Journal of Parallel and Distributed Computing 132 (2019) 363–382

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

MIDAS: Multilinear detection at scale

Saliya Ekanayake [a], Jose Cadena [b,*], Udayanga Wickramasinghe [c], Anil Vullikanti [d]

[a] Lawrence Berkeley National Laboratory, United States
[b] Lawrence Livermore National Laboratory, United States
[c] Indiana University, United States
[d] University of Virginia, United States

HIGHLIGHTS

- Finding subgraphs is an important primitive in network analysis.
- It is possible to find "small" subgraphs optimally, but it takes exponential time.
- Existing parallel algorithms find subgraphs of size up to 12.
- We propose a distributed algorithm that scales to subgraphs of size 18.
- Our algorithm can be applied to find subtrees and for anomaly detection tasks.

ARTICLE INFO

Article history:
Received 23 July 2018
Received in revised form 1 April 2019
Accepted 2 April 2019
Available online 31 May 2019

Keywords:
Subgraph isomorphism
Distributed graph algorithms
Graph scan statistics
Multilinear detection
Parameterized complexity

ABSTRACT

We focus on two classes of problems in graph mining: (1) finding trees and (2) anomaly detection in complex networks using scan statistics. These are fundamental problems in a broad class of applications. Most of the parallel algorithms for such problems are either based on heuristics, which do not scale very well, or use techniques like color coding, which have a high memory overhead. In this paper, we develop a novel approach for parallelizing both these classes of problems, using an algebraic representation of subgraphs as monomials—this methodology involves detecting multilinear terms in multivariate polynomials. Our algorithms show good scaling over a large regime, and they run on networks with close to half one billion edges. The resulting parallel algorithm for trees is able to scale to subgraphs of size 18, which has not been done before, and it significantly outperforms the best prior color coding based method (FASCIA) by more than two orders of magnitude. Our algorithm for network scan statistics is the first such parallelization, and it is able to handle a broad class of scan statistics functions with the same approach.

Published by Elsevier Inc.

1. Introduction

Many problems in graph mining and social network analysis can be reduced to questions about different kinds of subgraphs. Two important classes of such problems, which are the focus of our paper, are: (1) *Detecting and counting subgraphs*, such as paths and trees, of a given size k—these are used for characterizing different kinds of networks, especially in biological models [3,23]. (2) *Anomaly detection in network data using graph scan statistics*, which involves finding connected subgraphs of size k, optimizing different kinds of anomaly score functions [12,22,27,31,38]—this arises in a number of applications, such as social network analysis, epidemiology, finance, and bio-surveillance [2].

Both problems are computationally very challenging. For instance, exact detection of paths is NP-hard [19] and the corresponding counting problem is #P-hard [40]. Development of parallel algorithms for these problems is an active area of research, and many parallel algorithms exist for counting simple local subgraphs [5,6] or approximately counting maximal cliques [14,33]. Finding trees is more challenging than finding local subgraphs, such as triangles, and a number of heuristics have been developed. One of the few techniques that gives rigorous approximation guarantees is *color coding* [3,4,23]. Parallel adaptations have been developed using MapReduce [44] and MPI [36,37]. The MPI-based FASCIA algorithm [36,37] is the state-of-the-art in terms of counting trees in massive networks, scaling to finding trees with up to 12 vertices in networks with one billion edges. However, it seems very challenging to scale the color coding method to larger subgraphs, even on small networks. The main reason is that the time and space complexity of the color coding technique scale as $(2e)^k$ and 2^k (times a polynomial factor in the graph size),

* Corresponding author.
E-mail addresses: esaliya@lbl.gov (S. Ekanayake), cadenapico1@llnl.gov (J. Cadena), uswickra@iu.edu (U. Wickramasinghe), vsakumar@virginia.edu (A. Vullikanti).

https://doi.org/10.1016/j.jpdc.2019.04.006
0743-7315/Published by Elsevier Inc.

respectively, where k denotes the subgraph size. In this paper, we take the first steps towards beating this bound, which has remained a significant open problem since [37]. Our approach involves parallelization of a powerful algebraic technique for detecting multilinear terms in a multivariate polynomial, developed by Koutis [24] and Williams [42].

Finding subgraphs that maximize most scan statistics are NP-hard problems, in general [10]. There is a lot of work on adapting heuristics based on steiner connectivity, but these do not give any rigorous approximation guarantees. The first rigorous result for network scan statistics was also designed using color coding [10]; however, this has not been parallelized because of its high memory overhead.

In this paper, we develop a distributed algorithm for multilinear detection, which immediately leads to highly scalable algorithms for both paths and trees, and network scan statistics. Our work combines two conference papers: [17] and [16]. Our main contributions are:

1. MIDAS**: Distributed algorithm for multilinear detection and applications to subgraph analysis.** We develop distributed algorithms in the MPI model for finding paths and trees, and optimizing scan statistics on networks, through detection of multilinear terms with k variables of the form $x_{i_1}x_{i_2}\ldots x_{i_k}$ (i.e., a term in which all variables have exponent 1) in a multivariate polynomial $P(x_1,\ldots,x_n)$. The sequential algorithm uses a matrix representation of a group algebra [24,42], which has a structure that lends itself to parallelization. As we discuss in Section 4, the obvious ways to try to parallelize multilinear detection do not scale well; instead, we find that a careful interplay between the degree of partitioning as well as batching a set of iterations yields better results. We give rigorous bounds on the performance in terms of the time and space complexity, which scale as $O(2^k)$ and $O(k)$, respectively, compared with $O(2^k e^k)$ and $O(2^k)$ for color coding, respectively [36,37,44]. For random graphs, we show a rigorous scaling with N, the number of processors for $N \leq 2^k$. For network scan statistics, we also examine the performance of the multilinear detection approach in the "think-like-a-vertex" Giraph framework, but we found it to have limited scalability due to high communication overheads.

2. Cache optimization and weak scalability. MIDAS partitions the graph G into N_1 parts. The computation involves 2^k iterations, and N_1 processors together perform one iteration—this allows us to schedule N/N_1 such computations to occur in parallel. The total compute time exhibits good weak scaling. Additionally, our data structures for supporting Galois field operations during the iterations support a temporal cache locality, which actually leads to a reduction in the compute time as N_1 increases. On the other hand, the communication cost increases with N_1, leading to an optimal value of N_1 for the best performance.

3. Experimental results and case studies. We evaluate our results on a number of real and synthetic networks with up to 250 million edges. The reduced memory footprint allows us to scale to finding subgraphs of size 18, which has not been done before. Our algorithms for both problems show reasonable scaling up to 512 processors, supporting our theoretical analysis. The running time grows linearly with the network size and as 2^k for the subgraph size k. We also illustrate applications of our algorithms to two anomaly detection tasks: (1) finding congested highways in a traffic network and (2) discovering clusters of low vaccination in the state of Minnesota.

4. Comparison with prior methods. Our algorithm for finding paths is four orders of magnitude faster than FASCIA, the state of the art method based on color coding [36,37]. One additional advantage is that our parallel algorithm based on multilinear detection is conceptually much simpler than color coding algorithms, and it requires far less bookkeeping.

2. Preliminaries

2.1. Problem formulation and notation

We will focus on the following two classes of problems in this paper, but our approach can be extended more broadly.

2.1.1. Finding paths and trees

Given a graph $G = (V, E)$ with $n = |V|$, $m = |E|$, and a subgraph $H = (V^H, E^H)$, with $k = |V^H|$, find a mapping $f : V^H \to V$, such that $(i, j) \in E_H$ *only if* $(f(i), f(j)) \in E$. Such a mapping is referred to as a *non-induced embedding* of H in G.

Problem 1 (k*-Tree*)**.** Given a weighted graph $G = (V, E)$ with a weight vector $\mathbf{w}$, and a tree denoted by $H = (V^H, E^H)$ with $|V^H| = k$, the objective is to determine if there exists an embedding of H in G.

We will consider the following approximate version of Problem 1: determine if a non-induced embedding exists with probability at least $1-\epsilon$, where $\epsilon \in (0, 1)$ is a parameter. Other common variants of this problem are: (1) counting all embeddings, and (2) finding a maximum weight embedding in a weighted version of the graph; our approach can be extended to these variants.

2.1.2. Anomaly detection using graph scan statistics

We assume that we are given an undirected graph $G = (V, E)$, where V is a set of n vertices or nodes, and E is a set of m edges. Each vertex $v \in V$ has two values associated with it: (1) a *baseline count*, $b(v)$, which indicates the count that we expect to see at the node v – e.g., the number of people in a county corresponding to node v – and (2) an *event count* or *weight*, $w(v)$, which indicates how many occurrences of an event of interest are seen at the node—e.g., the number of cases of a disease in a county. These values vary over time, but we will not indicate the time in the notation in order to keep it simple.

Graph scan statistics are among the most commonly used methods for detecting anomalies or "hotspots" in network data [12,22,27,31,38]. This approach formalizes anomaly detection as a hypothesis testing problem. Under the null hypothesis H_0, it is *business as usual*, and the event counts for all nodes are generated proportionally to their baseline counts. Under the alternative hypothesis $H_1(S)$, counts of a majority of the vertices are generated (again) with rate proportional to the baseline counts, but there exists a small connected subset $S \subseteq V$ of vertices for which the counts are generated at a higher rate than expected. Then, the goal is to find a set of vertices S that maximizes an appropriate scan statistic function $F(S)$, typically a log-likelihood ratio that compares event counts to baseline counts. We define a scan statistic in terms of the event and baseline counts of a node set:

$$F(S) = F(W(S), B(S), \boldsymbol{\theta}),$$

where $W(S) = \sum_{v\in S} w(v)$ is the total event count or *weight* of S, $B(S) = \sum_{v\in S} b(v)$ is the baseline count of the set, and θ represents possible additional arguments to F.

Depending on the assumptions that are satisfied by the data, there are two broad types of scan statistics: parametric and non-parametric. A well-known example of the former is the Kulldorff scan statistic commonly used in disease surveillance [15,25,26,32], which is defined as

$$W(S)\log\left(\frac{W(S)}{B(S)}\right) + (W(V) - W(S))\log\left(\frac{W(V) - W(S)}{B(V) - B(S)}\right) - B(V)\log\left(\frac{W(V)}{B(V)}\right),$$

with $\boldsymbol{\theta} = (W(V), B(V))$. An example of a non-parametric function is the Berk–Jones scan statistic (BJ) [8] used for civil unrest events and network intrusion detection [12,29]. In this setting, each node v has a p-value $p(v) \in [0, 1]$, and, for a significance level α, the event count $w(v)$ is 1 if $p(v) < \alpha$ (i.e., the node is significant) and 0 otherwise, and the baseline count is $b(v) = 1$ for all nodes. This scan statistic is defined as

$$\max_{\alpha \le \alpha_{max}} B(S)\left[\frac{W(S)}{B(S)}\log\left(\frac{W(S)/B(S)}{\alpha}\right) + \left(1-\frac{W(S)}{B(S)}\right)\log\left(\frac{1-W(S)/B(S)}{1-\alpha}\right)\right],$$

with $\boldsymbol{\theta} = \alpha_{max}$.

From the discussion above, the graph anomaly detection task can be posed as the following constrained optimization problem: Given a graph $G' = (V', E')$, a scan statistic $F(\cdot)$, and the associated counts for vertices – represented by vectors **w** and **b** – find a connected subset $S' \subseteq V'$ that maximizes $F(S') = F(W(S'), B(S'), \theta)$. For brevity, we will focus on non-parametric scan statistics in the rest of the paper, using the BJ statistic as a working example. As discussed above, for this function, $B(S') = |S'|$ is the size of the set.

Finding solutions with constraints on the effective size. Following [10], we will consider connected subgraphs S' that maximize $F(S')$, with $|S'| \le k'$, where k' is a parameter. The graph G' can be transformed $G' \to G$ by merging connected subsets S'_v of anomalous nodes in G' to form "supernodes" v in G using the refinement method discussed in [10]; we use $w(v)$ to denote the number of anomalous nodes in S'_v. This is illustrated in Fig. 1. We refer to $k_{\text{eff}} = |S|$ as the "effective solution size" for the set $\cup_{v\in S} S'_v$, which has size $|\cup_{v\in S} S'_v| \gg k_{\text{eff}}$. In the rest of the paper, we assume that we are given the preprocessed graph G as input, and the goal is to find a connected subset S that maximizes $F(S)$, with $|S| \le k_{\text{eff}}$; this will be denoted by k, in order to simplify the notation. Formally, the focus of the paper is the following problem:

Problem 2. Given a graph $G = (V, E)$, a scan statistic $F(\cdot)$, the associated counts for vertices—**w** and **b**—and a parameter $k \ll |V|$, find a connected subset $S \subseteq V$ that maximizes $F(S) = F(W(S), B(S), \theta)$ with $B(S) \le k$.

3. k-multilinear detection and sequential algorithm

We describe the sequential multilinear detection algorithm. We start with some introduction to group algebras and Galois fields and end with the sequential algorithm for finding k-paths. Because of the limited space, we only describe the main ideas here and refer to [24,42] for more details.

Let $X = x_1, \ldots, x_n$ be a set of variables, and let $P(X)$ be a polynomial, which is a sum of monomials on X. We will denote $P(X) = \sum_S \Pi_{i\in S} x_i$ as a monomial, where the sum is over multisets S. An example of a polynomial on six variables is $P(x_1, x_2, x_3, x_4, x_5, x_6) = x_1^2x_2 + x_2x_3x_4 + x_3x_4x_5 + x_5x_6$. A monomial is called *multilinear* or *square-free* if all its variables have exponent 1, and its *degree* is the sum of the exponents of all its variables. For instance, in the example above, $x_2x_3x_4$, $x_3x_4x_5$, and x_5x_6 are multilinear monomials, but $x_1^2x_2$ is not multilinear. Given variables $X = x_1, \ldots, x_n$ and a polynomial $P(X)$, the goal in the k-Multilinear Detection (k-MLD) problem is to decide whether or not $P(X)$ has a multilinear monomial of degree exactly k.

We note that the polynomial $P(X)$ may have an arbitrary number of terms – i.e., exponential on the size of n – therefore, the problem is not as simple as writing the polynomial explicitly and checking each term. Rather, we assume that $P(X)$ is given succinctly in a recursive form, and the "yes"/"no" decision has to be made without unrolling this recursion. In general, we also have a weight w_S for each multinomial $\Pi_{i\in S} x_i$. Formally, we have the following problem.

Problem 3 (k-MLD *Problem*)**.** Given a polynomial $P(\cdot)$ defined recursively, in which each monomial has degree at most k and weight w_S, determine: (1) if $P(\cdot)$ has a multilinear term of degree k, and (2) the maximum weight of any multilinear term, if one exists.

3.1. Group algebras

We discuss some notation from group algebras that is crucial for the paper. Let $\mathbb{Z}_2^k$ be the group formed by all the k-dimensional binary vectors, and define the group multiplication operation as entry-wise XOR. For example, $\mathbb{Z}_2^2$ consists of the vectors $v_0 = (0, 0)$, $v_1 = (0, 1)$, $v_2 = (1, 0)$, $v_3 = (1, 1)$. We note that v_0 is the multiplicative identity of the group, and each element is its own multiplicative inverse: $v_i \cdot v_i = v_0$. Now, we define a group algebra $\mathbb{Z}_2[\mathbb{Z}_2^k]$. Each element in the group algebra is a sum of elements from $\mathbb{Z}_2^k$ with coefficients from $\mathbb{Z}_2$ (i.e., either 1 or 0): $\sum_{v\in\mathbb{Z}_2^k} a_v v$, where $a_v \in \{0, 1\}$. The addition operator of the group algebra is

$$\sum_{v\in\mathbb{Z}_2^k} a_v v + \sum_{v\in\mathbb{Z}_2^k} b_v v = \sum_{v\in\mathbb{Z}_2^k} (a_v + b_v)v,$$

where the addition of the coefficients is modulo 2, and the multiplication is defined as

$$\left(\sum_{v\in\mathbb{Z}_2^k} a_v v\right)\left(\sum_{u\in\mathbb{Z}_2^k} b_u u\right) = \sum_{v\in\mathbb{Z}_2^k} (a_v \cdot b_u)(v \cdot u).$$

Example. For $k = 2$, the group algebra $\mathbb{Z}_2[\mathbb{Z}_2^k]$ has $2^{2^2} = 16$ elements, such as

$x_1 = 0\cdot\begin{bmatrix}0\\0\end{bmatrix} + 1\cdot\begin{bmatrix}0\\1\end{bmatrix} + 1\cdot\begin{bmatrix}1\\0\end{bmatrix} + 0\cdot\begin{bmatrix}1\\1\end{bmatrix}$, which we also write as $\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}$

$x_2 = \begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}$

We have

$x_1 + x_2 = \begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix} + 2\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}$

$x_1x_2 = \left(\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}\right)\cdot\left(\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix}$

It is easy to check that

$x_1^2x_2 = 0\cdot\begin{bmatrix}0\\0\end{bmatrix} + 0\cdot\begin{bmatrix}0\\1\end{bmatrix} + 0\cdot\begin{bmatrix}1\\0\end{bmatrix} + 0\cdot\begin{bmatrix}1\\1\end{bmatrix} = \vec{0}$ (additive identify)

3.2. Sequential algorithm for multilinear detection

We briefly discuss the algorithm of Koutis [24], which forms the basis of our paper. An important property is that for any $v_i \in \mathbb{Z}_2^k$, the square of the term $(v_0 + v_i) \in \mathbb{Z}_2[\mathbb{Z}_2^k]$ evaluates to 0:

$$(v_0+v_i)^2 = v_0^2 + 2(v_0\cdot v_i) + v_i^2 = v_0 + (0 \mod 2)v_i + v_0 = 2v_0 = 0.$$

We can show that, if we choose a $v_i \in \mathbb{Z}_2^k$ uniformly at random and set $x_i = v_0 + v_i$, then a multilinear monomial does **not** evaluate to $\vec{0}$ with some probability, whereas a monomial with squares is always $\vec{0}$ (as in the box above). The algorithm was later refined in [42] by evaluating the polynomial over the group algebra $GF(2^{3+\log_2 k})[\mathbb{Z}_2^k]$, where $GF(p)$ is the Galois field of order

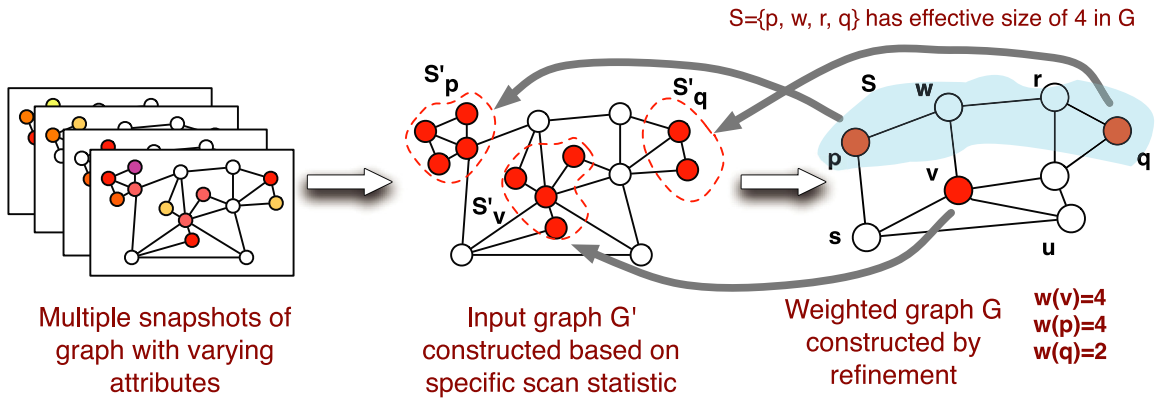

Fig. 1. Example of an input graph G' (constructed using the scan statistics), and the reduced graph G constructed using the refinement process of [10]. The anomalous nodes in G' are shown in red. (Super-)node p in the reduced graph G represents a set of four anomalous nodes in G', denoted by the set S'_p and has weight $W(p) = 4$. The subset $S = \{p, w, r, q\}$ in the graph G corresponds to a subset of size 8 in G', but it has effective size 4 in G. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

p [30], and this is the version that we implement. But, to simplify the discussion below, we assume that we are working on the group algebra $\mathbb{Z}_2[\mathbb{Z}_2^k]$.

A polynomial $P(x_1, \ldots, x_n)$ with variables from $\mathbb{Z}_2[\mathbb{Z}_2^k]$ can be evaluated in time $O(2^k poly(n))$ and space $O(2^k poly(n))$, resulting in Theorem 1.

Theorem 1 (*Koutis [24] and Williams [42]*). *There exists an algorithm that, given an instance $P(x_1, \ldots, x_n)$ of the k-MLD problem, correctly returns "no" if $P(X)$ does not contain a k multilinear term. Otherwise, if $P(X)$ has a k multilinear term, it returns "yes" with probability at least 1/5. The algorithm has time complexity $O(2^k poly(n))$ and space complexity $O(2^k poly(n))$.*

3.3. Implementation using a matrix representation of $\mathbb{Z}_2[\mathbb{Z}_2^k]$

Theorem 1 performs operations in the group algebra $\mathbb{Z}_2[\mathbb{Z}_2^k]$, which takes $O(2^k poly(n))$ space. Koutis [24] showed that the space complexity can be reduced to $O(kpoly(n))$ by using the idea of matrix representations. The main idea is that every element of the group algebra can be represented as a $2^k \times 2^k$ matrix, and the polynomial $P(X)$ evaluates to $\bar{0}$ if and only if the trace of its corresponding matrix representation is $0 \mod 2^{k+1}$. We can compute the trace by evaluating the polynomial over the group of all integers 2^k times, once for each element of the diagonal. For each variable $x_i = v_0 + v_i$, the tth diagonal element in the corresponding matrix representation is $1 + (-1)^{v^T t_{bin}}$, where t_{bin} is the k-bit binary representation of t.

3.4. Application to k-path

As an example of the multilinear detection technique, we now describe a sequential algorithm for the k-Path problem, which is a special case of Problem 1. We are given a graph $G(V, E)$ and a parameter k, and the algorithm decides whether or not the graph has a path of length k. First, we reduce k-Path to a k-MLD instance (this follows from [24,42]). Given a graph $G(V, E)$, let x_i denote a variable associated with each node $i \in V$. We define polynomials $P(i, j)$ for all $i \in V, j \leq k$ in the following manner.

- $P(i, 1) = x_i$ for all $i \in V$
- For $j > 1$, $P(i, j) = \sum_{u \in \text{NBR}(i)} P(i, 1)P(u, j - 1)$
- Define the polynomial $P(x_1, \ldots, x_n) = \sum_i P(i, k)$

Intuitively, a polynomial $P(i, j)$ encodes all the possible walks of length j ending at node i. Each monomial in $P(i, j)$ corresponds to one walk. It can be verified that the graph G has a path of length k if and only if the polynomial $P(x_1, \ldots, x_n)$ has a multilinear term—i.e., a walk with no repeated vertices.

Algorithm 1 presents the full procedure. With the matrix representation, the polynomial for k-Path is evaluated over 2^k iterations (lines 6–12). In each iteration, we first initialize $P(i, 1)$ (lines 7–8). From there, we compute recursively $P(i, j)$, a polynomial where each term contains x_i and has degree j (lines 9–11). The computation of $P(i, j)$ for a node i uses data from the immediate neighbors of i and all the polynomials of degree $j - 1$, which have already been computed at this point. The two applications that we consider in Sections 5 and 6 have this structure.

Algorithm 1 MULTILINEARDETECTPATH($G(V, E)$, k).

1: **Input**: Graph $G(V, E)$ and parameter k
2: **Output**: "Yes" if G has a k-Path, "No" otherwise.
3:
4: For each node i, pick a random vector $v_i \in \mathbb{Z}_2^k$
5: $P = 0$
6: **for** $t = 0$ to 2^{k-1}
7: ▸ **Base case**
8: **for** $i \in V$ **do**
9: $P(i, 1) = 1 + (-1)^{v_i^T \cdot t_{\text{bin}}}$
10: ▸ **Inductive step**
11: **for** $i \in V, j = 2$ to k **do**
12: $P(i, j) = \sum_{u \in \text{NBR}(i)} P(i, 1)P(u, j - 1)$
13: $P(k) = \sum_i P(i, k)$ for $i \in V$
14: $P = P + P(k) \mod 2^{k+1}$
15: **return** "Yes" if $P \neq 0$, else "No"

4. Proposed parallel algorithm MIDAS for k-path

Opportunities and challenges for parallelization. Part of the outer for loop in lines 6–14 involves iterations that are uncoupled, in the sense that they can be done separately, as long as we are able to sum up the result P from each iteration, modulo 2^{k+1}. This gives us an easy source of parallelism, namely, run each iteration in parallel. However, this approach would not be viable if the graph does not fit in one processor's memory. Further, even for small graphs that do fit in main memory, we find that parallelizing the node computation improves running time (Section 7). The computation in the inductive step of Algorithm 1 has a *local* structure: a vertex only needs to data from its immediate neighbors in the graph. An alternative approach is to partition the graph into N parts, and then try to parallelize the local computation. However, neither extreme works well, and

Table 1
Summary of notation.

Symbol	Description
N	Total number of processors
N_1	Number of parts in graph partitioning
N_2	Size of each phase
Phase	Group of N_2 iterations for which communication is done simultaneously
Batch	A set of N/N_1 phases
$\mathcal{P}$	Partition of G in N_1 parts, $G^1, \ldots, G^{N_1}$

instead, MIDAS partitions the graph into N_1 parts, and runs N/N_1 iterations in parallel. This approach can lead to significant savings on the computation time, but it has a high communication overhead. Since the values exchanged between nodes are small, we introduce an idea of combining the communications of multiple iterations together as a way to reduce the overhead.

4.1. Overview of algorithm MIDAS

Let N denote the total number of processors or parallel units available. Quantities N_1 and N_2 are parameters for controlling the parallelism in different parts of the algorithm. We assume $2^k/N_2$ and N/N_1 are integers, in order to avoid cluttering the notation using ceiling and floor of these quantities. The algorithm involves solving a dynamic program 2^k times; these 2^k loops are independent, and we divide them into *phases* of size N_2 each, so that a total of $2^k/N_2$ phases have to be run. These are run in $2^k/(N_2N/N_1)$ *batches*, where each batch involves running N/N_1 phases. A phase involves a call to the subroutine ParEvaluatePolynomialPath. See Fig. 2 for an illustration of this structure and Table 1 for a summary of notation.

We partition the graph G into N_1 parts, denoted by $\mathcal{P} = \{G^1, \ldots, G^{N_1}\}$; desirable properties of the partition will be discussed later. For a partition j, let Deg(j) be the *degree* of j, defined as the number of edges connecting nodes in j to nodes in some other partition:

$$\text{Deg}(j) = |\{(u, v) : (u, v) \in G, u \in G^i, v \notin G^i\}|,$$

and let maxdeg $= \max_j \text{Deg}(j)$. Also, let maxload $= \max_j |G^j|$ be the maximum "load" or number of vertices on any partition. We will analyze the performance of our algorithm in terms of maxload and maxdeg.

We describe the main steps of MIDAS below.

1. The algorithm starts with the partitioning $\mathcal{P}$ of the graph G.
2. The algorithm runs $\log 1/\epsilon \log 5/4$ rounds, each of which involves 2^k iterations. Here, $\epsilon \in (0, 1)$ is a parameter, which governs the success probability.[1] Each such round with 2^k iterations is partitioned into $2^k/N_2$ phases in the while loop in lines 8–12 of MIDAS, which are completely independent of other phases.
3. In the tth phase, Algorithm ParEvaluatePolynomialPath uses a vector of size N_2 to store polynomials $\langle P(i, tN_2, j), \ldots, P(i, (t+1)N_2 - 1, j)\rangle$ for each node i and $j \in [1, k]$. $P(i, q, j)$ corresponds to the polynomial of node i and degree j for the qth diagonal element in the matrix representation.
4. In the tth phase, for each node i, we use the vector for each neighbor u of i to compute $\langle P(i, tN_2, j), \ldots, P(i, (t+1)N_2 - 1, j)\rangle$. If u is in the same partition, then its data is available on that processor. For every neighbor u in a different partition, u has to send a message with $\langle P(u, tN_2, j-1), \ldots, P(u, (t+1)N_2 - 1, j-1)\rangle$, introducing a communication overhead.
5. We use $\text{sum}_t^\ell = \sum_{i\in V} P(i, tN_2, k) + \cdots + \sum_{i\in V} P(i, (t+1)N_2 - 1, k)$ to denote the sum of the polynomial evaluations for phase t, within round ℓ. These are summed up over all the phases within round ℓ to compute the total, denoted by P^ℓ.

Algorithm 2 MIDAS(G, k, ϵ, N_1, N_2).

1: **Input**: Graph $G = (V, E)$, parameter k, confidence parameter $\epsilon \in (0, 1)$, parameters N_1 and N_2, which guide the parallelism.
2: **Output**: "Yes" if G has a k-Path, "No" otherwise.
3:
4: Let $v_i \in \mathbb{Z}_2^k$ be a random vector for each node i
5: Let $P = 0$ be the polynomial
6: Let N_1 denote the number of processors used for each iteration. Let $\mathcal{P} = \{G^1, \ldots, G^{N_1}\}$ denote the corresponding partition of the graph into N_1 parts.
7: **for** $\ell = 1$ to $(\log 1/\epsilon)/(\log 5/4)$
8: $P^\ell = 0$
9: **while** $s \le \frac{2^k/N_2}{N/N_1}$ **do**
10: **for** $t = sN/N_1$ to $(s+1)N/N_1$ **do in parallel**
11: $\text{sum}_t^\ell = \text{ParEvaluatePolynomial}(G, k, \mathbf{v}, t, N_2, N_1, \mathcal{P})$
12: MpiBarrier
13: $P^\ell = P^\ell + \sum_{t=sN/N_1}^{(s+1)N/N_1} \text{sum}_t^\ell \mod 2^{k+1}$ (MpiReduce)
14: **if** $P^\ell \neq 0$ for some ℓ
15: **return** "Yes"
16: **else**
17: **return** "No"

Algorithm 3 ParEvaluatePolynomialPath($G, k, \mathbf{v}, t, N_2, N_1, \mathcal{P}$).

1: **Input:** Graph G, parameter k, random assignment $\mathbf{v}$, phase number t, number of iterations within phase N_2, number of partitions N_1, and partitioning $\mathcal{P}$
2: **Output:** The value of the polynomial corresponding to k-path in the iterations within a phase
3:
4: **for** each processor s **do in parallel**
5: **Base case**
6: **for** node $i \in G^s$ and iteration $q \in [tN_2, (t+1)N_2 - 1]$ **do**
7: $P(i, q, 1) = 1 + (-1)^{v_i^T \cdot q_{\text{bin}}}$
8: **Recursive step**
9: **for** $j = 2$ to k **do**
10: **for** node $i \in G^s$ **do**
11: **for** all q set $P(i, q, j) = 0$
12: **for** each incoming message $\langle u, P(u, q, j-1)\rangle$ **do**
13: $P(i, q, j) = P(i, q, j) + P(i, q, 1)P(u, q, j-1)$
14: **Send result to neighbors**
15: **for** $u \in \text{Nbr}(i) \setminus G^s$ **do**
16: **Send** $\langle i, P(i, q, j)\rangle$
17: MpiBarrier
18: **return** $\sum_q \sum_i P(i, q, k)$

4.2. Computation and communication complexity

Recall the definition of maxdeg corresponding to the partitioning $\mathcal{P}$. Further, let c_1 and c_2 denote the time for unit computation at any node in G and the unit communication along any edge, respectively, in the Algorithm ParEvaluatePolynomialPath. The time and communication complexity of algorithm MIDAS is summarized below in terms of these parameters.

Theorem 2. *For any $\epsilon \in (0, 1)$, Algorithm* MIDAS *solves the k-Path problem for an instance G, k with probability at least $1-\epsilon$. The total time for computation and communication are*

[1] As per Theorem 1, if there exists a k-path in the graph, we find it with probability 1/5, so we need to run the algorithm multiple times

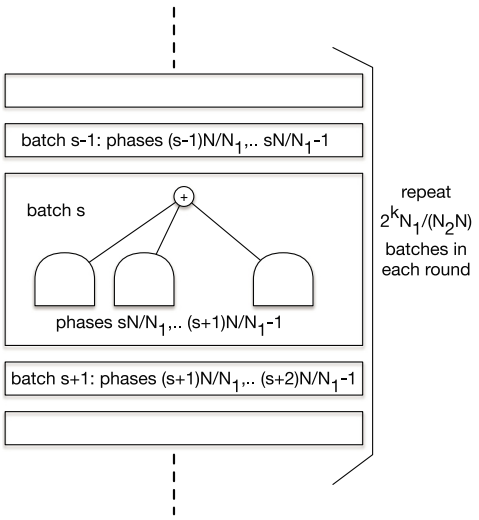

Fig. 2. Schematic structure of MIDAS: we run $(\log 1/\epsilon)/(\log 5/4)$ rounds. Each round is partitioned into $2^k N_1/(N_2 N)$ batches, and each batch involves N/N_1 phases being run simultaneously. Each phase involves an evaluation of the polynomial on N_2 iterations in algorithm ParEvaluatePolynomial, which are then summed up.

$O\left(c_1 \frac{2^k N_1}{N} k\text{MAXLOAD} \log 1/\epsilon\right)$ *and* $O\left(c_2 \frac{2^k N_1}{N N_2} k\text{MAXDEG} \log 1/\epsilon\right)$*, respectively.*

Proof (*Sketch*)**.** First, we argue the correctness. The call to ParEvaluatePolynomialPath evaluates the polynomial bottom up in parallel for all iterations in the tth phase, namely iterations $tN_2, \ldots, (t+1)N_2 - 1$. The vector $\langle P(tN_2, k), \ldots, P((t+1)N_2 - 1, k)\rangle$ is the final evaluation of the polynomial for each iteration in this phase. Each call to ParEvaluatePolynomialPath in MIDAS returns the sum of these values for phase t. Each round of MIDAS, corresponding to the value of ℓ in the outer for loop in lines 6–12 in the sequential algorithm, goes over all phases, and calls ParEvaluatePolynomialPath. Therefore, within round ℓ, P^ℓ denotes the sum of the polynomial evaluation over all the 2^k iterations within that round. If $P(\cdot)$ has a multilinear term, from [24,42], $P^\ell \neq 0 \bmod 2^{k+1}$ with probability at least 1/5. This implies that $\Pr[P^\ell = 0,\ \forall \ell] = (\frac{4}{5})^{(\log 1/\epsilon)/(\log 5/4)} \leq \epsilon$, so that with probability at least $1 - \epsilon$, MIDAS returns "Yes" if G has a k-path. On the other hand, if $P(\cdot)$ has no multilinear term, then with probability 1, $P^\ell = 0$ for all ℓ. Therefore, MIDAS correctly solves the k-Path problem with probability at least $1 - \epsilon$.

Next, we consider the computation and communication time complexity. The algorithm ParEvaluatePolynomialPath computes the polynomial for each degree up to k within each iteration. Therefore, the computation time in a phase t is $O(c_1 k \max_j |G^j| N_2) = O(c_1 k\text{MAXLOAD} N_2)$, which is the maximum time for any processor. Therefore, the total compute time over all the rounds is $O(\frac{2^k/N_2}{N/N_1} c_1 k\text{MAXLOAD} N_2)$, which corresponds to the bound in the theorem. After the evaluation in the recursive step, the results have to be sent on all neighbors, for every pair of processors s, s'. Therefore, the maximum number of messages in one iteration of the loop in lines 8–15 is MAXDEG, and the total number of messages, over all the rounds, is $O(\frac{2^k/N_2}{N/N_1}\text{MAXDEG}) = O(\frac{2^k N_1}{N N_2}\text{MAXDEG})$. □

The recursive step in ParEvaluatePolynomialPath has properties of a highly memory-bound process. Recall that the polynomial multiplication terms are summed up for each of the incoming messages. This computation loop may be subject to locality effects of the memory sub-system and pipelining by the logical processor. Therefore, selecting appropriate values for N_2 is important to leverage fast memory access[2] and achieve the desired parallel performance.

Lemma 1. *For a graph $G = (V, E)$ drawn from the Erdös–Renyi model, $G(n, p)$, the computation and communication times for a random partition are $O(c_1 \frac{2^k nk}{N} \log 1/\epsilon)$ and $O(c_2 \log 1/\epsilon \frac{2^k mk}{N N_2})$, respectively, with high probability.*

Proof (*Sketch*)**.** For a random partition into N_1 parts of equal size, we have $\text{MAXLOAD} = n/N_1$, and the bound for the total compute time follows from Theorem 2. Since $G \in G(n, p)$, we have that $\text{MAXDEG} = O(\frac{n}{N_1}(n - \frac{n}{N_1} p)) = O(m/N_1)$, with high probability, and the lemma follows. □

4.3. Witness extraction

In the case that the input graph to MIDAS does contain a k-path, we may be interested in retrieving the nodes in the path. We can recover this subgraph by calling MIDAS multiple times. As a naive procedure, we can remove a node v and all its incident edges from the input graph, and then we can use MIDAS to check whether the resulting graph still contains a k-path. If the answer is "Yes", the node is not necessary to obtain a k-path in G; otherwise, we put the node back in the graph. We continue until we have removed everything but the nodes in a k-path.

It is easy to verify that the process above takes $O(n)$ sequential calls to MIDAS, which would be very inefficient for large graphs. Instead, we use the *witness extraction* algorithm of Bjorklund et al. [9], which requires $O(k \log n)$ queries in expectation. The algorithm has two phases. In the first phase, the graph is recursively split into two halfs, and we try to discard one of these splits if a call to MIDAS returns "Yes". Because MIDAS returns "No" with some probability even when there is a k-path, at the end of this phase, we will end up with a superset S' of the target subgraph. Then, in phase 2, we try to remove each node in S' until we are left with only the k-path. For completeness, the algorithm of [9] is presented below in Algorithm 4.

5. Extension to trees of size *k*

We now describe how MIDAS for the k-path problem can be extended to trees. We discuss how the corresponding poly-

[2] Cache affinity in terms of spatial and temporal locality results in fast memory access.

Algorithm 4 WitnessExtraction(G, k, ϵ, N_1, N_2).

Input: Graph $G = (V, E)$ containing a k-path, parameter k, confidence parameter $\epsilon \in (0, 1)$, parameters N_1 and N_2 for MIDAS.
Output: Set of nodes S that form a k-path in G

Phase 1
Let $S' = \emptyset$
Let Q be an empty queue
Let $V' = V$
Insert V' into Q
while Q is not empty **do**
  Pop the next element A from Q
  if $|A| = 1$ **then**
    $S' = S' \cup A$
  else
    Let A_1, A_2 be a partition of A, such that $||A_1| - |A_2|| \leq 1$
    Let H be the subgraph of G induced by $V' \setminus A_1$
    if MIDAS(H, k, ϵ, N_1, N_2) returns "Yes" **then**
      ▷ Discard A_1
      Let $V' = V' \setminus A_1$
      Insert A_2 into Q
    else
      Let H be the subgraph of G induced by $V' \setminus A_2$
      if MIDAS(H, k, ϵ, N_1, N_2) returns "Yes" **then**
        ▷ Discard A_2
        Let $V' = V' \setminus A_2$
        Insert A_1 into Q
      else
        Insert A_1 and A_2 into Q

Phase 2
Let S be a queue initialized with all the elements in S'
while $|S| > k$ **do**
  Pop the next element v from S
  Let H be the subgraph of G induced by S
  if MIDAS(H, k, ϵ, N_1, N_2) returns "No" **then**
    Insert v into S
return S

nomials are constructed recursively and evaluated in the subroutine ParEvaluatePolynomialTree; the main Algorithm MIDAS remains unchanged.

5.1. k-tree

We describe how an instance of k-Tree with graph $G = (V, E)$ and tree $H = (V^H, E^H)$ is reduced to a k-MLD instance. We consider the tree H to be rooted, and let root(H) be the root node, selected arbitrarily. We consider a hierarchical structure among subtrees of H in the following manner: consider any node $u \in$ Nbr(root(H)). Let H_1 and H_2 denote the subtrees or *children* obtained upon deleting the edge $(u, \text{root}(H))$, with root(H) $\in H_1$ and $u \in H_2$. We set root(H_1) = root(H) and root(H_2) = u. This process is illustrated in Fig. 3. The subtrees H_1 and H_2 are further partitioned in a recursive manner until all trees have a single node. We define the polynomials $P(i, H')$, which will correspond to all layouts (not necessarily isomorphisms) of H' with root(H') = i for all nodes $i \in V$, as follows:

- If H' consists of a single node, $P(i, H') = x_i$
- Else, $P(i, H') = \sum_{u \in \text{Nbr}(i)} P(i, H'_1)P(u, H'_2)$, where H'_1 and H'_2 are the children of H'.
- Finally, we have $P(x_1, \ldots, x_n) = \sum_{i \in V} P(i, H)$

By using ideas from [3], it can be verified that the tree H is a subgraph of G if and only if the polynomial $P(x_1, \ldots, x_n)$ has a multilinear term. Algorithm ParEvaluatePolynomialTree evaluates this polynomial analogous to Algorithm 5 from Section 4. The performance of MIDAS using ParEvaluatePolynomialTree is summarized below.

Lemma 2. *For any $\epsilon \in (0, 1)$, Algorithm* MIDAS, *using* ParEvaluatePolynomialTree, *solves the k-*Tree *problem for an instance (G, H) with probability at least $1 - \epsilon$. The total time for computation and communication are $O\left(c_1 \frac{2^k N_1}{N} |\mathcal{T}| \text{maxload} \log 1/\epsilon\right)$ and $O\left(c_2 \frac{2^k N_1}{N N_2} |\mathcal{T}| \text{maxdeg} \log 1/\epsilon\right)$, respectively.*

Algorithm 5 ParEvaluatePolynomialTree($G(V, E)$, $H(V^H, E^H)$, $\mathbf{v}$, t, N_2, N_1, $\mathcal{P}$).

Input: Graph $G(V, E)$, tree $H(V^H, E^H)$ with k vertices, random assignment $\mathbf{v}$, phase number t, number of iterations within phase N_2, number of partitions N_1, and partitioning $\mathcal{P}$
Output: The value of the polynomial corresponding to k-tree in the iterations within a phase

Let $\mathcal{T}$ be a collection of subtrees of H sorted by size
for each processor s**do in parallel**
  for each subtree $j \in \mathcal{T}$ **do**
    for node $i \in G^s$ and iteration $q \in [tN_2, (t+1)N_2 - 1]$ **do**
      if $|j| = 1$ **then**
        $P(i, q, j) = 1 + (-1)^{v_i^T \cdot q_{\text{bin}}}$
      else
        set $P(i, q, j) = 0$
        let j' and j'' be the children of subtree j
          for each incoming message $\langle u, P(u, q, j'') \rangle$ **do**
            $P(i, q, j) = P(i, q, j) + P(i, q, j')P(u, q, j'')$
      Send result to neighbors
      for $u \in$ Nbr(i) $\setminus G^s$ **do**
        Send $\langle i, P(i, q, j) \rangle$
MpiBarrier
return $\sum_q \sum_i P(i, q, H)$

More generally, we consider a weighted version of the problem, where the goal is to find an embedding of the maximum weight. We also note that the algorithm can be generalized to find motifs of bounded treewidth, similar to algorithms based on the color coding technique. [4].

6. Extension to network scan statistics

We now consider the problem of optimizing scan statistics on networks, which was defined in Section 2. In this case, we first start with a description of the sequential algorithm, since the problem is much more complex, and several ideas are needed for the reduction to multilinear detection.

As in Section 5, we discuss how the corresponding polynomials are constructed recursively and evaluated in the subroutine ParEvaluatePolynomialScanStat; the main algorithm MIDAS remains unchanged.

Let $W(V) = \sum_{i \in V} w(i)$ be the total weight of the nodes in G. For each node i, we define a variable x_i, and we construct a polynomial over the set of variables $\{x_i : i \in V\}$. Every term – i.e., monomial – in this polynomial will represent a connected subgraph of size at most k and weight at most $W(V)$. For $j \leq k$ and $z \leq W(V)$, let $P_s(i, j, z)$ be the polynomial corresponding to a subgraph (1) containing node i, (2) of size j, and (3) total weight z. We will describe the recurrence relations for these polynomials; these will be implemented in a space efficient manner using the matrix representation, as in the case of k-paths in Section 3.3.

6.1. Basic-MultilinearScan: An optimal, but slow solution

We first give the intuition behind the algorithm Basic-MultilinearScan and the high level steps, before describ-

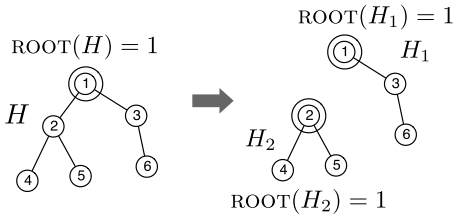

Fig. 3. Tree H with $\text{ROOT}(H) = 1$ is decomposed into trees H_1 and H_2 by removing edge $(1, 2)$: $\text{ROOT}(H_1) = 1$ and $\text{ROOT}(H_2) = 2$.

ing it formally in Algorithm 7.

- *Preprocessing step*: this converts a sequence of graphs with varying attributes to an input graph with event counts, as defined in Section 2. The algorithm starts by performing the graph transformation of [10], which transforms graph G' into a weighted graph G with a weight $w(v)$ for each node v. This corresponds to the **for** loop in lines 4–6 in Algorithm 7, which is run for each possible p-value $\alpha \in A$.
- *Construction and evaluation of the polynomials $P_s(v, i, j)$*: The subroutine ConnectedSubgraphSearch (Algorithm 6) constructs and evaluates the polynomial $P_s(v, i, j)$ for each node v, size $i \in K$ and weight $j \in R$ in the graph G, in line 6 of Algorithm 7. These polynomials are constructed recursively through a dynamic program. Only the results of their evaluation are stored, and not the complete representation of each $P(v, i, j)$. Finally, $P_s^{\alpha}(i, j) = \sum_v P_s(v, i, j)$ is returned to Algorithm Basic-MultilinearScan, for all i, j.
- *Finding the best solution*: The algorithm evaluates $F(j, i, \alpha)$ for each i, j such that $P_s^{\alpha}(i, j) \neq \bar{0}$, and for each α, and returns the best solution in line 7. Let $OPT(F, k) = \max_{S:i\leq k} F(W(S), i, \theta)$ be the optimal solution over connected subgraphs of H with size $i \leq k$. Then, our algorithm returns $OPT(F, k)$ with constant probability.

Algorithm 6 ConnectedSubgraphSearch($G(V, E)$, $\mathbf{w}$, k).

1: **Input**: Instance $(G(V, E), \mathbf{w})$ and parameter k
2: **Output**: Polynomial P_s, such that $P_s(i, j)$ is non-zero if G has a subgraph S with size $i \leq k$ and weight $j \leq W(V)$
3: For each node v, pick a random vector $x_v \in Q[\mathbb{Z}_2^k]$
4: $P_s(v, i, j) = \bar{0}$ for $i \in K, j \in R$
5: **for** $v \in V$ **do**
6: $\quad P_s(v, 1, w(v)) = x_v$
7: **for** $v \in V$, $i = 2$ to k, $j = 0$ to $W(V)$ **do**
8: $\quad P_s(v, i, j) = \sum_{u\in Nbr(v)} \sum_{i'=1}^{i-1} \sum_{j'=0}^{j} (P_s(v, i', j') \cdot P_s(u, i - i', j - j'))$
9: $P_s(i, j) = \sum_v P_s(v, i, j)$ for $i \in K, j \in R$
10: **return** M

Algorithm 7 Basic-MultilinearScan($(G'(V', E'), \mathbf{p}, \alpha_{max}), k$).

1: **Input**: Instance $(G'(V', E'), \mathbf{p}, \alpha_{max})$, parameter k
2: **Output**: Score $OPT(F, k)$
3: Let A be the set of p-values of nodes in V below α_{max}
4: **for** $\alpha \in A$ **do**
5: $\quad$ Run refinement step of [10] to create a weighted graph $G(V, E)$ on "super nodes". Let $w(v)$ denote the weight of node v.
6: $\quad P^{\alpha}$ = ConnectedSubgraphSearch($G(V, E)$, $\mathbf{w}$, k)
7: **return** $\max_{\alpha\in A} \max_{i,j:P_s^{\alpha}(i,j)\neq\bar{0}} F(j, i, \alpha)$

Theorem 3. *Let $F(\cdot)$ be a non-parametric scan statistic, as defined in Section 2. Algorithm* Basic-MultilinearScan *returns $OPT(F, k)$ defined above with probability at least $1/5$, in time $O(2^k|A|mk^2W(V)^2)$, and using space $O(knW(V))$, where A and is defined in line 3 of Algorithm 7.*

Proof. The proof of correctness of the algorithm involves three parts. (1) First, we have to argue that the dynamic program in MultilinearDetectPath correctly constructs the polynomial $P_s(v, i, j)$. (2) Then, we discuss the probability that the $P_s(i, j)$ entry returned by ConnectedSubgraphSearch is non-zero if a subgraph of size i and weight j does exist. (3) And, lastly, we show that our algorithm optimizes $F(\cdot)$ over all $i \in K, j \in R$.

(1) We argue that the procedure ConnectedSubgraphSearch correctly evaluates the polynomial $P_s(v, i, j)$ for a given random assignment of node variables $\mathbf{x}$, for all $v \in V, i \in K$, and $j \leq W(V)$. Our proof is by induction on i. The base case of $i = 1$ is trivial. For $i \geq 2$, the recursion forms a subgraph of size i by combining two subgraphs of size $i' < i$ and $)i - i') < i$, which we have already computed from the inductive step.

(2) We observe that, for any $i \in K$ and $j \in R$, if G does **not** contain a subgraph S with size i and $W(S)$ in group j, then algorithm ConnectedSubgraphSearch correctly sets $P_s(i, j) = \bar{0}$. Otherwise – if the subgraph does exist – by the correctness of ConnectedSubgraphSearch and Theorem 1, $P_s(i, j) \neq \bar{0}$ with probability at least $1/5$.

(3) Now, we argue that Basic-MultilinearScan returns $OPT(F, k)$, This follows because the algorithm tries every combination of i, j, and α for which $P_s^{\alpha}(i, j) \neq \bar{0}$—i.e., a subgraph of size i and weight j exists.

Finally, we analyze the time and space complexity. The space complexity follows from the size of the dynamic programming table, which is $k \times W(V) \times n = O(knW(V))$. For the running time bound, notice that calculating $P_s(v, i, j)$ for some i and j requires summing over all $i' < i$ and all $j' \leq j$; thus, the computation is quadratic in k and $W(V)$. Furthermore, we sum over all neighbors of node v. Therefore, the total running time of ConnectedSubgraphSearch is $O(mk^2 2^k W(V)^2)$, where the 2^k term is the time complexity of operations in the $Q[\mathbb{Z}_2^k]$ group algebra. Finally, ConnectedSubgraphSearch is invoked $|A|$ times by Basic-MultilinearScan, so the running time follows. □

Remark: In practice, the success probability of Basic-MultilinearScan in Theorem 3 can be improved to $1-\delta$ for any $\delta \in (0, 1)$ by simply running the algorithm $O(\log 1/\delta)$ times.

6.2. *Scaling* Basic-MultilinearScan *with logarithmic binning*

The quadratic dependence on $W(V)$ creates a bottleneck in ConnectedSubgraphSearch. To make the computation scalable, we will group the weights of the input graph to ConnectedSubgraphSearch by considering powers of $(1 + \epsilon)$, where $\epsilon > 0$ is an error parameter: if $w(v) \in [(1 + \epsilon)^{j-1}, (1 + \epsilon)^j)$, we say that v is in the weight group j; this definition is extended to the weight of a subgraph. For instance, if $\epsilon = 1$, nodes with weight in $[8, 16)$ are in group 4. The goal is to scale the weights of the input graph down by a logarithmic factor and run Algorithm 6 on this much smaller set of weights.

With a slight abuse of notation, let us redefine R as $\{0, 1, 2, \ldots, r\}$, where r is a *weight parameter*. Now the polynomial $P_s(v, i, j)$ corresponds to a subgraph (1) containing node v, (2) of size i,

and (3) total weight in $[(1+\epsilon)^{j-1}, (1+\epsilon)^{j})$. $P_s(v, i, 0)$ represents subgraphs of weight 0.

We need to modify ConnectedSubgraphSearch to satisfy condition (3). Algorithm 8 shows this modified version. In the base case (lines 4–6), we set $P_s(v, 1, j)$ to be x_v if node v is in group j and 0 otherwise. For $i \geq 2$ (lines 7–15), we consider the following cases:

Case 1. For $j = 0$, a polynomial $P_s(v, i, 0)$ represents connected subgraphs of weight 0 only. For a graph of size i to have weight 0, its two parts, of size i' and size $i - i'$ must both have weight 0 as well.

Case 2. For $j = 1$, a polynomial $P_s(v, i, 1)$ represents connected subgraphs of weight 1 only. For a graph of size i to have weight 1, one of its two parts must have weight 1, and the other must have weight 0.

Case 3. For $j \geq 2$, a polynomial $P_s(v, i, j)$ represents connected subgraphs of weight between $(1+\epsilon)^{j-1}$ and $(1+\epsilon)^{j}$. We could obtain a subgraph of size i with this weight in one of two ways. The first is to combine two subgraphs, each of weights between $(1+\epsilon)^{j-2}$ and $(1+\epsilon)^{j-1}$. Thus, the resulting subgraph will have weight at least $(1+\epsilon)^{j-1}$ and at most $(1+\epsilon)^{j}$ (exclusive). The second way is to combine a subgraph with weight between $(1+\epsilon)^{j-1}$ and $(1+\epsilon)^{j}$ with one of weight 0, as in the case for $j = 1$.

Algorithm 8 Approx-ConnectedSubgraphSearch($G, \mathbf{w}, k, r$).

1: **Input**: Instance $(G(V, E), \mathbf{w})$ and parameters k, r
2: **Output**: Polynomial P_s, such that $P_s(i, j)$ is non-zero if G has a subgraph S with size $i \leq k$ and weight group $j \leq r$
3: For each node v, pick a random vector $x_v \in Q[\mathbb{Z}_2^k]$
4: **for** $v \in V$ **do**
5: $\quad j = \lceil \log_{(1+\epsilon)}(w(v) + 1) \rceil$
6: $\quad P_s(v, 1, j) = x_v$
7: **for** $v \in V$, $i = 2$ to k, $j = 0$ to r **do**
8: $\quad$ **if** $j = 0$ **then**
9: $\quad\quad P_s(v, i, 0) = \sum_{u \in Nbr(v)} \sum_{i'=1}^{i} (P_s(v, i', 0) \cdot P_s(u, i - i', 0))$
10: $\quad$ **if** $j = 1$ **then**
11: $\quad\quad P_s(v, i, 1) = \sum_{u \in Nbr(v)} \sum_{i'=1}^{i} (P_s(v, i', 1) \cdot P_s(u, i - i', 0) +$
12: $\quad\quad\quad (P_s(v, i', 0) \cdot P_s(u, i - i', 1))$
13: $\quad$ **if** $j \geq 2$ **then**
14: $\quad\quad P_s(v, i, j) = \sum_{u \in Nbr(v)} \sum_{i'=1}^{i} (P_s(v, i', j - 1) \cdot P_s(u, i - i', j - 1) +$
15: $\quad\quad\quad P_s(v, i', j) \cdot P_s(u, i - i', 0) + P_s(v, i', 0) \cdot P_s(u, i - i', j))$
16: $P_s(i, j) = \sum_v P_s(v, i, j)$ for $i \in K, j \in R$
17: **return** P_s

Now, we present Approx-MultilinearScan in Algorithm 9. This algorithm takes an extra parameter, ϵ, and calls Approx-ConnectedSubgraph Search with weight parameter $r = \lceil \log_{(1+\epsilon)}(k w_{\max} + 1) \rceil$, for $w_{\max}$ defined in line 6. That way, r is the maximum (scaled down) weight of a subgraph of size k. Let $W_{1+\epsilon}(S) = (1+\epsilon)^{\lceil \log_{1+\epsilon}(W(S)+1) \rceil}$ be the approximate weight of S, and let $OPT(F, k, \epsilon) = \max_{S: i \leq k} F(W_{1+\epsilon}(S), i, \theta)$ be the optimal solution over connected subgraphs of G' with size $i \leq k$ and rounded weights. Then, we obtain the following result.

Algorithm 9 Approx-MultilinearScan($(G', \mathbf{p}, \alpha_{max}), k, \epsilon$).

1: **Input**: Instance $(G'(V', E'), \mathbf{p}, \alpha_{max})$, parameters k, ϵ
2: **Output**: Score $OPT(F, k, \epsilon)$
3: Let A be the set of p-values of nodes in V below α_{max}
4: **for** $\alpha \in A$ **do**
5: $\quad$ Run refinement step of [10] to create a weighted graph $G(V, E)$ on "super nodes". Let $w(v)$ denote the weight of node v
6: $\quad$ Let $w_{\max} = \max_v w(v)$ be the maximum weight
7: $\quad$ Let $r = \lceil \log_{(1+\epsilon)}(k w_{\max} + 1) \rceil$
8: $\quad P_s^{\alpha} =$ Approx-ConnectedSubgraphSearch($G, \mathbf{w}, k, r$)
9: **return** $\max_{\alpha \in A} \max_{i,j: P_s^{\alpha}(i,j) \neq 0} F((1+\epsilon)^j, i, \alpha)$

Theorem 4. *Let $F(\cdot)$ be a non-parametric scan statistic, as defined in Section 2. Algorithm* Approx-MultilinearScan *returns $OPT(F, k, \epsilon)$ with probability at least 1/5, in time $O(2^k |A| m k^2 \log_{1+\epsilon}(k w_{\max}))$, and using space $O(kn \log_{1+\epsilon}(k w_{\max}))$, where A and $w_{\max}$ are defined in lines 3 and 6 of Algorithm 9.*

6.3. Approx-MultilinearScan *in Pregel*

We implement Approx-MultilinearScan in the message-passing Pregel model [21], specifically in the Giraph framework. Our parallel algorithm, MultilinearScanGiraph, calls the subroutine ParConnectedSubgraphSearch (Algorithm 10); this is a parallel version of Approx-Connected SubgraphSearch. As in the case of the sequential version, this is described without the more efficient matrix representation—this requires a for loop with 2^k steps around the Pregel call. We have also explored a GraphX [21] version of the algorithm, which has a slightly different structure but performs poorly. We also note that Algorithm 10 differs from standard Pregel algorithms, such as PageRank, RandomWalk, and ConnectedComponents [20,41] in two important aspects. First, in these three algorithms the messages exchanged consists of a single integer or floating point number, whereas our algorithm requires sending integer arrays of size k. Second, in algorithms like PageRank, the number of total messages sent across vertices decreases as the algorithm progresses, whereas, in our case, we need to send a number of messages proportional to the number of edges throughout the entire computation.

Algorithm 10 exploits two levels of parallelism identifiable in the sequential implementation: (*i*) the outer **for** loop, which corresponds to the matrix representation (not shown here) can be easily parallelized; (*ii*) the **for** $v \in V$ loop in line 7 of Approx-ConnectedSubgraphSearch happens in parallel. A mapping of the elements in lines 7 to 15 of the sequential Approx-ConnectedSubgraphSearch algorithm to the Pregel model shown here is as follows. Each super step represents a step of the loop **for** $i = 2$ to k; each vertex within a super step represents the loop **for** $v \in V$; and the loop **for** $j = 0$ to r is carried out inside each vertex within a super step. The summation in line 16 of the Approx-ConnectedSubgraphSearch is carried out through an aggregate operation that collects results to the master vertex at the end of last super step. Note, in the Pregel implementation, a row of vertex computations within a super step occur in parallel, while the super steps follow one after the other.

Lemma 3. *The Giraph implementation,* MultilinearScanGiraph, *using Algorithm 10 runs in $O(2^k |A| k \log_{1+\epsilon}(k w_{\max}))$ super steps and gives a solution with the same guarantees as in Theorem 4.*

6.4. An MPI implementation of Approx-MultilinearScan using MIDAS

Finally, we examine our MPI approach using MIDAS, which is described in Algorithm 11. For completeness, we describe the algorithm using the matrix representation. The algorithm maintains variables $P_s(i, q, j, z)$ for every node $i, j \leq k$, total weight in $[(1+\epsilon)^{z-1}, (1+\epsilon)^{z})$, in iteration q within phase t.

7. Experiments

The experimental study of the proposed parallel algorithms covers the following topics.

- **Effect of partition size.** We analyze performance with partition size as N_1 is varied (Section 7.2).

Algorithm 10 ParConnectedSubgraphSearch($G(V, E)$, $\mathbf{w}$, k, r).

Input: Instance $(G(V, E), \mathbf{w})$ and parameters k, r
Output: Polynomial $P_s(i, j)$, such that $P_s(i, j)$ is non-zero if G has a subgraph of size $i \leq k$ and weight group $j \leq r$

For each node v, pick a random vector $x_v \in \mathbb{Z}_2^k$
return P_s = **pregel**($G(V, E)$, $\mathbf{w}$, $\mathbf{x}$, k, r, MCompute, VCompute)

Procedure MCompute(ℓ, k)
  Input: Super step number ℓ and parameter k
  if $\ell = k$ **then**
    $P_s =$ **GetAggregatedMessages**()
    for $v \in V$ **do**
      $P_s(i, j) = \sum_v P_s(v, i, j)$ for $i \in K, j \in R$
      Halt()
    return P_s

Procedure VCompute($\mathcal{M}, \ell, w(v), x_v, k, r$)
  Input: Set of messages $\mathcal{M}$ from set $Nbr(v)$ of neighbors of node v, super step number ℓ, node variables $w(v)$ and x_v, and parameters k, r.
  if $\ell = 0$ **then**
    $P_s(v, 1, j) = 0$, for $j \leq r$
    $P_s(v, 1, \lceil \log_{(1+\epsilon)}(w(v) + 1) \rceil) = x_v$
  else
    Compute $P_s(v, i, j)$ using the recurrence as in Approx-ConnectedSubgraphSearch
  if $\ell < k - 1$ **then**
    for $u \in Nbr(v)$ **do Send**($P_s(v, \cdot, \cdot)$, u)
  else
    Aggregate($P_s(v, \cdot, \cdot)$)
    VoteToHalt()

Algorithm 11 ParEvaluatePolynomialScanStat($G(V, E)$, k, $\mathbf{w}$, $\mathbf{v}$, ϵ, t, N_2, N_1, $\mathcal{P}$).

Input: Graph $G(V, E)$, parameter k, node weights $\mathbf{w}$, random assignment $\mathbf{v}$, parameter ϵ for logarithmic binning, phase number t, number of iterations within phase N_2, number of partitions N_1, and partitioning $\mathcal{P}$
Output: The value of the polynomial corresponding to the scan statistics in the iterations within a phase

for each processor s **do in parallel**
  for node $i \in G^s$ and iteration $q \in [tN_2, (t + 1)N_2 - 1]$ **do**
    $z = \lceil \log_{(1+\epsilon)}(w(i) + 1) \rceil$
    $P(i, q, 1, z) = 1 + (-1)^{v_i^T \cdot q_{\text{bin}}}$
  for $j = 2$ to k and $z = 0$ to $\lceil \log_{(1+\epsilon)}(kw_{\max} + 1) \rceil$ **do**
    Initialize $P(i, q, j, z) = 0$
    if $z = 0$ **then**
      for each incoming message $\langle u, P(u, q, j - j', 0) \rangle$ **do**
        $P(i, q, j, 0) = P(i, q, j, 0) + P(i, q, j', 0)P(u, q, j - j', 0)$
    if $z = 1$ **then**
      for each incoming message $\langle u, P(u, q, j - j', 0) \rangle$, $\langle u, P(u, q, j - j', 1) \rangle$ **do**
        $P(i, q, j, 1) = P(i, q, j, 1) + P(i, q, j', 1)P(u, q, j - j', 0)$
        $+ P(i, q, j', 0)P(u, q, j - j', 1)$
    if $z \geq 2$ **then**
      for each incoming message $\langle u, P(u, q, j - j', 0) \rangle$, $\langle u, P(u, q, j - j', z - 1) \rangle$ **do**
        $P(i, q, j, z) = P(i, q, j, z) + P(i, q, j', z - 1)P(u, q, j - j', z - 1)$
        $+ P(i, q, j', z)P(u, q, j - j', 0) + P(i, q, j', 0)P(u, q, j - j', z)$
    Send result to neighbors
    for $u \in \text{Nbr}(i) \setminus G^s$ **do**
      Send $\langle i, P(i, q, j, z) \rangle$
  MpiBarrier
return $\sum_q \sum_i P(i, q, k, z)$ for all $z \leq \lceil \log_{(1+\epsilon)}(kw_{\max} + 1) \rceil$

Table 2
Datasets used in our experiments.

Dataset	Nodes ($\times 10^6$)	Edges ($\times 10^6$)
miami	2.1	51.5
com-Orkut	3.1	234.3
random-1e6	1	13.8
random-1e7	10	161.8

- **Scalability with subgraph size.** We depict the total runtime as subgraph size is increased (Section 7.3).
- **Strong scaling.** We show the total runtime as the number of parallel processors is increased, thereby reducing the computing workload per process (Section 7.4).
- **MIDAS vs. FASCIA.** We present the runtime of our implementation compared to FASCIA (Section 7.5).
- **Scan Statistics and its applications.** We provide performance results for the parallel scan statistics algorithm and present an application (Section 7.6).

7.1. Experimental setup

7.1.1. Hardware

Experiments were conducted on Juliet, an Intel Haswell HPC cluster. Up to 32 nodes were used for the evaluation, where each node has 36 cores (2 sockets × 18 cores each). A node consists of 128GB of main memory and 56Gbps Infiniband interconnect. We also tested on another HPC cluster, Shadowfax–Haswell, where we used 32 nodes each with 32 cores (2 sockets × 16 cores each). Memory and interconnect of this cluster are similar to those of Juliet.

7.1.2. Datasets

We evaluate our algorithms on two large graphs: (1) a social contact network of Miami, commonly used in agent-based simulation studies [7], and 2) a snapshot of the Orkut social network.[3] In addition, we perform experiments in two Erdös–Renyi networks of 1 and 10 million nodes with an expected number of edges of $n \log n$, where n is the number of nodes. A summary of the datasets is provided in Table 2.

7.2. Effect of partition size

Our parallel k-Path algorithm exhibits two levels of parallelism: vertex and iteration. On one hand, the 2^k iterations are pleasingly parallel except for a global reduction at the end. The parallel vertex computation within each iteration, on the other hand, requires message passing between neighbors for $k - 1$ steps—in the case of trees, this would be the number of subtemplates instead of $k - 1$.

Given these two levels of parallelism and a total of N processes, we can split the 2^k iterations among $a = N/N_1$ parallel phases. Each phase decomposes the graph across N_1 processes and performs the k-path computation in parallel for $2^k/a$ of the 2^k iterations. To reduce the communication over computation cost, the algorithm packs a user defined N_2 number of iterations into one computation step, so each parallel phase only has to perform $2^k/(aN_2)$ compute and communication phases. To illustrate this with an example, consider the case of $k = 6$, $N = 128$, $N_1 = 32$, and $N_2 = 8$. The total number of iterations is $2^k = 64$. The number of parallel phases corresponding to $N_1 = 32$ is $a = N/N_1 = 128/32 = 4$. Each phase only needs to run $2^k/a = 64/4 = 16$ iterations. Since $N_2 = 8$, the 16 iterations can be completed in just $16/8 = 2$ batches.

[3] https://snap.stanford.edu/data/com-Orkut.html

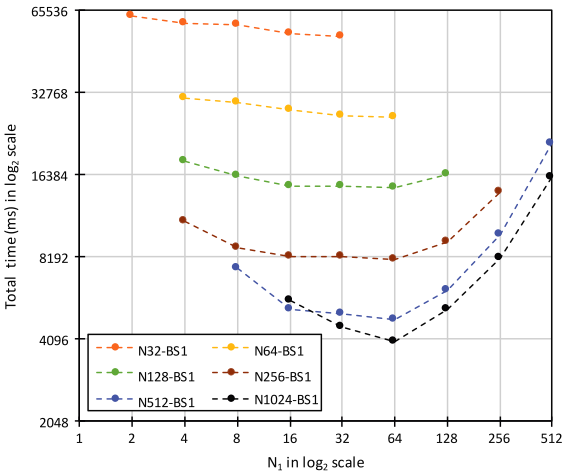

Fig. 4. k-path total runtime for random-1e6 and varying N_1. Note. $BS1 = N_2 = 1$.

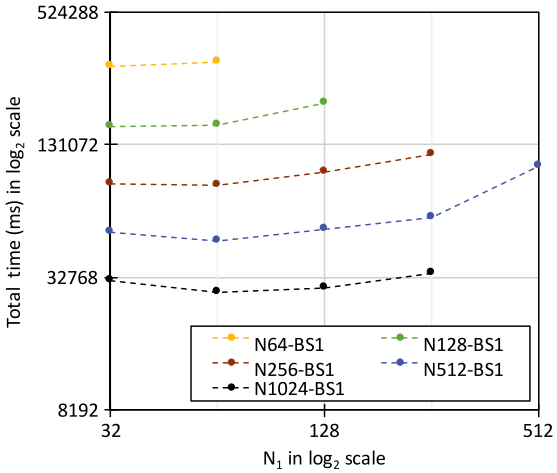

Fig. 5. k-path total runtime for com-Orkut and varying N_1. Note. $BS1 = N_2 = 1$.

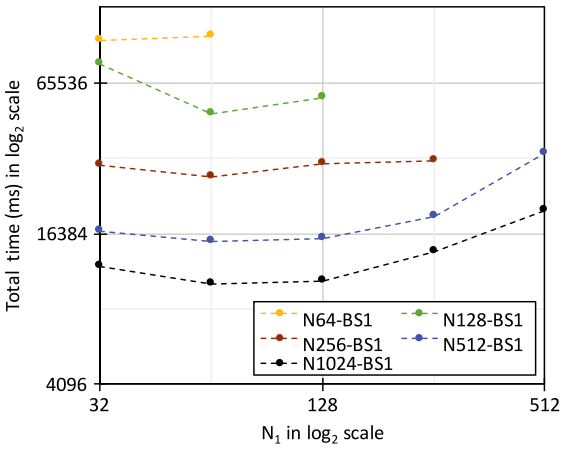

Fig. 6. k-path total runtime for miami network and varying N_1. Note. $BS1 = N_2 = 1$.

To understand the effects of N_1 and N_2 on performance for a given number of processes N, we conduct two types of experiments. In the first, N_1 is varied in the k-path algorithm for different networks. In Figs. 4–9, we show the total running time as a function of N. We further divide this set of experiments such that Figs. 4–6 show results for $N_2 = 1$, whereas Figs. 7–9 use $2^k N_1/N$ as N_2, which is the maximum number of iterations a phase has to perform. In the second type of experiments, we look at the effect of N_2 alone for our k-tree algorithm and for a fixed N. The results are reported in Figs. 10–13.

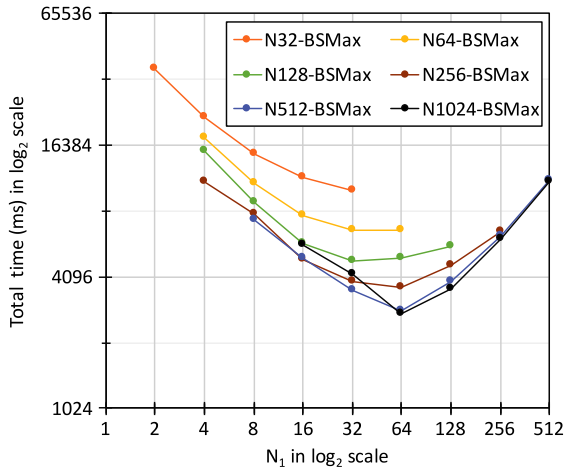

Fig. 7. k-path total runtime for random-1e6 and varying N_1. Note. $BSMax = N_2 = 2^k N_1/N$.

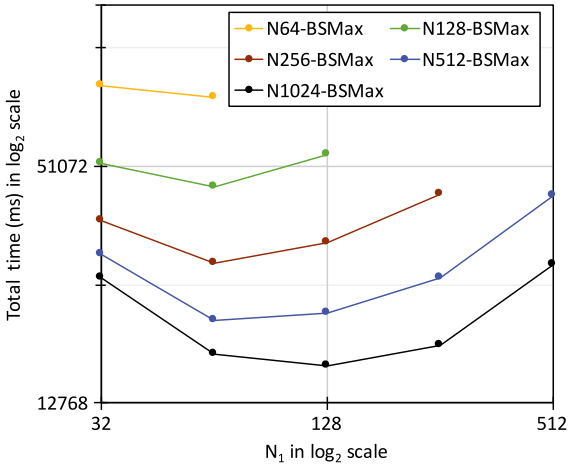

Fig. 8. k-path total runtime for com-Orkut and varying N_1. Note. $BSMax = N_2 = 2^k N_1/N$.

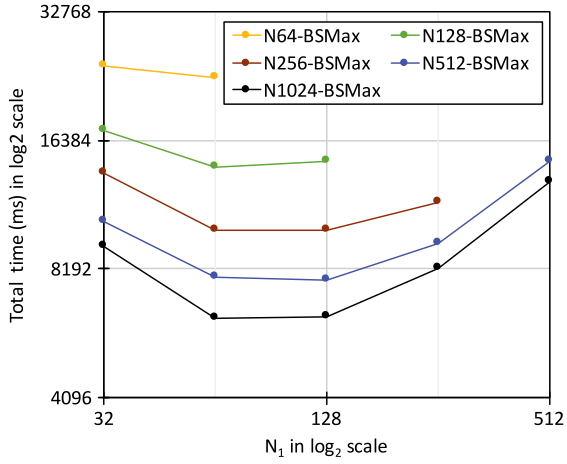

Fig. 9. k-path total runtime for miami and varying N_1. Note. $BSMax = N_2 = 2^k N_1/N$.

For $N_2 = 1$ (Figs. 4–6), we observe the existence of an optimal point (i.e., a minimum) between the two levels of parallelism – vertex and iteration – discussed before. The communication cost gradually increases when moving from one extreme end of parallelism to the other because of the increase in number of messages exchanged[4]. However, at the optimal point, the cost of communication can be sufficiently amortized by the amount of

4 Number of messages exchanged is $O(\log N_1)$ for small message sizes where N_1 ranges from $N_1 = 1$ to N

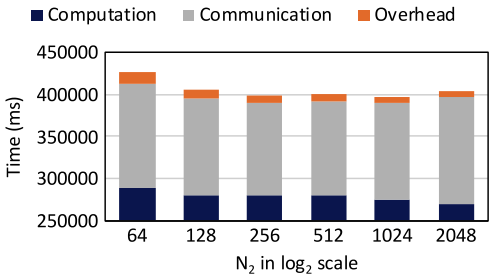

Fig. 10. MIDAS time breakdown for the k-tree problem with increasing N_2 and fixed $N_1 = 512$.

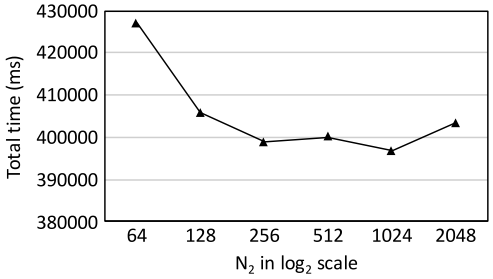

Fig. 11. MIDAS total time for the k-tree problem with increasing N_2 and fixed $N_1 = 512$.

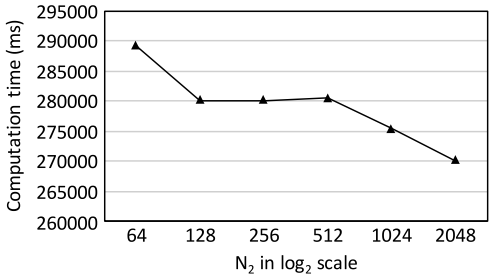

Fig. 12. MIDAS compute time for the k-tree problem with increasing N_2 and fixed $N_1 = 512$.

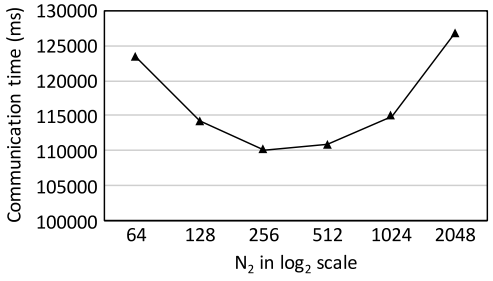

Fig. 13. MIDAS communication time for the k-tree problem with increasing N_2 and fixed $N_1 = 512$.

parallelism gained. In other words, the optimal setting for MIDAS can be found in a point between vertex and iteration parallelism.

In the $N_2 = 2^k N_1/N$ case (Figs. 7–9), we observe relative performance gains compared to $N_2 = 1$. To further understand this effect of N_2, we perform the second type of experiments involving N_2 only. Figs. 10–13 provide the breakdown of communication, computation, and total times while keeping N_1 fixed at 512 processors. We observe in the figures that computation time is reduced as N_2 is increased. This is expected due to the cached use of incoming messages in our algorithms. The communication time reduces too, but it increases again after a certain point, due in part to the increase in message size when N_2 increases.

Similar to N_1, N_2 also has an optimal point in performance as seen in the total time graph in Fig. 11. From these experiments, we find that keeping N_1 closer to the number of cores in one or two machines and N_2 closer to the maximum number of iterations of a phase leads to better performance. However, a more formal characterization of the trade-off between N_1 and N_2 is a topic for future work.

7.3. Scalability with subgraph size

In Fig. 16, we increase the subgraph size, k, in both FASCIA and MIDAS, while keeping N and N_1 fixed. Figs. 7–9 suggest

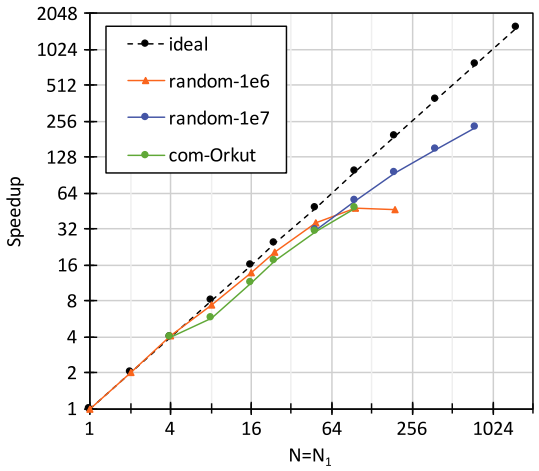

Fig. 14. MIDAS strong scaling for the k-path problem with increasing N and $N_1 = N$.

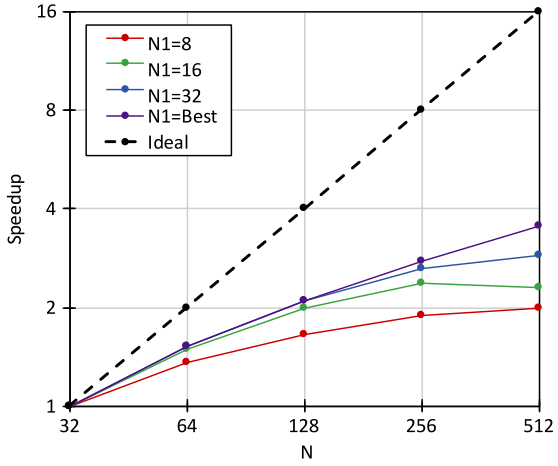

Fig. 15. MIDAS strong scaling for the k-path problem with increasing N, while N_1 is fixed.

it is best to keep N_2 as high as possible to leverage the cache locality benefits discussed above. However, the total message size communicated out of a process increases with N_2 leading to diminishing returns. Therefore, we have kept $N_2 < 1024$.

7.4. Strong scaling

The strong scaling of MIDAS can be investigated in two ways. The first is to fix N_1 and change N, thereby increasing the parallel phases to split the 2^k iterations. We could observe the effect of this behavior by examining the values along a fixed N_1 value in Figs. 4–9. Dividing the runtime corresponding to the minimum N (the top most line) by the given N gives speedup indicating the strong scalability of MIDAS.

Fig. 15 presents such speedup for a set of N_1 values over varying N. We observe the results do not necessarily scale linearly due to the fact that communication within a phase is dominant. We get the best speedup by going along points in Figs. 4–9 that gave the minimum runtime. This is shown as the $N1 =$ Best line in Fig. 15.

The other form of strong scalability we can test is by setting $N1 = N$. This produces a single phase and is the classic strong scaling of parallel graph algorithms. Fig. 14 presents the speedup values for different datasets. Even though the speedups are less than ideal, we still observe good performance up to a considerable number of processes.

7.5. MIDAS *vs.* *FASCIA*

Fig. 16 compares the running time of FASCIA to MIDAS for varying subgraph sizes. We see FASCIA fails to process subgraphs larger than 12 nodes on the random-1e6 graph, whereas MIDAS scales to over 18. Also, MIDAS shows a significant improvement over FASCIA in runtime. Fig. 17 shows MIDAS performance for different tree templates on random-1e6 and random-5e6. The details of the templates can be found in the FASCIA papers [36,37]. We note that the running times reported in Fig. 16 are for the version of MIDAS optimized for finding paths, whereas Fig. 17 corresponds to the algorithm for k-tree. The latter requires keeping track of more information in the dynamic programming step to decompose the target tree. Thus, the running times reported for paths in Fig. 17 are higher than those of comparable size in Fig. 16.

To further analyze the running time of MIDAS, we compare MIDAS to FASCIA, as shown in Fig. 18. MIDAS performs better and, particularly, it scales to larger templates than FASCIA. However, we observe that a single iteration of FASCIA is faster than one iteration of MIDAS as observed in Fig. 19. In implementing our algorithm, we have developed a generalized vertex-centric computing system, which made it possible to quickly implement k-path, k-tree, and scan statistics algorithms by only modifying the vertex logic. This kind of generality seem to cause the slightly lower performance per iteration compared to FASCIA.

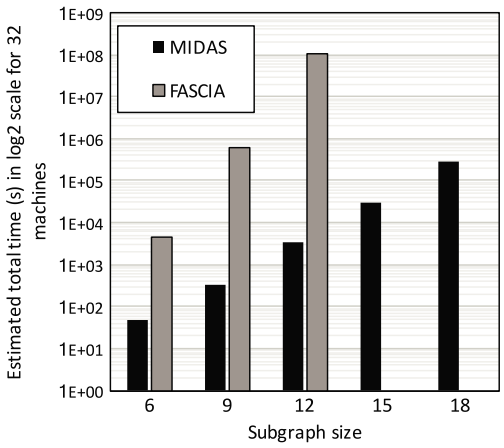

Fig. 16. MIDAS runtime compared to FASCIA for varying subgraph sizes in the k-path problem.

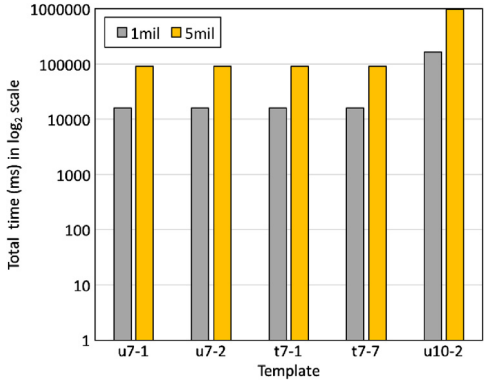

Fig. 17. k-tree total runtime for random 1e6 and 5e6 for different templates for $N = 16$.

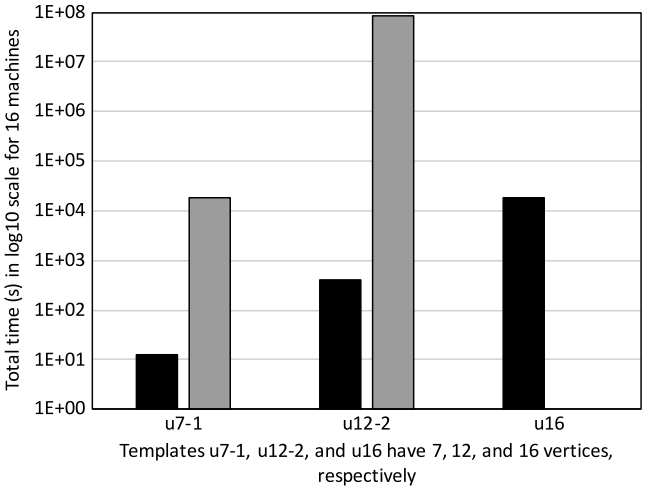

Fig. 18. MIDAS k-tree runtime compared to FASCIA for different templates.

7.6. Scan statistics optimization

7.6.1. MultilinearScanGiraph

First, we discuss the scalability of our Giraph algorithm. The only prior work on parallel algorithms for scan statistics is by Zhao et al. [43]. However, the authors show results only for datasets with up to 12,000 nodes. For comparison purpose, we consider 6 state-of-the-art sequential algorithms for scan statistics optimization:

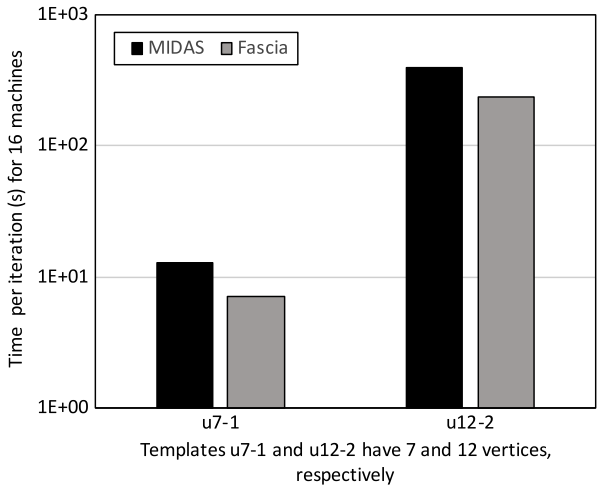

Fig. 19. Per iteration time of MIDAS k-tree compared to FASCIA for different templates.

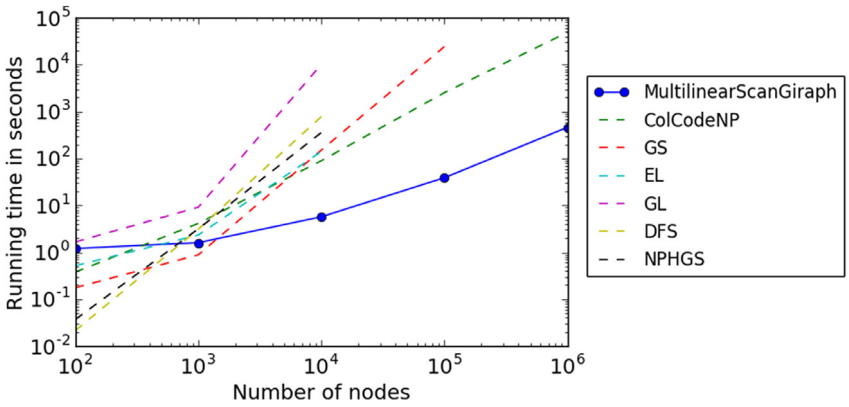

Fig. 20. Running times of state-of-the-art graph scan statistics algorithms.

1. `NPHGS` [12]
2. `AdditiveGraphScan` [39]
3. `DepthFirstScan` [38]
4. `GraphLaplacian` [35]
5. `EdgeLasso` [34]
6. `ColCodeNP` [10]

In Fig. 20, we show the running time of MultilinearScanGiraph and the baseline methods on a set of Erdős–Rényi graphs. MultilinearScanGiraph is the only one besides `ColCodeNP` – also a fixed-parameter tractable algorithm – that is able to process the instance of 10^6 nodes. All the other algorithms run out of memory or do not finish running even after 24 h.

We also analyze the performance of MultilinearScanGiraph as a function of k, the subgraph size. Fig. 21(a) shows the running time for two of the largest graphs found in SNAP, and Fig. 21(b) shows the variation of the running time with the number of nodes for random networks for different values of k. The communication cost in Giraph is the main bottleneck in our algorithm, which has a very different computing pattern from standard benchmarks for distributed systems. For instance, PageRank algorithms stop sending messages to specific nodes as the algorithm proceeds, when certain criteria are met, leading to reduced communication over time. In contrast, in our algorithm, in each super step, vertices always send the same amount of data to neighbors, which contributes to significant overheads. In as-Skitter, for instance, we observed over 80% overhead with 256 and 512 parallel tasks over 16 machines.

7.6.2. MIDAS

Even though it improves over the sequential state-of-the-art, MultilinearScanGiraph has limited scalability, especially for a distributed implementation. We now focus on MIDAS. In Fig. 22, we present strong scaling results for the scan statistics problem where N_1 is set to N. We do this for multiple datasets and observe considerable strong scalability similar to the k-Path problem in Fig. 14.

7.6.3. CaSe studies

Finding under-vaccinated clusters. We study under-vaccinated clusters in Minnesota, which have become a concern for public health officials in recent years. We use school immunization data from the Minnesota Department of Health.[5] This dataset contains immunization records of 7th-grade students for 870 public and private schools during the academic year 2015–16. For each school, the dataset provides the number of students enrolled

[5] http://www.health.state.mn.us/divs/idepc/immunize/stats/school/.

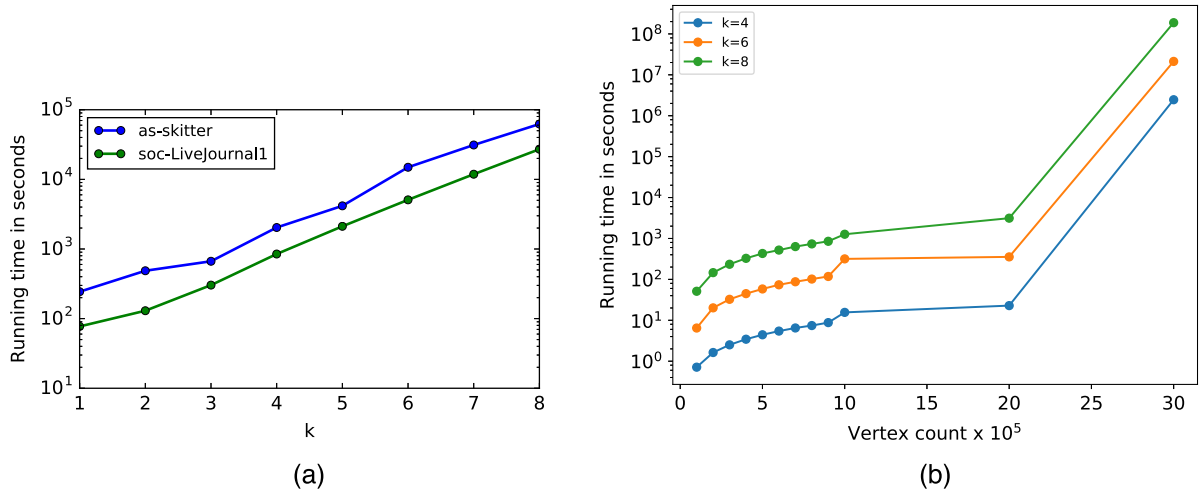

Fig. 21. Running time of MultilinearScanGraph as a function of (a) k for soc-LiveJournal1 and as-Skitter, and (b) number of nodes in random graphs.

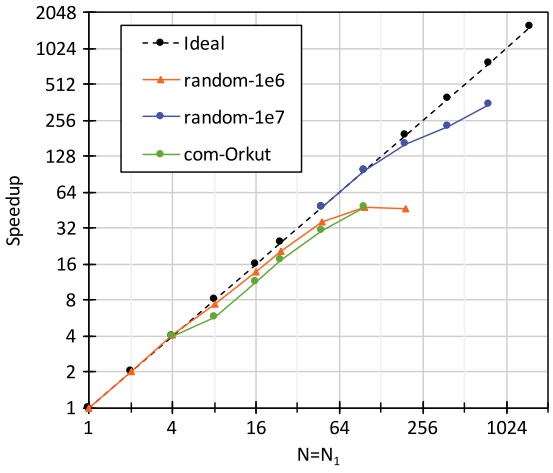

Fig. 22. MIDAS strong scaling for the scan statistics problem with increasing N and $N_1 = N$.

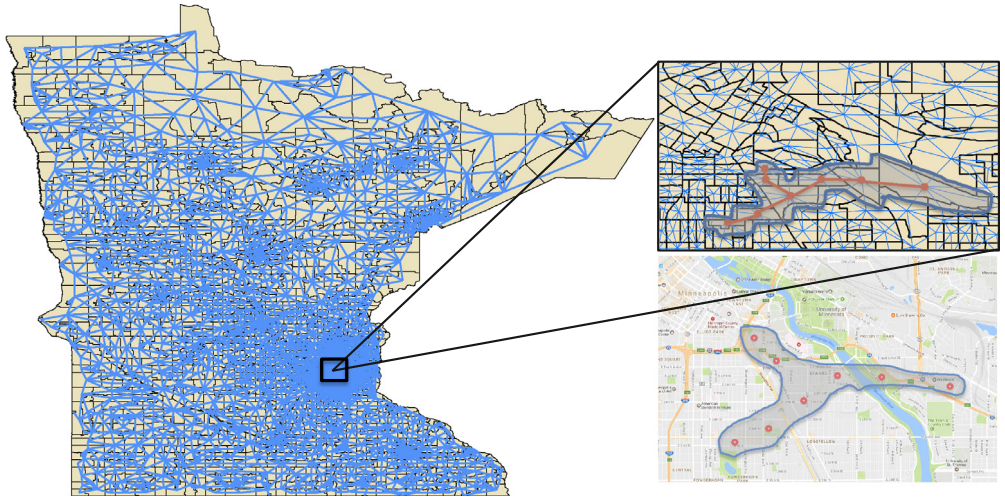

Fig. 23. Low-vaccination clusters in the census block group graph of Minnesota.

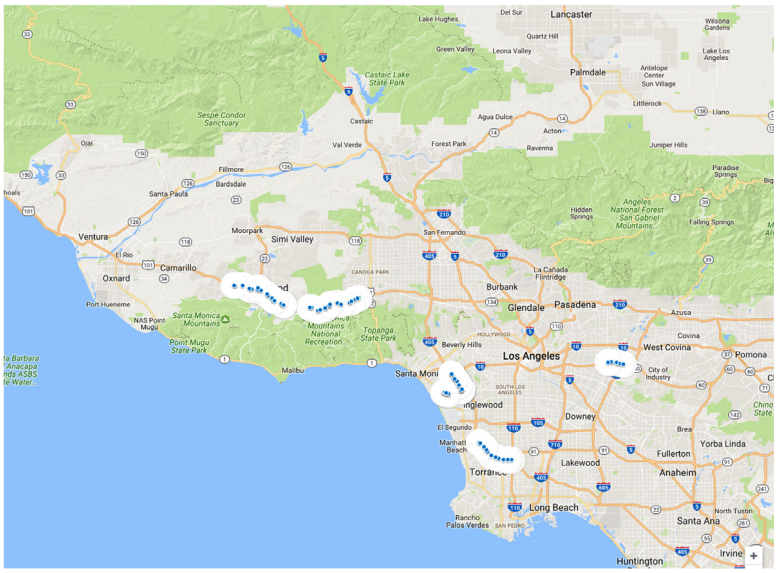

Fig. 24. Discovering highway segments with unexpected congestion in the Los Angeles road network.

in the school and the percentage of students who are vaccinated for various diseases. We focus on the Measles–Mumps–Rubella (MMR) vaccine, which has a 97.5% average vaccination rate statewide in Minnesota.

Our goal is to discover geographical clusters where the vaccination rate for MMR is much lower than the statewide average. We pose this problem as scan statistics optimization in the network of census block groups of Minnesota. We aggregate the school data at the block group level, and we construct a planar graph where nodes are block groups, and there is an edge between every pair of adjacent block groups. For a block group s, let $w(s)$ be the total number of **unvaccinated** children in the schools within s, and let $b(s)$ be the total enrollment. We model $w(s)$ as a sample from a Poisson distribution with parameter proportional to the number of enrolled children: $w(s) \sim Poisson(\lambda b(s))$, where $\lambda = 0.025$ is the statewide average unvaccination rate. Under this model, we derive a p-value for each block: $p(s) = 1 - \Pr(X \leq w(s)|\lambda b(s))$. In words, the p-value for s is the probability of seeing counts as extreme as $w(s)$ in that block group. Then, we find a subgraph with high anomaly score for the BJ scan statistic.

Fig. 23 shows the block group network and a connected subgraph discovered by our method. This cluster is to the south of University of Minnesota and downtown Minneapolis, and the unvaccination rate in the cluster is 6%, 2.5 times higher than expected. All prior work on finding undervaccinated clusters has only considered well-rounded shapes, e.g., [28], which would not find such a cluster.

Traffic congestion. We find clusters with unexpectedly low-moving traffic in the highway network of Los Angeles County.[6] Nodes in the graph are sensors next to the road that record the average speed and the number of vehicles passing through. We have 30-minute snapshots for May 2014. We assume that the average speed recorded by each sensor follows a normal distribution. Then, the p-value of a node i is the cumulative distribution function of a normal distribution with mean $\mu_i^{[1,t-1]}$ and standard deviation $\sigma_i^{[1,t-1]}$, where $\mu_i^{[1,t-1]}$ and $\sigma_i^{[1,t-1]}$ are, respectively, the sample mean and standard deviation for node i from snapshots 1 to $t-1$.

We use our algorithm with $k = 12$ on this dataset. In Fig. 24, we show with blue dots highway segments that our algorithm identifies as having *unexpectedly* low average speed during rush hour (16:00 to 19:00) on Friday May 9, 2014. These segments are not necessarily the ones with most congestion. For instance, the center of Los Angeles city has higher congestion; however, such activity is normal on Friday afternoons according to the previous snapshots. The clusters shown in the map are selected because they have significantly lower average speeds than in previous observations.

8. Related work

There is a vast literature on a variety of subgraph analysis problems, arising out of a number of applications, such as bioinformatics, security, social network analysis, epidemiology and finance (see [2] for a survey). We discuss two main directions here: subgraph isomorphism and clique enumeration—for which parallel algorithms exist – and anomaly detection – for which there has been limited work on parallel algorithms.

The basic frequent subgraph detection problem involves finding subgraphs having frequency higher than a threshold. Parallel approaches for this problem involve a "bottom-up" candidate generation approach, combined with careful pruning, which builds embeddings of larger subgraphs using all possible embeddings of smaller subgraphs [1,18]. While these results allow scaling to very large networks with millions of nodes, they give no guarantees on the performance. Our work is more closely related to the use of the color coding technique for finding tree-like subgraphs [3,23], which guarantees a fully polynomial time approximation to the number of embeddings with running time and space of $O(2^k e^k m \log n)$ and $O(2^k m)$, respectively. This has been parallelized using MapReduce [44] and OpenMP [36,37], enabling subgraph counting in graphs with tens of millions of nodes with rigorous guarantees. Slota et al. [36,37] use threading and techniques for reducing the memory footprint of the color coding dynamic programming tables in order to scale. Our approach uses algebraic methods for which the memory scales as $O(k)$ instead of $O(2^k)$, and the worst case running time scales as $O(2^k)$ instead of $O(2^k e^k)$, which gives the improved performance.

6 http://pems.dot.ca.gov/.

Parallel algorithms for counting more complex subgraphs have been proposed. In principle, the color coding technique can be applied to subgraphs of bounded treewidth t. However, the time complexity is proportional to $O(n^{t+1})$, making it impractical even for subgraphs with treewidth 2. Chakaravarthy et al. [11] propose a distributed algorithm to efficiently count cycles and other treewidth-2 subgraphs. They introduce several new ideas to decompose a query subgraph into smaller motifs and process the nodes in an order that significantly reduces computation time. Another area where parallel algorithms have been developed is for dense subgraph enumeration. There are several implementations for finding maximal cliques in parallel by careful partitioning, pruning and backtracking heuristics [5,13,14,33,43]. Our results do not extend to the clique enumeration problem.

Finally, there are only two prior works on parallel graph scan statistics [16,43]. However, these do not scale to very large instances.

9. Conclusions

We present a new class of distributed MPI-based algorithms for various subgraph detection problems based on the recent multilinear detection technique. Even with a naive partitioning scheme, we observe significant performance improvement over the state-of-the-art color coding based methods. Our algorithms are conceptually much simpler than color coding and scale to subgraphs of size 18, which has not been done before. Our method combines parallelization of two different parts of the sequential multilinear algorithm, with a batched communication, and a data structure that gives cache locality.

Acknowledgments

This work was partially supported by the following grants: DTRA CNIMS Contracts HDTRA1-11-D-0016-0010, HDTRA1-17-0118, and NSF grants IIS-1633028, ACI-1443054. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

References

[1] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, F. Jamour, Scalemine: scalable parallel frequent subgraph mining in a single large graph, in: SC, 2016.
[2] L. Akoglu, H. Tong, D. Koutra, Graph based anomaly detection and description: a survey, Data Min. Knowl. Discov. 29 (3) (2015) 626–688.
[3] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, S.C. Sahinalp, Biomolecular network motif counting and discovery by color coding, Bioinformatics 24 (13) (2008) i241–i249.
[4] N. Alon, R. Yuster, U. Zwick, Color-coding, J. ACM 42 (4) (1995) 844–856.
[5] D. Aparicio, P. Ribeiro, F.A. da Silva, Parallel subgraph counting for multicore architectures, in: IEEE ISPA, 2014.
[6] S. Arifuzzaman, M. Khan, M. Marathe, Patric: a parallel algorithm for counting triangles in massive networks, in: CIKM, 2013.
[7] C.L. Barrett, R.J. Beckman, M. Khan, V.A. Kumar, M.V. Marathe, P.E. Stretz, T. Dutta, B. Lewis, Generation and analysis of large synthetic social contact networks, in: Winter Simulation Conference, 2009.
[8] R.H. Berk, D.H. Jones, Goodness-of-fit test statistics that dominate the kolmogorov statistics, Z. Wahrscheinlichkeitstheor. Verwandte Geb. 47 (1) (1979) 47–59.
[9] A. Björklund, P. Kaski, Ł. Kowalik, Fast witness extraction using a decision oracle, in: European Symposium on Algorithms, Springer, 2014, pp. 149–160.
[10] J. Cadena, F. Chen, A. Vullikanti, Near-optimal and practical algorithms for graph scan statistics, in: SIAM Data Mining (SDM), 2017.
[11] V.T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, B. Schieber, Subgraph counting: color coding beyond trees, in: Parallel and Distributed Processing Symposium, 2016 IEEE International, Ieee, 2016, pp. 2–11.
[12] F. Chen, D. Neill, Non-parametric scan statistics for event detection and forecasting in heterogeneous social media graphs, in: KDD, 2014.
[13] J. Cheng, L. Zhu, Y. Ke, S. Chu, Fast algorithms for maximal clique enumeration with limited memory, in: Proc. SIGKDD, 2012.
[14] N. Du, B. Wu, L. Xu, B. Wang, P. Xin, Parallel algorithm for enumerating maximal cliques in complex network, Min. Complex Data (2009) 207–221.
[15] L. Duczmal, M. Kulldorff, L. Huang, Evaluation of spatial scan statistics for irregularly shaped clusters, J. Comput. Graph. Statist. 15 (2) (2006) 428–442.
[16] S. Ekanayake, J. Cadena, A. Vullikanti, Fast graph scan statistics optimization using algebraic fingerprints, in: Proc. IEEE BigData, 2017.
[17] S. Ekanayake, J. Cadena, U. Wickramasinghe, A.K. Vullikanti, MIDAS: multilinear detection at scale, in: The Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018.
[18] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, P. Kalnis, Grami: frequent subgraph and pattern mining in a single large graph, in: VLDB, 2014.
[19] M.R. Garey, Computers and Intractability: A Guide to the Theory of NP-Completeness, WH Freeman & Co, 1979.
[20] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: distributed graph-parallel computation on natural graphs, in: Proc. OSDI, 2012.
[21] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, GraphX: graph processing in a distributed dataflow framework, in: Proc OSDI, 2014.
[22] T. Hansen, F. Vandin, Finding mutated subnetworks associated with survival in cancer, 2016, arXiv preprint arXiv:1604.02467.
[23] F. Hüffner, S. Wernicke, T. Zichner, Algorithm engineering for color-coding with applications to signaling pathway detection, Algorithmica 52 (2) (2008) 114–132.
[24] I. Koutis, Faster algebraic algorithms for path and packing problems, in: Proc. ICALP, 2008.
[25] M. Kulldorff, A spatial scan statistic, Comm. Statist. Theory Methods 26 (6) (1997) 1481–1496.
[26] M. Kulldorff, T. Tango, P.J. Park, Power comparisons for disease clustering tests, Comput. Statist. Data Anal. 42 (4) (2003) 665–684.
[27] M.D. Leiserson, F. Vandin, H.-T. Wu, J.R. Dobson, J.V. Eldridge, J.L. Thomas, A. Papoutsaki, Y. Kim, B. Niu, M. McLellan, et al., Pan-cancer network analysis identifies combinations of rare somatic mutations across pathways and protein complexes, Nature Genet. 47 (2) (2015) 106–114.
[28] T.A. Lieu, G.T. Ray, N.P. Klein, C. Chung, M. Kulldorff, Geographic clusters in underimmunization and vaccine refusal, Pediatrics 135 (2) (2015) 280–289.
[29] E. McFowland, S. Speakman, D.B. Neill, Fast generalized subset scan for anomalous pattern detection, J. Mach. Learn. Res. 14 (1) (2013) 1533–1561.
[30] G. Mullen, C. Mummert, Finite fields and applications, Amer. Math. Soc. 3 (2007) 19–20.
[31] J. Neil, C. Hash, A. Brugh, M. Fisk, C.B. Storlie, Scan statistics for the online detection of locally anomalous subgraphs, Technometrics 55 (4) (2013) 403–414.
[32] D.B. Neill, Fast subset scan for spatial pattern detection, J. R. Stat. Soc. Ser. B Stat. Methodol. 74 (2) (2012) 337–360.
[33] M.C. Schmidt, N.F. Samatova, K. Thomas, B.-H. Park, A scalable, parallel algorithm for maximal clique enumeration, J. Parallel Distrib. Comput. 69 (4) (2009) 417–428.
[34] J. Sharpnack, A. Singh, A. Rinaldo, Sparsistency of the edge lasso over graphs., in: AISTATS, 2012.
[35] J. Sharpnack, A. Singh, A. Rinaldo, Changepoint detection over graphs with the spectral scan statistic, in: AISTATS, 2013.
[36] G.M. Slota, K. Madduri, Fast approximate subgraph counting and enumeration, in: ICPP, 2013.
[37] G.M. Slota, K. Madduri, Complex network analysis using parallel approximate motif counting, in: IEEE IPDPS, 2014.
[38] S. Speakman, E. McFowland III, D.B. Neill, Scalable detection of anomalous patterns with connectivity constraints, J. Comput. Graph. Statist. 24 (4) (2015) 1014–1033.
[39] S. Speakman, Y. Zhang, D.B. Neill, Dynamic pattern detection with temporal consistency and connectivity constraints, in: ICDM, 2013.
[40] L.G. Valiant, The complexity of enumeration and reliability problems, SIAM J. Comput. 8 (3) (1979) 410–421.
[41] J. Wei, K. Chen, Y. Zhou, Q. Zhou, J. He, Benchmarking of distributed computing engines: spark and graphlab for big data analytics, in: Proc. IEEE BigData, 2016.
[42] R. Williams, Finding paths of length k in $O(2^k)$ time, Inform. Process. Lett. 109 (6) (2009) 315–318.
[43] J. Zhao, J. Li, B. Zhou, F. Chen, P. Tomchik, W. Ju, Parallel algorithms for anomalous subgraph detection, Concurr. Comput.: Pract. Exper. 29 (3) (2017) e3769.
[44] Z. Zhao, G. Wang, A.R. Butt, M. Khan, V.A. Kumar, M.V. Marathe, Sahad: subgraph analysis in massive networks using hadoop, in: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, IEEE, 2012, pp. 390–401.

Saliya Ekanayake is a postdoctoral fellow in the Performance and Algorithms Research (PAR) group at Lawrence Berkeley National Laboratory. His research is on improving the performance of large scale deep learning systems and graph analytics. Prior to joining the Berkeley Lab, Saliya was a postdoctoral researcher in the Network Dynamics and Simulation Science Laboratory (NDSSL) at Virginia Tech, where he researched on developing parallel algorithms to perform efficient subgraph detection. Saliya's Ph.D. work includes a systematic study of performance in machine learning algorithms.

Jose Cadena is a research staff member in the Computational Engineering Division at Lawrence Livermore National Laboratory, Livermore, CA, USA. His research interests include graph mining, machine learning for network data, combinatorial optimization, and approximation algorithms. His work has been applied to problems on the domains of social media analytics, computational epidemiology, and anomaly detection on temporal graphs. He received the Ph.D. degree from the Department of Computer Science and the Biocomplexity Institute, Virginia Tech, Blacksburg, VA, USA.

Udyanga Wickramasinghe is a Ph.D. candidate at the Computer Science Department of Indiana University, USA. He received his M.S. in Computer Science from Indiana University in 2017 and B.S. in Computer Science and Engineering from the University of Moratuwa, Srilanka in 2010. His research interests are in the general areas of runtime/operating systems, computer architecture and high-performance computing. His thesis work involves optimizing and exploring direct remote memory access (RMA) techniques in high performance interconnects for low overhead message passing. He is also an open source enthusiast and was involved in open sources projects such as Hybrid MPI, High-Performance ParallelX (HPX), Apache Synapse and Apache Xerces.

Anil Vullikanti is a Professor at the Department of Computer Science and the Biocomplexity Institute & Initiative at the University of Virginia. His research interests include the broad areas of approximation and randomized algorithms and dynamical systems, and their applications to computational epidemiology, wireless networks, and interdependent infrastructures. He is the recipient of the Virginia Tech College of Engineering Faculty Fellow Award 2017, the Biocomplexity Institute of Virginia Tech Excellence in Research Award 2017, DOE Early Career award in 2010 and the NSF CAREER award in 2009.