GraphINC: Graph Pattern Mining at Network Speed

RANA HUSSEIN, University of Fribourg, Switzerland
ALBERTO LERNER, University of Fribourg, Switzerland
ANDRÉ RYSER, University of Fribourg, Switzerland
LUCAS DAVID BÜRGI*, ETH Zürich, Switzerland
ALBERT BLARER, ArmaSuisse, Switzerland
PHILIPPE CUDRÉ-MAUROUX, University of Fribourg, Switzerland

Graph Pattern Mining (GPM) is a class of algorithms that identifies given shapes within a graph, e.g., cliques of a certain size. Any area of a graph can contain a shape of interest, but in real-world graphs, these shapes tend to be concentrated in areas deemed *skewed*. Because mining skewed areas can dominate GPM computations, the overwhelming majority of state-of-the-art GPM techniques break such areas into many small parts and load balance them across servers. This paper takes a diametrically opposite approach: we suggest a framework that concentrates rather than divides the skewed areas.

Our framework, called GraphINC, relies on two key innovations. First, it introduces a new graph partitioning scheme capable of separating the skewed area from the rest of the graph. Second, it offloads the skewed part onto a new class of hardware accelerator, a programmable network switch. We implemented our framework to leverage a commercial 100 Gbps switch and obtained results 6.5 to 52.4× faster thanks to our novel offloading technique.

CCS Concepts: • **Information systems → Database management system engines**; **Storage architectures**; **Data management systems**.

Additional Key Words and Phrases: graph pattern mining, in-network computing

ACM Reference Format:
Rana Hussein, Alberto Lerner, André Ryser, Lucas David Bürgi, Albert Blarer, and Philippe Cudré-Mauroux. 2023. GraphINC: Graph Pattern Mining at Network Speed. *Proc. ACM Manag. Data* 1, 2, Article 184 (June 2023), 28 pages. https://doi.org/10.1145/3589329

1 INTRODUCTION

The mining of graph patterns such as cliques, motifs, and frequent subgraphs is an essential primitive in several classes of computation. For instance, finding cliques is used in discovering communities in social networks [4, 21, 24], motif counting is used in bioinformatics for extracting patterns from gene networks [1, 55], frequent subgraph mining is used in finding patterns in chemical compounds [22], to mention a few.

Pattern mining tasks as the above require significant computing resources [8]. Often, the resource in demand is not memory; machines nowadays can support multi-terabyte graphs with large DRAM pools [36]. The work involved in enumerating all the candidate subgraphs and testing them for a pattern is what is computationally intensive. Consider, for instance, the *k-clique listing* task [14],

*The author performed the work while at the University of Fribourg.

Authors' addresses: Rana Hussein, University of Fribourg, Switzerland; Alberto Lerner, University of Fribourg, Switzerland; André Ryser, University of Fribourg, Switzerland; Lucas David Bürgi, ETH Zürich, Switzerland; Albert Blarer, ArmaSuisse, Switzerland; Philippe Cudré-Mauroux, University of Fribourg, Switzerland.

2836-6573/2023/6-ART184
https://doi.org/10.1145/3589329

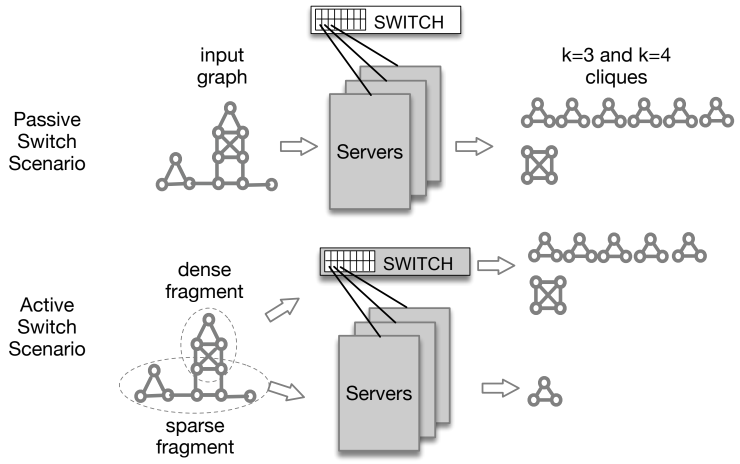


Fig. 1. (Top) GPM tasks can be performed entirely by a cluster of servers interconnected by a switch. (Bottom) With increasingly common programmable switches, this equipment can participate in a GPM computation, relieving the servers from processing the heavier portions of the graph.

where a k-clique is a fully interconnected subgraph with k nodes. Consider further the Mico dataset [25], which contains a graph with authors as nodes and co-authorship information as edges. With around 100k vertices and 1M edges, Mico is relatively small but contains approximately 984M subgraphs of size 4, and a staggering 36B subgraphs of size 5. To deal with such demands, GPM systems partition the graph across a clusters of servers using what is called a *subgraph-centric processing model* [71]. Figure 1 (Top) depicts this scenario.

This processing model has two salient characteristics. First, it looks for the desired patterns iteratively. For example, consider the k-cliques task again, now assuming $k = 5$. An algorithm could start with 2-node subgraphs (edges) and extend each with an additional neighbor node, forming subgraphs of size 3. The algorithm could then extend the 3-node subgraphs by one node, forming subgraphs of size 4—and so on. At every iteration, the algorithm would discard subgraphs that are not cliques. An iteration in this model is called a *super-step* and is usually implemented in a BSP-like style, i.e., the servers rendez-vous at the end of each super-step [74]. The rendez-vous brings us to the second hallmark of subgraph-centric algorithms: servers may exchange subgraphs at that point. This feature is critical to load balance work across servers, as the servers working on denser areas of the graph would invariably produce more subgraphs than the servers working in sparser areas.

This phenomenon is called *skew* and it occurs systematically in real-world graphs. GPM algorithms deploy techniques to deal with skew, such as load balancing and work stealing [10, 23, 77]. However, these redistribution techniques require costly subgraphs reshuffling. Ultimately, to load balance is better than to leave skew untreated, but it increases the algorithm's total runtime.

We claim that a more efficient method to handle skew in distributed GPM algorithms is to *offload it to the switch that interconnects the servers.* As we will discuss shortly, the switch is a powerful computational device, and with the recent advent of In-Network Computing (INC), it became programmable [7, 48, 62]. This means that the switch can, hypothetically, be assigned a portion of the graph to mine and be taught (via software) to do so, as Figure 1 (Bottom) depicts. We emphasize, however, that shifting work to the switch requires much more than simply porting software. The switch adopts a unique computational model that makes expressing many traditional data structures and stateful computations challenging [28].

In this paper, we show that, thanks to an innovative set of data structures and especially-designed switch algorithms, GPM skew processing is a task a programmable switch can handle. To the best of our knowledge, this is the first piece of work demonstrating that graph mining is viable and can benefit from in-network computing's potential. The advantage of moving computations to the switch is that this device is designed to be as performant – in terms of packet processing – as all the servers connected to it combined. If a computation can be expressed in terms of packet processing, it will likely run faster on the switch than on the servers.

We packaged our solution as an easy-to-use framework that we call GraphINC. The framework divides a GPM task among servers and the programmable switch to which they are connected and coordinates the task's execution. We implemented the k-cliques and k-motifs finding tasks on the framework, which can be easily extended to accommodate other tasks.

The performance improvements that the GraphINC framework delivers are sizable. For instance, in our experiments using an actual 100 Gbps programmable switch [38], the k-cliques finding task runs 6.5 to 52.4× faster when assisted by the switch than with a state-of-the-art, server-only algorithm. These results do not include an advanced implementation technique we call *super-linear pipelines* that are on the verge of becoming more feasible with the next generation of 400 Gbps switches that are entering the market [3, 16]. Moreover, we note that the 800 Gbps standard was already ratified [18] and that our advanced techniques will leverage that speed difference.

In summary, the paper makes the following contributions:

- We introduce the GraphINC framework, which pairs a server-based GPM algorithm with a switch-based counterpart (§ 3, § 4).
- We propose a new *graph cut* algorithm that identifies the adequate size and portion of the graph to assign to the switch (§ 5).
- We suggest INC-friendly graph representations that allow a programmable switch to explore skewed areas (§ 6).
- We implement and evaluate the GraphINC framework in a distributed, high-speed platform and conduct extensive experiments using an actual programmable hardware switch (§ 7).

This paper also discusses the related work (§ 8) and states our conclusions (§ 9). We start by introducing some necessary background on programmable switches and our motivation.

2 BACKGROUND AND MOTIVATION

A programmable hardware switch is a computing platform unlike any other. It is divided into two semi-independent units, a *control plane* and a *data plane*, as Figure 2 (Top) depicts. The control plane is responsible for management tasks, e.g., bringing switch ports up or down. It usually consists of an x86 machine, an Intel Xeon in our case, and it can run a common Linux distribution. The control plane functionality is available through C and Python libraries provided by the switch manufacturer.

The data plane is the component that receives packets from the network ports and forwards them back to their destination ports. The forwarding decision is the result of a computation—a networking protocol. In a programmable switch, the networking protocols are expressed as programs. These switches come with SDKs that can compile such programs into binaries they can run.

To explain how to program the data plane, we need to introduce a few concepts. The reason is that the programming model the switch supports is quite unique. The data plane consists of shared-nothing units called *Match-Action Units* (MAUs or, interchangeably, *stages*) arranged in a pipeline [7]. An MAU is for a switch what a core is for a general-purpose x86 CPU. MAUs, however, have many constraints. Chief among them is that they can only send their results to the next MAU in the pipeline and can only receive input from the previous one.

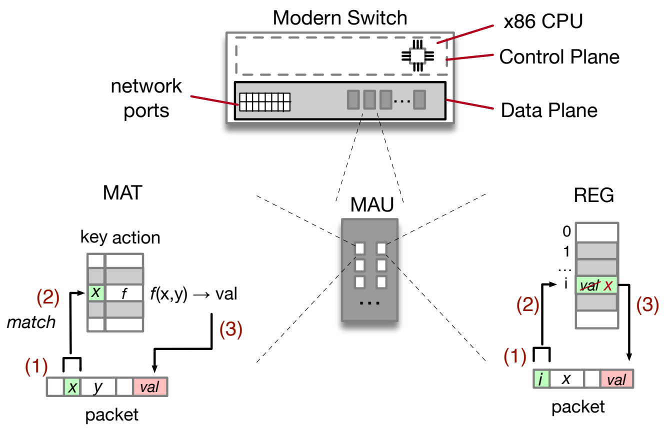


Fig. 2. (Top) The switch is composed of a control plane and a data plane. The data plane has a pipeline of Match-Action Units (MAUs) with two types of storage: Match-Action Tables (MATs) and Registers (REGs). (Bottom) A MAT can read and alter a packet by: (1) selecting the field(s) to match; (2) performing the match, e.g., via equality comparison, and if an entry is found; (3) executing the matched entry's action, altering the packet's contents. A register works similarly, although the access to registers is positional.

Each MAU can locally implement an abstraction called *Match-Action Tables* (MATs). A MAT is where packet processing takes place. It can execute a function consisting of a lookup operation (the match) against a locally stored table and the application of a side-effect (the action) associated with the matched value. Figure 2 (Bottom, Left) shows how the operation works.

MATs can be programmed by deciding (a) which packet value(s) to use in the lookup operation, (b) what calculation to perform as the action, and (c) where to store the result. For instance, a switch must decrement the TTL (time-to-live) field of an IP packet while forwarding it [17]. It can recognize IP packets via the EtherType field. Therefore, the table would have an entry that would match when the EtherType is 0x0800, i.e., an IP packet. When matched, the table would trigger a TTL decrement. In Figure 2's MAT, x would be 0x0800, y would be TTL, and f would be TTL $-$ 1. The TTL field would be overwritten with the new value in this example.

One important aspect of MATs is that their contents cannot be updated as part of an action. However, another construct called a *Register* (REG) allows updates. For instance, Figure 2 (Bottom, Right) shows a register updating an entry using a value lifted from a packet and copying the old entry's value back into the packet.

A program on the switch consists of a sequence of match-action tables and registers configurations. Such programs can be written using a language such as P4 [6], an open standard, or using proprietary languages such as Broadcom's NPL [57], Huawei's POF [68], or Xilinx's PX [9]. One advantage of P4 is its wide support, for instance, in switches from different manufacturers [2, 15], network interface cards (NICs) [56], and even in software switches [59].

The switch executes a program by moving each incoming packet through that program's MAU/REG pipeline, invoking all MATs and REGs along the way. Because packets can only move into one direction—the next stage on the pipeline—this computing paradigm is sometimes called *feed-forward model* [67].

One interesting property of commercial programmable switches is that they impose a strict pipelining discipline, i.e., each MAU/REG takes the same amount of time with every packet. This is possible because the P4 compiler can cap the actions' length to a given maximum set of steps. Programs with actions of longer durations simply fail to compile. This uniformity allows the switch to move all packets traversing the switch to their respective next MAU/REG in lock-step. In other words, forwarding a packet through a pipeline incurs latency but does not affect bandwidth. If a series of packets arrive at the maximum bandwidth (called *line rate*), they will be forwarded at the same rate. Therefore, the programs that compile successfully will handle packets at network speed[1].

Think Like a Packet. Networking protocols can be added or modified in a programmable switch simply by describing these protocols as a sequence of matches and actions. Researchers and developers quickly realized that this computing model could express logic beyond networking protocols, allowing them to use the switch as an application platform. In particular, several examples exist of switches supporting data-intensive systems (§ 8). However, converting algorithms from general-purpose machines to this computing model requires a paradigm change. The computations now have to be described as side-effects of forwarding a packet. We call this mindset informally as "think like a packet."

We argue that the subgraph-centric model that is commonly used to implement GPM systems can be expressed this way–but not without significant challenges in algorithm and data structure design for the feed-forward computing model. In the rest of the paper, we describe how we overcame these challenges.

3 GRAPHINC FRAMEWORK OVERVIEW

The GraphINC framework is designed to run a GPM task efficiently across a set of servers and the switch that interconnects them. To understand how it does so, it helps to show the execution flow the framework adopts. This flow is depicted in Figure 3.

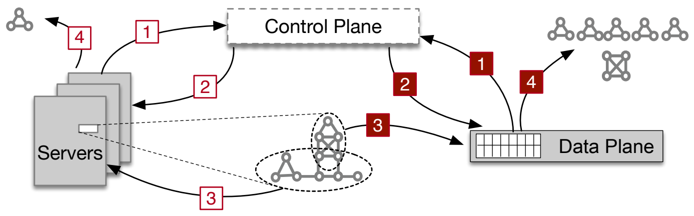


Fig. 3. The GraphINC framework's main workflow. The servers and the switch operate independently but request work in a similar way. (1) A server or the switch's data plane becomes available and (2) is assigned a specific graph fragment by a component running on the switch's control plane. (3) The servers or the switch access their assigned portion of the graph (a copy of which is held by any server) and enumerate their subgraphs. (4) The subgraphs that present the desired patterns are emitted.

At first, a graph is partitioned into *fragments*, i.e., a subset of graph nodes from which to start looking for patterns. We discuss the partitioning shortly (§ 5.1), but for now we note that the dense regions of the graph are separated from sparse ones. $\boxed{1}$ An available server solicits a graph fragment on which to work (§ 4.1). $\boxed{2}$ The requesting server is assigned a new fragment, $\boxed{3}$ and it starts processing that fragment (§ 4.4). $\boxed{4}$ The resulting patterns are output by the server.

The workflow on the switch is somewhat similar, even though the individual steps are performed differently. $\boxed{1}$ The switch's control plane detects that the data plane has finished processing a fragment. $\boxed{2}$ It then instructs a random server to transmit the next fragment (via a RDMA[2] operation) to the switch (§ 4.3). For each edge in the assigned fragment, $\boxed{3}$ the switch enumerates

[1]Hence the title of the paper.
[2]Remote Direct Memory Access is a fast networking technology in which data is transmitted and received without CPU intervention.

its derived subgraphs and tests the latter for the desired patterns (§ 4.2). Lastly, 4 the switch emits each subgraph presenting a desired pattern as an individual packet. The packets are received by some pre-assigned servers that add the subgraph to their result set.

Design Discussion. The flow above is the result of several design decisions adopted by the GraphINC framework. We now present these decisions and the rationales behind them.

- ***The GraphINC framework embraces two different computing models and provides facilities to each of them as well as mechanisms to bridge both.*** The first issue we faced concerns how specialized the switch's computing model is w.r.t. the computing model implemented by an x86 machine. Rather than try to unify the models, we decided very early in the design process that we would support both. In practice, this means that the GraphINC framework operates with two implementations of a GPM task: one that is structured to run on regular servers and one written for feed-forward processing. The framework coordinates the two implementations seamlessly.
- ***The switch should operate independently from the servers.*** The next issue we encountered was the disparity between the processing power of the servers and the switch. By construction, the switch is as powerful in terms of packet processing as all the servers combined. If the switch were to wait for any server to synchronize about work to do, this might cause some relatively long idle time on the switch. The GraphINC framework avoids any such wait by having work ready for the switch and the servers independently.
- ***The fragment allocation decisions are done by a centralized, global entity located on the switch's control plane.*** This was a somewhat opportunistic decision since the switch's control plane presents an exceptionally well-placed site from which to interact with the entire platform. The control plane can monitor the data plane (via an API) without incurring any overhead to packet forwarding and can be directly accessed by any server since they are connected.
- ***The fragment size decisions are made dynamically and the switch is always assigned computationally heavy fragments.*** The granularity of the tasks varies throughout the execution in such way that the servers and the switch finish roughly simultaneously. The fragments coming from dense areas of the graph are given to the switch, since it can perform computationally-intensive work faster than any server.

4 FRAMEWORK'S COMPONENTS

The GraphINC framework relies on many components, but four of them stand out. Two of these components are *task-agnostic*, i.e., they remain the same independent of the GPM task being executed (§ 4.1, § 4.3). The other two contain some customizable areas that are *task-sensitive* (§ 4.2, § 4.4). New tasks can be supported by the GraphINC framework simply by adjusting the latter (§ 4.5).

4.1 Algorithm Control

The Algorithm Control is responsible for selecting the next graph fragment to be processed. It does so without holding the actual graph. Instead, it maintains a list of graph nodes ordered by ID and keeps track of which nodes were already processed, as Figure 4 depicts. As we discuss shortly, fragment assignment is task agnostic but the servers' requests are treated differently than switch ones.

Upon a server request, the Algorithm Control chooses the next unprocessed fragment by traversing the nodes list backwards. Because of how the nodeIDs are assigned (§ 5.1), this means that servers get sparse areas of the graph. The fragment size is measured in number of vertices and is determined heuristically. Smaller fragments create some overhead because of the more frequent

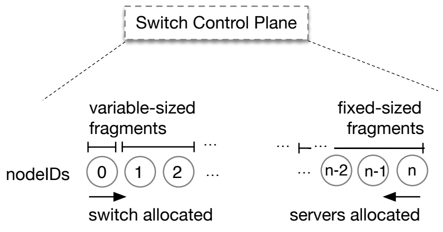


Fig. 4. Node allocations done by the Algorithm Control.

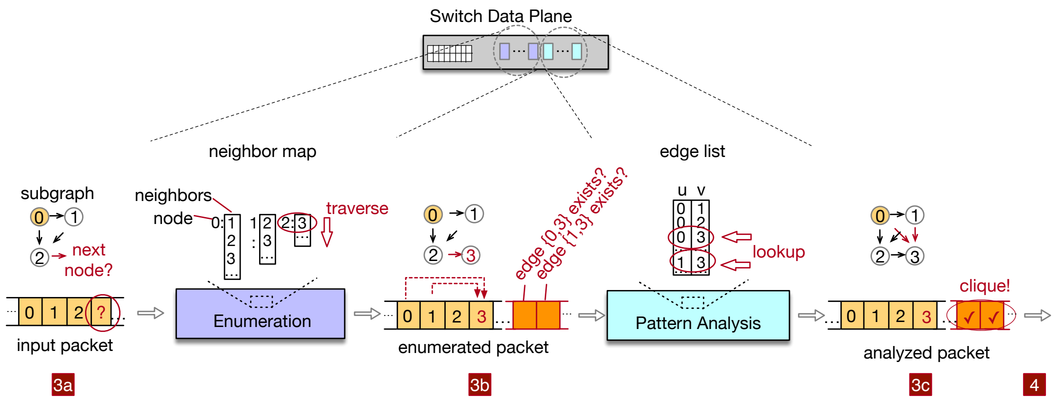


Fig. 5. 3a A packet containing an input subgraph enters the pipeline. 3b The Enumeration process adds/changes edges on the current subgraph using the neighbor map. 3c The Pattern Analysis process checks for potential edges connected to the newly added vertex. 4 If the necessary edges are present, the subgraph contains the desired pattern (e.g., a clique) and is emitted.

fragment requests. Larger fragments risk overwhelming a server and making it lag behind, slowing down the other servers.

Upon a switch request, the Algorithm Control selects the subsequently available nodes by traversing the nodes list forwards. This means that the switch receives the densest areas of the graph. The sizing of a switch fragment requires some explanations, as follows. The switch is unlikely to be able to hold an entire graph, but it does not need so to process a single fragment. All it needs is to hold the areas of the graph reachable by the assigned fragment (§ 6.2). The sizes of these areas are known because the switch fragments are pre-processed in advance; the GraphINC framework uses a small number of designated cores for that (§ 5.4). Therefore, by the time of fragment assignment, the Algorithm Control knows how many unprocessed nodes can be given at once to the switch.

4.2 Graph Transport and Processing Protocol

The GraphINC framework performs a GPM task on the switch by encoding the task's logic into a series of MATs and REGs—a networking protocol of sorts. The logic is roughly divided into two blocks: subgraph enumeration and pattern analysis. Figure 5 depicts how the MATs and REGs are divided according to the role they play in the GPM logic. We call this protocol *Graph TRansport and Processing protocol*, or GTRP ("*gee-trip*"). It is worth noting that the GTRP protocol can co-exist with other networking protocols, i.e., a switch running GTRP is still capable of executing all the usual networking protocols.

The GTRP protocol presupposes that the graph areas necessary to process a given fragment are loaded onto the switch (§ 6.1). The graph is represented by two data structures: a *neighbor map*, which supports the subgraph enumerating process, and an *edge list*, which supports the pattern analysis one. These areas are re-loaded whenever a new fragment is assigned to the switch. Once the graph area is loaded, the GTRP protocol is designed to: 3a take the assigned fragment's edges as an input packet, 3b enumerate the next derived subgraph from each input packet, 3c test it for the presence of the desired pattern, and 4 output the pattern in a packet if the tests succeeded.

This process is iterative. To obtain the next subgraph, the switch re-injects the packet at the beginning of the pipeline. This operation is called *recirculation* and is extremely efficient in commercial switches. Enumeration is a depth-first process; eventually no further subgraph could be generated. In that case, the packet is dropped. When all packets/subgraphs resulting from a fragment are explored, the data plane has completed processing that fragment.

The GTRP protocol is one of those components that have parts that are task-agnostic and parts that are not. All GPM tasks use the same neighbor map and edge list data structures. However, the enumeration and pattern analysis themselves are task-specific. The framework comes with some pre-programmed tasks, and we will explain how to encode new ones shortly (§ 4.5).

4.3 GTRP Packet Format

The GTRP protocol introduces its own packet frame format. The packet frame carries application-level information (as opposed to networking level) and, therefore, can be laid upon different transport protocols. We use RDMA-over-Converged Ethernet (RoCE) in this paper [34]. The GTRP packet frame information can be divided into three areas, as Figure 6 depicts at a high level.

The first area of the frame carries a subgraph. It contains an array of k nodeID entries, where k is the size of the pattern being matched, as shown in Figure 5 (light-orange area under 3a, 3b, and 3c). This area is used for input, e.g., sending an edge or subgraph to the switch, or output, e.g., to deliver a clique to a server. The next area of a GTRP frame is dedicated to pattern analysis. It consists of an array that stores the results of different tests, one entry per test, as shown in Figure 5 (dark-orange area under 3b and 3c). The remaining area of a GTRP frame is dedicated to flow control information. It carries several fields that help the switch determine how to process any given packet. For instance, it contains information on whether the subgraph area's data should be loaded onto a switch data structure or whether it should be used as input for the enumeration process.

4.4 Server-Side Execution

The server-side portion of the framework consists of a simple but powerful programming abstraction: a *super-step stack*. A super-step stack can be seen as a query plan that combines two types of operators: a projection over a user-defined function and an exchange operator [29]. The projection's function can be used to enumerate increasingly larger subgraphs. The exchange operators can move subgraphs up the stack or, as we will discuss shortly, across super-step stacks. The super-step stack abstraction is modeled after BSP-like processes in which servers alternate computing and data reshuffling (rendez-vous) steps.

Figure 7 shows how a super-step stack can implement k-cliques listing. The first projection of a super-step stack takes an assigned fragment's edges (i.e., 2-node subgraphs) as input and produces 3-node subgraphs. The following projection takes 3-node subgraphs as input and produces 4-node subgraphs. Each projection function forwards the cliques it found using an exchange operator.

The GraphINC framework can execute several super-step stack instances in parallel, each processing a different graph fragment. This allows a mining task to be distributed across servers

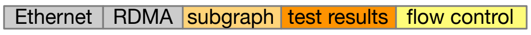


Fig. 6. Headers (gray) and content of a GTRP packet.

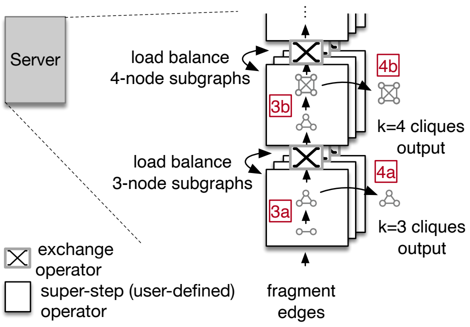

Server
load balance 4-node subgraphs
load balance 3-node subgraphs
3b
4b
k=4 cliques output
3a
4a
k=3 cliques output
exchange operator
super-step (user-defined) operator
fragment edges

Fig. 7. The super-step stack consists of user-defined operators connected by exchange operators. $\boxed{3a}$ The first step expands the subgraphs from 2 to 3 nodes and $\boxed{4a}$ selects the $k = 3$ cliques. $\boxed{3b}\boxed{4b}$ The process repeats for 3 to 4 nodes subgraphs, $k = 4$ cliques—and so on.

and to take advantage of multi-core machines. Load balancing in the GraphINC framework occurs by redistributing subgraphs at each exchange operator. To reduce the overhead of load balancing, the GraphINC framework restricts the exchange operators to reshuffle load only across super-stack stacks of the same machine.

As with other GraphINC framework components, portions of the super-stack are task-specific. The user-defined functions are task-specific, but the exchanges are generic.

4.5 Supporting New Tasks

The GraphINC framework comes pre-loaded with some tasks, e.g., k-cliques listing and k-motifs listing. New tasks are easy to implement because several components inside the framework are task-agnostic, i.e., they can be reused across tasks, as discussed above. These components include switch data plane artifacts such as the neighbor map and edge list data structures, the GTRP packet frame format, the boilerplate logic supporting enumeration and pattern analysis, and other components such as the Algorithm Control and the super-step stack structure.

To implement a new task on the GraphINC framework, the programmer would need to roughly provide the following components: the user-define functions to use on the super-step stack and the enumeration and pattern analysis logic on the switch. They would also need to provide server logic to pre-process the fragments potentially destined for the switch. Such pre-processing involves pruning and formatting that will be discussed shortly (§ 5.4).

5 THE SWITCH CUT

Having described the framework, the next question is, naturally, how to determine a *graph cut*, a partitioning in which the switch receives the dense, potentially skewed portions of the graph while the servers take the relatively sparse and easy-to-compute areas. We start by giving the intuition of such a cut (§ 5.1), followed by a formal definition (§ 5.2). We then show how to determine the cut (§ 5.3) and discuss some pre-processing associated to it (§ 5.4).

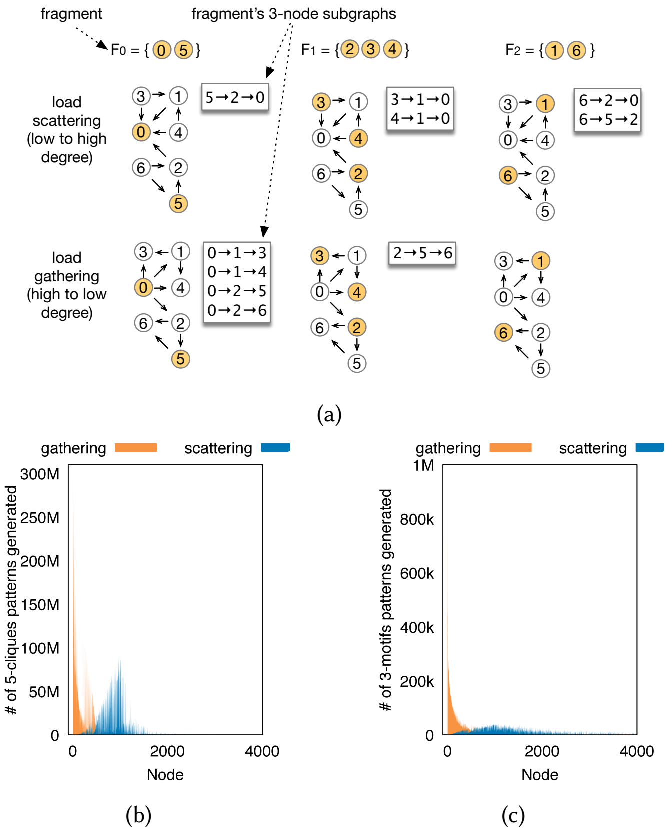


Fig. 8. (a) (Top) A directed graph where nodes of lower degree point to higher degree ones. (Bottom) The switch cut concentrates the skew in fewer nodes by inverting the direction. (b) Number of Cliques and (c) Motifs generated from each node when enumerating using load gathering or scattering on the Mico dataset.

5.1 Switch Cut Intuition

To balance the load among servers and switch, the GraphINC framework partitions the graph in node sets. We call these node sets *fragments*. We can simply partition the graph into fragments, but the search space would be unnecessarily large, producing redundant results. For instance, if we performed the k-clique listing task, the cliques that span fragments would be enumerated more than once. Instead, before determining the fragments, the GraphINC framework performs two procedures: (a) it reorders the nodes by descending degree, and (b) it converts the graph into a directed one. Using a directed graph reduces the enumeration space in GPM algorithms [12, 20, 39, 53, 64].

The difference here is that the direction commonly used has been *ascending*; nodes go from a lower degree to a higher one. Figure 8a (Top) illustrates this method. We call such graphs *load scattering* because they tend to distribute the load better. In contrast, Figure 8a (Bottom) shows that the descending order gathers the load on fewer nodes. Since we want to concentrate skew on the switch, the descending order suits our purposes.

To illustrate these effects further, we execute the k-clique finding task on a dataset (Mico) and count the number of cliques generated from each node. Figure 8b shows a plot of the results. We observe that the descending approach "moves" most cliques to earlier nodes. We verified that these effects also held on a different task such as k-motifs. It entails finding all the occurrences of connected patterns having k-nodes. For example, a 3-motif contains two patterns: a triangle and a wedge. We show in Figure 8c that the load-gathering approach also works for motifs.

5.2 Switch Cut Definition

Now that we have given the intuition about the switch cut, we formally describe it.

Let $G = (V, E)$ be an undirected graph where V is the set of nodes and E the set of edges. The edge that joins nodes u and v is denoted by (u, v). Let $d(v)$ denote the degree of a node v. For any permutation $\pi : V \longrightarrow \{1, ..., |V|\}$, let $G_\pi = (V, E_\pi)$ be the directed graph derived from G, where $\forall u, v \in V$, $(u, v) \in E_\pi$ **iif** $(u, v) \in E$ *and* $\pi(u) < \pi(v)$.

In particular, we are interested in permutations that are monotonous with respect to the order on V induced by d. We denote by π_{d^+} increasing permutation, i.e., that satisfies $d(u) < d(v) \Rightarrow \pi_{d^+}(u) < \pi_{d^+}(v)$. Note that the associated graph $G_{d^+} = (V, E_{\pi_{d^+}})$ have edges that point from lower to higher degree nodes.

Conversely, we denote by π_{d^-} decreasing permutations. The graphs generated with these permutation are denoted $G_{d^-} = (V, E_{\pi_{d^-}})$ and thus have edges that point from higher to lower degree nodes. These graph have the same enumeration properties than the load scattering ones but have the opposite effect in terms of skew; They tend to concentrate the load. We call graphs associated to decreasing permutations *load gathering*.

In the following, a subset of nodes $F_i \subset V$ of the graph G_{d^-} is called a fragment. Finally, for any $K \geq 1$, for any partition $\mathcal{F} = \cup_{i=0}^{K-1} F_i$ of V, and for any $0 \leq k < K$ we call S a *switch cut* for

$$S \doteq (G_{d^-}, \mathcal{F}, k)$$

where the fragments $F_0, ..., F_k \in \mathcal{F}$ are assigned to the switch.

5.3 Switch Cut Determination

The ideal cut should divide the work evenly between the servers and the switch. By doing so, the GraphINC framework seeks to engage the power that both these computing elements can deliver. Putting it differently, the cut should be so that the time the switch takes to process its share is approximately the time the servers take to process the remaining portion of the graph. Figure 9 shows such an ideal cut for performing the 5-cliques listing on three different datasets.

The charts show the effect on the execution time that assigning an increasingly large cut to the switch takes. We start the cut by assigning 100 nodes to the switch. The servers' runtime dominates the computation. As we increase the load on the switch, its runtime becomes dominant. We want the cut to be somewhere in between.

To determine how precise the cut needs to be, we calculated the effects of missing it by ± 10%, 20%, or 30% of the nodes. Figure 9 shows these margins as horizontal lines in each chart. It also shows the loss of performance at each end of line. For example, if the cut is 30% smaller, i.e., if we assign less work than we could to the switch, we have a performance loss of 39.6%, 18.8%, and 55.3% on the Mico, Skitter, and Wiki datasets, respectively. Interestingly, if we over-assign work to the switch by the same proportion, the performance penalties are much smaller: 4.4%, 5.2%, and 15.7%.

Either way, determining the precise cut would require predicting the runtime of the switch and servers. We believe this would be somewhat imprecise. Instead, we designed the GraphINC framework to determine the cut dynamically, as follows. The switch starts processing the graph by the low nodeID nodes (more precisely, by the fragments that contain those nodes) and proceeds in ascending order. The servers start processing the graph by the fragments containing the highest IDs and proceed descendingly.

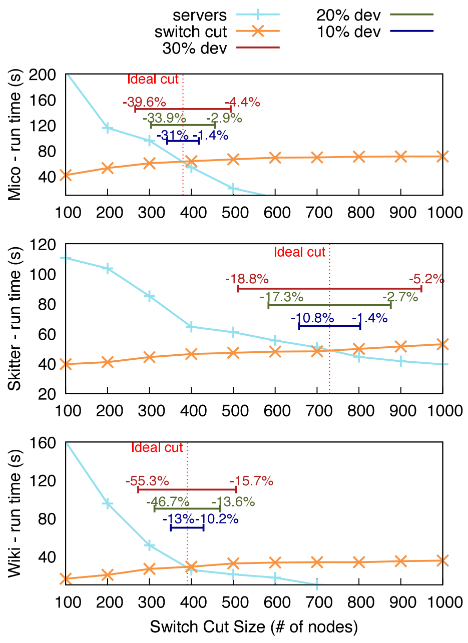


Fig. 9. Effect on performance when deviating from the ideal cut point for $k = 5$ cliques by ± 10%, 20%, and 30% nodes. The effects are similar in $k = 4$ motifs, omitted due to space restrictions, where early deviations cause from 8.8% to 31.1% performance penalties, and late deviations, 1.5% to 2.2%.

Eventually, both the switch and the servers will try to process a few remaining fragments. The algorithm control (§ 4.1) will prioritize the switch. We will show in Section 7 that our approach gets very close to the ideal cut and systematically errs by giving more work to the switch.

5.4 Switch Cut Pre-Processing

In contrast to a server, the switch does not hold the entire graph throughout a GPM computation. Instead, it stores the portion of the graph necessary to execute an assigned fragment. Before processing a fragment, the switch requests a designated server to select and format the relevant data (§ 6). The server transfers the requested data to the switch using RDMA [35], an efficient networking protocol that relies on the network card hardware rather than the server's CPU to perform transfers.

Even so, selecting and formatting data takes some computing power from the server. The GraphINC framework uses a small number of cores in designated servers to pre-process fragments ahead of time before they are assigned to the switch. Our experiments showed that we could use one core per physical pipeline on the switch for pre-processing—4 cores in total (out of 256 in our setting). This amounts to less than 0.1% of server capacity.

6 SWITCH DESIGN

There are several ways to lay out the GraphINC framework's data structures and graph manipulation logic on a programmable switch. In this section, we present the choices and trade-offs we made. We start with the design of the two main data structures (§ 6.1), and then discuss powerful *pruning techniques* that reduce their footprint (§ 6.2). These techniques can be so effective that sometimes resources go unused. We present novel ways to deploy graph logic on the switch that leverage these resources to obtain better performance in what we call *super-linear* pipelines (§ 6.3).

6.1 Switch Data Structures

The GraphINC framework uses two main data structures on the switch: a neighbor map for subgraph enumeration and (one or more) edge list(s) for pattern analysis. Because of size restrictions, it would be infeasible to fit these structures on the switch for entire graphs. Instead, the GraphINC framework processes one graph fragment at a time, i.e., it only enumerates subgraphs and tests patterns that start at the nodes in that fragment. Therefore, a fragment's neighbor map and edge list may omit non-reachable nodes. There may still be highly connected nodes that could reach a significant fraction of the graph. We will discuss this case shortly, but first, let us introduce the main data structures on the switch.

The Neighbor Map. The classic way to represent a neighbor map is called Compressed Sparse Row format (CSR). In this format, a list of a node's edges is laid out on an array. The CSR format also requires an index to locate where each node's neighbor list starts and ends. Figure 10 (Left) shows one way to implement the index and how to pack the arrays together on the switch.

The figure also shows one alternative implementation in which we would not allow a neighbor map to cross stages. The vast majority of the nodes' neighbors fit within a register[3] and, if space is left, we can still pack other neighbor lists in it—as long as none of them crosses stages. Figure 10 (Right) shows this representation. It may seem that since storage space is at a premium on the switch, we should opt for the organization that saves the most space.

However, the complexity of the programs that a switch supports is also limited. The first organization requires extra code to cross stages, whereas the second only needs to know where a neighbor list ends once a traversal starts. We opted for more straightforward logic in the GraphINC framework, i.e., the implementation shown in Figure 10 (Right). Even if some registers may not be complete, this strategy allows us to trade a small amount of space to reduce the size/complexity of our program.

The Edge List. The edge list contains a sequence of (u, v) node pairs. It is used by the pattern analysis algorithm, for instance, to check if a recently enumerated subgraph contains only fully-connected nodes. If the necessary edges are present, the subgraph is a clique (§ 4.2). The edge list is usually large, and even after pruning the unreachable nodes, it requires many registers to store. Our framework lays out the list across several stages by hashing its contents as Figure 11 (Left) illustrates.

Pattern analysis, however, involves multiple tests. When the enumeration process adds a new node to a subgraph, that node may need to have many edges to the rest of the subgraph, depending on the pattern we are mining. Unfortunately, the feed-forward model gives a packet only one chance at an action, after which it should move to the next stage (§ 2). If the list is laid out in a given stage, once the packet moves, it loses access to it. In other words, we can only use the edge list to test for one edge at a time. This means that the GraphINC framework must recirculate the packet once more for every additional test the pattern requires.

The GraphINC framework tries to avoid recirculation by reserving as many stages as possible to the edge list. In many cases, there are stages enough to have more than one copy of it, as Figure 11 (Right) depicts. Each extra copy of the edge list allows an extra edge test to be done. Generally, we need $k - 2$ copies of the list to fit on the switch, k being the size of the pattern we are mining.

[3]The size of a register is a piece of information under NDA but is adequate for our use cases.

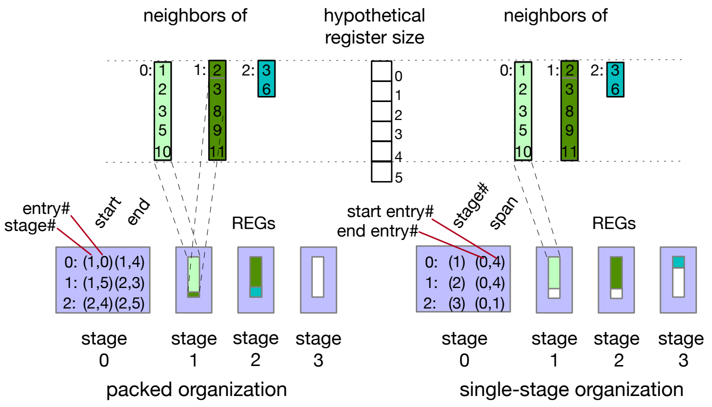


Fig. 10. A tradeoff between space and complexity: (Left) the packed organization maps node 0's neighbor list (light green) to a register in stage 1 and, since there is space left, it uses it for the following neighbor list, node 1's (dark green). (Right) the single-register organization maps any node's neighbor list to a unique register, potentially leaving some space unused. Note that indexing the neighbor map (stage 0) differs in each organization. It takes less register space on the packed one to store neighbors but more logic to find/traverse them because of the checks needed for when a list crosses stages.

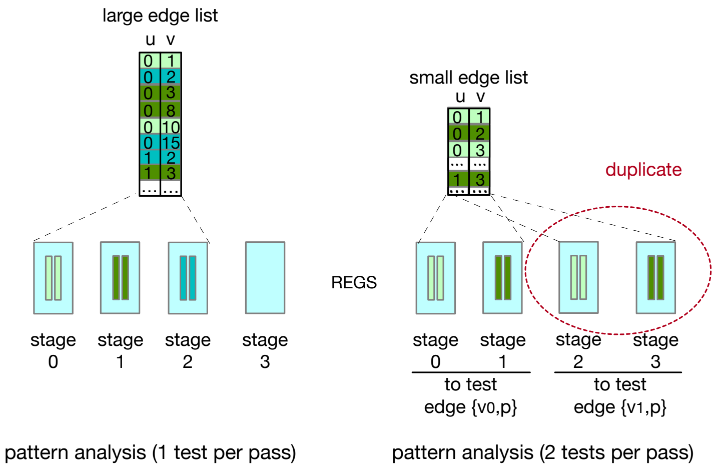


Fig. 11. A tradeoff between space and performance. (Left) A large edge list needs to be broken into several stages (e.g., by hashing its contents, shown here in different tones of green). (Right) Smaller edge lists could be stored in fewer stages and potentially be duplicated if there are enough stages left. Every replica allows the switch to perform an additional test within the same pass. With enough replicas, a task can be implemented without recirculation.

We can control the size of the edge list by limiting the number of nodes assigned to a fragment. It is even possible to have a one-node fragment. If this node's edge list is still too big, we can sub-partition it further by considering a fragment covering only a few edges. Using such a technique, the neighbor maps and edge lists for sub-partitions could be arbitrarily smaller. That said, sub-node partitioning was unnecessary for all the datasets we tested.

6.2 Pruning Techniques

Besides pruning the unreachable nodes given a fragment, there are task-specific techniques to reduce the graph size even further. We introduce two pruning techniques here for the k-clique listing but note that other tasks have pruning opportunities of their own, which we omitted due to space restrictions.

Testing Order Technique. We present this technique through an example. Suppose we have two instances of the edge list, as Figure 11 (Right) shows, and the switch fragment contains node 0, and that the enumeration process just added node 3 to the subgraph $\{0, 1, 2\}$. By definition, larger cliques are only extended from smaller cliques. Therefore $\{0, 1, 2\}$ is a clique. To test the new subgraph, we need to discover if the edges $\{0, 3\}$ and $\{1, 3\}$ exist. We can then assess if $\{0, 1, 2, 3\}$ is also a clique.

In general, the pattern analysis process needs to look for edges between $\{v_0, p\}, \{v_1, p\}, ..., \{v_{k-3}, p\}$ where p is the newly added node, v_i are the nodes in the subgraph that was extended, and k is the size of the pattern. Because of our partitioning scheme, we know that v_0 will only contain the fragment's node(s). In our example, it is node 0. We also know, by convention, that the membership tests for an edge to this node will be performed using the first copy of the edge list. Therefore, the first edge list needs to contain only the edges originating from nodes in the current fragment.

Moreover, we know that all other membership tests will be performed in other edge list copies. Accordingly, the second edge list contains all edges except the ones originating from the fragment node with the smallest node ID. This observation results in a substantial compression opportunity. For the Mico dataset, on average across all fragments, the first stage in pattern analysis holds 21× less state than the second stage.

Clique Property Technique. By definition, the switch does not need to store neighbors that are more than $k - 1$ hops away from the fragment's node(s) it is considering. These neighbors never form cliques of size k that originate from the fragment's nodes. Therefore, these *distant* neighbors can be pruned from the graph representation for that fragment.

However, not all nodes within k hops need to be considered. The Clique Property also determines that a 4-node clique can only be formed from nodes participating in 3-node cliques. In other words, the opportunity exists to prune nodes not participating in any 3-node clique. This requires calculating 3-node cliques as part of the pruning process before shipping a fragment to the switch.

The Clique Property pruning applies to the neighbor map and the edge lists. This gives us a reduction in fragment sizes ranging from a factor of 2 to 9 times. The Clique Property and the Testing Order techniques are orthogonal and can yield compound benefits.

Discussion. Although the properties we presented here were discussed in the context of the k-cliques listing task, some of their ideas can be generalized. The order of tests performed in an arbitrary pattern can be determined upfront, just as we did with k-cliques. The established order can then help prune the edge list instances. Moreover, the idea of pruning nodes beyond k hops applies to all tasks that mine for fixed-sized patterns.

The pruning techniques we suggest here can be performed during the pre-processing time, as long as we keep the calculation necessary in check. For instance, we limit the pruning to look at only 3-node cliques for the Clique Property as opposed to larger cliques. The small addition in computational effort is offset by the reduction in size of the data structures that are produced.

6.3 Super- and Sub-Linear Pipelines

As discussed above, a pipeline may have several copies of the edge list. We call this pipeline *linear* since it can enumerate and test one subgraph per pipeline traversal. We also mentioned that if the edge list is too large, we may need to recirculate a packet several times to complete the pattern analysis. We call such a pipeline *sub-linear* since many passes are needed for each subgraph. Figure 12 depicts these pipeline layouts. The figure also shows that, at times, we can leverage our pruning techniques to compress the data structures enough to enumerate and evaluate several subgraphs per pass. These are deemed *super-linear* pipelines.

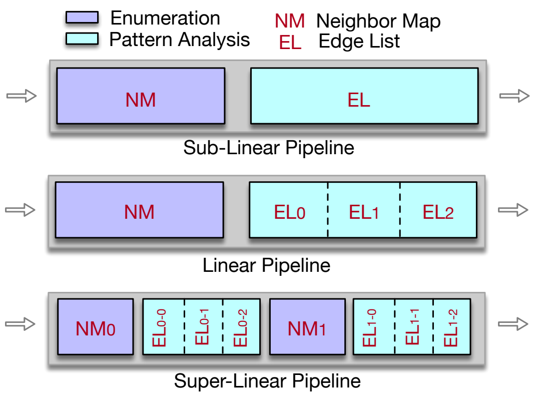


Fig. 12. Example of the different types of pipelines for $k = 5$ cliques finding. (Top) The sub-linear pipeline requires multiple passes because it can only store one edge list. (Center) The linear pipeline enumerates and tests one subgraph per pass. Therefore it needs $k - 2$ copies of the edge list. (Bottom) The super-linear one can process two subgraphs per pass because it has a copy of the entire pipeline.

The super- and sub-linear pipeline possibilities give the GraphINC framework great flexibility. One may choose to engage more resources on the switch to obtain speed. Others may choose to process with as little switch area as possible or to process more computationally intensive tasks, for instance, increasing k in our k-clique listing running example. We will show shortly what types of pipelines are possible given a dataset, the desired k, and the type of switch available (§ 7.3).

7 EXPERIMENTS

We carried out six sets of experiments, each aiming to answer a specific question about the GraphINC framework, as follows:

- How does the switch contribute to the overall performance of a GPM task (§ 7.1)?
- How does the cut assigned to the switch influence that performance (§ 7.2)?
- What is the performance of different types of pipelines (§ 7.3)?
- How scalable is the GraphINC framework when running increasingly complex GPM tasks (§ 7.4)?
- How does the GraphINC framework's performance numbers translate (or not) across a different task (§ 7.5)?
- How does the switch performance compare to other accelerators (§ 7.6)?

Setup. We run our experiments on a cluster of 16 servers. Each server contains two sockets, each with an 8-cores 2.1 GHz Intel Xeon CPU E5-2620v4, and a total of 128 GB of main memory. The servers run Ubuntu Linux 22.04.1 and are interconnected through a programmable hardware switch with 100 Gbps ports based on the Tofino 1 chip [38]. We use Mellanox's ConnectX-5 100 Gbps network cards on all the servers.

We program the switch using P4 to behave like a regular packet forwarding device when running baseline experiments. Otherwise, the switch is programmed to run the GraphINC framework in addition to its packet forwarding logic. Either way, the servers run the server-side portion of the GraphINC framework, written in C. Unless mentioned otherwise, we use the server-side portion of the framework to implement the *k-clique listing* task (cf. § 4.4).

We use several known graph datasets in our experiments: the Mico dataset models co-authorship information; the Skitter dataset captures Internet topology; the Wiki dataset represents communications over wikipedia (i.e., discussions pages); the Patents dataset represents patents and their citations. Table 1 lists the key characteristics of these graphs.

Graph	|V(G)|	|E(G)|	$\overline{d}$	4-nodes		5-nodes	
				cliques	subgraphs	cliques	subgraphs
Mico [25]	100k	1.08M	22	515M	984M	19B	36B
Skitter [50]	1.7M	11M	13	149M	2.1B	1.1B	27B
Wiki [50]	2.3M	4.6M	3	65M	4.8B	383M	49B
Patents [31]	3.7M	16M	10	3.5M	68M	3M	44M

Table 1. Dataset Properties, where $\overline{d}$ is the average degree.

The columns 4-nodes and 5-nodes in Table 1 refer to the number of cliques and subgraphs generated for those values of k. These numbers help us put in perspective the amount of computation involved for each dataset.

7.1 Effects of Engaging the Switch

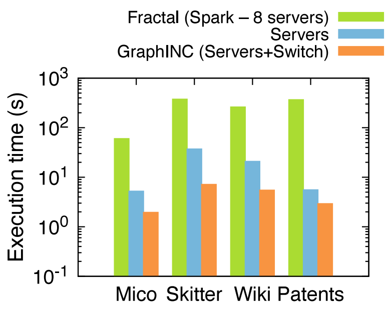
This first experiment compares the performance of executing the k-cliques listing task with $k = 4$ and $k = 5$. We compare the GraphINC framework with two different baseline systems. The first one is called Fractal, a state-of-the-art GPM system built on top of Spark [23] and OpenJDK8. We list Fractal's results running on 8 or 16 machines because, curiously, the latter may be worse than the former. The second baseline is the GraphINC framework itself but running with the switch in a passive mode, i.e., using the switch only to exchange data between servers. We denote this version by 'Servers' since no acceleration is deployed in this case.

In contrast, the GraphINC framework version uses the switch to process the skewed portion of the graph, but otherwise, it uses the same server-side code as the 'Servers' version. We use the dynamic cut approach to determine the size of the switch cut (§ 5). The three systems run on the same machines and switch. We tested these systems with the datasets described in Table 1 and show the results of this experiment in Figure 13 (in log scale).

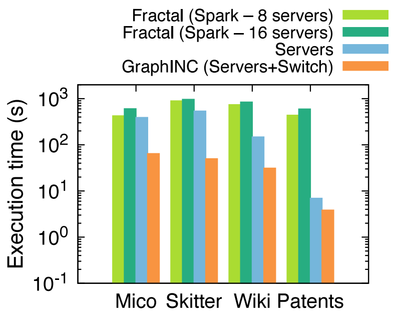
The charts show that the switch-accelerated version of the k-cliques is faster in all cases. The reason is that, given the switch assistance, the servers are required to process a smaller, more uniform portion of the graph. For the three datasets, Mico, Skitter, and Wiki, the difference in speed between GraphINC and Fractal ranges from 6.58×, for Mico and $k = 5$, to 52.49×, for Skitter and $k = 4$. The difference in speed between GraphINC and the servers ranges from 2.67× for Mico $k = 4$ to 10.79× for Skitter and $k = 5$.

For the Patents dataset, we notice a different behavior compared to the other datasets. Patents is peculiar in that its number of cliques and subgraphs decreases as the pattern size increases, as shown in Table 1. Moreover, most of the cliques do not originate from a small portion of high-degree nodes.

These factors explain the behavior seen in Figure 13 for the Patents dataset: there is no substantial difference between the runtime of $k = 4$ and $k = 5$. The GraphINC framework still outperforms



(a) k = 4



(b) k = 5

Fig. 13. Comparison of running the k-cliques task entirely on servers or sharing the workload with the switch.

Fractal by 125.5× and 113.73× for k = 4 and k = 5, respectively, but it improves the servers performance by only 1.9× and 1.8× for k = 4 and k = 5, respectively.

7.2 Effects of Varying the Cut Size

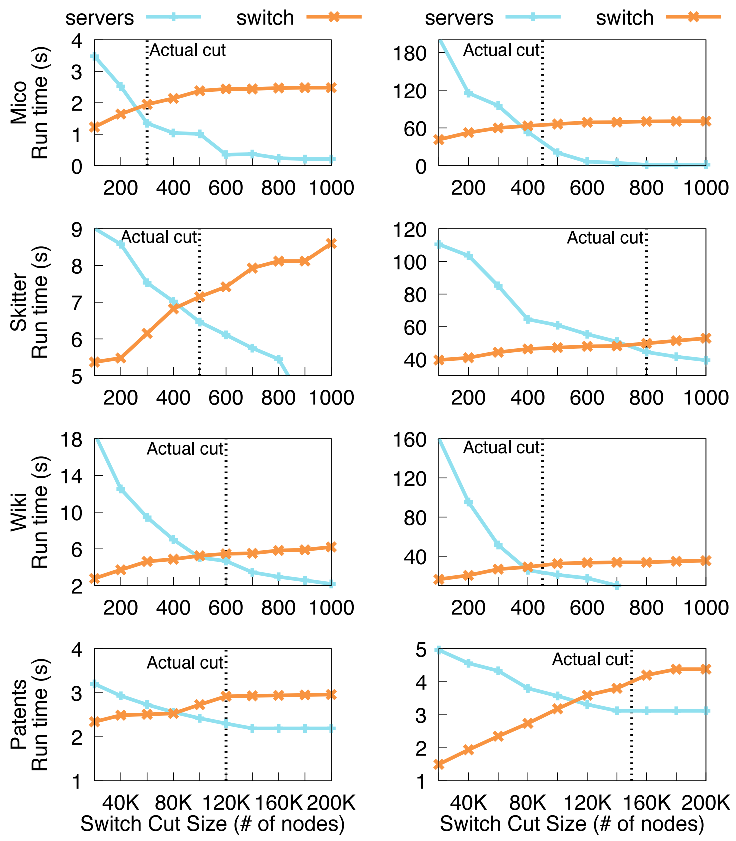
This experiment expands on the results of the previous one as follows. We run the GraphINC framework again on all the datasets of Table 1, calculating both k = 4 and k = 5 cliques finding task, but now we vary the switch cut manually from 100 to 1000 nodes. (We extend the experiment to 200k nodes for the Patents dataset.) For each cut size, we record the time the servers take with the switch acceleration. For comparison, we annotate each experiment with the actual switch cut (dynamic) that would be chosen by the GraphINC framework. Figure 14 shows the experiment's results.

As expected, the larger the cut, the longer the switch takes to complete its part of the computations and, conversely, the easier it becomes for the servers. Recall that the nodes are ordered by degree and that the skew is concentrated on early nodes. Therefore, the switch handles the most computationally intensive nodes, even in the small cuts. We observe that, as expected, the GraphINC framework determines the dynamic cut conservatively by erring on this side of assigning more work to the switch (§ 5.3). The charts show that this decision affects the performance only marginally. Ultimately, this penalty is small given that the alternative, early cuts, would have worse performance consequences.

7.3 Performance of Different Types of Pipelines

For certain datasets and k sizes, it is possible to duplicate logic and data structures on the switch such that more than one subgraph can be handled in a single pass, i.e., by traversing all stages in the switch once (§ 6.3). In other cases, we need several passes on the switch to process a single subgraph. Table 2 describes which pipelines are possible for the Tofino 1 generation of switches as well as the projected results for the next generation of switching silicon, Tofino 2 [37], which has more stages per pipeline.

The table uses a numeric scale to denote how much processing is done by a pipelined implementation. A zero indicates that the pipeline is linear, and precisely one subgraph is processed at every pass. A positive number means the pipeline is super-linear and can process an additional number of subgraphs. For instance, a *+1* means this is a 2-subgraph super-linear pipeline; a *+2* denotes a 3-subgraph pipeline, and so on. A *<+1* means that we cannot implement a super-linear pipeline but miss by very little. The × sign means a certain pipeline implementation is not feasible. A negative



(a) $k = 4$ (b) $k = 5$

Fig. 14. Effect of varying the size of the offloaded fragment for different datasets and values of k.

number means the pipeline is sub-linear. For instance, a -1 means each subgraph requires one additional recirculation to be fully processed.

		Tofino 1			Tofino 2			
		sub-linear	linear	super-linear	sub linear	linear	super linear	×
Mico	k=5		🟩	+2		🟩	+6	-2
	k=6		🟩	<+1		🟩	+3	-1
	k=7		🟩	<+1		🟩	+3	+0
Skitter	k=5		🟩	×		🟩	+1	+1
	k=6	-1	×	×		🟩	×	+2
	k=7	-2	×	×		🟩	×	+3
Wiki	k=5		🟩	<+1		🟩	+3	+4
	k=6		🟩	<+1		🟩	+2	+5
	k=7	-1	×	×		🟩	+2	+6

Table 2. Pipelines types supported by Tofino 1 and 2 chips. A positive number indicates the additional number of subgraphs per pass that super-linear pipelines can implement (cf. Fig. 12). A negative number indicates the amount of recirculations needed on a sub-linear pipeline. An × means the pipeline cannot be fully implemented.

Table 2 shows we can implement 3-subgraph super-linear pipelines for Mico and $k = 5$. In other words, we can fit three instances of the enumeration-testing logic (cf. Figure 5) in one pipeline. This means we can also implement a 2-subgraph super-linear version of that pipeline by omitting one enumeration-testing logic instance, and a linear version, by omitting two. Therefore, we decided to compare the performance scalability of these three different implementations. The idea behind this experiment is to quantify the performance benefits of traversing more subgraphs per pass. Figure 15 shows the result of this experiment.

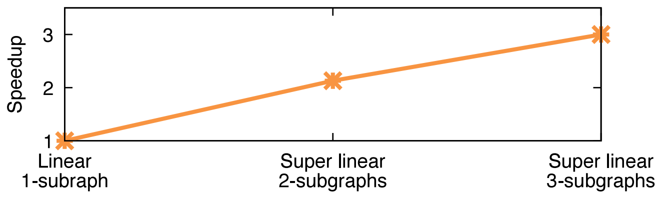


Fig. 15. Effect of pipeline optimization.

We observe that the super-linear pipelines are 2.13× and 3× faster than the linear one. This is due to the reduced number of packet recirculations required to perform the enumeration. For instance, the recirculation is reduced by 2.8× when using the faster super-linear pipeline compared to the linear one.

7.4 Projected Scalability of the Framework

In this experiment, we investigate how the GraphINC framework's performance evolves when we execute the k-clique listing task for increasingly large clique sizes. We use Fractal with 16 servers as the baseline. Figure 16 (Left) shows the results for the Mico dataset while Figure 16 (Right) shows the results for the Wiki one. Note that the y-scale in those charts is logarithmic.

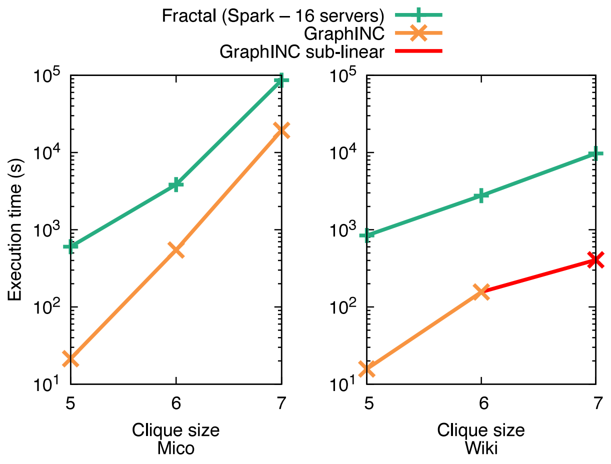


Fig. 16. Performance for cliques of size [5-7] on Mico (left) and Wiki (right). Cliques of size 7 are calculated using sub-linear pipeline on Wiki.

For the Mico experiments, we use an actual 3-subgraph implementation for $k = 5$. However, as shown in Table 2, for $k = 6$ and $k = 7$ super-linear pipelines are barely possible, i.e., there is a good amount of space left on the switch after implementing a linear pipeline but not enough to implement a 2-subgraph one. We decided to assess the potential of a 2-subgraph pipeline by making some implementation modifications. These modifications execute in the same way as an actual 2-subgraph super-linear pipeline once the data is loaded into the switch data structures. However, loading data fragments in this setup is not possible in a fast way. The measured times we show do not consider this extra loading time, but it simulates accurately the times we expect to see on a Tofino 2 switch.

For the Wiki experiments, we performed similar modifications. As Table 2 shows, the $k = 5$ and $k = 6$ could not be super-linear but by very little, so we bypass the data loading times as above. The $k = 7$ is an actual sub-linear pipeline. The results indicate that the GraphINC framework is consistently faster than Fractal for cliques of sizes [5-7] for both the Mico and the Wiki dataset.

7.5 Performance on a Different Task

In this experiment, we evaluate the switch performance on a different task: the k-motifs finding task. We run two sets of experiments, the first for size 3-motifs, which contains 2 patterns, and the second for size 4-motifs, with 6 patterns. Figure 17 shows the different patterns involved in this task. Like in the k-cliques listing task, the GraphINC framework assigns the skewed portion of the graph to the switch and the remaining nodes to the servers. We use the Mico dataset in this experiment and two baselines: Fractal and the non-accelerated version of GraphINC framework ('Servers'). Figure 18 and 19 show the results of this experiment. Note that the y-scale in Figure 18 is logarithmic.

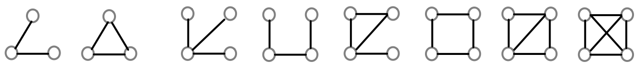


Fig. 17. (Left) 3-motifs patterns and (Right) 4-motifs ones.

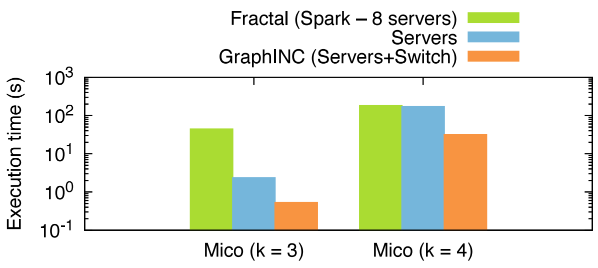
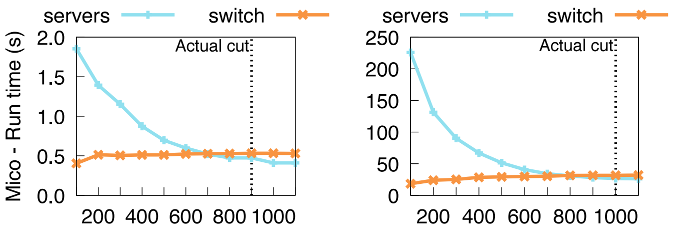


Fig. 18. Comparison of running the k-motif task entirely on the servers or sharing the workload with the switch.

The results in Figure 18 show that GraphINC outperforms Fractal by 83× in the case of $k = 3$, and 5.7× in the case of $k = 4$. GraphINC also outperforms the servers by 4.4× for $k = 3$, and 5.4× for $k = 4$. These acceleration numbers are consistent with the numbers we obtained for k-cliques (§ 7.1), i.e., our framework outperforms the baselines on both tasks by similar margins.

Figure 19 shows the effect of varying the cut size on the run time for the motifs task. Like in the k-cliques listing task, most results originate from the high-degree nodes, and, as expected, we see the GraphINC framework assigning the 'actual cut' (dynamic) to the right side of when the curves cross, i.e., the ideal cut.



(a) $k = 3$ (b) $k = 4$

Fig. 19. Effect of varying the size of the offloaded fragment for different values of k.

7.6 Comparison to Other Accelerators

In this experiment, we evaluate the relative performance of different accelerators for the k-clique listing task. We start with Pangolin, a state-of-the-art Graph Pattern Mining System [12], which runs on GPUs and CPUs. For the GPU experiment, we installed an NVIDIA GTX TITAN X GPU (12GB memory) with CUDA 10.2 on one of our servers. We compare the performance of Pangolin in a stand-alone CPU and in a stand-alone GPU with the GraphINC framework using a single switch. We vary the sizes of the cliques we are mining from $k = 4$ to $k = 6$. We used the Mico dataset in this experiment, and the switch was loaded with a $k = 4$ and $k = 5$ 3-subgraph super-linear pipeline and a $k = 6$ linear one. Figure 20 shows the results of this experiment. Note that the y-scale in that chart is logarithmic.

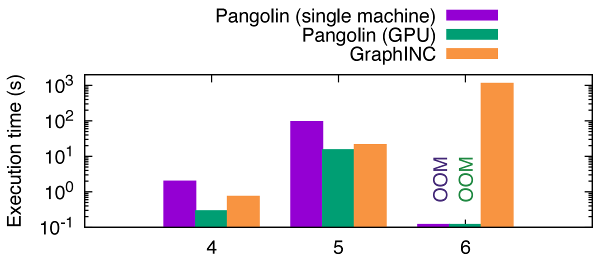


Fig. 20. Comparing a switch and GPU as accelerators for finding cliques in the Mico dataset.

The results show that, for smaller cliques, the switch performs better than the CPU but worse than the GPU for $k = 4$ and $k = 5$. The real difference appears in $k = 6$, as the size of the intermediate results for processing cliques grow (e.g., the number of subgraph candidates). Pangolin depends heavily on optimized in-memory tables whose size is relative to the size of such intermediate results. Consequently, it runs out of memory when processing $k = 6$. The memory limitation, however, is temporary, as newer generation GPUs tend to come with more memory than the previous generation.

Either way, we should note that there are promising partitioning techniques to fit graphs beyond the GPU memory [30] or that use multiple GPUs [11]. As future work, we plan to compare advanced GPU techniques to a version of GraphINC that utilizes the next generation of programmable switches. Independently of each platform's relative merits, the work we present here provides system builders with an additional acceleration option that did not exist before. The switch may be an existing equipment rather than an addition of a card, and using it would not alter the power consumption of the entire platform, as adding accelerator cards would.

Next, we evaluate the performance of GraphINC compared to GraphPi, a state-of-the-art distributed Graph Pattern Matching system [65]. Note that pattern matching and mining systems are different. GraphPi receives one pattern at a time and generates an optimal matching order to eliminate all redundant computations for this pattern, whereas the GraphINC framework understands that several patterns may be involved in a task. For this reason, to evaluate the task of finding 5-cliques in GraphPi, we input clique patterns of sizes 3, 4, and 5 and aggregate the results. Figure 21 shows the results of peforming the k-cliques task on the different systems for $k = 4$ and $k = 5$.

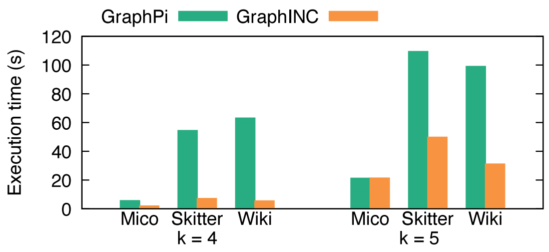


Fig. 21. Comparing GraphPi and GraphINC systems when performing the k-cliques finding task.

The results show that the GraphINC framework provides improvements ranging from 2.19× for Skitter $k = 5$ to 11.53× for Wiki $k = 4$. There is one specific case in which GraphPi's results and ours are the same, in Mico $k = 5$. This may be due to the highly crafted server-side optimizations that GraphPi performs that are absent in our server-side scheme. Our servers stack could also adopt these optimizations, but even so, the switch fundamentally takes over a portion of the work from the servers. If the portion chosen is computationally intensive—e.g., the skew—the experiments we presented here show that the acceleration provided by moving the computation to the switch is substantial.

8 RELATED WORK

To the best of our knowledge, the GraphINC framework is the first to demonstrate how to accelerate GPM tasks using In-Network Computing and programmable switches. However, INC has also been shown to benefit other database areas such as query execution [40, 47, 48, 61, 66, 72, 73], data aggregation [27, 46, 63, 78], data caching [44], replication [52, 82], transactions support [19, 41, 43, 51], and benchmarking [42, 49], to cite a few areas. As programmable switches replace *fixed-function* ones in major networking equipment vendors, we expect to see other uses of INC applied towards data analytics in general and graphs in particular.

There exist numerous algorithms tackling GPM tasks in the literature. We divide them into three classes according to their focus: single-machine approaches, distributed systems approaches and hardware-based acceleration approaches. We discuss each of them in turn.

Single-Machine Approaches. The approaches in this category run on a single machine and try to leverage the multi-core architecture in which typical CPUs are structured. The advantage of keeping to a single machine is that the processes can communicate with light-weight methods such as shared-memory [12, 39, 53, 54, 76]. Most of these algorithm apply, like our approach, techniques to reorder vertices to reduce the search space and achieve load balancing. Note, however, that while most of them reorder vertices to scatter the skew, we reorder vertices to concentrate it (§ 5.1).

Some relational systems can express graph pattern matching as SQL-like queries [26, 33]. These systems usually implement a join algorithm called *worst-case optimized join* [58, 75] in which the join is solved without enumerating unnecessary subgraphs. This is equivalent to the depth-first search that the GraphINC framework performs on the switch.

Distributed Systems Approaches. In this category, systems try to parallelize the processing of the task and distribute it across a set of machines while incurring a minimum amount of communication [10, 23, 60, 69, 70, 77, 80]. The core aspect about these approaches is that they need to handle skew in a dynamic fashion. They typically use a range of techniques that revolve around the idea of dynamic work stealing across cores and workers. By definition, these systems deploy a switch to interconnect the servers and, therefore, our technique can be seen as complementary. It is conceivable for these approaches to resort to the offloading techniques we present here should the switch they deploy be programmable.

GPU- and Hardware-based Accelerator Approaches. Like ours, this class of algorithms resorts to specialized hardware for GPM computations. The most common platform in this category are GPU-based [11, 12, 30, 79], but further work also proposes accelerators such as Processing-In-Memory [5], or even custom ASICs [13, 32, 45, 81]. The way in which the hardware is used varies greatly. For instance, some accelerators, like ours, require a change of paradigm, e.g., the use of set-centric algorithms in the PIM work [5]. Some accelerators, unlike ours, run the entirety of the GPM task, e.g., Pangolin [12]. The most notable difference to our approach is that the GraphINC framework does not alter how the servers operate; it simply reduces the load by dynamically assigning portions of the graph to the switch.

9 CONCLUSION

This paper introduced the GraphINC framework, a high-performance distributed system that supports Graph Pattern Mining tasks. The most salient feature of our framework is that it utilizes a new technology called In-Network Computing, in which the network is at once the element that transports data and a computing unit that can process it. In particular, the GraphINC framework leverages off-the-shelf programmable switches to offload the computations on the critical, heavy portions of the graph.

Adopting the GraphINC framework has at least two significant advantages. First, it effectively benefits from hardware acceleration without adding new hardware. Strictly speaking, the hardware in question—the switch—already exists in a distributed system setting, and programmables switches are bound to become prevalent, given their flexibility and similar costs. Second, our framework provides significant performance improvements by relieving the servers from processing the skewed portion of the graph.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Stefan Manegold for their constructive comments during the review process. This work has received funding from the Swiss State Secretariat for Education (SERI) in the context of the SmartEdge EU project (grant agreement No. 101092908), and from ArmaSuisse.

REFERENCES

[1] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[2] Arista. 2020. High Performance Multi-function Programmable Platforms. https://www.arista.com/en/products/7170-series.

[3] Arista. 2022. 400G Solutions. https://www.arista.com/en/products/400g-solutions.

[4] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[5] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. 2021. SISA: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 282–297.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[8] Sarra Bouhenni, Said Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. 2021. A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–35.

[9] Gordon Brebner and Weirong Jiang. 2014. High-speed packet processing using reconfigurable computing. *IEEE Micro* 34, 1 (2014), 8–18.

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*. 1–12.

[11] Xuhao Chen et al. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.

[12] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1190–1205.

[13] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. 2021. FlexMiner: A pattern-aware accelerator for graph pattern mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 581–594.

[14] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[15] Cisco 2020. Cisco Nexus 34180YC and 3464C Programmable Switches Data Sheet. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html.

[16] Cisco. 2022. 400G Data Center Networking. https://www.cisco.com/c/en/us/solutions/data-center/high-capacity-400g-data-center-networking/index.html.

[17] Douglas E Comer. 2006. *Internetworking con TCP/IP*. Vol. 1. Pearson.

[18] Ethernet Technology Consortium. 2020. 800G Specification. https://ethernettechnologyconsortium.org/wp-content/uploads/2021/10/Ethernet-Technology-Consortium_800G-Specification_r1.1.pdf.

[19] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1726–1738.

[20] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[21] Imre Derényi, Gergely Palla, and Tamás Vicsek. 2005. Clique percolation in random networks. *Physical review letters* 94, 16 (2005), 160202.

[22] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (2005), 1036–1050.

[23] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*. 1357–1374.

[24] Dongsheng Duan, Yuhua Li, Ruixuan Li, and Zhengding Lu. 2012. Incremental K-clique clustering in dynamic social networks. *Artificial Intelligence Review* 38, 2 (2012), 129–147.

[25] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* 7, 7 (2014), 517–528.

[26] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 12 (2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

[27] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-network aggregation for shared machine learning clusters. *Proceedings of Machine Learning and Systems* 3 (2021), 829–844.

[28] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 153–159.

[29] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. 102–111.
[30] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.
[31] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. 2001. The NBER patent citation data file: Lessons, insights and methodological tools.
[32] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.
[33] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *The Semantic Web – ISWC 2019*. Springer International Publishing, 258–275.
[34] Infiniband Architecture Specification Annex A16 [n. d.]. Infiniband Architecture Specification–Annex A16: RoCE. https://www.infinibandta.org/ibta-specifications-download/.
[35] Infiniband Architecture Specifications [n. d.]. Infiniband Architecture Specification. https://www.infinibandta.org/ibta-specifications-download/.
[36] Intel. 2020. Intel Xeon Platinum 8380HL Processor. https://ark.intel.com/content/www/us/en/ark/products/205684/intel-xeon-platinum-8380hl-processor-38-5m-cache-2-90-ghz.html.
[37] Intel. 2022. Intel® Tofino 2: Second-generation P4-programmable Ethernet switch. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.
[38] Inventec. 2018. Invente D10064 Programmable Switch Data Sheet. https://productline.inventec.com/Switch/Download/D10064.pdf.
[39] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[40] Matthias Jasny, Tobias Ziegler, Lasse Thostrup, and Carsten Binnig. 2022. P4DB – The Case for In-Network OLTP. In *Proceedings of the 2022 ACM SIGMOD international conference on Management of data*.
[41] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. 2021. In-network support for transaction triaging. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1626–1639.
[42] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *Proceedings of the Symposium on SDN Research*.
[43] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 35–49.
[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.
[45] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. 2020. The triejax architecture: Accelerating graph operations through relational joins. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1217–1231.
[46] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*. 741–761.
[47] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *Proceedings of the 9th Conference on Innovative Data System Research*.
[48] Alberto Lerner, Rana Hussein, André Ryser, Sangjin Lee, and Philippe Cudré-Mauroux. 2020. Networking and Storage: The Next Computing Elements in Exascale Systems? *IEEE Data Engineering Bulletin* 43, 1 (2020), 60–71.
[49] Alberto Lerner, Matthias Jasny, Theo Jepsen, Carsten Binnig, and Philippe Cudré-Mauroux. 2022. DBMS Annihilator: A High-Performance Database Workload Generator in Action. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3682–3685.
[50] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.
[51] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 104–120.
[52] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with in-Network Coherence Directories. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*.
[53] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877* (2019).

[54] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[55] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[56] Netronome. 2017. Programming NFP with P4 and C. https://www.netronome.com/media/redactor_files/WP_Programming_with_P4_and_C.pdf.

[57] Network Programming Language [n. d.]. Network Programming Language. https://nplang.org/.

[58] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (mar 2018). https://doi.org/10.1145/3180143

[59] Ben Pfaff, Debnil Sur, Leonid Ryzhyk, and Mihai Budiu. 2022. P4 in open vswitch with ofp4. In *P4 Workshop*.

[60] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal* 25, 2 (2016), 125–150.

[61] André Ryser, Alberto Lerner, Alex Forencich, and Philippe Cudré-Mauroux. 2022. D-RDMA: Bringing Zero-Copy RDMA to Database Systems. In *Proceedings of the 9th Conference on Innovative Data System Research*.

[62] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 150–156.

[63] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.

[64] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 625–636.

[65] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[66] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[67] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 15–28.

[68] Haoyu Song. 2013. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. 127–132.

[69] Nilothpal Talukder and Mohammed J Zaki. 2016. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery* 30, 5 (2016), 1024–1052.

[70] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[71] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[72] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2407–2422.

[73] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. 2018. LaKe: The Power of In-Network Computing. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8.

[74] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[75] Todd L Veldhuizen. 2012. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481* (2012).

[76] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 763–782.

[77] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. 2020. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1369–1380.

[78] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using Trio: Juniper Networks' Programmable Chipset - for Emerging in-Network Applications. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. 633–648.

[79] Xintian Yang, Srinivasan Parthasarathy, and P Sadayappan. 2011. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. *Proceedings of the VLDB Endowment* 4, 4 (2011).

[80] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data*. 2049–2062.

[81] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A locality-aware energy-efficient accelerator for graph mining applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 895–907.

[82] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proceedings of the VLDB Endowment* 13, 3 (2019), 376–389.

Received November 2022; revised February 2023; accepted March 2023