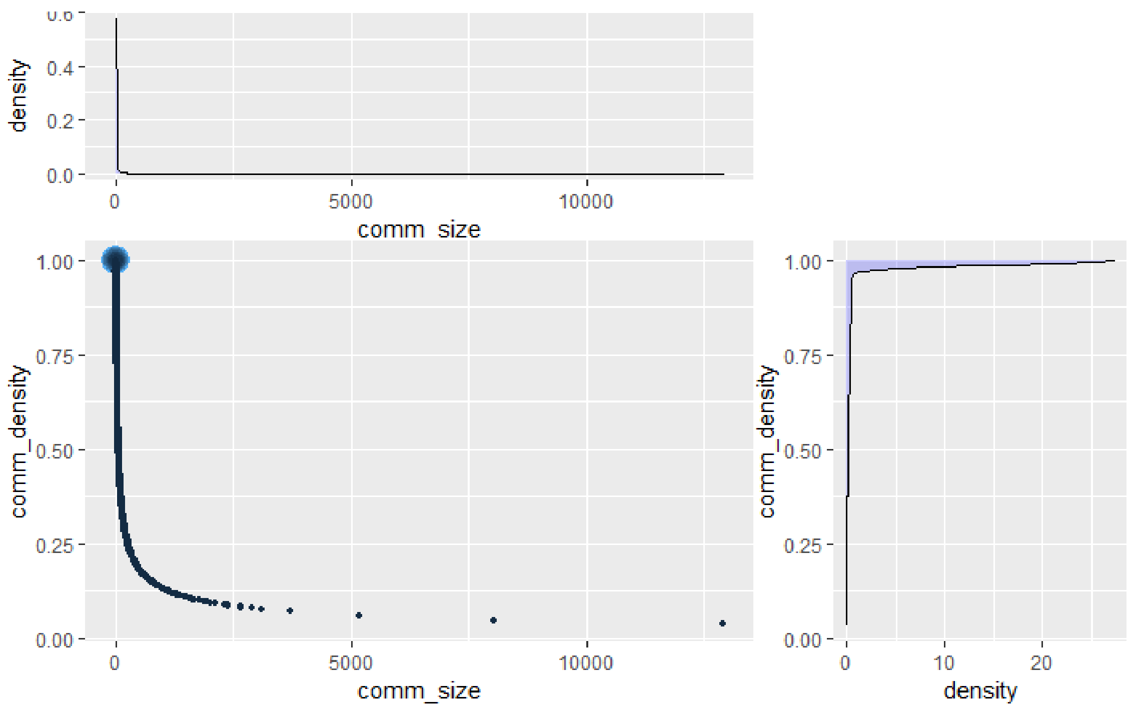


Figure 6.6: Plot showing the community sizes and densities produced by the CKB model.

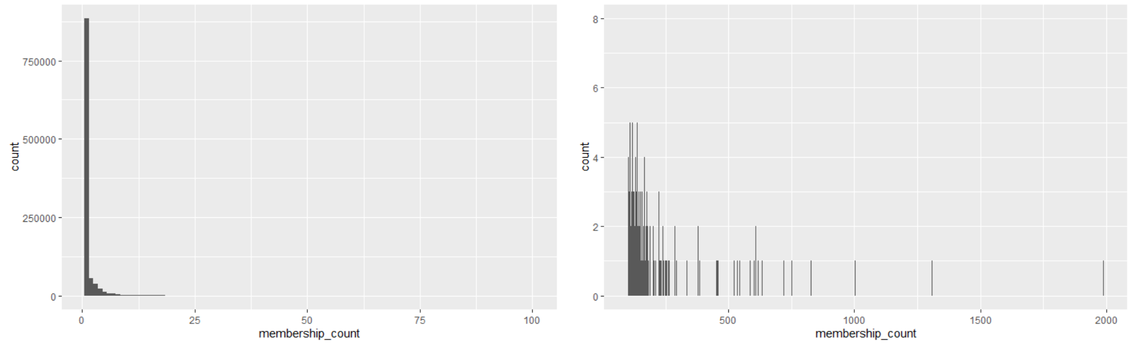


Figure 6.7: Histograms of membership counts of vertices in the CKB graph. The plot is split into values at most 100 and values larger than 100 for better visibility.

membership counts are power law distributed, there are very few vertices with very large degree. They have a large membership count and are members in large communities. With this the most connected vertex of the graph achieves a degree that is a lot larger than the size of the largest community.
The internal vertex degrees are plotted in Figure 6.9 on the left. They behave similar to the internal degrees in the PL-community models. Small communities up to a size of 16 are guaranteed to be fully connected. Members of larger communities can have lower internal degree, but by design the expected degree increases at a sub-linear rate with increasing community size.
Note that the range of different inner degrees for large communities is larger than for the PL-community models in Figure 6.4. This range is not distributed symmetrically around the expected inner degree, but there are vertices whose internal degree is much larger than that of most members. These vertices have a large membership count and are also members of communities overlapping the large community. As such edges of different overlapping communities are counted towards the internal degree, causing higher internal degrees than would correspond to the intra-community edge probability of the considered community.

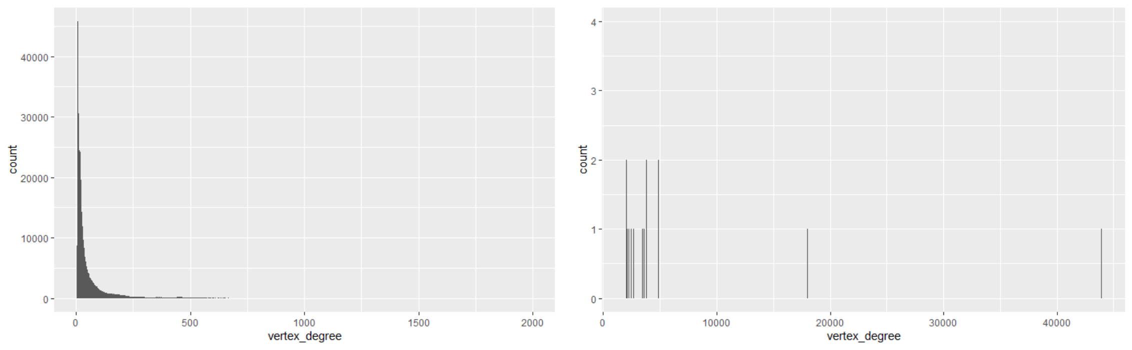


Figure 6.8: Histograms of vertex degrees in the CKB graph. The plot is split into values smaller than 2000 and values at least 2000 for better visibility.

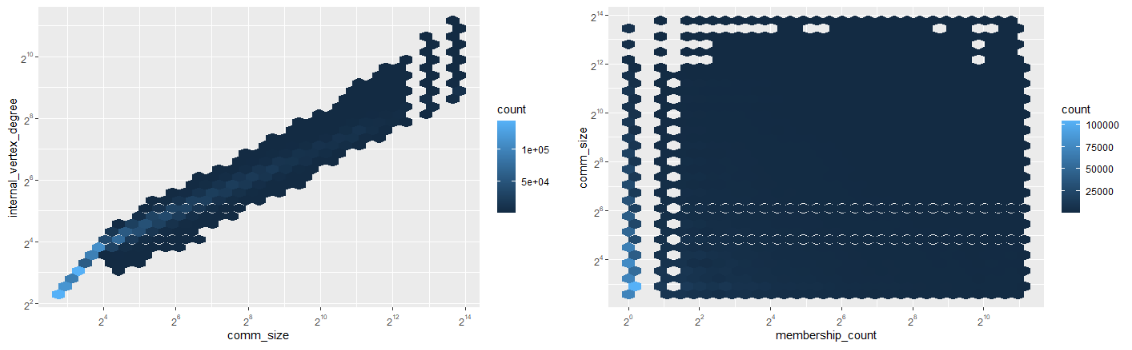


Figure 6.9: Plot of internal degrees against community size (left) and membership count against community size (right) of the CKB graph.

This does not happen at a noticeable scale for models where communities do not overlap or vertices are only in a very limited number of communities.
The right plot of Figure 6.9 shows that the assignment of vertices to communities does not produce any notable bias toward assigning vertices of a given membership count to communities of a certain size. Vertices located in only a single community are found in communities of all sizes. Similarly, vertices with a large membership count show no tendency towards being located in larger or smaller communities.

6.2 Scaling Tests

For a parallelized implementation scaling is important. This section explores how the runtime of the generators scales with increased problem size and increased usage of PEs.

6.2.1 Test Environment and Setup

All scaling tests were run on a remote machine with two 64-Core AMD EPYC$^{\text{tm}}$ Zen2 7742 CPUs. It has 1024GB of DDR4 EEC RAM organised as 16×64GB.

Times for the scaling tests are excluding IO and MPI overhead. Measurements begin from the moment that the generator is started and end when it has finished generating its edges. For each model multiple parameter sets are tested and each of these parameter sets is run with three different seeds. For those situations where edge count is used to put runtimes into perspective, the sum of generated edges across all PEs will be given. This is not necessarily the number of edges of the final graph, since some models will under some circumstances generate the same edge multiple times on different PEs. All scaling tests were run using the undirected version of the models and did not allow loops.
For strong scaling, the parameters of the graph to be generated stay unaltered. The number of PEs used gets increased gradually. Tests are run for all PE counts that are powers of 2 up to 128 and there is a run with 6 PEs, showcasing that the PE count is not restricted to powers of 2. The main metric showing the quality of the parallelization is the speedup. It divides the runtime of the sequential approach, which is the run with a single PE, by the runtime of the current run. Perfect scaling would result in the speedup being equal to the number of PEs used.
Weak scaling explores how the runtime of the generators develops when the workload for each PE remains more or less the same, but the number of PEs used is increased. The edge count of the graph is hard to control, especially as the interface expanded from the original KaGen library only specifies vertex count by powers of 2. Therefore, the vertex count of the graph will be scaled with the number of PEs used. Since the edge count grows superlinearly for some models and parameter sets, some models will have an increased average workload per PE. To take this into consideration, the accumulated runtime of all PEs will be plotted against the number of generated edges. Perfect scaling would result in the same runtime for all runs, or at least in constant workload per edge generated.

6.2.2 Scaling Results Overview

The basic models scale nearly perfectly, both when generating the same graph with more PEs and when increasing the size of the generated graph while keeping the workload per PE fixed.

For all models with power law distributed community sizes the workload is not balanced perfectly among the PEs used. Each PE generates edges for a fixed number of vertices. But vertices in larger communities have higher expected degree, so PEs generating edges for members of larger communities have to do more work.
This is not a fundamental issue of these generators. The expected numbers of communities of any size is known. With experimentally obtained knowledge about how much work is related to edge generation and how much work is inherent to each PE or has to be done per community handled, an improved re-balancing of the workload among the PEs should be possible.

With probabilistic assignment overlapping communities can be generated in comparable time to not overlapping communities. This means that graphs with the same combined size of all communities can be generated with similar effort.

The CKB model tends to spend a notable amount of work on the membership assignment routine when there are many small communities. Hence, settings with small communities tend to scale with the size of the structures storing power law sampled values, which roughly scales in $\mathcal{O}(\sqrt{n})$. Even with a re-balancing of the workload distribution, this would always negatively impact the weak scaling of this generator.

6.2.3 Basic Models

Both the equal communities model and the unique communities model have the same parameters and were tested with the same parameter sets.

Strong Scaling

The first parameter set tested here covers a normal scenario. It has communities with slightly more than 2000 members each. A vertex is expected to have roughly 200 edges to members of the same community and 0.4 edges to vertices in different communities.
The second parameter set has a much lower number of communities, which are therefore much larger. The probability for edges within a community was set to get a comparable, but slightly higher vertex degree than with the first parameter set.
The third and last parameter set is similar to the first, but generates graphs with a significantly higher density.
The exact parameters used can be seen in Table 6.4.

Parameter Set	n	c	p_i	p_o
1	2^{28}	130000	0.1	$1.4 \cdot 10^{-9}$
2	2^{28}	1000	0.001	$1.4 \cdot 10^{-9}$
3	2^{28}	130000	0.75	$2.8 \cdot 10^{-9}$

Table 6.4: Parameter sets used to test strong scaling for the equal communities model.

Plots visualizing the scaling behavior of the equal-communities model can be seen in Figure 6.10. The absolute runtime in seconds plotted in the upper diagram is the average of the three differently seeded runs. Both parameter-set 1 and 2 have similar density and run in comparable time. The higher density graph generated with parameter-set 3 takes more time to account for the additional edges generated. Scaling is practically perfect for all parameter sets. The dip in speedup seen for 128 PEs can be observed for all models tested and is probably a negative effect related to using all available PEs.

Scaling of the unique communities model is also nearly perfect. Absolute runtimes are slightly higher than those of the equal communities model which does not permute the vertex ids before pushing generated edges to the graph. The plots of the strong scaling for unique communities are displayed in Figure 6.11.

Weak Scaling

The parameter sets to test weak scaling are similar to the ones used for strong scaling.
The first parameter set tests a setting with slightly more than 2000 vertices per community and a fixed intra-community edge probability. The intra-community edge probability scales with the vertex count to maintain the expected number of intra-community edges per vertex.
The second parameter set keeps the number of communities constant at 1000. To maintain a comparable vertex degree, the probability of edges within and between communities scale with the vertex count.

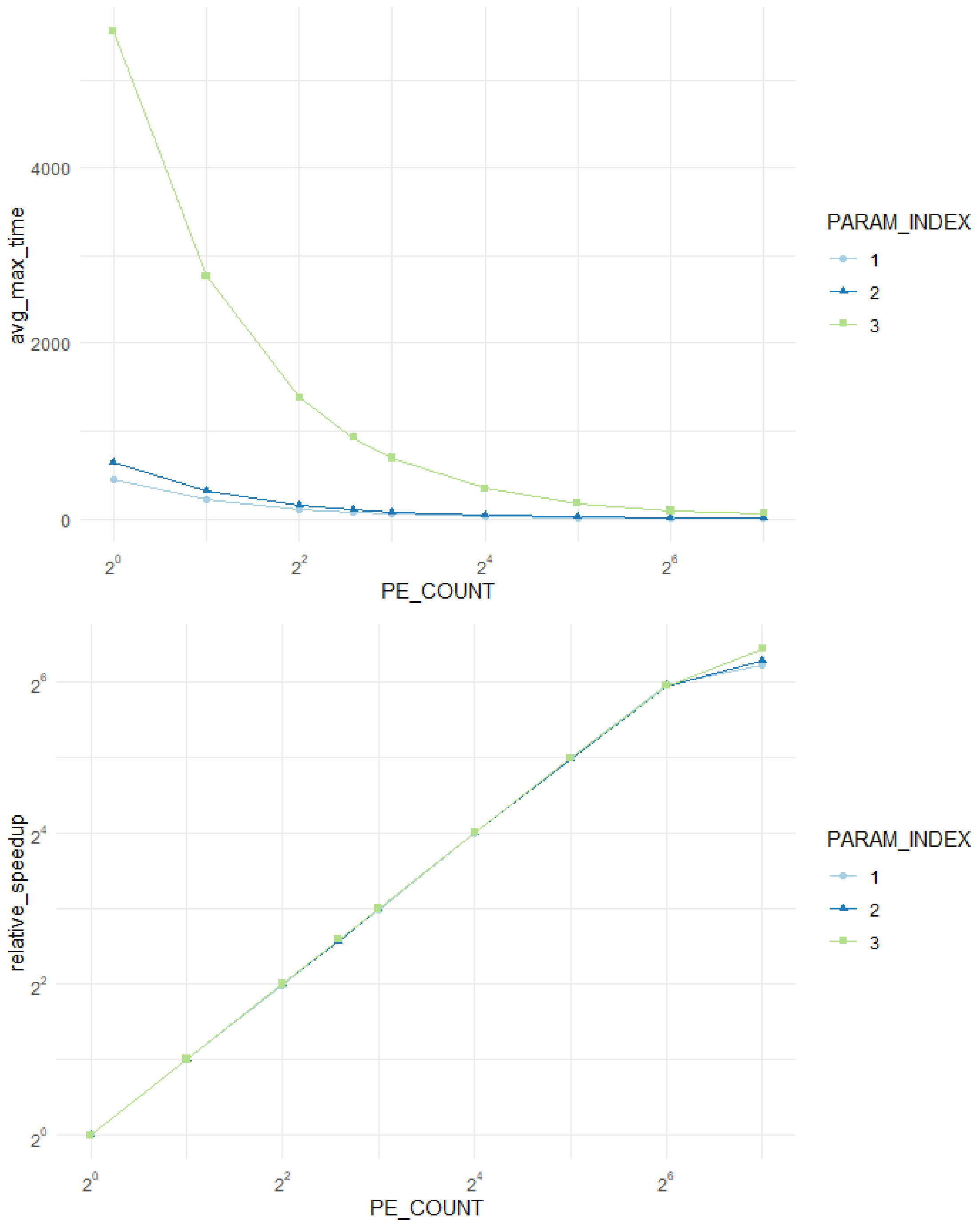


Figure 6.10: Plots visualizing the near perfect strong scaling of the equal communities model. The top graphic plots total runtime in seconds against PE count. The lower one shows the speedup.

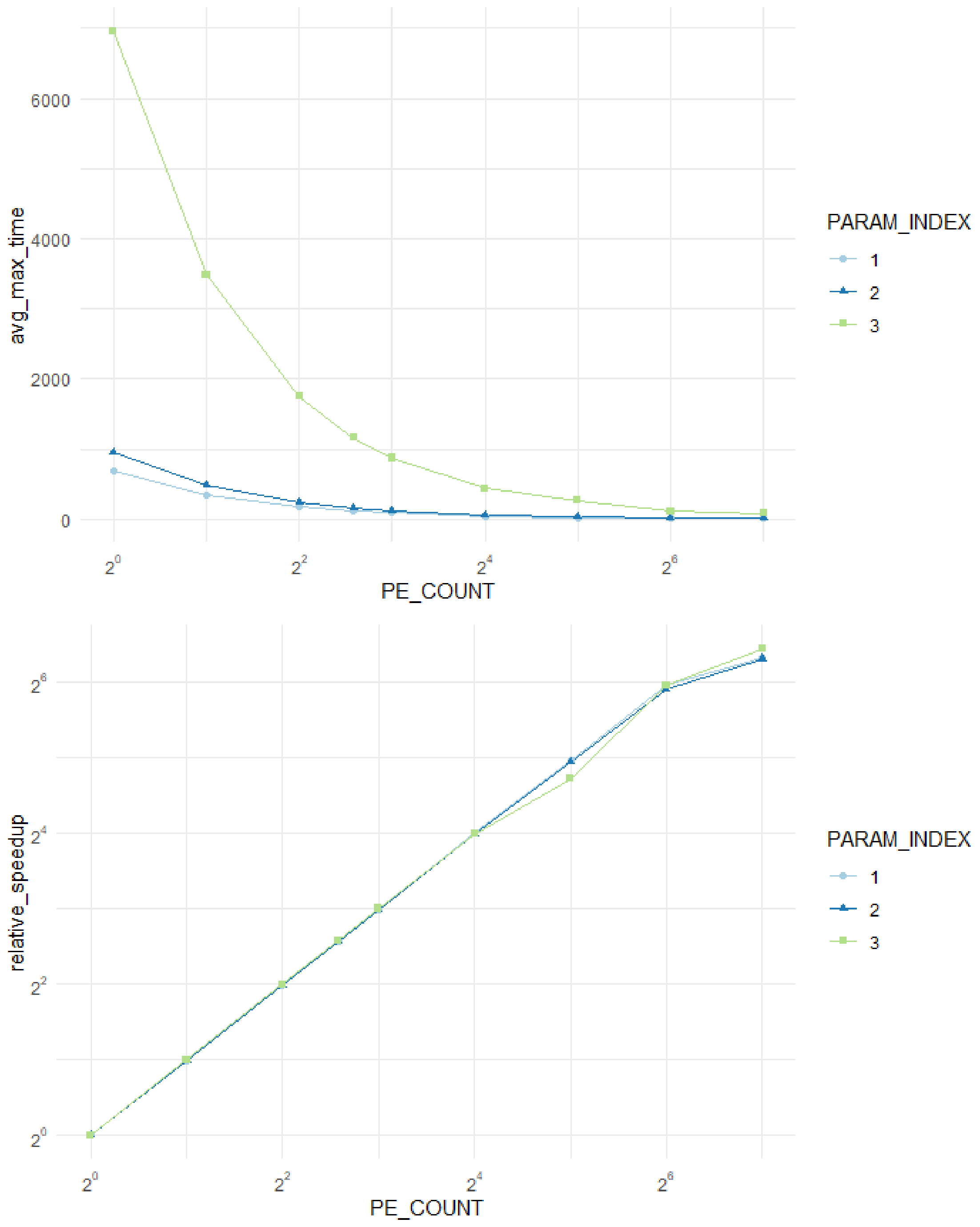


Figure 6.11: Plots visualizing the near perfect strong scaling of the unique communities model. The top graphic plots total runtime in seconds against PE count. The lower one shows the speedup.

The third parameter set scales parameters like the first. The intra-community edge probability is increased to produce graphs with a higher density.
The parameters used are shown in Table 6.5. For the scaling parameters the parameter for sequential and maximally parallel computation are given, p_o decreases with increased vertex count. They all scale linearly with vertex count and are specified with two significant digits.

Parameter Set	n	c	p_i	p_o
1	$2^{24} - 2^{31}$	$8200 - 1000000$	0.1	$1.2 \cdot 10^{-7} - 9.3 \cdot 10^{-10}$
2	$2^{24} - 2^{31}$	1000	$0.16 - 0.0013$	$1.2 \cdot 10^{-7} - 9.3 \cdot 10^{-10}$
3	$2^{24} - 2^{31}$	$8200 - 1000000$	0.75	$1.2 \cdot 10^{-7} - 9.3 \cdot 10^{-10}$

Table 6.5: Parameter sets used to test weak scaling for the equal communities model.

Weak scaling for the equal communities model is also nearly perfect. The average runtime over all seeded runs of each parameter set is plotted the upper graphic of Figure 6.12. The increased runtime for 128 PEs is experienced by all models and unlikely to hint towards bad scaling of the algorithms used. Here the second parameter set takes time comparable to the third, because its intra-community edge probabilities are one order of magnitude larger than they were for the strong scaling tests.

As with strong scaling, there is no notable difference in scaling behaviour between the basic models. Again, the absolute runtime of the unique communities model is a bit higher than that of the equal communities model. The plots visualizing runtimes for the unique communities model are shown in the lower plot of Figure 6.12.

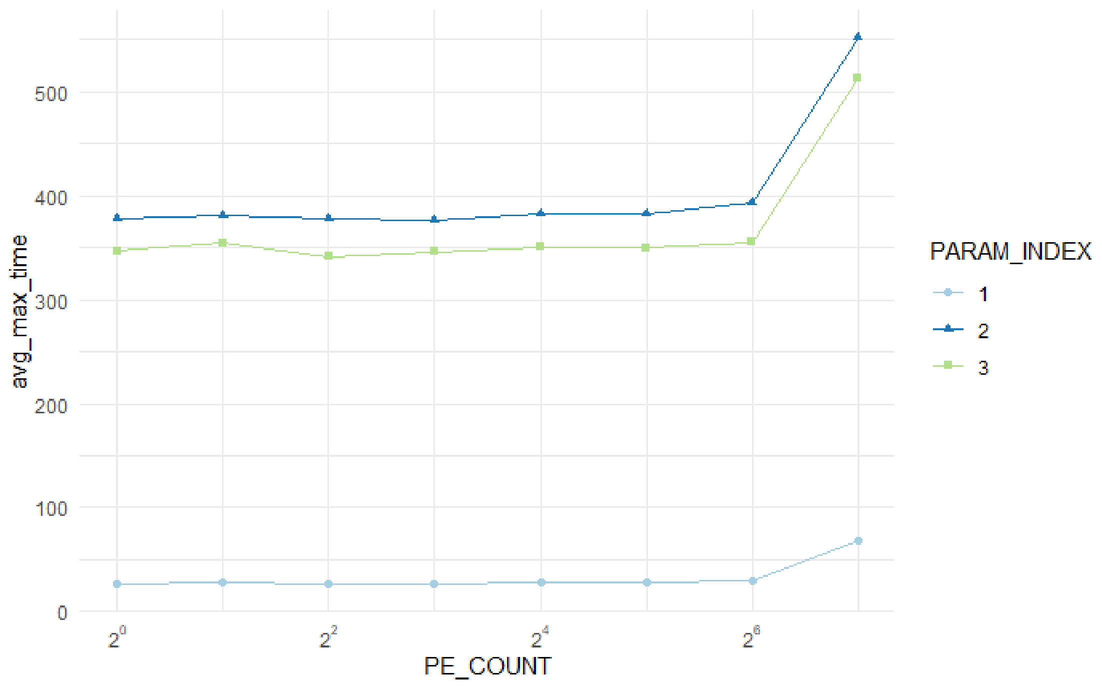
6.2.4 PL-Communities Model

Scaling of Sqrt-Based Sampling

The sampling of community sizes from a power law with the sqrt-based approach of Subsection 3.1.3 works by sampling and discarding batches of samples. A batch is a recursive sampling run for adding or removing a fixed number of communities. This allows efficient sampling to reach an intended combined size of all samples. The runtime of each batch is in $\mathcal{O}(\sqrt{n})$ as discussed in the description of the approach. But due to the extreme variance of the power law distribution, no sufficiently good bound for the number of batches needed to be sampled or discarded could be given.
So here some experimental evaluation is conducted to check the hypothesis, that only few batches have to be computed. For this two sets of tests were run with different exponents and the comparably large maximum community size of a tenth of the vertex count. Smaller maximum community sizes only make the power law distribution more predictable. Therefore, sampling runs are less likely to over- or under-sample drastically and the necessity to run more batches is reduced.
The tests were run for vertex counts between 2^{20} and 2^{50} for all powers of 2. Each parameter-set was tested in three runs with seeds from 1 to 3. The parameters used can be seen in Table 6.6.

n	c_{min}	c_{max}	k_{comm}
$2^{20} - 2^{50}$	6	$n/10$	2.5
$2^{20} - 2^{50}$	6	$n/10$	1

Table 6.6: Parameters used to test the scaling of the sqrt-based sampling routine.



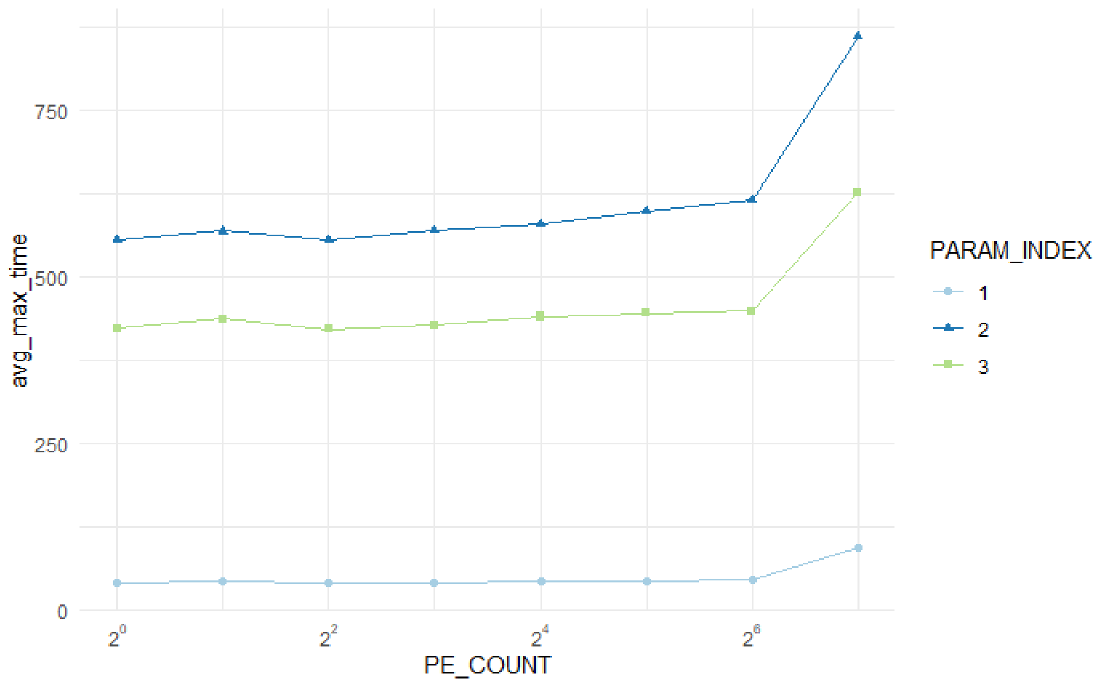


Figure 6.12: Plots visualizing the near perfect weak scaling of the basic model. They plot total runtime in seconds against PE count. The upper graphic belongs to the equal-communities model and the lower plot visualizes the scaling of the unique communities model.

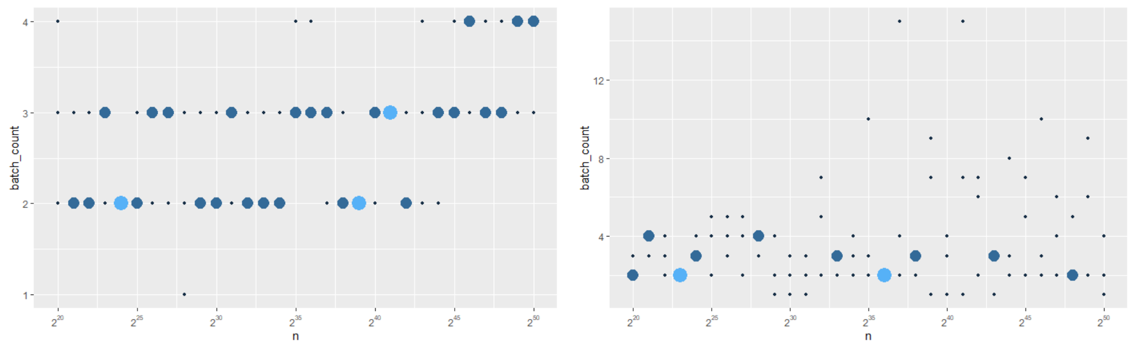


Figure 6.13: Plots of how many batches needed to be sampled during sqrt-based sampling. The left plot visualize the tests done with exponent 2.5, the tests of the right plot were done with exponent 1.

The results of these tests can be seen in Figure 6.13. As expected only very few batches have to be sampled. For the tests with exponent 2.5 at most 4 runs to sample or discard community sizes were needed. For the runs with exponent 1 and higher variance 15 batches were always sufficient. There is a tendency for larger sampling runs to require more batches, but there are still large sampling runs completed with a single batch.

Strong Scaling

The first parameter set for strong scaling tests is a normal setting. Community sizes can be between 6 and one tenth of the vertex count. The exponent of the power law distribution for community sizes and the parameters to calculate intra-community edge probability are the default settings of the CKB model. p_o is set to have the expectation of inter-community edges at roughly 2 per vertex.
The second parameter set tests performance for larger communities. The exponent of the power law distribution to draw communities is set to 1, resulting in a more even spread of samples across the permitted sample range. Also, the minimum community size is set to 2000 to get only community sizes beginning at that larger size.
The third parameter set produces graphs with a lower density. For this the maximum community size is set lower than for the other tests preventing large communities with higher internal vertex degrees. The parameters to calculate the intra-community edge probability were also adapted to reduce both the base probability and to slow the rate at which internal degrees grow with increased community size.
A list of all parameters used for these strong scaling tests is given in Table 6.7.

Parameter	**Set 1**	**Set 2**	**Set 3**
n	2^{24}	2^{24}	2^{24}
c_{min}	6	2000	6
c_{max}	$1.6 \cdot 10^6$	$1.6 \cdot 10^6$	20000
k_{comm}	2.5	1	2.5
α	4	4	3
γ	0.5	0.5	0.666
p_o	$1.2 \cdot 10^{-7}$	$1.2 \cdot 10^{-7}$	$1.2 \cdot 10^{-7}$

Table 6.7: Parameter sets used to test strong scaling for the PL-communities model.

The strong scaling behaviour of the PL-communities model is plotted in Figure 6.14. The speedup is no longer nearly perfect. To a small degree this is due to the power-law sampling

process, that has to be done on every PE. But looking at the accumulated runtime of all PEs normalized by the sum of edges generated across all PEs in Figure 6.15 paints a different picture. The normalized scaling looks near perfect, the time taken per edge generated stays almost the same. The peak at full PE usage is an artefact seen for all models and not related to this model.
So there are two factors negatively impacting the speedup here. The PL-communities model still has every PE generate all edges for the vertices assigned to it. This results in rectangle chunks being processed twice by different PEs. For chunks containing almost exclusively inter-community edges there are hardly any edges generated twice, which is why this effect is not noticeable for the basic models. But communities sampled from a power law distribution can become a lot larger, so substantial parts of communities will end up in rectangle chunks and their edges are generated twice. This is why parameter set 3 suffers least from this effect. Its community sizes are capped at a much lower value.
The other issue negatively impacting speedup for all models using power law distributions is imperfect load balancing across the PEs. Each PE processes the same number of vertices, but vertices in smaller communities have less incident edges. So all work related to edge generation – and not sampling community sizes or assigning members – is balanced unevenly. PEs handling vertices in larger communities have to generate more edges, which takes more time. The resulting effects can be seen best for parameter set 1, where the edge count imbalance is greatest. Figure 6.16 shows a box plot of the runtimes of each PE from the run of parameter set 1 with seed 1. For runs with at least eight PEs there is always at least one PE taking notably more time, because it generates notably more edges than the PEs only handling members of small communities.
Note that imperfect load balancing is not an inherent issue of this approach. The current implementation takes a very simple way of dividing up the workload. With data that indicates how much of the work of each PE falls to edge generation and how much is sampling and assignment, it should be possible to find a better workload distribution. The runtime of edge generation depends on how many communities and vertices are processed. The runtime of community assignment and sampling depends on how many communities and vertices are processed or is fixed per PE. So knowing the parameters of the generator configuration, allows estimating how much work has to be done overall and then distributing vertices among the PEs in a way that considers the different workload necessary to process them.

Weak Scaling

The parameters used for weak scaling tests fulfil a similar role as the parameters used for strong scaling. Maximum community size is not set to 10% of the vertex count, but to 1.2% and p_o is set to produce an expected 16 edges per vertex. So these parameters would be default for a setting 8 times smaller.
The first parameter set is still the regular setting with high power law exponent for community sizes and scaling maximum community size. Intra community edge probability is parametrised in the usual way as well.
The second parameter set is testing performance with large communities. So the power law exponent for sampling community sizes is set to 1 and the minimum community size is set to 2000. The maximum community size scales with the vertex count.
The third parameter set generates lower density graphs. The maximum community size is fixed and does not scale with vertex count. Parameters to control intra-community edge probability are manipulated in the same way as in the strong scaling tests.
A list of the parameters used can be seen in Table 6.8. Scaling parameters scale linearly with vertex count, values are given explicitly for the smallest and largest setting, values for p_o decrease with increasing vertex count.

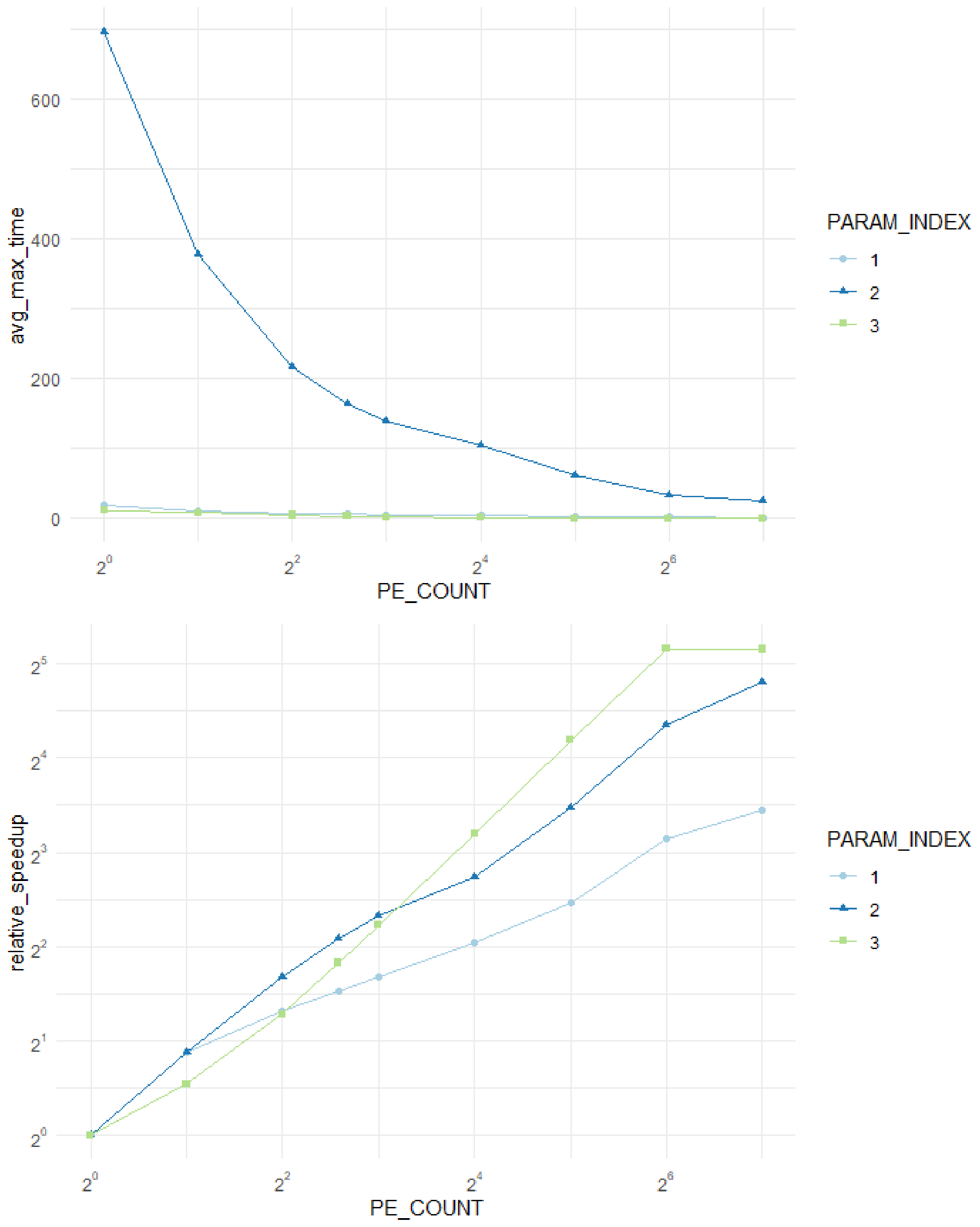


Figure 6.14: Plots visualizing the strong scaling of the PL-communities model. The top graphic plots total runtime in seconds against PE count. The lower one shows the speedup.

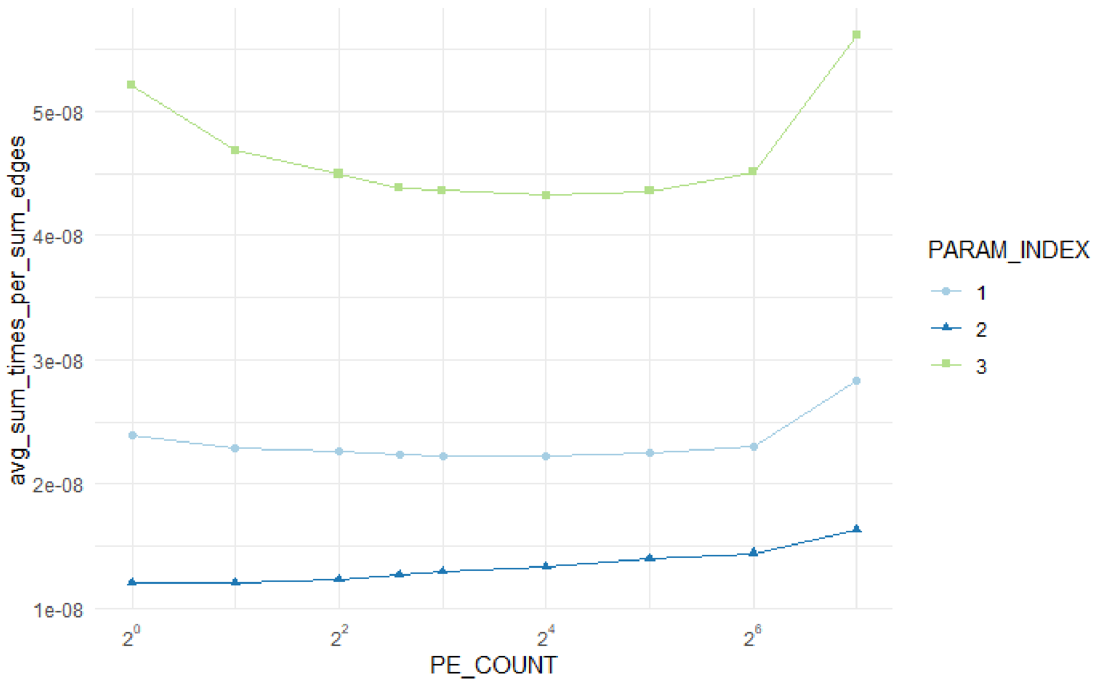


Figure 6.15: Plots visualizing the combined runtime of all PEs divided by the total number of edges generated by all PEs for the strong scaling tests of the PL-communities model.

Parameter	**Set** 1	**Set** 2	**Set** 3
n	$2^{23} - 2^{30}$	$2^{23} - 2^{30}$	$2^{23} - 2^{30}$
c_{min}	6	2000	6
c_{max}	$1.0 \cdot 10^5 - 1.3 \cdot 10^7$	$1.0 \cdot 10^5 - 1.3 \cdot 10^7$	20000
k_{comm}	2.5	1	2.5
α	4	4	3
γ	0.5	0.5	0.666
p_o	$1.9 \cdot 10^{-6} - 1.5 \cdot 10^{-8}$	$1.9 \cdot 10^{-6} - 1.5 \cdot 10^{-8}$	$1.9 \cdot 10^{-6} - 1.5 \cdot 10^{-8}$

Table 6.8: Parameter sets used to test weak scaling for the PL-communities model.

The weak scaling behaviour is similar to the strong scaling behaviour. Except for parameter set 3, the runtimes visualised in Figure 6.17 seem to scale poorly. The issues of load balancing and calculating edges twice were already discussed for the strong scaling tests. The parameter sets with scaling maximum community size also naturally produce higher density graphs, additionally increasing the total workload. The accumulated runtimes of all PEs normalized by the total number of edges generated are shown in Figure 6.18. These plots suggest that the scaling of work per edge is still very good, but generating more edges twice and having few PEs work notably longer than most PEs, significantly worsens the actual runtimes.

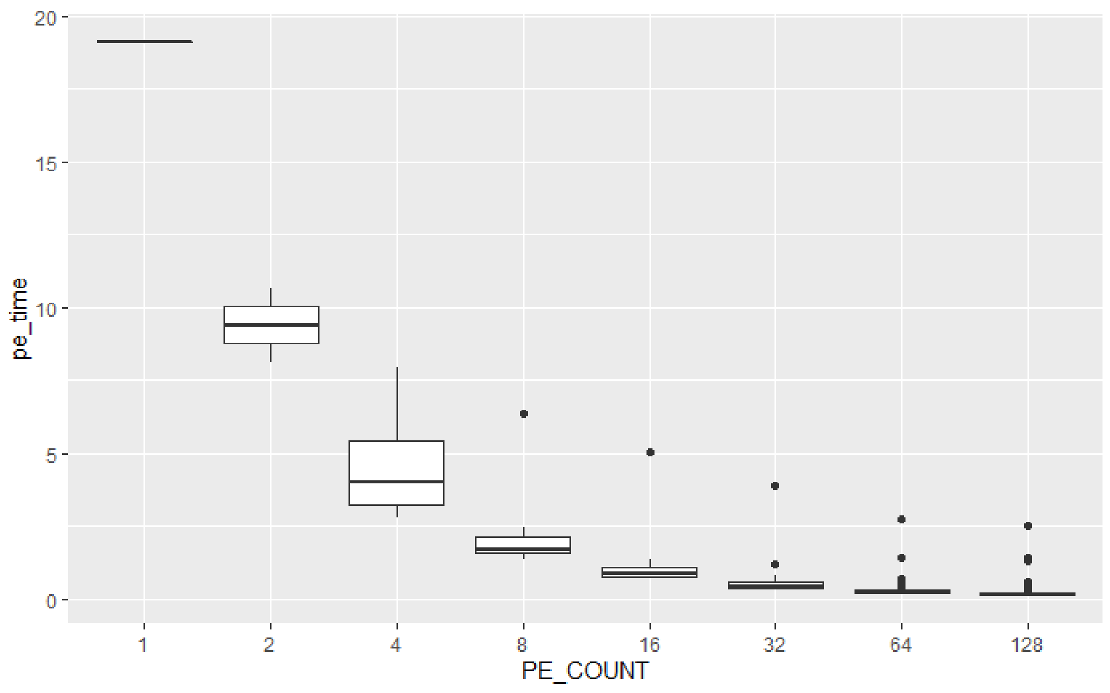


Figure 6.16: Box plot showing the load imbalance between different PEs in the PL-communities model. The data for this plot is from parameter-set 1 of the strong scaling tests with seed 1.

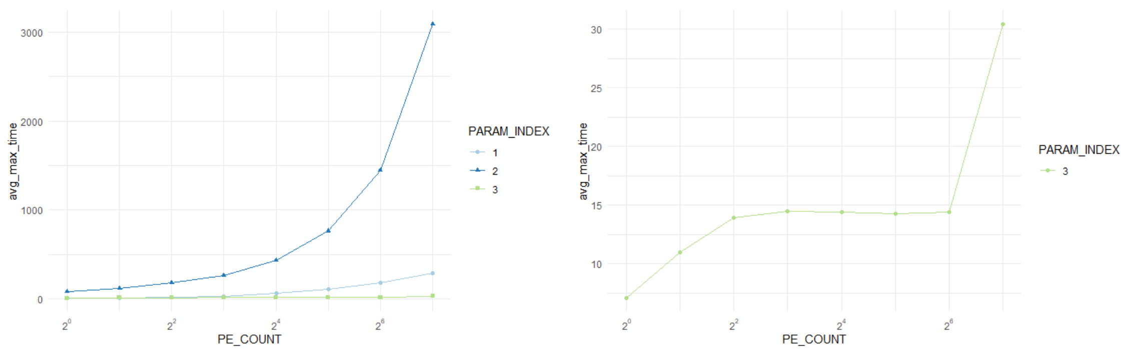


Figure 6.17: Plots visualizing the weak scaling of the PL-communities model. It plots total runtime in seconds against PE count. The right plot is a zoom in on the result of parameter-set 3.

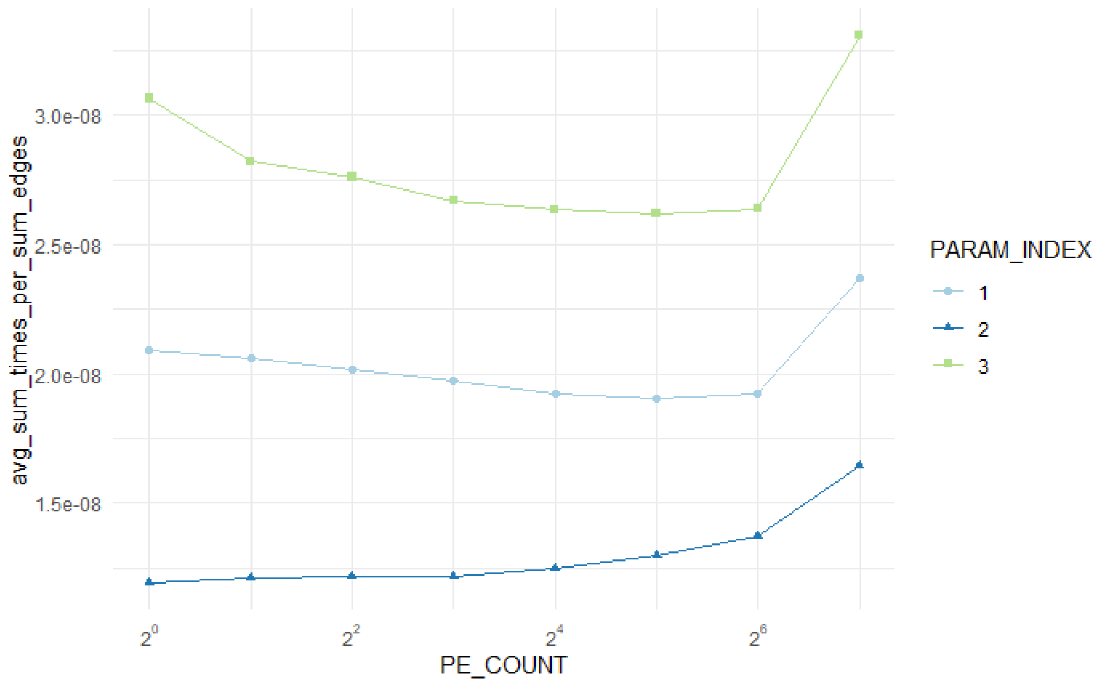


Figure 6.18: Plots visualizing the combined runtime of all PEs divided by the total number of edges generated by all PEs for the weak scaling tests of the PL-communities model.

6.2.5 PL-Multicommunity Model

Strong Scaling

The first parameter set for strong scaling tests of the PL-multicommunity model is a normal setting. Community sizes are at least 6 and at most one tenth of the vertex count. The communities' power law distribution has exponent 2.5 as per default in the CKB model. The parameters setting the intra-community edge probability are also set to default values. The inter-community edge probability is set to produce 2 inter-community edges per vertex in expectation. Each vertex is set to be in three communities on average.
The second parameter set produces graphs with large communities. Minimum community size is set to 2000 and the power-law distribution to sample community sizes has exponent 1. Other parameters are similar to the first parameter set.
The third parameter set tests performance for lower density. The maximum community size is set relatively low at 20000. Intra community degree of vertices is set lower and scales slower with community size than in the other parameter sets. The expected membership count for each vertex is higher than for the other tests.
The parameters used are listed in Table 6.9.

The speedup for the PL-multicommunity model is visualised in Figure 6.19. Absolute runtimes and combined runtimes per total number of edges generated is shown in Figure 6.20. Similar to the PL-communities model without overlapping communities, the speedup is not optimal. The best scaling parameter sets have already lost factor two when working with 64 PEs. The runtimes per edge generated are basically constant. The increase in runtime when using all PEs can be observed in all models and is unlikely to indicate bad scaling of any model.
Models without overlapping communities do not generate blocks of edges twice. But the distribution of the work among the PEs is the same as for the PL-communities model. So the issue that PEs processing members of large communities have to generate more

Parameter	Set 1	Set 2	Set 3
n	2^{24}	2^{24}	2^{24}
c_{min}	6	2000	6
c_{max}	$1.6 \cdot 10^6$	$1.6 \cdot 10^6$	20000
vertex_mc	3	3	5
k_{comm}	2.5	1	2.5
α	4	4	3
γ	0.5	0.5	0.666
p_o	$1.2 \cdot 10^{-7}$	$1.2 \cdot 10^{-7}$	$1.2 \cdot 10^{-7}$

Table 6.9: Parameter sets used to test strong scaling for the PL-multicommunity model.

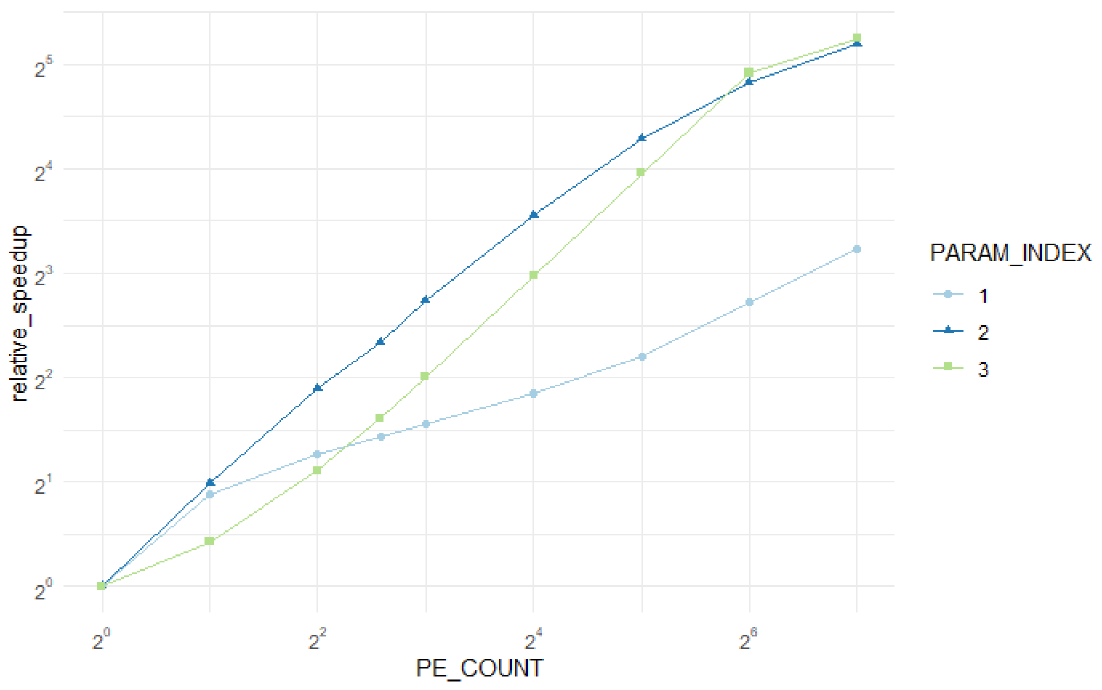


Figure 6.19: Plot visualizing the strong scaling speedup of the PL-multicommunity model.

edges also occurs here and slows down the runtime as those PEs take notably more time than PEs generating edges for members of smaller communities. The parameter sets with reduced range of permitted community sizes are affected less by this, thus parameter set 1 scales the worst of the tested ones.
Compared to the not overlapping PL-communities, scaling is very similar. Absolute runtimes per edge are also comparable. The main reason why the PL-multicommunity tests took longer is that vertices are located in multiple communities and are having intra-community edges generated for all those communities. So for a fair comparison in runtime, the membership count of the vertices should be handled as a multiplier to the vertex count. Then absolute runtimes are also similar for the overlapping and the not overlapping model.

Weak Scaling

Weak scaling is tested with similar parameter-sets as strong scaling.
The first parameter set is a normal setting. Maximum community size scales to always be roughly one tenth of the vertex count. The inter-community edge probability also scales with the vertex count to produce an expected inter-community degree around 2 for each vertex.

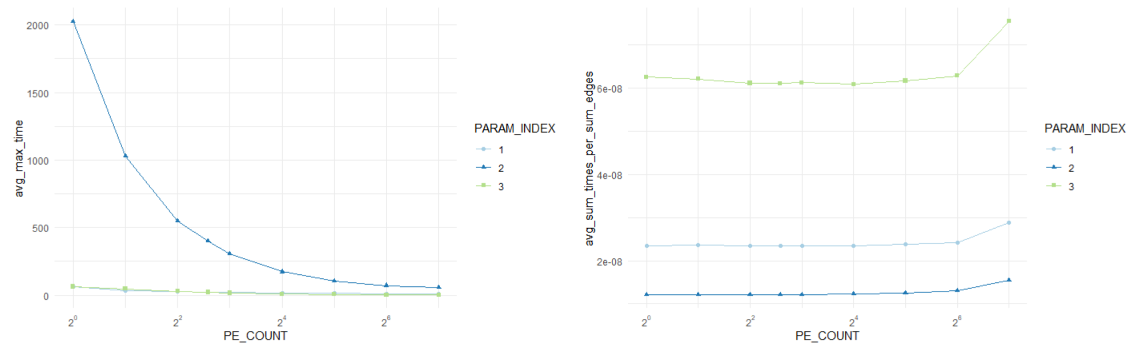


Figure 6.20: Runtime plots for the strong scaling tests of the PL-multicommunity model. The left plot shows the absolute runtime, the right plot shows the combined runtime of all PEs divided by the total number of edges generated by all PEs.

The second parameter set has large communities. The minimum community size is set to 2000 and the power law samples community sizes with a less extreme exponent. Maximum community size and inter-edge probability scale just as in parameter-set 1.
The third parameter set produces graphs of a lower density. Maximum community size is fixed at 20000. The intra-community edge probability is lower and parametrized to grow more slowly with increased community size. Inter-edge probability scales the same as in the other parameter sets. There are 5 communities per vertex for this parameter set rather than the 3 of the previous parameter sets.
The parameters are listed in Table 6.10. For scaling parameters, the values for the sequential run and the run with 128 PEs are given, the values of p_o decrease with increasing vertex count. They scale linearly with the vertex count of the generated graph.

Parameter	**Set** 1	**Set** 2	**Set** 3
n	$2^{21} - 2^{28}$	$2^{21} - 2^{28}$	$2^{21} - 2^{28}$
c_{min}	6	2000	6
c_{max}	$1.0 \cdot 10^5 - 1.3 \cdot 10^7$	$1.0 \cdot 10^5 - 1.3 \cdot 10^7$	20000
vertex_mc	3	3	5
k_{comm}	2.5	1	2.5
α	4	4	3
γ	0.5	0.5	0.666
p_o	$1.9 \cdot 10^{-6} - 1.5 \cdot 10^{-8}$	$1.9 \cdot 10^{-6} - 1.5 \cdot 10^{-8}$	$1.9 \cdot 10^{-6} - 1.5 \cdot 10^{-8}$

Table 6.10: Parameter sets used to test weak scaling for the PL-multicommunity model.

As with strong scaling, the weak scaling of the PL-multicommunity model is not perfect. Plots of the runtimes can be seen in Figure 6.21. The accumulated runtime of all PEs per generated edge is basically constant. The increase in runtime for using 128 PEs has been observed for all models and parameter sets and does not seem to hint at bad scaling.
Similar to strong scaling, the reason for suboptimal scaling of absolute runtimes is the poor balancing of the workload across the PEs used. PEs generating edges for vertices in small communities have to generate less edges than those working with vertices in larger communities.

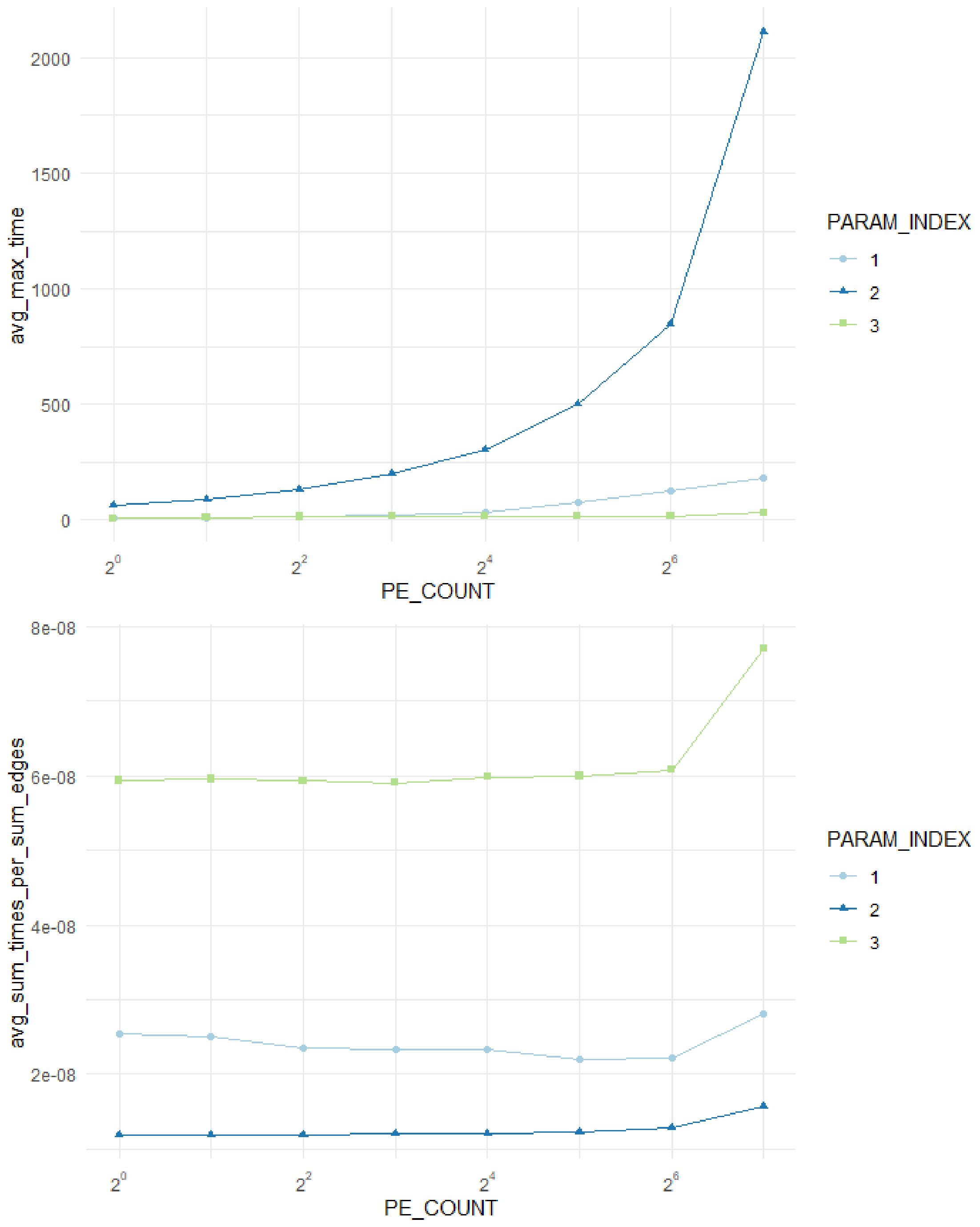


Figure 6.21: Plots visualizing the weak scaling of the PL-multicommunity model. The top plot shows absolute runtimes. Below that, the accumulated runtime across all PEs is plotted, divided by the number of edges generated.

6.2.6 CKB Graphs

Strong Scaling

The first parameter set testing the strong scaling of the communication-free implementation of the CKB model is a normal setting. Community sizes go from 6 to one tenth of the vertex count. Membership counts can be between 1 and one tenth of the vertex count. The exponent of the power law distributions to sample community sizes and membership counts are both set to 2.5, which is the default value given in the original CKB model [CKB+14]. The parameters to calculate intra-community edge probabilities are also set to their default values. The inter-community edge probability is set to produce an expected inter-community degree of 2 for each vertex. Pre-assignment is used to guarantee that each vertex is in at least one community.
The second parameter set is almost exactly the same. It is the normal setting, but without using pre-assignment.
The third parameter set produces graphs with large communities. The communities' power law has a lower exponent and the minimum community size is set to 2000.
The fourth parameter set test performance for lower density. The maximum community size is set slightly lower than in the normal setting. The parameters governing intra-community edge probability are set to decrease the initial probability and to slow the rate at which it scales with community size. The exponent of the power law distribution sampling membership counts is set to 2. To account for the increase in average membership count caused by this, the vertex count of this parameter set is lowered.
The fifth parameter set uses the original CKB parameters. Notable differences to the normal setting are that the maximum community size and membership count are set comparably low at 10000. Also, the smallest permitted community size is 2, which means that many communities will just consist of a single edge.
Table 6.11 lists all parameters used for the strong scaling tests.

Parameter	**Set** 1	**Set** 2	**Set** 3	**Set** 4	**Set** 5
n	2^{22}	2^{22}	2^{22}	2^{20}	2^{22}
c_{min}	6	6	2000	6	2
c_{max}	$4.0 \cdot 10^5$	$4.0 \cdot 10^5$	$4.0 \cdot 10^5$	$1.0 \cdot 10^5$	10000
v_{min}	1	1	1	1	1
v_{max}	$4.0 \cdot 10^5$	$4.0 \cdot 10^5$	100	$1.0 \cdot 10^5$	10000
k_{comm}	2.5	2.5	1	2.5	2.5
k_{vertex}	2.5	2.5	2.5	2	2.5
α	4	4	4	3	4
γ	0.5	0.5	0.5	0.666	0.5
p_o	$4.8 \cdot 10^{-7}$	$4.8 \cdot 10^{-7}$	$4.8 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$4.8 \cdot 10^{-7}$
pre_assign	true	false	true	true	true

Table 6.11: Parameter sets used to test strong scaling for the CKB model.

The speedup for strong scaling of the CKB model is plotted in Table 6.22. Similar to the other models with power law distributed community sizes, the scaling is not perfect. Looking at the accumulated runtimes across all PEs normalized by the number of edges generated in Figure 6.23 on the right, it can be seen that the workload does not increase. Instead, the imperfect scaling is caused by suboptimal load balancing across the PEs.
As described when analysing the scaling of the PL-communities model, PEs handle edge generation for a fixed number of vertices, or – in models with overlapping communities – members in communities. Members of smaller communities have a lower degree than

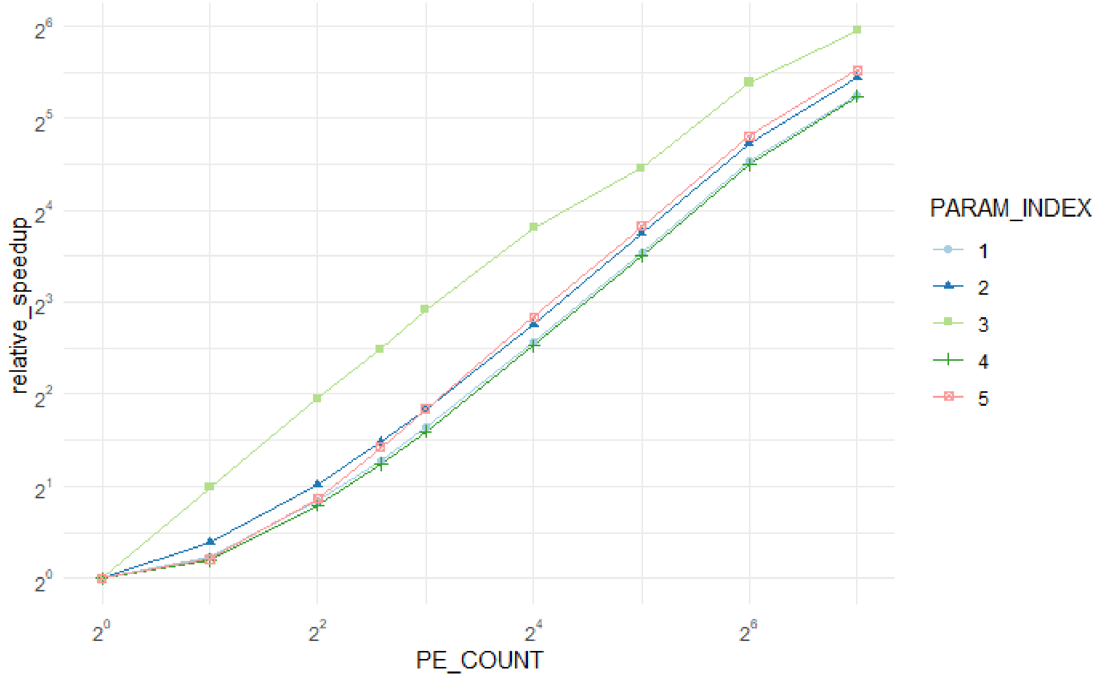


Figure 6.22: Plot visualizing the strong scaling speedup of the CKB model.

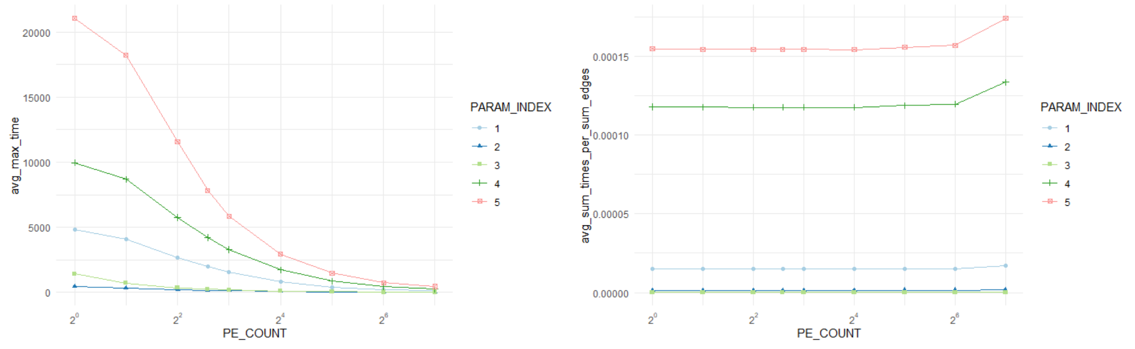
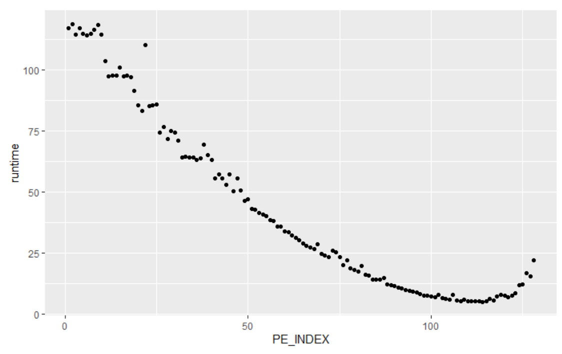


Figure 6.23: Runtime plots for the strong scaling tests of the CKB model. The left plot shows the absolute runtime, the right plot shows the combined runtime of all PEs divided by the total number of edges generated by all PEs.

members of large communities. So PEs handling members of large communities have to generate more edges. In addition, the assignment process has to be done per community partially handled by the PE. While there is some cropping during the assignment process that makes assignment for smaller communities less work intensive, some parts of assignment and especially pre-assignment have to be done per community. So PEs only processing very small communities have to do more work for the assignment process.
Depending on the parametrization, either PEs handling predominantly small or large communities will be slowest. For settings with a comparably low minimum community size, the runtime is dominated by the assignment process. So PEs processing members of small communities that have to do the assignment process for more communities tend to take the longest.
Figure 6.24 plots the runtimes of all PEs for the run of parameter sets 1 and 3 with seed 1 and 128 PEs. PEs with small index tend to process smaller communities. For communities larger than $\sqrt{n}$ this is not necessarily true, because communities are only sorted by size if they are stored by how often communities of that size were sampled.



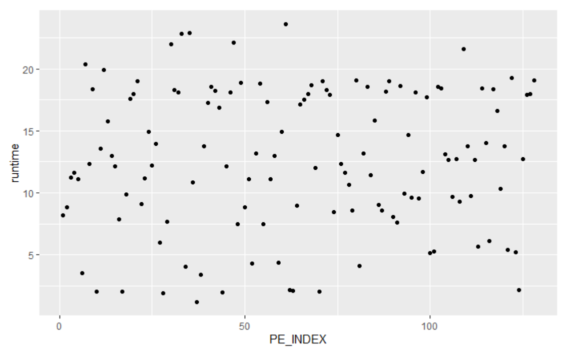


Figure 6.24: Plot visualizing runtime of each PE for parameter-set 1 with seed 1 and 128 PEs on the left and for parameter-set 3 with seed 1 and 128 PEs on the right.

For the normal setting of parameter set 1 it can be seen that the PEs processing the smallest communities take longest. The PEs generating edges for the largest communities also take longer than those working on a moderate number of medium-sized communities, but for this parametrization they are still a lot faster than the PEs handling many small communities.
The large community setting with parameter set 3 has too few medium-sized communities to produce a large imbalance in community densities for communities smaller than $\sqrt{n}$. While the balancing is obviously not perfect, there is no more clear tendency that could be fixed by simply assigning lower index PEs fewer members. So without knowing the exact sizes of the sampled communities, the balancing of parameter set 3 can not be improved. As such its scaling gives an idea of how all models with power law distributed community sizes should scale if load balancing would consider different densities and assignment costs of the communities.

Looking at the absolute runtimes plotted in Figure 6.23 the effect of certain parameters on the runtime can be estimated. The default CKB setting of parameter set 5 is slowest, because it generates by far the most communities due to the very small minimum community size. The large community setting of parameter set 3 runs the fastest of all settings with pre-assignment. The lower density graphs generated with parameter set 4 do not run faster than the default setting. Due to the decreased exponent for the power law distribution of membership counts, the number of member community pairs is larger than in the default setting. Both these observations signal that for the normal settings with pre-assignment the assignment of members to communities dominates the workload.
The run without pre-assignment was fastest. Most notably it was roughly ten times faster than the run with the same parameters with pre-assignment. At least for settings with many small communities, pre-assignment requires a lot of additional work to lessen the distortion of the intended membership count distribution caused by the probabilistic assignment.

Weak Scaling

The parameter sets for weak scaling tests are similar to the ones used for strong scaling tests.
The first parameter set is a normal setting. The maximum community size scales with the vertex count to allow communities containing one tenth of all vertices. The maximum membership count scales the same way. The inter-community edge probability also scales with the vertex count to get an expected 2 inter-community edges per vertex.
The second parameter set tests performance in the normal setting without using pre-assignment.

Parameter	**Set** 1	**Set** 2	**Set** 3
n	$2^{18} - 2^{25}$	$2^{18} - 2^{25}$	$2^{18} - 2^{25}$
$c_{\min}$	6	6	2000
$c_{\max}$	$25000 - 3.2 \cdot 10^6$	$25000 - 3.2 \cdot 10^6$	$25000 - 3.2 \cdot 10^6$
$v_{\min}$	1	1	1
$v_{\max}$	$25000 - 3.2 \cdot 10^6$	$25000 - 3.2 \cdot 10^6$	$25000 - 3.2 \cdot 10^6$
k_{comm}	2.5	2.5	1
k_{vertex}	2.5	2.5	2.5
α	4	4	4
γ	0.5	0.5	0.5
p_o	$7.6 \cdot 10^{-6} - 6.0 \cdot 10^{-8}$	$7.6 \cdot 10^{-6} - 6.0 \cdot 10^{-8}$	$7.6 \cdot 10^{-6} - 6.0 \cdot 10^{-8}$
pre_assign	true	false	true

Parameter	**Set** 4	**Set** 5
n	$2^{16} - 2^{23}$	$2^{15} - 2^{22}$
$c_{\min}$	6	2
$c_{\max}$	$1.0 \cdot 10^5$	10000
$v_{\min}$	1	1
$v_{\max}$	$1.0 \cdot 10^5$	10000
k_{comm}	2.5	2.5
k_{vertex}	2	2.5
α	3	4
γ	0.666	0.5
p_o	$7.6 \cdot 10^{-6} - 6.0 \cdot 10^{-8}$	$7.6 \cdot 10^{-6} - 6.0 \cdot 10^{-8}$
pre_assign	true	true

Table 6.12: Parameter sets used to test weak scaling for the CKB model.

The third parameter set generates graphs with large communities. Maximum membership count and community size as well as inter-community edge probability scale with the vertex count. The minimum community size is set to 2000 and community sizes are sampled from a power law distribution with exponent 1.
The fourth parameter set is the lower density setting. Only the inter-community edge probability scales with the vertex count. Maximum membership count and community size are set to 100000. The parameters controlling the intra-community edge probabilities are set to start at a lower probability and decrease the rate at which the probabilities grow with increasing community size compared to the normal setting.
The fifth parameter set is the setting given in the original CKB paper [CKB+14]. Only the inter-community edge probability scales to produce around 2 inter-community edges per vertex. The other parameters are fixed with a very low minimum community size of 2 and relatively small maximum community size and membership count.
A list of the exact parameters used is shown in Table 6.12. All scaling parameters scale linearly with the vertex count. For scaling parameters the values for the sequential run and the run with all 128 PEs is given, values for p_o decrease with increasing vertex count.

The runtimes of the weak scaling tests are plotted in Figure 6.25. Scaling is not perfect, as the runtime increases significantly with the use of more PEs. The accumulated workload of all PEs divided by the number of generated edges is shown in figure 6.26. Unlike with the other models, these plots do not show that scaling would be nearly perfect with better balancing of the workload. Except for parameter set 3 the average workload per generated edge increases with the number of PEs used. While the effects of bad workload balancing

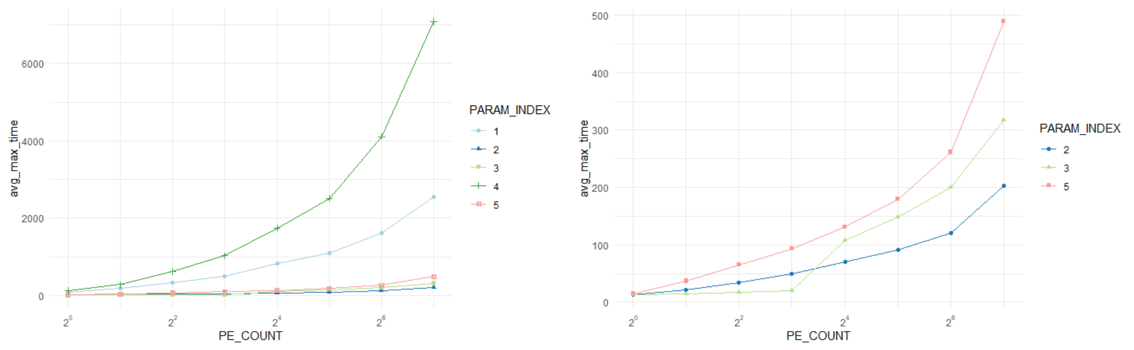
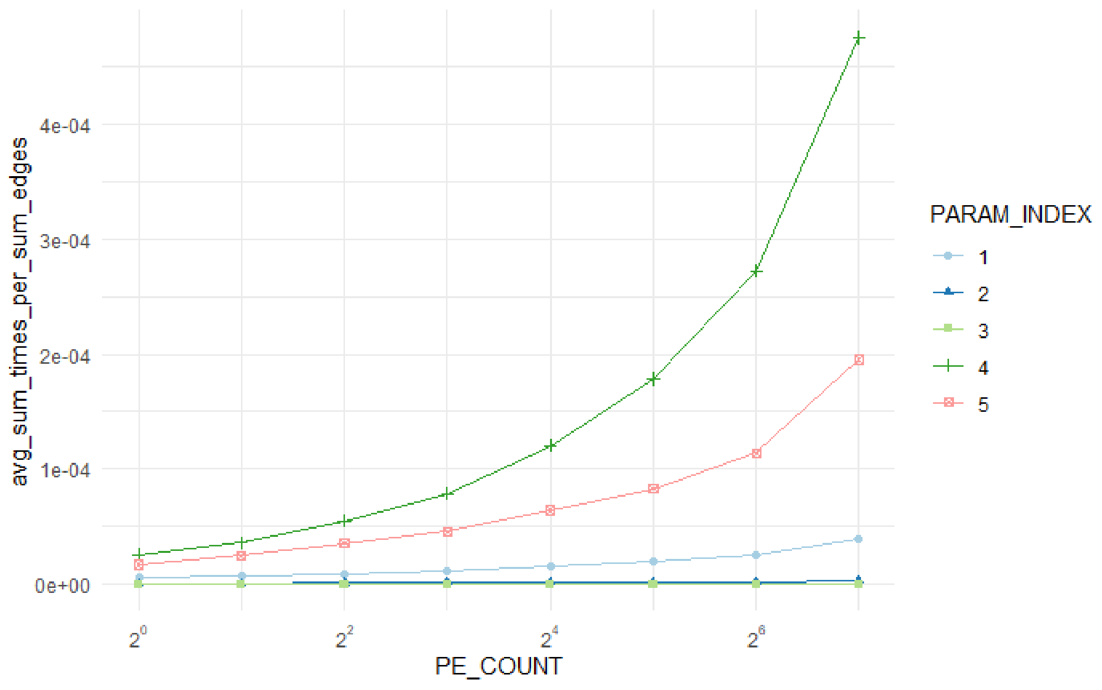


Figure 6.25: Plots visualizing the weak scaling of the CKB model. The right plot is a zoomed-in version of the left plot.

are still present in the CKB model as shown during the strong scaling tests, there are additional factors negatively impacting scaling here.
During the strong scaling tests it was discussed, that parameter sets with small minimum community size spend most time assigning vertices to communities and not generating edges. Since the assignment encompasses working through the compact data structure storing membership counts, it is expected that the runtime of the assignment scales with the size of that data structure. The size of the data structure storing membership counts is in $\mathcal{O}(\sqrt{n})$. So for settings where most of the work done is related to the assignment process, the runtime should also scale in $\mathcal{O}(\sqrt{n})$. This is around the rate at which the workload per edge grows. So the implementation does achieve the expected scaling, but that scaling is not optimal.
For parameter set 3 the workload is not dominated by the assignment process. The graphs generated there are made up of large communities, which drastically reduces the number of communities and the number of times that the assignment has to be run. The workload per edge slightly decreases with increased PE count as seen in Figure 6.26. So for graphs where the generating process is not dominated by the vertex assignment, good scaling could be achieved with better load balancing.
There is a very notable anomaly in the lower plot of Figure 6.26. When using 16 PEs the workload per edge more than triples compared to using 8 PEs. Parameter-set 3 has set its minimum community size to 2000. The data structure storing power law sampled data differentiates between samples smaller and larger than $\sqrt{n \cdot \mathrm{exp\_mc}}$. For the graphs generated with 8 or less PEs, no community sizes smaller than that threshold exist. Using 16 or more PEs places most community sizes below that threshold. So the additional complexity of handling cases for both halves of the community size storing data structure causes this jump in total workload.



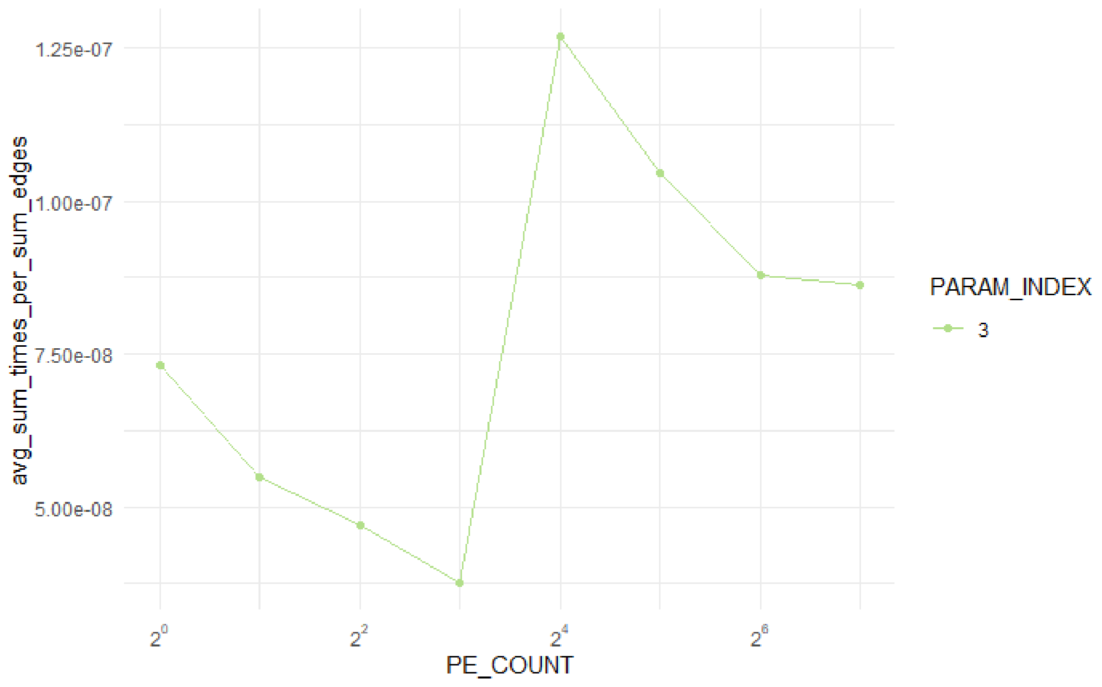


Figure 6.26: Plots visualizing the weak scaling of the CKB model plotting the accumulated runtime of all PEs divided by the number of generated edges. The lower plot is a zoomed-in version of the upper plot. It shows an increase in workload from the point where PL-samples both smaller and larger than $\sqrt{n}$ exist.

7. Conclusion

This chapter summarizes the achieved results and mentions potential improvements hinted at but not implemented or tested in this thesis.

7.1 Summary

In this thesis five communication-free generators for graphs with community structure were developed, implemented and tested. The generators cover models from a fixed number of equally sized communities over power law sampled communities to a communication free generator for CKB graphs. The complexity of the models was increased gradually, allowing the usage of techniques developed for previous generators.
The generators expand the KaGen library [FLS+18]. The generator for $G(n,p)$ graphs introduced there was used as a foundation to build the basic models. The added structure of communities resulted in a need for routines to split the adjacency matrix of the graph into homogeneous blocks. With that, the techniques of the $G(n,p)$ generator could be used to generate graphs with communities.
For sampling from a power law distribution two different approaches were considered. With sqrt-based sampling, an algorithm was developed to sample from a power law distribution to reach a fixed combined size of all samples. It stores results in a compressed format which reduces space and time demands of the sampling process and of processes working on the sampled data. This allowed sampling community sizes from a power law distribution and creating the PL-communities model.
To move to models with overlapping communities, two different ways of assigning vertices to communities were looked at. While group based assignment promises fast runtimes while guaranteeing exact membership counts and community sizes, it creates unwanted structure in the way that communities can overlap. So the probabilistic assignment was used to assign members to communities in models with overlapping communities. The first of these models features power law distributed community sizes and a fixed number of intended communities per vertex.
For the communication-free CKB generator, the group based assignment process was generalised. Negative effects of the probabilistic assignment routine were reduced by the optional tool of pre-assignment. It allows ensuring that each vertex is a member of at least one community. This makes isolated vertices a lot less likely and therefore allows generating graphs with better connectivity. Combining the techniques introduced resulted in a communication-free generator for CKB graphs.

The generated graphs were tested. For the metrics checked, all models produced graphs whose properties were as anticipated. This verifies the generators and their implementation to a certain extent.
Testing the performance of the implemented generators consisted of weak and strong scaling tests for all models with different parameter-sets. The basic models were found to scale basically perfectly.
For models with power law distributed community sizes, imperfect load balancing between PEs caused a decrease in speedup for increased degrees of parallelization. However, the workload per PE stayed mostly constant, suggesting that working with an improved workload distribution could allow very good scaling for those generators.
The runtimes of most runs of the CKB generator were dominated by the membership assignment process. So in addition to imperfect workload balancing the workload per PE was found to increase roughly with the size of the data structures storing sampled community sizes and membership counts. So even with better workload balancing the workload for the CKB generator using probabilistic assignment with pre-assignment grows with about $\mathcal{O}(\sqrt{n})$.

7.2 Outlook

Since the scaling of the more complex models is negatively impacted by uneven workload distribution, a good next step would be to look into better ways to assign tasks to the PEs. The scaling test conducted here should provide a basis for which parameters impact the runtime of each PE, thus allowing a better balancing of runtimes depending on the parameters of the run. Such a re-distribution of workloads would have to be done for each affected model individually or at least separately for the PL-communities models and the CKB model, since tests suggest that different PEs are responsible for the slowdown in these models.
The group based assignment process was not implemented and therefore not tested. Future research may want to look into this, especially since it may provide better scaling for the CKB model than the probabilistic assignment with pre-assignment which was used for the tested implementation. Apart from implementing a version of the generators with group based assignment and evaluating their scaling, the community structure created should be analysed. It seems like the effects to the overlap of different communities caused by group based assignment should be notable and especially for large communities create unwanted structures in community overlap. But testing would provide more knowledge on the actual effects and may find scenarios in which these effects are more acceptable than the slight distortion to the membership counts' power law distribution produced by using probabilistic assignment with pre-assignment.

Bibliography

[ALPH01] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Phys. Rev. E*, 64:046135, Sep 2001.

[Car15] Corrie J Carstens. Proof of uniform sampling of binary matrices with fixed row sums and column sums for the fast curveball algorithm. *Physical Review E*, 91(4):042812, 2015.

[CK10] Szymon Chojnacki and Mieczysław Kłopotek. Random graph generator for bipartite networks modeling. *arXiv preprint arXiv:1010.5943*, 2010.

[CKB+14] Kyrylo Chykhradze, Anton Korshunov, Nazar Buzun, Roman Pastukhov, Nikolay Kuzyurin, Denis Turdakov, and Hangkyu Kim. Distributed generation of billion-node social graphs with overlapping community structure. In Pierluigi Contucci, Ronaldo Menezes, Andrea Omicini, and Julia Poncela-Casasnovas, editors, *Complex Networks V*, pages 199–208, Cham, 2014. Springer International Publishing.

[FLS+18] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[KPPS14] Tamara G Kolda, Ali Pinar, Todd Plantenga, and Comandur Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, 2014.

[Kra96] Manfred Krause. A simple proof of the gale-ryser theorem. *The American Mathematical Monthly*, 103(4):335–337, 1996.

[LF09] Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys. Rev. E*, 80:016118, Jul 2009.

[New05] MEJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[Pei15] Tiago P Peixoto. Model selection and hypothesis testing for large-scale network models with overlapping groups. *Physical Review X*, 5(1):011033, 2015.

[RG12] Bradley S Rees and Keith B Gallagher. Overlapping community detection using a community optimized graph swarm. *Social Network Analysis and Mining*, 2(4):405–417, 2012.

[SLHS+18] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient parallel random sampling—vectorized, cache-efficient, and online. *ACM Trans. Math. Softw.*, 44(3), January 2018.

[WCL$^+$16] Xuyun Wen, Wei-Neng Chen, Ying Lin, Tianlong Gu, Huaxiang Zhang, Yun Li, Yilong Yin, and Jun Zhang. A maximal clique based multiobjective evolutionary algorithm for overlapping community detection. *IEEE Transactions on Evolutionary Computation*, 21(3):363–377, 2016.

[XYLZ14] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1673–1681. Curran Associates, Inc., 2014.