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

Daniel Allendorf† Ulrich Meyer‡ Manuel Penschuck§ Hung Tran¶

November 15, 2022

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.

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).

Supplemental 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 [28] 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.

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 [11, 4, 7].

A well-known property of observed networks from various domains is scale-freeness, typically associated with a powerlaw degree distribution. Barabási and Albert [8] 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., [16, 12, 25, 23, 21, 18, 26]). 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 [9, 29, 33, 24, 16, 3] (see also [30] for a recent survey). Some observed networks however, can be better explained with polynomial preferential attachment [26].

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

**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 [9]. Their algorithm exploits a special structure of linear preferential attachment: given a list generated graph [9]. Their algorithm exploits a special structure of linear preferential attachment: given a list generated graph [9].

In particular, we forbid loops and multi-edges, e.g., our algorithms generate simple graphs.

For the external memory setting (Section 4), we extend the algorithm of [29] 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 $O(sort(n_0 + n))$ I/Os and even applies to the most general case where $P(h) \propto f(d)$ for any $f: \mathbb{N} \to \mathbb{R}$.

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.

For the analysis of our external memory algorithms we use the commonly accepted I/O-model of Aggarwal [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 $\Theta(n/B)$ I/Os, (ii) sorting $n$ consecutive items requires $\Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os, and (iii) pushing and popping $n$ items into an external priority queue requires $O(sort(n))$ I/Os [5].

### 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 straightforward by drawing multiple hosts per node and rejecting duplicates.

**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 graph. 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 $\alpha \geq 1$, 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 $n_0 = o(n)$ as $n \to \infty$ by using known properties of the asymptotic degree distributions for $\alpha < 1$ and $\alpha \geq 1$. 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 $O(1/\sqrt{n_0})$. 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 fix 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 $W = \sum_{i \in S} f(j)$ gives the proper normalization. The initial construction of our data structure is comparable to a variant of the alias method [13]. We first initialize an empty list or array $P$, 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.

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

<sup>3</sup>We refer the interested reader to [24] for an analysis of the degree distributions of non-linear preferential attachment graphs.

<sup>4</sup>An efficient implementation requires fast sampling from $P$ and inserting into $P$; an array with table doubling (e.g. [14]) is an easy choice with constant expected/amortized time per operation.

<sup>5</sup>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 $\alpha > 0$
- **For** $i \in [1, m_n]$, **do**
  - **if** $i$ is accepted from $P_{i-1}$, **then**
    - Accept with probability $w_i = |P_i - |P_{i-1} - W_i| v)$
    - $w_i(v) := d_{i-1}(v)/(c_{i-1}(v)$ and $c_i(v) := \#occurrences of v in P_{i-1};$
  - **until** accepted;
    - $V_i \leftarrow V_{i-1} \cup \{v_n_i\};$
  - $E_i \leftarrow E_i \cup \{\{v_n_i, h\}\};$
  - $P_i \leftarrow P_{i-1} + \{v_n_i\};$
- **while** $w_i > W_i/n_i$, **do**
  - $P_i \leftarrow P_i - \{h\};$
- **Return** $G_N = (V_N, E_N);$

**Copyright © 2023 by SIAM**

**Unauthorized reproduction of this article is prohibited.**
each node $v$ of the seed graph to the optimal count $c(v) = [d_0(v)^\alpha n_0/W_0]$. 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 $w_{i-1}(v) = d_{i-1}(v)^\alpha/e_{i-1}(v)$, 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 2.1.** 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

$$p(h) = \frac{c(h)}{|P|} \frac{w(h)}{\max_{v \in V} w(v)} = \frac{d(h)^\alpha}{|P| \max_{v \in V} w(v)}.$$ 

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

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

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

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

as claimed. \(\square\)

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

**Lemma 2.1.** 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 $O(n_0 + N)$.

**Proof.** The initial size of $P$ is at most

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

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

$$\max \left\{ \left\lfloor \frac{d_i(h)^\alpha n_i}{W_i} \right\rfloor - \left\lfloor \frac{d_j(h)^\alpha n_j}{W_j} \right\rfloor \right\}$$

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

Now, we distinguish two cases. For $\alpha \leq 1$, we have

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

Now, observe that $d_j(h)^\alpha/W_j$ 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 $d_j(h)^\alpha/W_j$. 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 $\alpha > 1$. Due to space constraints, we give the full proof for this case in the extended version (see title page). However a simple argument suffices if $n_0 = o(n)$, e.g. if the contribution of the seed graph is vanishing. Then as $n \to \infty$ a single node emerges which obtains almost all links and has expected degree $\Delta = \Theta(n)$ [24]. In each further step $i$ we then have $W_i/n_i = \Omega(\Delta^\alpha/n_i) = \Omega(n_i^{-1})$ and the increase in the count of the chosen host node $h$ is

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

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

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

which shows the claim. \(\square\)

We now show the main result of this section.

**Theorem 2.2.** Given a seed graph $G_0 = (V_0, E_0)$ with $n_0 = |V_0|$ nodes, SeqPolyPA adds $N$ new nodes to $G_0$ in expected time $O(n_0 + N)$.

**Proof.** The initialization in line 1 of Algorithm 1 takes at most time $O(|P_0|) = O(n_0)$ (see Lemma 2.1). 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. 2.1 the expected number of proposals until a node is accepted in step $i$ is

$$E[T] = \frac{1}{1-q} \frac{W_i}{W_{i-1}} = \frac{|P_{i-1}| \max_{v \in V_{i-1}} w_{i-1}(v)}{W_{i-1}}.$$
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 2, 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

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

**Algorithm 2: ParPolyPA**

**Data:** Seed graph $G = (V, E)$, with $n_0 = |V|$, $m_0 = |E|$, requested nodes $N$, $\alpha > 0$

1. for $1 \leq p \leq P$ do in parallel  
   // Init $P$
   2. $P_p \leftarrow ([v_{p-1}^{(1)}], \ldots, [v_{p}^{(1)}, \ldots, v_p^{(2)}]$

3. $s \leftarrow n_0$;

4. while $s < N + 1$ do  
   // Start new batch
   5. $l \leftarrow N + 1$;
   6. for $1 \leq p \leq P$ do in parallel
   7. // Phase 1
   8. $H \leftarrow [];$
   9. for $i \in [s + p, s + p + P, \ldots]$ do
   10. with prob. $\frac{W_i^{\alpha} - W_{i-1}^{\alpha}}{W_i}$, $l \leftarrow \min\{i, l\}$
   11. Sample $h$ with $P_i(h) = d_s(h)^\alpha/W_s$;
   12. $H \leftarrow H + [(v_{n_0+i}, h, i)]$
   13. barrier;
   14. for $(v, h, i) \in H$ where $i < l$ do
   15. $V \leftarrow V \cup \{v\}$, $E \leftarrow E \cup \{v, h\}$
   16. $P_p \leftarrow P_p + [v]$;
   17. while $w(h) > W_s/n_s$ do
   18. $P_p \leftarrow P_p + [h]$;
   19. barrier;
   20. if $p$ responsible then
   21. with prob. $\frac{W_i^{\alpha} - W_{i-1}^{\alpha}}{W_i}$, Sample $h$ with
   22. $P_i(h) = (d_s(h)^\alpha - d_i(h)^\alpha)/(W_i - W_s)$
   23. else with $P_i(h) = d_i(h)^\alpha/W_i$;
   24. $V \leftarrow V \cup \{v_i\}$, $E \leftarrow E \cup \{v_i, h\}$
   25. $P_p \leftarrow P_p + [v_i]$;
   26. while $w(h) > W_i/n_i$ do
   27. $P_p \leftarrow P_p + [h]$;

28. $s \leftarrow l$;

29. 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 $\sqrt{n}$ or $\sqrt{N}$ 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 $P = \Theta(\sqrt{n})$ processors if $\alpha \leq 1$, and $P = \Theta(\sqrt{N})$ if $\alpha > 1$.

### 3.1 ParPolyPA

We now describe the parallel algorithm in detail, see also Algorithm 2. First, let $P$ denote the number of PUs (processors), and let $1 \leq p \leq P$ de-
note the $p$-th PU. Then, PU $p$ adds all new nodes at positions $n_0 + kP + p$ and all new edges at positions $n_0 + kP + p$ where $0 \leq k \leq N/P$. Before any samples are drawn, PU $p$ receives an $n_0/P$ 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 $W_i/W'_i$ where

$$W'_i = \begin{cases} W_s + 2(i - s) & \text{if } \alpha \leq 1 \\ W_s + 2((\Delta_s + i - s)^\alpha - \Delta_s^\alpha) & \text{if } \alpha > 1 \end{cases}$$

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 $r \in \{1, \ldots, P\}$ and $c \in \{1, \ldots, S\}$ where $S = \max_p |P_p|$, 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 $i \geq l$ 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 $W'_i$. 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 3.1.** 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 $P_2(h) = (d_l(h)^\alpha - d_s(h)^\alpha)/(W_l - W_s)$, 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

$$P(h) = \frac{W_s}{W_l} P_1(h) + \frac{W_l - W_s}{W'_l} P_2(h) + \frac{W'_l - W_l}{W'_l} P_3(h)$$

$$= \frac{d_s(h)^\alpha}{W_l} + \frac{W_l - W_s}{W'_l} \frac{d_l(h)^\alpha}{W_l} + \frac{W'_l - W_l}{W'_l} \frac{d_l(h)^\alpha}{W_l}$$

$$= \frac{d_l(h)^\alpha}{W_l}.$$

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

$$W_l \leq W_s + l - s + ((\Delta_s + l - s)^\alpha - \Delta_s^\alpha) \leq W_s + 2((\Delta_s + l - s)^\alpha - \Delta_s^\alpha) = W'_l$$

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

**Theorem 3.2.** 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 $O((\sqrt{N} + N/P) \log P)$ if $\alpha \leq 1$ or $\alpha > 1$ and $N = \omega(n_0)$.

**Proof.** Computing $W_0$ and initializing the lists $P_1, \ldots, P_P$ in parallel takes time $O(n_0/P \log(P))$. In the following, we show that the remainder of ParPolyPA can be implemented to process a batch of length $R$ in time $O((1 + R/P) \log P)$. By Observation 1 and $\Delta_N = \Theta(n)$ if $N = \omega(n_0)$, we expect $E[R] = \Theta(\sqrt{n})$ resulting in $O(\sqrt{N})$ 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 $O((1 + R/P) \log P)$.

The runtime of the second phase is dominated by the concurrent writes to the shared data structure 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 $O((1 + R/P) \log P)$ time per batch.

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

4 I/O-Efficient Algorithm

In this section, we extend the I/O-efficient algorithm of [20] to the general case. The algorithm transfers the main idea of [2] to the external memory setting. Rather than reading from 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 monotonically 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 $\ell$ 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 $\ell$ 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 $\ell$ de-

---

Algorithm 3: EM-GenPA

**Data:** Seed graph $G = (V, E)$, with $n_0 = |V|$, $m_0 = |E|$, requested nodes $N$

```
1 foreach $v \in V$ do // Init
    2 $\text{PQ}_M$.push(ExMsg(d_0(v), 0, v))
    3 $c_{d_0(v)} \leftarrow c_{d_0(v)} + 1$

4 foreach $i \in [1, \ldots, N]$ do // Phase 1
    5 $\text{PQ}_M$.push(ExMsg(\ell, \ell(i+1), v_{n_0+i}))

6 foreach $j \in [1, \ldots, \ell]$ do
    repeat
        7 Sample degree $d := d(h)$ of host $h$
        8 proportionally to $c_d f(d)$
        9 Accept with prob. $(c_d - s_d)/c_d$
        until accepted:
        10 $s_d(h) \leftarrow s_d(h) + 1$
        11 Store HostReq(v_{n_0+i}, j, d(h))

12 Update counters $c_d$ with changes $s_d$
13 Sort requests ascendingly by (degree, node)
14 for increasing degree $d$ do // Phase 2
15    $R_{\text{min}} := 0$, $R_{\text{max}} := 1$
16    $\text{HostReq}(v_{n_0+i}, j, d_j)$ with $d_j = d$ do
17        $t \leftarrow \ell_i + j$
18        while ExMsg(d', t', v) $\leftarrow \text{PQ}_M$.top() where $d' = d$ and $t' < t$
19            $R_v \leftarrow$ uniformly from $[R_{\text{min}}, R_{\text{max}}]$
20            $\text{PQ}_U$.push($R_v, v$)
21        $\text{PQ}_M$.pop()
22        $(R_u, u) \leftarrow \text{PQ}_U$.pop()
23        $R_{\text{min}} \leftarrow R_u$
24        $E \leftarrow E \cup \{\langle v_{n_0+i}, u \rangle \}$
25    26 Empty $\text{PQ}_U$
```
degrees. While sampling these requests we generate tuples \( \text{HostReq}(v_{n_0+i}, j, d_j) \) 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 hosts 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 \( \text{PQ}_U \) and \( \text{PQ}_M \) employing standard external memory techniques \[27\]. While \( \text{PQ}_U \) is simply used as a means to retrieve uniform samples of its currently held messages, \( \text{PQ}_M \) 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 \( \text{ExMsg}(d_0(v), 0, v) \) into \( \text{PQ}_M \) reflecting the information that at time \( t = 0 \) node \( v \) has degree \( d_0(v) \) in \( G_0 \). Similarly, we insert messages \( \text{ExMsg}(\ell, \ell(i+1), v_{n_0+i}) \) into \( \text{PQ}_M \) for all \( i \in \{1, \ldots, N\} \), hinting that at time \( t = \ell(i+1) \), after its addition, node \( v_{n_0+i} \) indeed has \( \ell \) neighbors. We then process requests ascendingly by degree.

When processing host request \( \text{HostReq}(v_{n_0+i}, j, d_j) \) we compute \( t = \ell i + j \) and push all vertices of messages from \( \text{PQ}_M \) with degree \( d_j \) and time \( t' < t \) into \( \text{PQ}_U \). More concretely, when processing all requests with degree \( d \) it is necessary to keep two values \( R_{\text{min}} \) and \( R_{\text{max}} \) and insert nodes into \( \text{PQ}_U \) with a weight drawn uniformly at random from \([R_{\text{min}}, R_{\text{max}}]\). After all suitable nodes have been inserted into \( \text{PQ}_U \), we pop the node with smallest weight \( R \) and connect \( v_{n_0+i} \) to it. Now that all remaining nodes in \( \text{PQ}_U \) have a weight of at least \( R \), we update \( R_{\text{min}} \leftarrow R \) 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 \( \text{ExMsg}(d+1, \ell(i+1), u) \) into \( \text{PQ}_M \). All remaining unmatched nodes in \( \text{PQ}_U \) will be unmatched, hence \( \text{PQ}_U \) 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 \[20\].

Lemma 4.1. 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 \( \text{HostReq}(v_{n_0+i}, j, d_j) \) 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 \( M(d+1,t) = M(d+1,t-1) \cup \{u\} \).

Therefore, when considering two requests \( \text{HostReq}(v_{n_0+a}, j, d) \) and \( \text{HostReq}(v_{n_0+b}, j', d') \) 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 Fig. 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 Fig. 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 4.1. 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 \( O(\text{sort}(n_0 + m)) \) I/Os if the maximum number of unique degrees fits into internal memory.

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

In the second phase, each request is read in ascending fashion requiring a single scan, incurring \( \Theta(\text{scan}(m)) \) 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 \( O(d_{\text{out}}(v) - d_0(v)) \) times into the priority-queues. Hence, in total \( O(n_0 + m) \) messages are produced incurring \( O(\text{sort}(n_0 + m)) \) I/Os when implementing the priority-queues using Buffer Trees as the underlying data structure \[9\].

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited.
5 Implementations

5.1 Implementation for Internal Memory We implement SeqPolyPA, ParPolyPA, and MVNPolyPa (see below) in the programming language Rust\(^7\) 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 \[28\]. This data structure samples weighted items from a universe of size \(N\) in expected time \(O(\log^* N)\) and supports updates in time \(O(2^{\log^* N})\). 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 crate\(^8\) 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 \(d^a\) for small degrees \(d\)) are memoized. The sampling of \(\ell\) different hosts per new node is supported by rejecting repeated hosts. Additionally, we investigate MVNPolyPa\(^9\), which temporarily removes hosts from the data structure.

Our ParPolyPA implementation uses parallel threads operating on a shared memory. Synchronization is implemented via hurdles barriers.\(^9\) All concurrently updated values are accessed either via fetch-and-add or compare-exchange primitives using the acquire-release semantics.\(^1\) 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 Fig.\(^2\)).

5.2 Implementation for External Memory We implement EM-GenPA in C++ using the STXXL\(^10\) library 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 \(\ell\), we use a hybrid decision tree for the first phase. More concretely, we manage the smallest degrees in the range of \([1, \sqrt{n}]\) statically and any larger degree dynamically where \(P(x)\) is the smallest power of two greater or equal to \(x\). Even for \(n_0 + n = 2^{10}\) 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 \[32\].

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\(^4\) To focus on this process, we compute the results but do not write out the graph to memory.

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)\(^11\) on an

\[\text{https://rust-lang.org}\] performance roughly on par with C.
\[\text{https://github.com/jonhoo/hurdles}\]
\[\text{https://github.com/jonhoo/hurdles}\]
\[\text{https://github.com/bingmann/stxxl}\]
\[\text{https://github.com/jonhoo/hurdles}\]
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.

**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 Fig. 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 \( \ell \) and \( \alpha \). Despite near-constant asymptotic predictions, all implementations show a consistent 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.

**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 Fig. 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 Fig. 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 [10, Sec. 5], one can also merge a small number batches into an epoch, 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 [29] for the linear case to our implementation of the algorithm of Brandes and Batagelj [9] provided by Networkit [34] which we refer to by NK-BA. Analogously to the experiments presented in [29], 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 Fig. 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 Fig. 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 $\alpha$ due to the increasingly more skewed degree distributions.

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 [28]. 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


Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited.