Parallel and I/O-Efficient Algorithms for Non-Linear Preferential Attachment

Daniel Allendorf ⊠ Goethe University Frankfurt, Germany

Ulrich Meyer \square Goethe University Frankfurt, Germany Manuel Penschuck \square

Goethe University Frankfurt, Germany

Hung Tran \square Goethe University Frankfurt, Germany

— Abstract -

Preferential attachment lies at the heart of many network models aiming to replicate features of real world networks. To simulate the attachment process, conduct statistical tests, or obtain input data for benchmarks, efficient algorithms are required that are capable of generating large graphs according to these models.

Existing graph generators are optimized for the most simple model, where new nodes that arrive in the network are connected to earlier nodes with a probability $P(h) \propto d$ that depends linearly on the degree d of the earlier node h. Yet, some networks are better explained by a more general attachment probability $P(h) \propto f(d)$ for some function $f: \mathbb{N} \to \mathbb{R}$. Here, the polynomial case $f(d) = d^{\alpha}$ where $\alpha \in \mathbb{R}_{>0}$ is of particular interest.

In this paper, we present efficient algorithms that generate graphs according to the more general models. We first design a simple yet optimal sequential algorithm for the polynomial model. We then parallelize the algorithm by identifying batches of independent samples and obtain a near-optimal speedup when adding many nodes. In addition, we present an I/O-efficient algorithm that can even be used for the fully general model. To showcase the efficiency and scalability of our algorithms, we conduct an experimental study and compare their performance to existing solutions.

2012 ACM Subject Classification Mathematics of computing \rightarrow Random graphs

Keywords and phrases Random Graphs, Graph Generator, Preferential Attachment

Supplementary Material The internal memory algorithms are maintained at https://github.com/massive-graphs/nonlinear-preferential-attachment. The implementation of the dynamic weighted sampling data structure of [29] is independently maintained as the Rust crate https://crates.io/crates/dynamic-weighted-index. The external memory algorithms are available at https://github.com/massive-graphs/extmem-nlpa. An archive containing the frozen source code of all experiments and most of the raw data collected can be found on https://zenodo.org/record/7318118.

Funding This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/5-1 (FOR 2975 — Algorithms, Dynamics, and Information Flow in Networks

1 Introduction

Networks govern almost every aspect of our modern life ranging from natural processes in our ecosystem, over urban infrastructure, to communication systems. As such, network models have been studied by many scientific communities and many resort to random graphs as the mathematical tool to capture them [12, 4, 8].

A well-known property of observed networks from various domains is scale-freeness, typically associated with a powerlaw degree distribution. Barabási and Albert [9] proposed a

simple random graph model to show that growth and a sampling bias known as *preferential* attachment produce such degree distributions.

In fact, a plethora of random graph models rely on preferential attachment (e.g., [17, 13, 26, 24, 22, 19, 27]). At heart, they grow a graph instance, by iteratively introducing new nodes which are connected to existing nodes, so-called *hosts*. Then, preferential attachment describes a positive feedback loop on, for instance, the host's degree; nodes with higher degrees are favored as hosts and their selection, in turn, increases their odds of being sampled again.

Most aforementioned models use *linear* preferential attachment and, consequently, sampling algorithms are optimized for this special case [10, 30, 34, 2, 7, 3] (see also [31] for a recent survey). Some observed networks however, can be better explained with *polynomial* preferential attachment [27].

In this paper, we focus on efficient algorithmic techniques of sampling hosts for non-linear preferential attachment. Hence, we assume the following random graph model but note that it is straight-forward to adopt our algorithms to most established variants¹.

▶ **Definition 1** (Preferential attachment). We start with an arbitrary so-called seed graph G_0 with n_0 nodes and m_0 edges. We then iteratively add N new nodes $v_{n_0+1}, \ldots, v_{n_0+N}$ and connect each to $\ell \leq n_0$ different hosts. The resulting graph G_N has $n = n_0 + N$ nodes and $m = m_0 + N\ell$ edges.

The probability to select a node h with degree d as host is governed by $P(h) \propto f(d)$. In the case where $f(d) = d^{\alpha}$ for some constant $\alpha \in \mathbb{R}_{>0}$, we speak of polynomial preferential attachment, and if $\alpha = 1$, of linear preferential attachment.

Related Work

A wide variety of algorithms have been proposed for the linear case. Brandes and Batagelj gave the first algorithm to run in time linear in the size of the generated graph [10]. Their algorithm exploits a special structure of linear preferential attachment: given a list of the edges in the graph, we may simply select an edge $e = \{u, v\}$ uniformly at random and then toss a fair coin to decide on either u or v as host. [34] gave a communication-free distributed parallelization of the algorithm, and [30] proposed a parallel I/O-efficient variant for use in external memory. A different shared memory parallel algorithm was also proposed by [7], and [2, 3] gave algorithms for distributed memory.

Note that we may simulate preferential attachment by using a dynamic data structure that allows fast sampling from a discrete distribution. While rather complex solutions for this more general problem exist (e.g. see [32, 20, 29]), none provide guarantees on their performance in external memory, and we are not aware of any parallelization that is practical for our use case.

Our contribution

We first present a sequential algorithm for internal memory (see Section 2). Our algorithm runs in expected time linear in the size of the generated graph and can be used for any case $P(h) \propto d^{\alpha}$ where $\alpha \in \mathbb{R}_{\geq 0}$. It builds on a simple data structure for drawing samples from a dynamic distribution that is especially suited to the structure of updates in preferential attachment but may be of independent interest.

 $^{^{1}}$ In particular, we forbid loops and multi-edges, e.g. our algorithms generate *simple* graphs.

In Section 3, we show how to identify batches of independent samples for the polynomial model. As independent samples can be drawn concurrently, we may use this technique to parallelize a sequential algorithm for the model. Consequently, we apply this technique to our sequential algorithm and obtain a shared memory parallel algorithm with an expected runtime of $\mathcal{O}((\sqrt{N} + N/\mathcal{P}) \log \mathcal{P})$.

For the external memory setting (Section 4), we extend the algorithm of [30] by using a two-step process: in the first phase, we sample the degrees of the hosts of each node, and in the second phase, we sample the hosts of the nodes. Our algorithm requires $\mathcal{O}(\operatorname{sort}(n_0 + m))$ I/Os and even applies to the most general case where $P(h) \propto f(d)$ for any $f \colon \mathbb{N} \to \mathbb{R}$.

In an empirical study (Section 6), we demonstrate the efficiency and scalability of our algorithms. We find that our sequential algorithms incur little slowdown over existing solutions for the easier linear case. In addition, our parallel algorithm obtains speed-ups over the sequential algorithm of 32 to 46 using 63 processors.

Preliminaries and notation

Given a graph G = (V, E) and a node $v \in V$, define the *degree* $d_G(v) = |\{u : \{u, v\} \in E\}|$ as the number of edges incident to node v, and let $\Delta_G = \max_{v \in V} d_G(v)$ denote the maximum degree in G. For a sequence of graphs G_1, \ldots, G_N , we also write $d_{G_i}(v)$ as $d_i(v)$ and Δ_{G_i} as Δ_i or drop the subscript if clear by context.

We analyze our parallel algorithm using the CREW-PRAM model (see [23]) on \mathcal{P} processors (PUs). This machine model allows concurrent reads of the same memory address in constant time, but disallows parallel writes to this same address. Potentially conflicting writes to the same addresses can, however, be simulated in time $\mathcal{O}(\log \mathcal{P})$ per access [23]. We use two methods of conflict resolution, namely minimum (storing the smallest value written to an address) and summation (storing the sum of values); both are practical, since they are either directly supported by modern parallel computers or can be efficiently simulated.

For the analysis of our external memory algorithms we use the commonly accepted I/O-model of Aggarwal and Vitter [1]. The model features a memory hierarchy consisting of two layers, namely the fast internal memory holding up to M items, and a slow disk of unbounded size. Data between the layers is transferred using so-called I/Os, where each I/O transfers a block of B consecutive items. Performance is measured in the number of I/Os it requires. Common tasks of many algorithms include: (i) scanning n consecutive items requires scan $(n) := \Theta(n/B)$ I/Os, (ii) sorting n consecutive items requires sort $(n) := \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os, and (iii) pushing and popping n items into an external priority queue requires $\mathcal{O}(\operatorname{sort}(n))$ I/Os [6].

2 Sequential Algorithm

In this section, we describe a sequential internal memory algorithm for polynomial preferential attachment, i.e. the probability to select node h as host is $P(h) \propto d(h)^{\alpha}$ for a constant $\alpha \in \mathbb{R}_{\geq 0}$. A parallelization is discussed in Section 3. To simplify the description, we assume that new nodes only select one host $(\ell = 1)$; generalization is straight-forward by drawing multiple hosts per node and rejecting duplicates.

We also note that the algorithm can in principle be used for any function f(d) that is non-decreasing. However, analyzing the runtime for other f(d) is not trivial. In particular, Lemma 2 requires the pre-asymptotic degree distribution of the generated graphs to have certain properties, but the exact distribution is not known even in the polynomial case.

2.1 Sampling Method

To simulate preferential attachment, we require an efficient sampling method which selects a host in each step. Depending on the seed graph and parameters chosen, the host distribution P(h) may also undergo significant changes as more and more nodes are added to the graph². Therefore, the method used should be capable of adapting to these changes in order to be efficient for adding any number of nodes to a graph.

Our sampling method builds on rejection sampling, a general technique for sampling from a target distribution by using an easier proposal distribution. To showcase rejection sampling, we consider a simple but suboptimal scheme for polynomial preferential attachment. Let the graph be given as an array of edges $E = [e_1, \ldots, e_m]$ in arbitrary order. Now, sample an edge $e_i = \{u, v\}$ uniformly at random, then, randomly choose either u or v using a fair coin. Based on the observation that node h with degree d(h) appears d(h) times in E, we propose h with probability d(h)/2m. Now, if $\alpha < 1$, accept h with probability $(1/d(h))^{1-\alpha}$, or, if $\alpha \ge 1$, accept h with probability $(d(h)/\Delta)^{\alpha-1}$, otherwise, restart with a new proposal.

Observe that this scheme implements rejection sampling with the host distribution of linear preferential attachment as proposal distribution. It can be shown that this scheme provides samples in constant time if $n_0 = \mathcal{O}(1)$ as $n \to \infty$ by using known properties of the asymptotic degree distributions for $\alpha < 1$ and $\alpha \ge 1$. However, the scheme is inefficient if we wish to add only a few nodes to a larger seed graph. For instance, consider running the scheme for $\alpha = 2$ on a seed graph of one node with degree $\sqrt{n_0} - 1$ and $n_0 - 1$ nodes with degree 1. It is straightforward to check that this results in an initial acceptance probability of $\mathcal{O}(1/\sqrt{n_0})$. The issue is that the seed graph degree distribution differs significantly from the limit degree distribution, which causes the acceptance probability to be small until we have added many times over the initial number of nodes.

To remedy this issue, we combine rejection sampling with a dynamic proposal distribution that adapts to the target distribution. To this end, let $S = \{1, \ldots, n\}$ be a set of indices we wish to sample from and let $f: S \to \mathbb{R}_{\geq 0}$ be a function giving the weight of each element, i.e. the distribution on S is f(i)/W where $W = \sum_{j \in S} f(j)$ gives the proper normalization. The initial construction of our data structure is comparable to a variant of the alias method [14]. We first initialize an empty list or array³ P, and then for each element $i \in S$, add i exactly $c(i) = \lceil f(i)n/W \rceil$ times to P. We call c(i) the *count* of i. As P is used to maintain the proposal distribution, we refer to P as the *proposal list*⁴. Having constructed P, it is easy to verify that we may sample an element according to the target distribution by first selecting a uniform random element i from P, and then accepting with probability proportional to f(i)/c(i). If the distribution changes, we update P as follows: to add a new element i, we calculate c(i) as during the construction phase and add i exactly c(i) times to P. If the weight of an existing element i increases, we recalculate its count c(i) and add i to P accordingly.

2.2 SeqPolyPA

Our SeqPolyPA algorithm implements Section 2.1 as detailed in Algorithm 1. First initialize the proposal list P_0 by setting the count of each node v of the seed graph to the optimal

² We refer the interested reader to [25] for an analysis of the degree distributions of non-linear preferential attachment graphs.

³ An efficient implementation requires fast sampling from P and inserting into P; an array with table doubling (e.g. [16]) is an easy choice with constant expected/amortized time per operation.

 $^{^{4}}$ We may also think of P as a compression of the target distribution. Rejection sampling then serves the purpose of correcting any errors caused by the loss of information.

Algorithm 1 SeqPolyPA **Data:** Seed graph $G_0 = (V_0, E_0)$, with $n_0 = |V_0|$, $m_0 = |E_0|$, requested nodes N, parameter $\alpha > 0$ 1 $P_0 \leftarrow [v_1, \ldots,$ v_i, \ldots, v_i $, \ldots, v_{n_0}];$ $\left[d_0(v_i)^{\alpha} n_0 / W_0 \right]$ times **2** for $i \in [1, ..., N]$ do // Add new node repeat // Sample a host 3 Select candidate h uniformly from P_{i-1} ; 4 Accept with probability $w_{i-1}(h) / \max_{v \in V_{i-1}} w_{i-1}(v)$ where 5 $w_{i-1}(v) := d_{i-1}(v)^{\alpha}/c_{i-1}(v)$ and $c_{i-1}(v) := \#$ occurences of v in P_{i-1} ; **until** *accepted*; 6 $V_i \leftarrow V_{i-1} \cup \{v_{n_0+i}\} ;$ // Update graph 7 $E_i \leftarrow E_{i-1} \cup \{\{v_{n_0+i}, h\}\};$ 8 $P_i \leftarrow P_{i-1} + [v_{n_0+i}] ;$ // Update P 9 while $w_i(h) > W_i/n_i$ do 10 $P_i \leftarrow P_i + [h];$ 11 **12** Return $G_N = (V_N, E_N);$

count $c(v) = \lceil d_0(v)^{\alpha} n_0/W_0 \rceil$. Now, in each step *i*, a new node v_{n_0+i} arrives and has to be connected to the graph by linking it to some earlier node *h*. To this end, select a candidate *h* uniformly at random from P_{i-1} . Then, accept *h* with probability $w_{i-1}(h)/\max_{v \in V_{i-1}} w_{i-1}(v)$ where $w_{i-1}(v) = d_{i-1}(v)^{\alpha}/c_{i-1}(v)$, otherwise, reject *h* and restart with a new proposal. Once a host *h* has been accepted, add the new node v_{n_0+i} to V_{i-1} , and add the edge $\{v_{n_0+i}, h\}$ to E_{i-1} to obtain the new graph $G_i = (V_i, E_i)$. To reflect the new distribution induced by G_i , adjust *P* as follows: first, add v_{n_0+i} to P_{i-1} to obtain P_i , then, add *h* to P_i until $w_i(h) \leq W_i/n_i$.

In the following, we establish that SeqPolyPA produces the correct output distribution.

▶ **Theorem 1.** SeqPolyPA samples host $h \in V$ with probability $d(h)^{\alpha}/W$.

Proof. The probability p(h) that node h is accepted after a single proposal is

$$p(h) = \underbrace{\frac{c(h)}{|P|}}_{\text{propose } h} \underbrace{\frac{w(h)}{\max_{v \in V} w(v)}}_{\text{accept } h} = \frac{d(h)^{\alpha}}{|P| \max_{v \in V} w(v)}$$

Let $W' := |P| \max_{v \in V} w(v)$. Then, we have

$$W' = |P| \max_{v \in V} w(v) \ge \sum_{v \in V} c(v)w(v) = \sum_{v \in V} d(v)^{\alpha} = W.$$

With the remaining probability q = 1 - W/W', the first proposal is rejected, and another node is proposed. Thus, the overall probability of sampling node h is

$$p(h) + p(h)q + \dots = p(h)\sum_{k=0}^{\infty} q^k = p(h)\frac{1}{1-q} = \frac{d(h)^{\alpha}}{W}$$

as claimed.

Next, we show that SeqPolyPA runs in expected time linear in the size of the generated graph. We first analyze the memory usage.

<

▶ Lemma 2. Given a seed graph $G_0 = (V_0, E_0)$ with $n_0 = |V_0|$ nodes, SeqPolyPA adds N new nodes to G_0 using a proposal list P of expected size $\mathcal{O}(n_0 + N)$ if $\alpha \leq 1$ or $\alpha > 1$ and $n_0 = \mathcal{O}(1)^5$.

Proof. The initial size of P is at most

$$|P_0| = \sum_{v \in V_0} \left\lceil \frac{d_0(v)^{\alpha} n_0}{W_0} \right\rceil \le \sum_{v \in V_0} \left(1 + \frac{d_0(v)^{\alpha} n_0}{W_0} \right) = 2n_0.$$

Then, in each step i, the size of P increases by 1 for the new node and by the increase in the count of the host node h. The increase in the count is

$$\max\left\{0, \left\lceil \frac{d_i(h)^{\alpha} n_i}{W_i} \right\rceil - \left\lceil \frac{d_j(h)^{\alpha} n_j}{W_j} \right\rceil\right\}$$

where j < i is the last step in which node h was sampled.

Now, we distinguish two cases. For $\alpha \leq 1$, we have $d_i(h)^{\alpha} = (d_j(h) + 1)^{\alpha} \leq d_j(h)^{\alpha} + 1$, and using $n_i/W_i \leq 1$ and $W_i > W_j$, we obtain

$$\left\lceil \frac{d_i(h)^{\alpha} n_i}{W_i} \right\rceil - \left\lceil \frac{d_j(h)^{\alpha} n_j}{W_j} \right\rceil < 2 + \frac{d_j(h)^{\alpha}}{W_i} \left(n_i - n_j \right).$$

Now, observe that $d_j(h)^{\alpha}/W_i$ is the probability that h is sampled in step i, and as W is non-decreasing, the probability that h is sampled in each step between j and i is at least $d_j(h)^{\alpha}/W_i$. Thus, the expected length of a run until h is sampled is at most $W_i/d_j(h)^{\alpha}$, and as by definition, i - j is the length of this run, we have $E[i - j] = E[(n_0 + i) - (n_0 + j)] =$ $E[n_i - n_j] \leq W_i/d_j(h)^{\alpha}$. Therefore, the expected increase is only constant, which shows the claim.

The other case is $\alpha > 1$ and $n_0 = \mathcal{O}(1)$. In this case as a single node emerges which obtains almost all links and has expected degree $\Delta = \Theta(n)$ as $n \to \infty$ [25]. In each further step *i* we then have $W_i/n_i = \Omega(\Delta^{\alpha}/n_i) = \Omega(n_i^{\alpha-1})$ and the increase in the count of the chosen host node *h* is

$$\left\lceil \frac{d_i(h)^{\alpha} n_i}{W_i} \right\rceil - \left\lceil \frac{d_j(h)^{\alpha} n_j}{W_j} \right\rceil = \mathcal{O}\left(\frac{d_i(h)^{\alpha} - d_j(h)^{\alpha}}{n_i^{\alpha - 1}}\right)$$

In addition, we have $d_i(h) = d_j(h) + 1 < n_i + 1$, and it is straightforward to check that $(x+1)^{\alpha} = x^{\alpha} + \mathcal{O}(x^{\alpha-1})$ for any $x > \alpha > 1$, so it follows that

$$\frac{d_i(h)^{\alpha} - d_j(h)^{\alpha}}{n_i^{\alpha - 1}} < \frac{(n_i + 1)^{\alpha} - n_i^{\alpha}}{n_i^{\alpha - 1}} = \mathcal{O}(1) \,.$$

This implies that in each further step, the increase in |P| is only constant, and thus we have |P| = O(n).

We now show the main result of this section.

▶ Theorem 3. Given a seed graph $G_0 = (V_0, E_0)$ with $n_0 = |V_0|$ nodes, SeqPolyPA adds N new nodes to G_0 in expected time $\mathcal{O}(n_0 + N)$.

⁵ We remark that this is a correction over the conference version of the article [5] in which the condition $n_0 = \mathcal{O}(1)$ if $\alpha > 1$ was missing.

Proof. The initialization in line 1 of Algorithm 1 takes at most time $\mathcal{O}(|P_0|) = \mathcal{O}(n_0)$ (see Lemma 2). The outer loop in lines 2 - 13 terminates after N steps. The first inner loop in lines 3 - 6 terminates once a node is accepted. Reusing definitions of Thm. 1, the expected number of proposals until a node is accepted in step i is

$$E[T] = \frac{1}{1-q} = \frac{W'_{i-1}}{W_{i-1}} = \frac{|P_{i-1}| \max_{v \in V_{i-1}} w_{i-1}(v)}{W_{i-1}}.$$

Recall that a weight w(v) can increase only if node v is accepted, at which point, we add v to P until $w(v) \leq W/n$. Then we have $\max_{v \in V_{i-1}} w_{i-1}(v) \leq W_j/n_j$ where j < i is the last step in which the node with the maximum w(v) was accepted, and we obtain

$$E[T] \le \frac{W_j}{n_j} \frac{|P_{i-1}|}{W_{i-1}} = \mathcal{O}\left(\frac{W_j}{n_j} \frac{n_{i-1}}{W_{i-1}}\right)$$

where the last equality follows from the upper bound on |P| given by Lemma 2.

Now if $W_j/n_j \leq W_{i-1}/n_{i-1}$, then $E[T] = \mathcal{O}(1)$ as desired. The other case is $W_j/n_j \geq w(v) > W_{i-1}/n_{i-1}$. In this case, we can show that v is sampled again adjusting its weight before E[T] grows too large. Note that since $W_{i-1}/n_{i-1} < w(v) \leq W_j/n_j$, there has to be some step $j < k \leq i-1$ with $W_k/n_k < w(v) < W_{k-1}/n_{k-1}$. Since $w(v) = d(v)^{\alpha}/c(v) > W_k/n_k$ implies that $d(v)^{\alpha} > W_k/n_k$, the probability of sampling v in step k is at least $1/n_k$. In addition, since W/n has to decrease for E[T] to increase, we still have $w(v) > W_l/n_l$ in some step l > k with $n_l = Cn_k$ for some C > 1. This implies that the probability of sampling v in step l is at least $1/n_l = 1/Cn_k$. We now examine the expected number of times that v is sampled between step k and l and find

$$\frac{1}{n_k} + \frac{1}{n_{k+1}} + \dots + \frac{1}{Cn_k} = \mathbf{H}_{Cn_k} - \mathbf{H}_{n_{k-1}}$$

where \mathbf{H}_i denotes the *i*-th harmonic number. Using $\mathbf{H}_{Cn_k} - \mathbf{H}_{n_{k-1}} = \ln(C) + o(1)$, we find that for C = e node v is expected to be sampled once, and thus in expectation, E[T] cannot grow larger than

$$E[T] = \mathcal{O}\left(\frac{en_k}{n_k}\right) = \mathcal{O}(1)$$

For the second inner loop in lines 10 - 12, it suffices to use the bound on |P| given by Lemma 2. As the bound gives $|P_N| = \mathcal{O}(n_0 + N)$ after N steps, and each iteration of the loop increases the size of P by 1, the total time spent in this loop is $\mathcal{O}(n_0 + N)$.

3 Parallel Algorithm

In this section, we describe an efficient parallelization of the SeqPolyPA algorithm given in Section 2.2. The parallel algorithm ParPolyPA builds on the following observation.

▶ **Observation 1.** Let G_0, \ldots, G_N be a sequence of graphs generated with polynomial preferential attachment for some $\alpha \in \mathbb{R}_{>0}$ as per Definition 1, and let $W_i = \sum_{v \in V_i} d_i(v)^{\alpha}$ denote the sum of the node weights after step *i*. Then, we have

$$\frac{W_{i+1} - W_i}{W_{i+1}} = \begin{cases} \mathcal{O}(1/n_i) & \text{if } \alpha \le 1.\\ \mathcal{O}(1/\Delta_i) & \text{if } \alpha > 1. \end{cases}$$

Algorithm 2 ParPolyPA **Data:** Seed graph G = (V, E), with $n_0 = |V|$, $m_0 = |E|$, requested nodes $N, \alpha > 0$ 1 for $1 \le p \le \mathcal{P}$ do in parallel // Init P $P_p \leftarrow \begin{bmatrix} v_{\frac{(p-1)n_0}{\mathcal{P}}}, \dots, & \underbrace{v_i, \dots, v_i}_{\mathcal{P}} \end{bmatrix}, \dots, \underbrace{v_{pn_0}}_{\mathcal{P}} \end{bmatrix}$ $\mathbf{2}$ $\begin{bmatrix} \mathbf{d}_0(v_i)^{\alpha} n_0 / W_0 \end{bmatrix}$ **3** $s \leftarrow n_0;$ **4 while** s < N + 1 **do** // Start new batch $l \leftarrow N + 1;$ 5 for $1 \le p \le \mathcal{P}$ do in parallel 6 $H \leftarrow [];$ // Phase 1 $\mathbf{7}$ for $i \in [s + p, s + p + \mathcal{P}, \dots, l]$ do 8 with prob. $\frac{W'_i - W_s}{W'_i}$, $l \leftarrow \min\{i, l\}$; 9 Sample h with $P_1(h) = d_s(h)^{\alpha}/W_s$; 10 $H \leftarrow H + [(v_{n_0+i}, h, i)];$ 11 barrier; // Phase 2 12 for $(v, h, i) \in H$ where i < l do $\mathbf{13}$ $V \leftarrow V \cup \{v\}, E \leftarrow E \cup \{\{v, h\}\};$ $\mathbf{14}$ $P_p \leftarrow P_p + [v];$ 15while $w(h) > W_s/n_s$ do 16 $P_p \leftarrow P_p + [h];$ 17// Phase 3 barrier: 18 if p responsible then 19 with prob. $\frac{W_l - W_s}{W_l - W_s}$, Sample h with $P_2(h) = (d_l(h)^{\alpha} - d_s(h)^{\alpha})/(W_l - W_s)$ $\mathbf{20}$ else with $P_3(h) = d_l(h)^{\alpha}/W_l$; $V \leftarrow V \cup \{v_l\}, E \leftarrow E \cup \{\{v_l, h\}\};$ 21 $P_p \leftarrow P_p + [v_l];$ 22 while $w(h) > W_l/n_l$ do 23 $P_p \leftarrow P_p + [h];$ $\mathbf{24}$ $s \leftarrow l;$ $\mathbf{25}$ **26** Return G = (V, E);

Observation 1 suggests that a sample drawn in step i + 1 is independent from any changes to the distribution caused by step i with a rather large probability. Extending this principle to a batch of $\sqrt{n_i}$ or $\sqrt{\Delta_i}$ samples, we see that all samples in the batch are independent from changes to the distribution caused within the same batch with a non-vanishing probability. Thus, we may draw all samples in a batch independently in parallel until the first dependent sample, which gives an efficient parallelization for $\mathcal{P} = \Theta(\sqrt{n})$ processors if $\alpha \leq 1$, and $\mathcal{P} = \Theta(\sqrt{\Delta_N})$ if $\alpha > 1$.

3.1 ParPolyPA

We now describe the parallel algorithm in detail, see also Algorithm 2. First, let \mathcal{P} denote the number of PUs (processors), and let $1 \leq p \leq \mathcal{P}$ denote the *p*-th PU. Then, PU *p* adds all new nodes at positions $n_0 + k\mathcal{P} + p$ and all new edges at positions $m_0 + k\mathcal{P} + p$ where

 $0 \le k \le N/\mathcal{P}^6$. Before any samples are drawn, PU p receives an n_0/\mathcal{P} share of the nodes in the seed graph, and initializes a proposal list P_p for its nodes as described in Section 2.2.

Next, all PUs enter the sampling stage. Sampling is done in batches, where each batch ends if a dependent sample is found. We indicate this event by an atomic variable l that is initially set to $l \leftarrow N + 1$, but may be updated with decreasing indices of dependent samples during the batch. We also let s denote the index of the first sample in each batch. Each batch consists of three phases. Note that the phases are synchronized, e.g. no PU may proceed to the next phase until all PUs have finished the current phase.

In the first phase, each PU draws its samples independently and stores them in a list of its hosts for this batch, but does not yet add any nodes or edges to the graph. Before sample *i*, PU *p* flips a biased coin that comes up heads with probability W_s/W'_i where

$$W'_{i} = \begin{cases} W_{s} + 2(i-s) & \text{if } \alpha \leq 1\\ W_{s} + 2\left(\left(\Delta_{s} + i - s\right)^{\alpha} - \Delta_{s}^{\alpha}\right) & \text{if } \alpha > 1 \end{cases}$$

gives an upper bound on W_i . If the result is heads, then we know that the sample has to come from the old distribution, i.e. node h should be sampled with probability $P_1(h) = d_s(h)^{\alpha}/W_s$. To this end, the PU draws two uniform random indices $r \in \{1, \ldots, \mathcal{P}\}$ and $c \in \{1, \ldots, \mathcal{S}\}$ where $\mathcal{S} = \max_p |P_p|$, and requests the c-th entry of list P_r of PU r. Note that it is possible for this request to fail if $c > |P_r|$, and in this case, two new indices are sampled. Once a candidate node h is found, h is accepted or rejected as in the sequential case, and once a sample is accepted, it is added to the list of hosts. If however, the result is tails, then the sample may have to come from new distribution, so the batch has to end before drawing this sample. Therefore, the PU atomically checks variable l, and if i < l, updates the lower bound by setting $l \leftarrow i$. The PU terminates the first phase if $i \ge l$ or all nodes have been processed; otherwise it continues with its next node.

In the second phase, each PU adds its new nodes and sampled edges to the graph, and updates its list P by adding its nodes, and any increase in the counts of its hosts caused by its edges.

In the third phase, the PU that found the first dependent sample l draws this sample (if any). Recall that we overestimated the probability that the sample was dependent by using the upper bound W'_l . To correct for this, we first flip a coin that comes up heads with probability $(W_l - W_s)/(W'_l - W_s)$. If the result is heads, the responsible PU draws the sample only from the weight added in the batch, i.e. node h is sampled with probability $P_2(h) = (d_l(h)^{\alpha} - d_s(h)^{\alpha})/(W_l - W_s)$. To this end, it selects a candidate node from one of the lists P, but only considers positions added during the batch. Consequently, a candidate is accepted with probability proportional to the increase in its weight divided by the increase in its count. Otherwise, if the result is tails, the sample is drawn from the new distribution with probability $P_3(h) = d_l(h)^{\alpha}/W_l$. Once the PU sampled a host, it adds the node and edge to the graph and updates its list P. Then, all PUs enter the next batch, or exit, if all N samples have been drawn.

We now prove the correctness of the necessary modifications to obtain ParPolyPA from SeqPolyPA.

▶ **Theorem 4.** ParPolyPA samples host $h \in V$ with probability $d(h)^{\alpha}/W$.

Proof. We distinguish three cases to sample node h. Either (1) h is sampled as independent sample with probability $P_1(h) = d_s(h)^{\alpha}/W_s$, or (2) as dependent sample with probability

⁶ For simplicity, we assume that \mathcal{P} divides n_0 and N.

 $P_2(h) = (d_l(h)^{\alpha} - d_s(h)^{\alpha})/(W_l - W_s)$, or (3) as sample from the new distribution with probability $P_3(h) = d_l(h)^{\alpha}/W_l$. Thus, the overall probability of sampling h is given by

$$\begin{split} P(h) &= \frac{W_s}{W'_l} P_1(h) + \frac{W_l - W_s}{W'_l} P_2(h) + \frac{W'_l - W_l}{W'_l} P_3(h) \\ &= \frac{d_s(h)^{\alpha}}{W'_l} + \frac{d_l(h)^{\alpha} - d_s(h)^{\alpha}}{W'_l} + \frac{W'_l - W_l}{W'_l} \frac{d_l(h)^{\alpha}}{W_l} \\ &= \frac{d_l(h)^{\alpha}}{W'_l} + \frac{W'_l - W_l}{W'_l} \frac{d_l(h)^{\alpha}}{W_l} \\ &= \frac{d_l(h)^{\alpha}}{W_l}. \end{split}$$

It only remains to show that W'_l is an upper bound on W_l . We first consider the case where $\alpha \leq 1$. Observe that in each step, W_s increases by 1 for the new node and by $(d(h) + 1)^{\alpha} - d(h)^{\alpha} \leq 1$ for the host h. Thus, the overall increase after l - s steps is at most 2(l - s), and $W_l \leq W_s + 2(l - s) = W'_l$ as claimed. In the other case, where $\alpha > 1$, W_s similarly increases by 1 for the new node and by $(d(h) + 1)^{\alpha} - d(h)^{\alpha}$ for the host h. In addition, it is easy to verify that $(d(h) + 1)^{\alpha} - d(h)^{\alpha}$ is maximized if $d(h) = \Delta$. Thus, we have

$$W_l \le W_s + l - s + (\Delta_s + l - s)^{\alpha} - \Delta_s^{\alpha}$$
$$\le W_s + 2((\Delta_s + l - s)^{\alpha} - \Delta_s^{\alpha})$$
$$= W'_t$$

as claimed.

The following theorem shows that ParPolyPA yields a near-optimal speed-up if $\alpha \leq 1$ or $\alpha > 1$ and N is large.

▶ **Theorem 5.** Given a seed graph $G_0 = (V_0, E_0)$ with $n_0 = |V_0|$, ParPolyPA adds N new nodes to G_0 in expected time $\mathcal{O}((\sqrt{N} + N/\mathcal{P})\log \mathcal{P})$ if $\alpha \leq 1$ or $\alpha > 1$ and $N = \omega(n_0)$.

Proof. Computing W_0 and initializing the lists $P_1, \ldots, P_{\mathcal{P}}$ in parallel takes time $\mathcal{O}(n_0/\mathcal{P}\log(\mathcal{P}))$.

In the following, we show that the remainder of ParPolyPA can be implemented to process a batch of length R in time $\mathcal{O}((1 + R/\mathcal{P}) \log \mathcal{P})$. By Observation 1 and $\Delta_N = \Theta(n)$ if $N = \omega(n_0)$, we expect $E[R] = \Theta(\sqrt{n_i})$ resulting in $\mathcal{O}(\sqrt{N})$ batches which, in combination, establishes the claim.

In the first phase, all samples are drawn independently and affect only each PU's local state. The only concurrent writes happen to the global variable l for which we use *minimum* as conflict resolution. Thus, the first phase can be executed in time $\mathcal{O}((1 + R/\mathcal{P}) \log \mathcal{P})$.

The runtime of the second phase is dominated by the concurrent writes to the shared data structuring maintaining the counts $c(\cdot)$ of the proposal list. Parallel writes to the same counter are summed up. There at most $\Theta(R)$ such updates, accounting for $\mathcal{O}((1 + R/\mathcal{P})\log\mathcal{P})$ time per batch.

Finally, in the third phase, only one PU draws one sample which takes constant time.

4 I/O-Efficient Algorithm

In this section, we extend the I/O-efficient algorithm of [30] to the general case. The algorithm transfers the main idea of [10] to the external memory setting. Rather than reading from

```
Algorithm 3 EM-GenPA
     Data: Seed graph G = (V, E), with n_0 = |V|, m_0 = |E|, requested nodes N
 ı foreach v \in V do
                                                                                                                      // Init
          \mathsf{PQ}_M.\mathrm{push}(\mathsf{ExMsg}\langle \mathrm{d}_0(v), 0, v \rangle)
  2
          c_{d_0(v)} \leftarrow c_{d_0(v)} + 1
  3
                                                                                                                 // Phase 1
 4 foreach i \in [1, \ldots, N] do
          \mathsf{PQ}_M.\mathrm{push}(\mathsf{ExMsg}\langle \ell, \ell(i+1), v_{n_0+i} \rangle)
  5
          for each j \in [1, \ldots, \ell] do
  6
               repeat
  7
                    Sample degree d := d(h) of host h proportionally to c_d f(d)
  8
                    Accept with prob. (c_d - s_d)/c_d
  9
               until accepted;
10
              s_{d(h)} \leftarrow s_{d(h)} + 1
Store HostReq\langle v_{n_0+i}, j, d(h) \rangle
11
12
          Update counters c_d with changes s_d
13
14 Sort requests ascendingly by (degree, node)
15 for increasing degree d do
                                                                                                                 // Phase 2
          R_{\min} \leftarrow 0, R_{\max} \leftarrow 1
16
          for each HostReq\langle v_{n_0+i}, j, d_j \rangle with d_j = d do
\mathbf{17}
               t \leftarrow \ell i + j
18
               while \mathsf{ExMsg}\langle d', t', v \rangle \leftarrow \mathsf{PQ}_M.\mathsf{top}() where d' = d and t' < t do
19
                     R_v \leftarrow \text{uniformly from } [R_{\min}, R_{\max}]
20
                     \mathsf{PQ}_U.\mathrm{push}(R_v, v)
\mathbf{21}
                    PQ_M.pop()
 22
               (R_u, u) \leftarrow \mathsf{PQ}_U.\mathsf{pop}()
23
               R_{\min} \leftarrow R_u
24
               E \leftarrow E \cup \{\{v_{n_0+i}, u\}\}
\mathbf{25}
          Empty PQ_U
26
```

random positions of the edge list, it emulates the same process by precomputing all necessary read operations and sorts them by the memory address they are read from. As the algorithm produces the edge list monotonously moving from beginning to end, it scans through the sorted read requests and forwards still cached values to the corresponding target positions using an I/O-efficient priority-queue.

In order to extend the algorithm to the general case, we split the sampling of hosts into a two-step process, see also Algorithm 3. First, for each new node v_{n_0+i} we only sample the degrees of the ℓ different hosts and collectively save this information for the second phase, e.g. that node v_{n_0+i} requested a host with degree d. By postponing the actual sampling of the hosts, we can bulk all nodes with the same degree and therefore distribute them I/O-efficiently to the incoming nodes.

As the first phase only samples node degrees, it suffices to group nodes with the same degree d and represent each group by their counter c_d and weight $c_d \cdot f(d)$. Therefore, initially each seed node $v \in V_0$ contributes weight $f(d_0(v))$ to the overall weight of its corresponding group. A host request for degree d is then sampled proportionally to $c_d \cdot f(d)$. To reflect the actual generation process, we remember the number of sampled hosts s_d from degree

d arising from a new node v_{n_0+i} . This is necessary, as any subsequent host request needs to seek a different node to faithfully represent the model. By adding rejection sampling we correct the probability distribution a posteriori, i.e. if degree d is sampled we finally accept with probability $(c_d - s_d)/c_d$ and restart otherwise. After generating all ℓ host requests we update the corresponding counters (c_d, c_{d+1}) to $(c_d - s_d, c_{d+1} + s_d)$ for at most ℓ degrees. While sampling these requests we generate tuples $\mathsf{HostReq}\langle v_{n_0+i}, j, d_j \rangle$ for $j \in \{1, \ldots, \ell\}$ representing that node v_{n_0+i} requested as j-th host a node with degree d_j .

After collecting all host requests, we sort all requests by host degree first and new node second. Subsequently in the second phase, we fulfill each request for a host of degree d by uniformly sampling from the set of existing nodes with degree d at that time using two I/O-efficient priority-queues PQ_U and PQ_M employing standard external memory techniques [28]. While PQ_U is simply used as a means to retrieve uniform samples of its currently held messages, PQ_M is used to gather all matching nodes. To initialize, given the seed graph G_0 we insert for each node $v \in V_0$ a message $ExMsg\langle d_0(v), 0, v \rangle$ into PQ_M reflecting the information that at time t = 0 node v has degree $d_0(v)$ in G_0 . Similarly, we insert messages $ExMsg\langle \ell, \ell(i+1), v_{n_0+i} \rangle$ into PQ_M for all $i \in \{1, \ldots, N\}$, hinting that at time $\ell(i+1)$, after its addition, node v_{n_0+i} indeed has ℓ neighbors. We then process requests ascendingly by degree.

When processing host request $\operatorname{HostReq}\langle v_{n_0+i}, j, d_j \rangle$ we compute $t = \ell i + j$ and push all vertices of messages from PQ_M with degree d_j and time t' < t into PQ_U . More concretely, when processing all requests with degree d it is necessary to keep two values R_{\min} and R_{\max} and insert nodes into PQ_U with a weight drawn uniformly at random from $[R_{\min}, R_{\max}]$. After all suitable nodes have been inserted into PQ_U , we pop the node with smallest weight R and connect v_{n_0+i} to it⁷. Now that all remaining nodes in PQ_U have a weight of at least R, we update $R_{\min} \leftarrow R$ to preserve uniformity for any following node. Essentially, for a request targeted at time t and degree d we provide all nodes that exist as nodes with degree d up to time t and uniformly sample from them. After fulfilling the request with node u, we forward u as a potential partner to requests for hosts with degree d + 1 by adding a message $\operatorname{ExMsg}\langle d+1, \ell(i+1), u\rangle$ into PQ_M . All remaining unmatched nodes in PQ_U will stay unmatched, hence PQ_U is simply emptied and the algorithm proceeds to requests for the next larger degree.

By lazily resolving the actual hosts in the second step and only sampling the degrees in the first, the algorithm only needs to keep the set of currently existing degree groups in internal memory, i.e. the respective degree and its multiplicity in the current graph. Note that a graph represented by its set of unique degrees D has $\Omega(|D|^2)$ edges which even under pessimistic assumptions amounts to an output graph with more than 1 PB of size [21].

▶ Lemma 6. Processing the host requests sorted ascendingly by degree first and new node second correctly retains the output distribution.

Proof. Let M(d,t) be the set of nodes with degree d after time t where M(d,0) is given by the nodes of the seed graph G_0 with degree d. Processing a host request $\mathsf{HostReq}\langle v_{n_0+i}, j, d \rangle$ of node v for degree d at time $t = \ell i + j$ uniformly samples a node u of M(d, t - 1) and forwards it to the next degree group reflecting the equalities $M(d,t) = M(d,t-1) \setminus \{u\}$ and $M(d+1,t) = M(d+1,t-1) \cup \{u\}$.

⁷ Due to interchangeability, all inserted nodes have equal probability to have minimum weight and are thus sampled uniformly at random.



Figure 1 Dependency graph of direct dependencies where the degrees (3, 1, 2, 2, 1, 3, 2, 3, 1) are requested in order from left to right.

Therefore, when considering two requests $\mathsf{HostReq}\langle v_{n_0+a}, j, d \rangle$ and $\mathsf{HostReq}\langle v_{n_0+b}, j', d' \rangle$ where the first is generated before the second, it is clear that the latter can only depend on the former when $d \leq d'$, implying a DAG of dependencies of the degree requests. In particular, it is only necessary to consider direct dependencies, see Figure 1 for an example.

Processing the requests is correct as long as it conforms to the DAG of dependencies, i.e. is equivalent to any topological ordering of the requests. Naturally, processing requests ascendingly by time, is therefore correct. However, the order given by ascendingly sorting by degree first and time second also corresponds to a topological ordering proving the claim.

In Figure 1 both approaches are easily visualized. Processing by time corresponds to fulfilling the requests from left to right while processing by degree first and time second corresponds to fulfilling the requests from top to bottom, where early requests are prioritized if the degree is matching.

▶ **Theorem 7.** Given a seed graph $G_0 = (V_0, E_0)$ with $n_0 = |V_0|$, EM-GenPA adds *m* new edges to G_0 using $\mathcal{O}(\operatorname{sort}(n_0 + m))$ I/Os if the maximum number of unique degrees fits into internal memory.

Proof. The first phase produces $\Theta(m)$ many host requests incurring $\Theta(\operatorname{scan}(m))$ I/Os where the sampling can be done efficiently using a dynamic decision tree. Sorting the requests then takes $\mathcal{O}(\operatorname{sort}(m))$ I/Os.

In the second phase, each request is read in ascending fashion requiring a single scan, incurring $\Theta(\operatorname{scan}(m))$ I/Os. Fulfilling the host requests is done iteratively for increasing target degree. If a node is matched with a request for degree d, it is forwarded in time to requests for degree d + 1 or cleared otherwise. Thus, each node v in the output graph is inserted at least once and reinserted at most $\mathcal{O}(d_N(v) - d_0(v))$ times into the priority-queues. Hence, in total $\mathcal{O}(n_0 + m)$ messages are produced incurring $\mathcal{O}(\operatorname{sort}(n_0 + m))$ I/Os when implementing the priority-queues using Buffer Trees as the underlying data structure [6].

5 Implementations

5.1 Implementation for Internal Memory

We implement SeqPolyPA, ParPolyPA, and MVNPolyPa (see below) in the programming language RUST⁸ and almost exclusively use the SAFE language subset and avoid hardware-specific features.

For comparison with the state of the art, we consider the generator MVNPolyPa which relies on the first algorithm proposed in [29]. This data structure samples weighted items from a universe of size N in expected time $\mathcal{O}(\log^* N)$ and supports updates in time $\mathcal{O}(2^{\log^* N})$.

⁸ https://rust-lang.org; performance roughly on par with C.

While the authors propose further asymptotical improvements, preliminary experiments suggest that these translate into slower implementations. We consider the final implementation well tuned and added a few asymptotically sub-optimal changes (e.g. exploiting the finite precision of float point numbers and removing the hash maps originally used) that improve the practical performance significantly. The code is designed as a standalone crate⁹ and will be made independently available to the Rust-ecosystem for general sampling problems.

All implementations share a code base for common tasks. Non-integer computations are based on double-precision floating-point arithmetic. Repeated expensive operations (e.g. evaluating d^{α} for small degrees d) are memoized. The sampling of ℓ different hosts per new node is supported by rejecting repeated hosts. Additionally, we investigate MVNPolyPa^{remove}, which temporarily removes hosts from the data structure.

Our ParPolyPA implementation uses parallel threads operating on a shared memory. Synchronization is implemented via hurdles barriers¹⁰. All concurrently updated values are accessed either via fetch-and-add or compare-exchange primitives using the acquire-release semantics [15]. In contrast to the description in Algorithm 2, the code uses a single proposal list which is implemented as a contiguous vector. To avoid overheads and false-sharing, each threads reserves small blocks to write. It is also straight-forward to merge the third phase with the first phase of the next batch. This allows us to reduce the number of barriers required to two.

Observe that SeqPolyPA and ParPolyPA use contiguous node indices and that each connected node has at least one entry in the proposal list. This allows us to store the first entry of each node only implicitly, at least halving the memory size and number of access to the proposal list P (see Figure 2).

5.2 Implementation for External Memory

We implement EM-GenPA in C++ using the STXXL¹¹ library [18] which offers tuned external memory versions of fundamental operations like scanning and sorting. It additionally provides many implementations of different external memory data structures.

Since the degrees change incrementally where all incoming nodes have initial degree ℓ , we use a hybrid decision tree for the first phase. More concretely, we manage the smallest degrees in the range of $[1, P(\sqrt{n})]$ statically and any larger degree dynamically where P(x) is the smallest power of two greater or equal to x. Even for $n_0 + n = 2^{40}$ this amounts to less than 50 MB of memory for the statically managed degrees.

For the second phase we use the external memory priority-queues provided by STXXL based on [33].

6 Experiments

In this section, we study the previously discussed algorithms empirically. All generators produce simple graphs according to the preferential attachment model in Definition 1. To focus on this process, we compute the results but do not write out the graph to memory.

 $^{^{9}\,}$ Roughly speaking, the Rust equivalent of a software library.

¹⁰ https://github.com/jonhoo/hurdles

¹¹ We use a fork of STXXL that has been developed ahead of master https://github.com/bingmann/ stxxl.



Figure 2 Relative length $|P|/(n_0 + N)$ of the proposal list without the N implicitly stored positions.



Figure 3 Runtime per sample $t_{PA}/(N\ell)$ for different algorithms as function of N, ℓ , and α .

6.1 Internal Memory Algorithms

All internal memory experiments use the implementations described in Section 5.1 and are build with rustc 1.66.0-nightly (f83e0266c 2022-10-03)¹² on an Ubuntu 20.04 machine with an AMD EPYC 7702P processor with 64 cores and 512 GB of RAM. To focus our measurements on the sampling phase, we use small 1-regular seed graphs with $n_0 = 10\ell$ which have a negligible influence on the runtime and the structure of the resulting graph. We report the wall-time of the preferential-attachment process t_{PA} excluding initial setup costs (e.g. seed graph, initial allocation of buffers, et cetera); this leads to negligible biases between algorithms.

6.1.1 Sequential performance

In Section 2.1, we bound the expected size of the proposal list to be linear in the graph size. This analysis is consistent with Figure 2 (Appendix), which reports the proposal size divided by N. We observe no dependency in N over several orders of magnitude and find the proportionality factor to be upper bounded from above by 1 (recall that we store the first entry of each node only implicitly).

Figure 3 summaries the scaling behavior of SeqPolyPA, ParPolyPA, and MVNPolyPa in N. It reports the average time $t_{PA}/(N\ell)$ to obtain a single host for various combinations of ℓ and α . Despite near-constant asymptotic predictions, all implementations show a consistent

¹²See Cargo.lock for the exact versions of all dependencies.



Figure 4 Strong scaling of ParPolyPA as speed-up over SeqPolyPA for $N = 10^9$, $\ell = 1$, and $0.5 \le \alpha \le 1.5$.

deterioration of performance for larger instances. We attribute this to unstructured accesses to a growing memory area causing measurable increases in cache misses and back-end stalls. The additional steep rises in some of MVNPolyPa's plots are due to deeper recursions in the sampling data structure.

While our proposal list-based algorithms are fastest for $\alpha \leq 1$, MVNPolyPa performs well for sequential super-linear preferential attachment. This is due to the expected formation of $\Theta(\ell)$ high-degree nodes in this regime. These nodes have similar weights and are grouped together by MVNPolyPa which leads to high locality and very few cache misses during sampling.

Observe that it is trivial to hard-code this partition into SeqPolyPA and ParPolyPA to achieve similar performance. We opted against it in favor of cleaner measurements that describe the actual performance of the proposal list. These results are especially representative if the seed graph has a significant contribution to the resulting graph, i.e. if n_0/N is non-vanishing.

For $\ell = 10$, SeqPolyPA and ParPolyPA need to reject hosts that have been sampled multiple times. This incurs a small slow-down of less than $1.5 \times$. In this context, MVNPolyPa^{remove} exploits its fully dynamic sampling data structure to temporarily remove hosts (i.e. it explicitly samples without replacement). This is slightly beneficial in the super-linear regime with high locality, while rejection sampling is faster otherwise.

6.1.2 Parallel performance

In Observation 1, we motivate our parallelization by establishing that the number of batches (corresponding to the number of explicit synchronization points) scales as a square root of the graph size / maximum degree. This is supported by Figure 5, which almost perfectly matches this prediction.

Figure 4 reports the results for ParPolyPA with $N = 10^9$ as the speed-up over the sequential implementation SeqPolyPA. For $\alpha \leq 1$, we observe a near linear scaling with a speed-up of up-to 46 on 63 threads, dropping to 32 for $\alpha = 1.5$.

This is despite a comparable number of batches (see Figure 5). Instead we —slightly counter-intuitively— attribute the effect to the increased locality of the super-linear regime, as concurrent updates on high degree nodes lead to more frequent cache-invalidations. This can be mitigated using default techniques including randomized update sequences or hierarchical updates. Similarly to [11, Sec. 5], one can also merge a small number batches into an epoch,



Figure 5 The number of batches executed by ParPolyPA for various parameters. Observe the slope of 1/2 almost matching the prediction of Observation 1.



Figure 6 Running times of TFP-BA, NK-BA and EM-GenPA for $\ell = 10$ and increasing N.

and only update shared data at the epoch's end, leading to a trade-off between increased local overheads and reduced shared updates.

6.2 External Memory Algorithms

In order to assess the computational overhead given by the two-phase sampling, we compare the state-of-the-art sequential external memory algorithm TFP-BA [30] for the linear case to our implementation where we simply set f(d) = d. As an additional reference we consider a fast sequential internal memory implementation of the algorithm of Brandes and Batagelj [10] provided by NetworKit [35] which we refer to by NK-BA. Analogously to the experiments presented in [30], we use a small ring graph with $n_0 = 2\ell$ nodes as the seed graph for both algorithms and compare their running times for an increasing number of incoming nodes N. The benchmarks are built with GNU g++-9.4 and executed on a machine equipped with an AMD EPYC 7302P processor and 64 GB RAM running Ubuntu 20.04 using four 500 GB solid-state disks.

As illustrated in Figure 6, the performance of EM-GenPA is less than a factor of 2.32 slower than TFP-BA in the linear case and seems to be independent of M. Furthermore, this discrepancy becomes smaller when including writing the result to disk. Naturally, for graphs that fit into internal memory NK-BA is the fastest algorithm but becomes infeasible as soon as the edge list exceeds the available internal memory.

To study the influence of f on the running time of EM-GenPA, we additionally consider two polynomial models with exponent $\alpha \in \{0.5, 1.5\}$. In Figure 7 we see that EM-GenPA performs similarly well for all three models. However, it is noticeable that EM-GenPA performs better for models with a larger exponent α due to the increasingly more skewed degree distributions.



Figure 7 Running times of EM-GenPA for $\ell = 10, \alpha \in \{0.5, 1.0, 1.5\}$ and increasing N.

7 Conclusions

We present the sequential algorithm SeqPolyPA and the first efficient parallel algorithm ParPolyPA for polynomial preferential attachment. Furthermore, we present the first I/O-efficient algorithm EM-GenPA for general preferential attachment.

For a comparison with the state of the art, we engineer a sequential solution MVNPolyPa that relies on the fully dynamic sampling data structure proposed by [29]. We find that SeqPolyPA performs better or similarly well as MVNPolyPa; only for $\alpha > 1$ and adding many nodes, we find that the latter exhibits slightly better memory behavior due to the degenerate nature of the degree distribution in the limit. In addition, our parallel algorithm ParPolyPA obtains a speed-up of 46 for $\alpha \leq 1$ and 32 for $\alpha > 1$. We expect further improvements by using longer batches that may include multiple dependent samples.

Our experiments suggest that EM-GenPA performs similarly well to the external memory state-of-the-art algorithm TFP-BA for the linear case. Additionally, we investigate the performance of EM-GenPA for different polynomial models. For larger exponents, EM-GenPA performs better due to more favorable output degree distributions enabling more cache-efficient degree sampling and incurring less I/Os from the underlying priority-queues. While EM-GenPA is feasible for virtually any set of realistic input parameters it would still be interesting to lift the restriction that the degrees are kept internally.

— References

- Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. Commun. ACM, 31(9):1116–1127, 1988.
- 2 Md. Maksudul Alam, Maleq Khan, and Madhav V. Marathe. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In SC, pages 91:1–91:12. ACM, 2013.
- 3 Md. Maksudul Alam, Kalyan S. Perumalla, and Peter Sanders. Novel parallel algorithms for fast multi-gpu-based generation of massive scale-free networks. *Data Sci. Eng.*, 4(1):61–75, 2019.
- 4 R. Albert and A. L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, Jan 2002.
- 5 Daniel Allendorf, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Parallel and i/o-efficient algorithms for non-linear preferential attachment. In 2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), pages 65–76. SIAM, 2023.
- 6 Lars Arge. The buffer tree: A technique for designing batched external data structures. Algorithmica, 37(1):1–24, 2003.
- 7 Keyvan Azadbakht, Nikolaos Bezirgiannis, Frank S. de Boer, and Sadegh Aliakbary. A high-level and scalable approach for generating scale-free graphs using active objects. In SAC, pages 1244–1250. ACM, 2016.
- 8 Albert László Barabási et al. Network science. Cambridge university press, 2016.
- 9 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. Science, 286(5439):509–512, 1999.
- 10 Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Phys. Rev. E*, 71:036113, Mar 2005.
- 11 Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating population protocols in sub-constant time per interaction. In *ESA*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 12 B. Bollobás. Random graphs. Academic Press, 1985.
- 13 Béla Bollobás, Christian Borgs, Jennifer T. Chayes, and Oliver Riordan. Directed scale-free graphs. In SODA, pages 132–139. ACM/SIAM, 2003.
- 14 Karl Bringmann and Kasper Green Larsen. Succinct sampling from discrete distributions. In STOC, pages 775–782. ACM, 2013.
- 15 C++ Standards Committee et al. Iso international standard iso/iec 14882: 2011, programming language C++. Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep, 2011.
- 16 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.
- 17 Derek J. de Solla Price. A general theory of bibliometric and other cumulative advantage processes. J. Am. Soc. Inf. Sci., 27(5):292–306, 1976.
- 18 Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. Softw. Pract. Exp., 38(6):589–637, 2008.
- 19 S. N. Dorogovtsev and J. F. F. Mendes. Evolution of networks. Advances in Physics, 51(4):1079–1187, 2002.
- 20 Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. Maintaining discrete probability distributions optimally. In *ICALP*, volume 700 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1993.
- 21 Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/Oefficient generation of massive graphs following the *LFR* benchmark. *ACM J. Exp. Algorithmics*, 23, 2018.
- 22 Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 65:026107, Jan 2002.

- 23 Joseph JáJá. An introduction to parallel algorithms. *Reading, MA: Addison-Wesley*, 10:133889, 1992.
- 24 Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. The web as a graph: Measurements, models, and methods. In COCOON, volume 1627 of Lecture Notes in Computer Science, pages 1–17. Springer, 1999.
- 25 P. L. Krapivsky, S. Redner, and F. Leyvraz. Connectivity of growing random networks. *Phys. Rev. Lett.*, 85:4629–4632, Nov 2000.
- 26 P. L. Krapivsky, G. J. Rodgers, and S. Redner. Degree distributions of growing networks. *Phys. Rev. Lett.*, 86:5401–5404, Jun 2001.
- 27 Jérôme Kunegis, Marcel Blattner, and Christine Moser. Preferential attachment in online networks: measurement and explanations. In WebSci, pages 205–214. ACM, 2013.
- 28 Anil Maheshwari and Norbert Zeh. A survey of techniques for designing I/O-efficient algorithms. In Algorithms for Memory Hierarchies, volume 2625 of Lecture Notes in Computer Science, pages 36–61. Springer, 2002.
- 29 Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory Comput. Syst.*, 36(4):329–358, 2003.
- **30** Ulrich Meyer and Manuel Penschuck. Generating massive scale-free networks under resource constraints. In *ALENEX*, pages 39–52. SIAM, 2016.
- 31 Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation. CoRR, abs/2003.00736, 2020.
- 32 Sanguthevar Rajasekaran and Keith W. Ross. Fast algorithms for generating discrete random variates with changing distributions. *ACM Trans. Model. Comput. Simul.*, 3(1):1–19, 1993.
- Peter Sanders. Fast priority queues for cached memory. ACM J. Exp. Algorithmics, 5:7, 2000.
 Peter Sanders and Christian Schulz. Scalable generation of scale-free graphs. Inf. Process.
- 34 Peter Sanders and Christian Schulz. Scalable generation of scale-free graphs. Inf. Process. Lett., 116(7):489–491, 2016.
- 35 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Netw. Sci.*, 4(4):508–530, 2016.