FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators

Qihang Chen
chenqh19@mails.tsinghua.edu.cn
Institute for Interdisciplinary Information Sciences, Tsinghua University
Beijing, China

Boyu Tian
tby20@mails.tsinghua.edu.cn
Institute for Interdisciplinary Information Sciences, Tsinghua University
Beijing, China

Mingyu Gao
gaomy@tsinghua.edu.cn
Institute for Interdisciplinary Information Sciences, Tsinghua University
Beijing, China

ABSTRACT

Graph mining is an emerging application of high importance and also with high complexity, thus requiring e cient hardware acceleration. Current accelerator designs only utilize coarse-grained parallelism, leaving large room for further optimizations. Our key insight is to fully exploit ne-grained parallelism to overcome the existing issues of hardware underutilization, ine cient resource provision, and limited single-thread performance under imbalanced loads. Targeting pattern-aware graph mining algorithms, we rst comprehensively identify and analyze the abundant ne-grained parallelism at the branch, set, and segment levels during search tree exploration and set operations. We then propose a novel graph mining accelerator, FINGERS, which e ectively exploits these multiple levels of ne-grained parallelism to achieve signi cant performance improvements. FINGERS mainly enhances the design of each single processing element with parallel compute units for set operations, and e cient techniques for task scheduling, load balancing, and data aggregation. FINGERS outperforms the state-of-the-art design by 2"8 on average and up to 8"9 with the same chip area. We also demonstrate that di erent patterns and di erent graphs exhibit drastically di erent parallelism opportunities, justifying the necessity of exploiting all levels of ne-grained parallelism in FINGERS.

CCS CONCEPTS

• Computer systems organization ! Special purpose systems; • Hardware ! Hardware accelerators ; • Mathematics of computing ! Graph algorithms

KEYWORDS

hardware acceleration; graph mining; parallelism

ACM Reference Format:
Qihang Chen, Boyu Tian, and Mingyu Gao. 2022. FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 March 4, 2022, Lausanne, Switzerland. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507730

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro t or commercial advantage and that copies bear this notice and the full citation on the rst page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior speci c permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '22, February 28 March 4, 2022, Lausanne, Switzerland
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507730

1 INTRODUCTION

By locating all subgraphs that are isomorphic to certain user-de ned patterns in a given graph, graph mining is a powerful analytics tool that is widely used in di erent domains, ranging from social science [18, 19, 26, 58], bioinformatics [2, 13, 40, 41], to chem-informatics [15, 20, 31, 46]. Unlike conventional graph processing [24, 37, 42, 48, 51], graph mining is more challenging due to the much higher algorithmic complexity. As a result, both specialized software frameworks [29, 38, 39, 50, 53, 55] and hardware accelerators [11, 30, 57] were proposed to improve its processing e ciency.

State-of-the-art graph mining algorithms are pattern-aware [38, 39, 50], in the sense that they compile the user-de ned pattern into an optimized execution plan to guide the mining process. The execution is characterized as exploring a search tree starting from each individual vertex in the input graph, and using set intersection and subtraction operations to determine the next candidate vertex to extend on the partially matched subgraph. This paradigm greatly reduces memory footprints and achieves signi cant speedups over the previous pattern-oblivious approaches [53], and thus is a promising target for further hardware acceleration [4, 11, 30, 47].

However, existing hardware accelerators have not fully and effectively exploited the abundant parallelism in graph mining. The most recent work, FlexMiner [11], only leveraged coarse-grained parallelism among separate search trees rooted at di erent input graph vertices, and followed a depth- rst search (DFS) order to explore each tree. This results in several ine ciencies. First, the hardware processing elements (PEs) are underutilized due to long memory access stalls caused by the intrinsic dependencies in DFS, analogous to long pipeline stalls in conventional processors. Second, by having each PE execute serial tree search, the resources are ine ciently provisioned. The complex control logic and the expensive local caches dominate the area, analogous to expensive out-of-order scheduling overheads in conventional processors. Third, the search trees starting from di erent vertices may su er from load imbalance with real-world power-law graphs. This requires us to provide superior single-PE performance to alleviate the serial bottleneck.

To address these issues, our key insight is that we need to additionally exploit the multiple levels of ne-grained parallelism within each individual tree search inside each PE. This allows us to discover more independent workload to tolerate the pipeline stalls (analogous to multi-threading processors). We can also provision

more compute units to amortize the expensive control and caching modules with better area and power efficiencies (analogous to vector processors), as well as providing better single-PE performance through parallel processing.

In this paper, we first comprehensively identify and analyze the three levels of fine-grained parallelism in pattern-aware graph mining algorithms in addition to the existing coarse-grained parallelism (Section 3). Branch-level parallelism explores different search tree branches simultaneously instead of strictly following DFS. It introduces more independent tasks but also increases memory footprints. Set-level parallelism processes the multiple set operations in each task in parallel and improves data reuse. But different sets vary significantly in terms of sizes and computations. Segment-level parallelism further parallelizes each set operation by dividing each set into non-overlapping segments. However, processing on different segments has irregular dependencies on both input and output, complicating the hardware design.

We next propose a novel graph mining accelerator, FINGERS, which effectively exploits the aforementioned multiple levels of fine-grained parallelism to achieve significant performance improvements (Section 4). To our best knowledge, FINGERS is the first graph mining hardware accelerator that systematically optimizes fine-grained parallelism. Besides the coarse-grained parallelism across multiple PEs, FINGERS enhances the design inside each PE with multiple parallel compute units for set operations. We propose a pseudo-DFS order that determines the proper degree of branch-level parallelism with balanced benefits and overheads. We use novel segment pairing and load balancing methods to distribute workloads onto multiple compute units in a PE to leverage set-level and segment-level parallelism. We also design an efficient result aggregation mechanism that ensures correct communication and collaboration among the compute units. A single type of hardware unit for set intersection is used for all types of operations including (anti-)subtraction. The results are collected from multiple compute units in a bitvector format using bitwise OR.

We extensively evaluate FINGERS using widely used patterns and popular real-world graph datasets (Section 6). We demonstrate that with the same chip area budget, FINGERS outperforms the state-of-the-art design [11] by 2.8× on average, and up to 8.9×. Such improvements are mostly from the better PE architecture, where the single-PE performance improves 6.2× on average and up to 13.2×, at the cost of less than twice of the area. In particular, we find that different patterns and different graphs exhibit drastically different degrees of each fine-grained parallelism. For example, loosely connected patterns result in high set-level and segment-level parallelism, while dense clique listing primarily benefits from branch-level parallelism. This further justifies the necessity of exploiting all levels of fine-grained parallelism in FINGERS.

In summary, our key contributions in this paper include:

- We identify several inefficiencies in the state-of-the-art graph mining accelerators that only make use of coarse-grained parallelism, including hardware underutilization, inefficient resource provision, and limited single-thread performance under imbalanced loads.
- We present and analyze the three additional levels of fine-grained parallelism in pattern-aware graph mining, namely branch-level, set-level, and segment-level parallelism.
- We propose a novel graph mining accelerator, FINGERS, that effectively exploits the aforementioned multiple levels of fine-grained parallelism. FINGERS enhances the design inside each PE with parallel compute units for set operations, and efficient techniques for task scheduling, load balancing, and data aggregation.
- We evaluate FINGERS on different patterns and different graphs, compared against the state-of-the-art baseline. FINGERS achieves 2.8× on average, and up to 8.9× speedups with the same chip area budget.
- We demonstrate different patterns and graphs exhibit drastically different degrees of each fine-grained parallelism, further justifying the necessity of exploiting all levels of fine-grained parallelism in FINGERS.

2 BACKGROUND AND MOTIVATIONS

2.1 Graph Mining Algorithms and Systems

Given a graph $G(V, E)$ and a set of patterns $\{P_1, P_2, \ldots\}$, the graph mining algorithm tries to find all the subgraphs in G that are isomorphic to each of the patterns $P_i \in \{P_1, P_2, \ldots\}$. Such isomorphic subgraphs are called the *embeddings* of the pattern P_i. For example, in Figure 1, the tailed triangle pattern P_1 is described by four vertices $\{v_0, v_1, v_2, v_3\}$, and the graph mining algorithm searches in the input graph for all its embeddings, such as {2, 1, 3, 5}. There are two ways to define subgraphs in $G(V, E)$. An edge-induced subgraph $G'(V', E')$ has vertices $V' \subseteq V$ and edges $E' \subseteq E$. In contrast, a vertex-induced subgraph $G'(V', E')$ has vertices $V' \subseteq V$ and its edge set E' contains exactly those edges in G whose both endpoints are in V'.

Graph mining is powerful to solve many graph analytics problems. For example, the most general form, subgraph listing, either vertex-induced or edge-induced, aims to list all embeddings of a defined pattern P in a graph G. k-clique listing is a special but common case of subgraph listing, which only focuses on the pattern of the size-k clique, i.e., a complete subgraph with k vertices. k-motif counting on the other hand, only counts the number of occurrences for each size-k pattern rather than listing the embeddings. Our hardware supports all these variants.

Graph mining software systems. Previously, various graph mining problems usually used different hand-optimized special-purpose algorithms, such as for triangle counting [22, 25, 27, 43, 44], clique listing [12, 14, 21], motif counting [1, 35, 45], and subgraph listing [5, 6, 32, 33, 36, 49, 52]. However, it requires substantial programming efforts to hand-tune each algorithm individually for correctness and performance. Recently, many generic graph mining software frameworks have been proposed to improve programmability and also optimize performance in a generally applicable way. Representative examples of such systems include Arabesque [56], G-Miner [7], G-thinker [56], RStream [55], Fractal [16], AutoMine [39], GraphZero [38], GraphPi [50], Pangolin [10], Peregrine [29], Sandslash [9], DwarvesGraph [8], and aDFS [54].

The above graph mining frameworks can be classified into two main categories. Early systems are *pattern-oblivious* (also known as embedding-centric) [7, 10, 16, 53, 56]. They represent the graph mining problem as a large search tree, whose level-i contains all the possible embeddings of size-i that are extended from the parent level. The tree keeps extending in a breadth-first search (BFS) [53] or

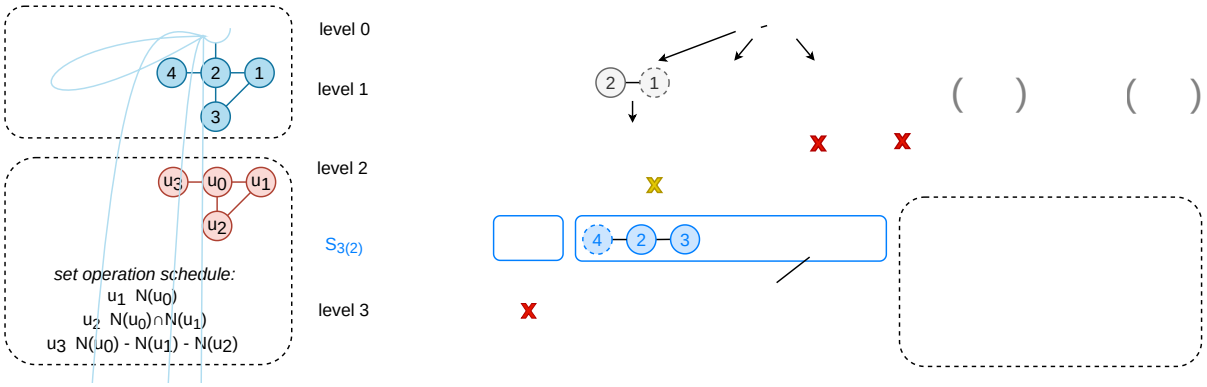

Figure 1: Pattern-aware graph mining algorithm for the tailed triangle pattern.

depth- rst search (DFS) [16, 54] manner. At each intermediate level, the partial embeddings that cannot match the pattern could be early pruned. At the leaf level, all nal embeddings go through expensive isomorphic checks. This leads to high computation complexity and exponential memory footprint growth.

In contrast, pattern-aware (a.k.a., set-centric) systems exploit the structure of the pattern to rst synthesize an e cient execution plan including the orders and the conditions of adding each new vertex into the candidate embedding [9, 29, 38, 39, 50]. Following such an optimized plan allows us to greatly reduce the time and space complexity. Our work focuses on these pattern-aware algorithms, which we further introduce in details below.

Pattern-aware graph mining algorithms. Figure 1 uses the tailed triangle pattern with four vertices as an example to illustrate the pattern-aware algorithm [38, 39]. We aim to mine vertex-induced subgraphs for now, and explain edge-induced cases later. Before mining on the input graph, the framework compiles the pattern to generate an execution plan as follows. First, it determines the order of vertices to consider in the pattern [38], which in our example is assumed to be $D_0 \cdot D_1 \cdot D_2 \cdot D_3$, i.e., D_i is an ancestor of and considered earlier than D_j only if $i < j$. Second, for each D_i to be added at each step, the compiler generates its set operation schedule (S_i), which describes the candidate vertex set that D_i can be mapped to according to the connections to its ancestors. Essentially, D_i should be a common neighbor to all connected ancestors, but exclude the neighbors of disconnected ancestors. For example, $S_3 = N(D_0) - N(D_1) - N(D_2)$ ($N(D)$ denotes the neighbors of D) in Figure 1 because D_3 is connected to D_0 but not D_1 and D_2. At each step, these set operation schedules will be materialized to concrete candidate vertex sets after the ancestors are determined during the search. Third, the plan also includes symmetric breaking restrictions if the pattern has non-trivial automorphisms [38]. As in Figure 1, D_1 and D_2 are symmetric in the pattern, so exchanging their mapped vertices results in automorphic embeddings. We would like to only count such embeddings once, and also prune the mining process early rather than extending both to the end, to save time and memory usage. To do so, a set of restrictions are applied to the mapped vertex IDs in the input graph, e.g., forcing $D_1 > D_2$. How to compile an optimized execution plan is an extensively studied topic

```
1  C0 = V;
2  for D0 ∈ C0 do
3      C1 = (N(v0) = N(D0));
4      for D1 ∈ C1 do
5          C2 = (N(v0) \ N(D0) \ N(D1));
6          C3' = (N(v1) − N(D0) − N(D1));
7          for D2 ∈ C2 do
8              if D2 < D1 then break;
9              C3 = (N(v2) − N(D0) − N(D1) − N(D2));
10             for D3 ∈ C3 do
11                 output {D0, D1, D2, D3}
```

Figure 2: Algorithm for mining the tailed triangle pattern.

in graph mining algorithms [29, 38, 39, 50]. Our hardware design is compatible with these proposals as it works with the generic execution plan format described above.

The execution plan guides the mining process on the search tree as shown in Figure 1. Figure 2 represents the algorithm as nested loops following the DFS manner. At each level i, we extend the current partial embedding with a new vertex from the input graph, which is mapped to D_i in the pattern. This new vertex is chosen from the candidate vertex set materialized from the set operation schedule (S_i), and different choices grow to multiple branches in the next level. For example, if at level 0 we choose $D_0 = 2$, then at level 1, D_1 can be any vertex in $C_1 = N(D_0) = \{1, 3, 4, 5\}$, corresponding to the four branches 2-1, 2-3, 2-4, 2-5 in Figure 1.

During the tree search, we obtain the concrete candidate vertex sets from the set operation schedules after each vertex is determined. Such materialization is done incrementally, i.e., for the candidate vertex set (S_j) of level j, at each previous level $i < j$, we incrementally update a partial result ($S_{j|i}$) as follows.

$$S_{j|i} = \begin{cases} S_{j|i} \cap N(D_i) & \text{intersection} \\ S_{j|i} - N(D_i) & \text{subtraction} \\ N(D_i) - S_{j|i} & \text{anti-subtraction} \end{cases} \quad (1)$$

from the root vertex 2 to branches 2-1 and 2-3 are two individual tasks; extending 2-3 to 2-3-1 is also a task. Each task thus involves updating multiple candidate vertex sets for the future tree levels. One of these sets becomes a fully materialized candidate vertex set, which is used to spawn new tasks for the next tree level. Consequently, the set and segment levels of parallelism are within each task, while the branch-level parallelism affects how many separate tasks are available to the PE.

Each PE has multiple intersect units (IUs) as the main compute units for various set operations. We use 24 IUs per PE as default, and explore the impact in Section 6.4. We store all candidate vertex sets during the tree search in the PE's private cache, and only spill to the shared cache if they overflow. These sets are always associated with specific tasks, so we can manage them in collaboration with the task stack. On the other hand, the neighbor lists of input graph vertices bypass the PE private cache and are only buffered in the shared cache. Their access patterns are mostly streaming (see Figure 3). In addition, such neighbor lists are larger than the candidate vertex sets and thus are harder to cache. Furthermore, our task scheduling can effectively tolerate their fetch latencies.

The PE realizes a 5-stage macro pipeline. (1) We first pop one or more tasks from the stack to schedule, and fetch their candidate vertex sets and vertex neighbor lists from the PE private cache and the shared cache, respectively (Section 4.1). (2) Next, the dispatcher controller receives the input sets and generates a head list from each, which contains the first elements (heads) from all segments of the set. During the head list generation, symmetric breaking restrictions are applied to prune entire segments with no valid vertices. (3) The head lists are then sent to the task dividers for segment pairing and load balancing (Section 4.2). The set operations are divided by segments into parallel workloads. (4) The task dividers then issue the workloads to the IUs for intersection or subtraction. (5) Finally, the results from separate IUs for the same set operation are properly aggregated in the result collector (Section 4.3). We apply the symmetric breaking restrictions again to filter invalid vertices in the results. The fully materialized candidate vertex sets become the new tasks and are pushed to the stack.

As FINGERS processes generic search tree structures, it naturally supports multi-pattern mining just like FlexMiner does [11]. The first few tree levels are common, until the point where different patterns diverge to separate tree trunks. The trunks of different patterns are treated similarly to different branches in single-pattern execution, but with additional marks to differentiate their patterns.

4.1 Resource Allocation and Task Scheduling

With multiple levels of parallelism, a key question is how to allocate the limited number of IUs in the PE. For example, with 4 IUs, we can process either 4 individual tasks, 4 set operations in one task, 4 segments in a single set operation, or any mix of these. We notice that a single task usually includes multiple set operations on multiple segments, therefore its set- and segment-level parallelism is already abundant. To simplify scheduling, a PE primarily parallelizes each task on its IUs, while sequentially executing different tasks.

To parallelize a task on the multiple IUs, we prioritize set-level parallelism to improve data reuse. We execute all the set operations simultaneously on the IUs, so the neighbor list of the newly

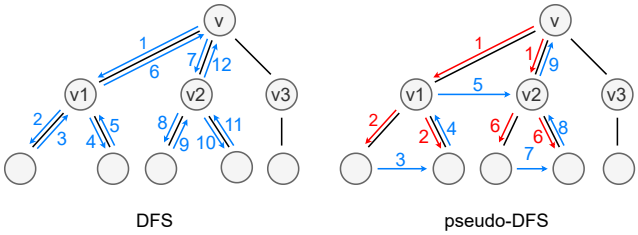

Figure 6: The pseudo-DFS task scheduling order to traverse search trees with appropriate branch-level parallelism.

extended vertex can be shared and maximally reused by all these operations as explained in Figure 3. Usually the number of set operations is smaller than the number of IUs in a PE (24), so each set operation would still use a few IUs, where the segment-level parallelism can be utilized. Because these set operations have diverse workloads, IUs are not uniformly allocated among them, but instead follow the load balance strategies introduced later in Section 4.2.

On the other hand, we do not need to heavily exploit branch-level parallelism to have multiple tasks execute in parallel. However, maintaining a small number of available tasks is still necessary in order to hide the memory access latency and keep the PE macro pipeline well utilized (Section 3.2). We propose the pseudo-DFS task scheduling order illustrated in Figure 6 as a slightly modified version of DFS, to traverse the tree with appropriate degrees of branch-level parallelism. Specifically, we gather a small number of tasks spawned from the same parent task as a task group before pushing them into the task stack. For example, E_1 and E_2 in Figure 6 form a task group, both spawned from the root E. The tasks in a group are popped from the stack together. Then we first execute those tasks whose input neighbor lists are already in the shared cache while fetching the others. Such selection is implicitly done without explicitly tracking the cache content. We simply issue all tasks to the cache; hits return immediately to resume the corresponding tasks, while misses cause a task to wait. The branch-level parallelized tasks also share the (v_0, v_1) data in the PE private cache. The task group size, i.e., the degree of branch-level parallelism, is a configurable parameter. It is chosen to fully utilize the hardware while preventing too large intermediate data. For simplicity, we set it as the minimum number of tasks to fully occupy the IUs in a PE, where the IU count needed for each task is estimated using the average sizes of the two input sets: vertex neighbor lists and candidate vertex sets. Using average set sizes is good enough, as different tasks may cancel each other's error. We observe that performance is insensitive to these parameters. Overall, pseudo-DFS is able to improve PE resource utilization while not exponentially expanding intermediate data.

4.2 Segment Pairing and Load Balancing

To distribute the multiple set operations in a task to the multiple IUs in a PE, the task dividers use the head lists of these sets for segment pairing and load balancing. Recall that a head list contains the first elements (heads) from all segments of the set. Each task divider can match a pair of head lists with up to 15 heads for the long set (corresponding to a vertex neighbor list of size $15 \times B = 240$) and up

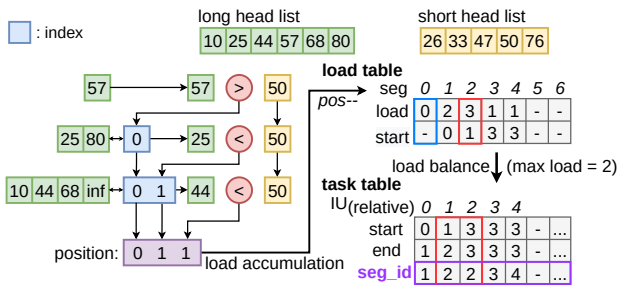

Figure 7: The task divider unit for input segment pairing and load balancing.

to 24 heads for the short set (corresponding to a candidate vertex set of size 24×96). Such limits are sufficient for most vertices in real-world graphs. Further increasing the head list sizes would increases the hardware cost. In rare cases with even longer head lists, they can be split into chunks to be matched on multiple task dividers or processed sequentially (see below).

To pair overlapped long and short segments, as in Figure 7, we organize the 15 long heads as a binary tree, and stream in each short head to compare with them. This hardware-based binary search is only used to match among the heads for segment pairing; the actual intersection/subtraction of paired segments uses the merge-based IUs as described in Section 4.3. The resultant index for the i-th short head, pos_i, indicates the position of a long head immediately larger than it, so the i-th short segment should be paired with the long segments from $pos_i - 1$ to $pos_{i+1} - 1$ (both included). We use this $pos_i - 1$ to index into a load table, and update the corresponding columns. The load table only needs 15 columns and 2 registers per column to store the number and the starting index of the short segments paired with each long segment. Such a small table can be implemented efficiently with separate registers for concurrent accesses.

The task divider next distributes the loads of the long segments to the IUs. We let each IU process one long segment with multiple paired short segments. Segments have fixed lengths, so the load can be described by the number of short segments. Each long segment with its paired short segments can be assigned to one or multiple IUs for load balancing, similar to Figure 4. We first omit any long segment that has no short segments paired with it (load = 0 in the load table, the blue box in Figure 7) except for anti-subtraction. Using a pre-determined maximum load threshold, we further split the too many short segments paired with the same long segment onto multiple IUs (load $= 3 > 2$ for long segment #2 in the load table, the red box). The load assignments after balanced are stored into a task table, which records the long segment index and the range of the paired short segments for each IU.

Coordination between multiple task dividers. Each PE has multiple task dividers (default 12 per PE). Each task has multiple set operations to be scheduled to the IUs. Also, some input sets could be quite large in power-law graphs and their head lists must be split into chunks. The task dividers share the same set of physical IUs in the PE, and they issue their task table columns to available IUs for execution. We further load balance between different task dividers by monitoring the last scheduled segment index in their task table, to approximately ensure similar progress among them.

Overheads of task dividers. The area of a task divider is mostly on the two tables, which are simply registers of a few hundreds of bytes. The latencies of each individual task divider and their coordination do not dominate the pipeline stages. The set operations on the IUs take relatively long time proportional to the entire set sizes, while the task dividers only work with the head lists that are shorter by a factor of B_2 or B_3.

4.3 Input Distribution and Result Aggregation

By leveraging segment-level parallelism, we distribute a single operation onto multiple IUs with the help of the task dividers in Section 4.2. To avoid excessive data bandwidth of transferring the relatively large and different segments to all IUs simultaneously, the PE distributes the input data in a round-robin fashion, one segment at a time, multicast to all IUs that require it. If the next IU has not finished its last workload, input data distribution would stall. Such a design is not a performance bottleneck, as it matches with the sequential processing style of the IUs, which also takes many cycles for set operations (see below).

On the output side, these separately computed results must be aggregated with caution. For example, assume a subtraction in Equation (1), where we subtract the long set from the short set. A short segment $\{11, 18\}$ is paired with two overlapped long segments $\{3, 5, 7, 12\}$ and $\{13, 15, 18, 22\}$. The differences are $\{11, 18\}$ and $\{11\}$. The final result should be the common subset $\{11\}$, which requires another round of intersection, potentially across many IUs. This issue also exists for anti-subtraction if multiple short segments pair with one long segment.

FINGERS resolves this difficulty with an efficient and unified solution for all three set operations. Note that with two sets A and B, $A - B = A \setminus (A \setminus B)$. We always compute the intersection of the two input segments on an IU regardless of the required operation, using a simple compare unit that streams in the two input lists similar to previous work [11, 47], as illustrated in Figure 8. The result is stored as a bitvector with the same length as the long segment, with different meanings for different operations. For intersection and anti-subtraction, the bitvector indicates whether each element in the long segment is in the intersection result; for subtraction (Figure 8), it indicates whether each element in the short segment is in the intersection result (padding with 1s).

To aggregate the results, we let the result collector sequentially receive the bitvector and its associated segment from each IU in a round-robin fashion, as shown in Figure 8. If the segment of the current IU does not match with the previous one, they belong to distinct ranges. We thus know that the previous segment has been fully collected and aggregated. It is now translated back to the list representation and concatenated with the existing result set. The new incoming segment stays in the result collector to wait for the next result.

Otherwise, if the same segment is received by the result collector, we need to aggregate the two bitvectors with bitwise OR. For intersection this is intuitive as we keep both results because $A \setminus B_1 \cap A \setminus B_2 = A \setminus (B_1 \cup B_2)$. For (anti-)subtraction, the results are the elements with 0s in the bitvector, as the set difference

Table 2: Area breakdown of one FINGERS PE.

Components	Area (mm^2)	% Area
24 Intersect Units	0.115	12.3%
12 Task Dividers	0.069	7.4%
2 Stream Buffers	0.214	22.9%
Private Cache	0.118	12.6%
Others	0.418	44.8%
PE Total	0.934	100%

contribute to a small portion of the area, and each IU takes only 0.005 mm^2. The task dividers are also cheap components. Most of the area is spent on the local caching buffers and the control logic. Such results validate our design principle that scaling a PE into fine-grained parallel processing incurs little extra cost.

FlexMiner uses a PE of 0.18 mm^2 under 15 nm. When scaled to the same technology, the FINGERS PE (0.26 mm^2 in 15 nm) is less than twice large as the FlexMiner PE, despite the much higher processing throughput (up to 13.2×, Section 6.2).

One FINGERS PE consumes 88.5 mW power on the compute logic and 85.6 mW on the caches. With 20 PEs per chip (see below), the total power of FINGERS would be just a few watts. FlexMiner did not report power results so we cannot directly compare, but it is likely the two behave similarly.

Our PE achieves 1 GHz frequency in 28 nm. We suspect it could match the 1.3 GHz of FlexMiner after scaled to 15 nm.

6.2 Single PE Performance

We first compare the single-PE performance between FINGERS and FlexMiner, to demonstrate that exploiting fine-grained parallelism within each PE can greatly improve performance. Figure 9 shows the results. The FINGERS PE outperforms the FlexMiner PE by 6.2× on average and up to 13.2× over all seven patterns and six graphs. We observe that different patterns and different graphs exhibit quite different speedups. This is mainly because of the different available types and drastically varying amounts of fine-grained parallelism that FINGERS can leverage in these benchmarks. We discuss the details below. Such diverse and irregular distribution necessitates FINGERS to simultaneously support all levels of fine-grained parallelism.

Impacts of patterns. Clique counting does not have set-level parallelism as the candidate vertex sets for all future levels are always identical, so `tc`, `4cl`, and `5cl` show lower improvements than the other patterns. Larger clique patterns use deeper search trees. The branch-level parallelism decreases as the tree goes deeper, because fewer vertices can be valid candidates that connect to all ancestors. This reduces the benefits of FINGERS in `As`, `Pa`, `Yo`, and `Or` when the clique size increases, but not in `Lj`, which has more large cliques and actually sees better performance. This is because the lower tree levels have better on-chip data reuse, as their demanded neighbor lists may already be fetched into the shared cache by the ancestors.

The `tt`, `cyc`, and `dia` patterns generally see the highest speedups among all patterns. This is because their set operation schedules involve (anti-)subtractions to produce large sets, which result in high parallelism that FINGERS can exploit. In particular, `tt` usually has large $|S_3|$ and $|S_2|$ (Figure 2), which leads to abundant segment-level parallelism when using it to calculate different S_3 later. `cyc` generates large S_2, which leads to high branch-level parallelism when used as the candidate vertex set for the next level. On the other hand, the subtraction operations in `dia` are only at the lower tree levels, where the sets are already shrunk small, so it shows lower benefits than the other two.

These three patterns all have size of 4 vertices, same as `4cl`. Their performance differences are attributed to the symmetric breaking restrictions. `4cl` has more connections and thus more restrictions, reducing valid vertices and hence tasks to process.

Finally, `3mc` is a multi-pattern task mining both triangles and wedges, which incurs more workloads than the single pattern of `tc`, and thus exhibits higher speedups.

Impacts of graphs. `As` and `Mi` are small graphs that all fit in the on-chip shared cache (see their tiny cache miss rates in Figure 13), so they enjoy higher improvements from the stronger PE design. `As` is actually so small that there are only a few embeddings matched with each pattern, so the independent workloads are limited. `Mi` has more cliques and thus even higher speedups. These small graphs are used to evaluate the on-chip parallelism and load balance capabilities.

On the other hand, `Yo` generally has the lowest speedups because of its lowest average degree (Table 1), which results in short neighbor lists and small sets, limiting the overall available parallelism for FINGERS. `Yo` is particularly good with `tt` and `cyc`, in which cases its candidate vertex sets are large (see above). `Pa` has very few high-degree vertices (i.e., low maximum degree, Table 1) and hence also limited benefits.

In contrast, `Lj` and `Or` are both very large graphs that exceed the on-chip cache size and are able to stress our memory system (Figure 13). They contain rich structures with many embedding matches, and thus exhibit abundant parallelism. `Or` has similar performance to `Lj` except for the patterns of large cliques, because it has fewer dense vertex clusters.

6.3 Overall Performance

With the additional coarse-grained parallelism at the PE level, we next demonstrate the overall performance improvements of FINGERS over FlexMiner in Figure 10. Since a FINGERS PE uses less than twice the area of a FlexMiner PE, for an iso-area evaluation we compare a 20-PE FINGERS chip with the 40-PE FlexMiner chip, which is its largest configuration used in the original paper [11]. Overall, FINGERS achieves on average 2.8× and up to 8.9× speedups over the baseline.

The general trends across patterns closely follow those in the single-PE setting as Figure 9. The impacts of input graphs become more significant. As mentioned above, `As` and `Mi` are small graphs fitting in the shared cache, so they see good scalability with more PEs. Their speedups are roughly half of those in Figure 9 due to the half number of PEs, but still reach 4.2× on average. `Yo` and `Pa` are large graphs with most vertices having low degrees, leading to large amount of data but limited work, especially in `tc`, `4cl`, and `5cl`. So their memory access latency cannot be fully hidden by computations even with the help of branch-level parallelism in FINGERS, leading to limited speedups. Although `Lj` and `Or` are

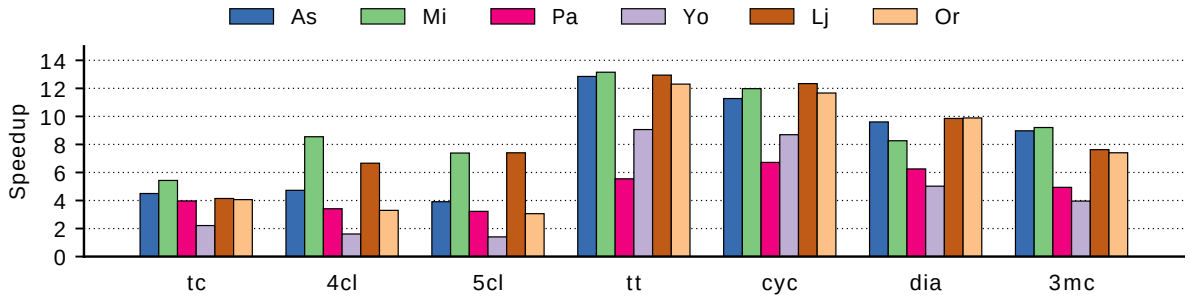

Figure 9: Single-PE speedups of FINGERS over FlexMiner.

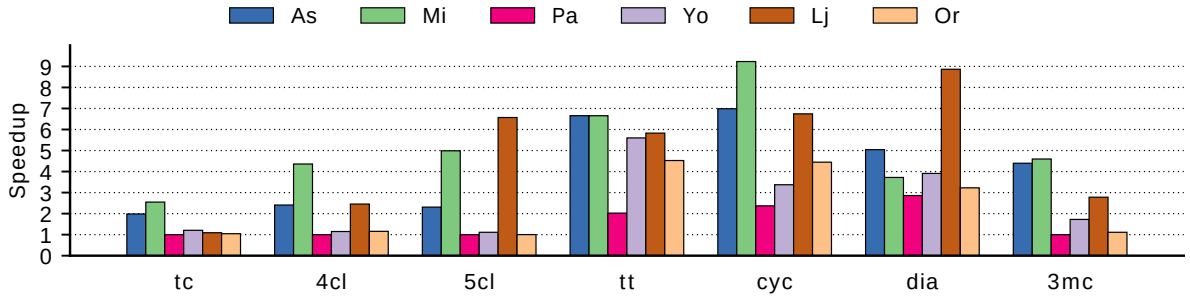

Figure 10: Overall speedups of FINGERS with 20 PEs over FlexMiner with 40 PEs.

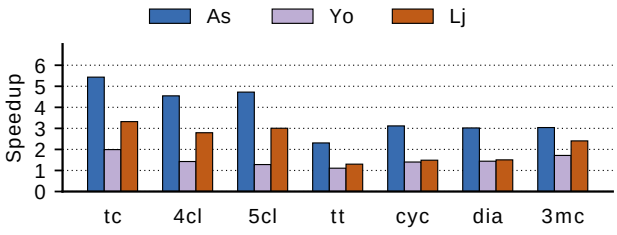

Figure 11: Speedups from branch-level parallelism and pseudo-DFS order.

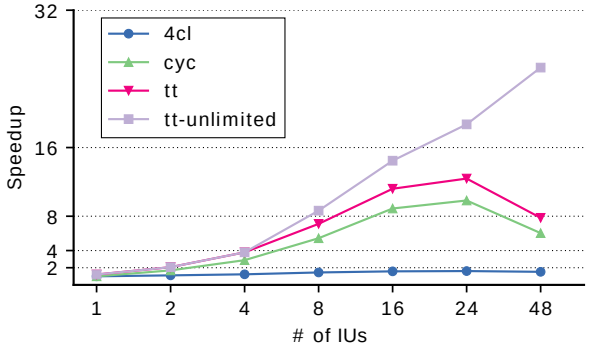

Figure 12: PE scalability in terms of numbers of IUs with **Yo**.

also large, they exhibit more parallelism due to higher degrees, and thus their speedups are higher. One orthogonal way to improve memory access performance would be to simultaneously schedule search trees with nearby starting root vertices on di erent PEs, so they access similar vertices with high locality in the shared cache. Alternatively, FINGERS can be combined with processing-in-memory technologies [4]. We leave this study as future work.

6.4 In-Depth Studies

Figure 11 presents the performance di erences when enabling and disabling the pseudo-DFS order used in FINGERS (Section 4.1) to re ect the impacts of the branch-level parallelism. We present the results of three graphs: `Mi`, `Pa`, `Or` are similar to `As`, `Yo`, `Lj`, respectively. We see that leveraging branch-level parallelism in FINGERS provides up to 5× performance gains. The `tc`, `4cl`, and `5cl` patterns bene t particularly signi cantly, because of their limited set-level and segment-level parallelism (see Section 6.2). Branch-level parallelism is the major chance they can leverage.

Figure 12 studies the scalability of the FINGERS PE in terms of number of IUs, which closely relates to the utilization of set-level and segment-level parallelism. Because using more IUs also increases area, we use an iso-area scaling approach, by keeping the product of the number of IUs and the long segment length constant, i.e., # IUs × long segment length = 48 × 16. We show three patterns with the challenging `Yo` graph whose bene ts are among the worst.

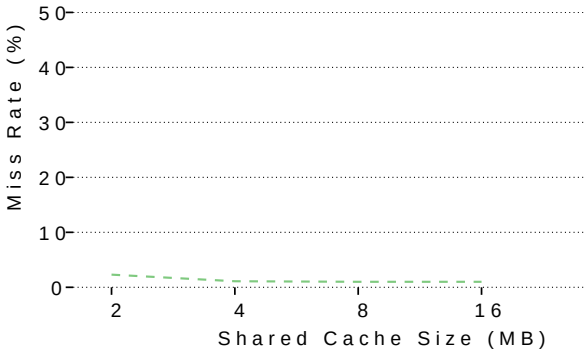

Figure 13: Shared cache miss curves (miss rate vs. capacity).

Table 3: IU utilization and load balance in one PE with Mi.

	tc	4cl	5cl	tt	cyc	dia	3mc
Active Rate	55.3%	80.8%	81.5%	94.7%	89.9%	88.9%	65.6%
Balance Rate	67.3%	66.4%	66.3%	68.2%	70.3%	71.4%	69.3%

tt and cyc exhibit good scalability until 16 to 24 IUs, and the performance drops at 48 IUs. This is limited by area. If we allow unlimited area, the performance can further improve, as shown in tt-unlimited. In contrast, 4cl has poor scalability. The reason is still its limited set/segment-level parallelism, and it should use branch-level parallelism instead (Figure 11). These results again motivate FINGERS to exploit all three levels of fine-grained parallelism.

To better understand the memory and shared cache behaviors in FINGERS, Figure 13 varies the shared cache capacity (default at 4 MB) and measures its miss rates when running the cyc pattern with three different graphs, using the same configurations as in Section 6.3. Mi (As similarly) is a small graph that fits in the cache, so its miss rates are consistently low in both FINGERS and FlexMiner. Although Yo (Pa similarly) is large and constantly requires memory accesses, it has a small average vertex degree (Table 1) that allows most neighbor lists to stay in the shared cache for multiple times of reuse by the requesting PE, before being evicted by other neighbor lists. So the miss behaviors are similar in both designs and insensitive to the cache capacity. In contrast, Lj (Or similarly) is a large graph and has long neighbor lists, so it exhibits higher cache pressure. FINGERS more efficiently uses the shared cache with lower miss rates than FlexMiner, because it has fewer individual PEs competing for capacity. The pseudo-DFS order also decreases the miss rates by prioritizing the tasks whose data are already in the cache. In addition, with set-level parallelism, a long neighbor list could be streamed only once from memory to a PE and simultaneously reused by multiple candidate vertex sets (Figure 3), without polluting the small PE private cache as in FlexMiner. These trends validate our analysis of different graphs in Sections 6.2 and 6.3.

Finally, Table 3 lists the utilization and the load balance behavior of the IUs in a PE, with the default 24 IUs. The active rate is the percentage of clock cycles during which workloads are assigned to the IUs. Recall that we assign each compute load to a subset of IUs (Section 4.2). Within these cycles on each subset of IUs that execute a compute load, we further study load balance using the balance rate, which is the sum of the individual IU active cycles divided by the total cycle of this load times the subset size. For example, assuming 4 IUs, and only 2 IUs are assigned a load executed for 10 cycles. Then in a 20-cycle period, the active rate is 25%. If in those 10 cycles, one IU is fully used but the other is only active for 5 cycles, then the balance rate is only 75%. We can see that in general the utilization is high (well exploiting branch-level parallelism) and the load balance is good (well exploiting set/segment-level parallelism).

7 CONCLUSIONS

In this paper we propose FINGERS, a novel graph mining accelerator that exploits fine-grained parallelism at the branch, set, and segment levels during search tree exploration and set operations of pattern-aware graph mining. FINGERS efficiently handles resource allocation, task scheduling, load balancing, and input/output communication with a pipelined processing element architecture enhanced with multiple compute units. Compared with a state-of-the-art baseline that only utilizes coarse-grained parallelism, FINGERS achieves up to 8.9× performance improvements at the same chip area, and benefits from different parallelism levels in different patterns and graphs.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and our shepherd, Johnathan Alsop, for their valuable suggestions. This work was supported by the National Natural Science Foundation of China (62072262). Mingyu Gao is the corresponding author.

REFERENCES

[1] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. 2015. Efficient Graphlet Counting for Large Networks. In International Conference on Data Mining (ICDM). 1–10.

[2] 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.

[3] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. ACM Transactions on Architecture and Code Optimization 14, 2 (2017).

[4] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 282–297.

[5] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In 2019 International Conference on Management of Data (SIGMOD). 1447–1462.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In 2016 International Conference on Management of Data (SIGMOD). 1199–1214.

[7] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-Oriented Graph Mining System. In 13th European Conference on Computer Systems (EuroSys). 1–12.

[8] Jingji Chen and Xuehai Qian. 2020. DwarvesGraph: A High-Performance Graph Mining System with Pattern Decomposition. arXiv preprint arXiv:2008.09682 (2020).

[9] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In International Conference on Supercomputing (ICS). 378–391.

[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An E cient and Flexible Graph Mining System on CPU and GPU. Proceedings of the VLDB Endowment 13, 8 (2020), 1190 1205.

[11] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, and Chanwoo Chung. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In 48th Annual International Symposium on Computer Architecture (ISCA). 594.

[12] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. SIAM J. Comput. 14, 1 (1985), 210 223.

[13] Young-Rae Cho and Aidong Zhang. 2009. Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks. IEEE Transactions on Information Technology in Biomedicine 14, 1 (2009), 30 36.

[14] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In 2018 World Wide Web Conference (WWW). 589 598.

[15] 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.

[16] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In 2019 International Conference on Management of Data (SIGMOD). 1357 1374.

[17] 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.

[18] Katherine Faust. 2010. A Puzzle Concerning Triads in Social Networks: Graph Constraints and the Triad Census. Social Networks 32, 3 (2010), 221 233.

[19] Ove Frank. 1988. Triad Count Statistics. Discrete Mathematics 72, 1 (1988), 141 149.

[20] Benoit Gaüzere, Luc Brun, and Didier Villemin. 2012. Two New Graphs Kernels in Chemoinformatics. Pattern Recognition Letters 33, 15 (2012), 2038 2047.

[21] Lukas Gianinazzi, Maciej Besta, Yannick Scha ner, and Torsten Hoe er. 2021. Parallel Algorithms for Finding Large Cliques in Sparse Graphs. In 33nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 243 253.

[22] Ilias Giechaskiel, George Panagopoulos, and Eiko Yoneki. 2015. PDTL: Parallel and Distributed Triangle Listing for Massive Graphs. In 44th International Conference on Parallel Processing (ICPP). 370 379.

[23] Bronwyn H Hall, Adam B Ja e, and Manuel Trajtenberg. 2001. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. Technical Report. National Bureau of Economic Research.

[24] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-E cient Accelerator for Graph Analytics. In 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 1 13.

[25] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. DistTC: High Performance Distributed Triangle Counting. In 2019 IEEE High Performance Extreme Computing Conference (HPEC). 1 7.

[26] Paul W Holland and Samuel Leinhardt. 1976. Local Structure in Social Networks. Sociological Methodology 7 (1976), 1 45.

[27] Yang Hu, Hang Liu, and H Howie Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 171 182.

[28] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions. Proceedings of the VLDB Endowment 8, 3 (2014), 293 304.

[29] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. In 15th European Conference on Computer Systems (EuroSys). 1 16.

[30] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. 2020. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 1217 1231.

[31] Hisashi Kashima, Hiroto Saigo, Masahiro Hattori, and Koji Tsuda. 2011. Graph Kernels for Chemoinformatics. In Chemoinformatics and Advanced Machine Learning Perspectives: Complex Computational Methods and Collaborative Techniques. IGI Global, 1 15.

[32] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In 2016 International Conference on Management of Data (SIGMOD). 1231 1245.

[33] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Cha , Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In 2018 International Conference on Management of Data (SIGMOD). 411 426.

[34] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection.

[35] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: A Motif Counting Algorithm for Uncertain Graphs. Proceedings of the VLDB Endowment 13, 2 (2019), 155 168.

[36] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. 2012. Distributed Graph Pattern Matching. In 21st International Conference on World Wide Web (WWW). 949 958.

[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In 2010 International Conference on Management of data (SIGMOD). 135 146.

[38] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. ACM SIGOPS Operating Systems Review 55, 1 (2021), 21 37.

[39] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In 27th ACM Symposium on Operating Systems Principles (SOSP). 509 523.

[40] Tijana Milenkovi¢, Weng Leong Ng, Wayne Hayes, and Nata²a Pr°ulj. 2010. Optimal Network Alignment with Graphlet Degree Vectors. Cancer Informatics 9 (2010), CIN S4744.

[41] Tijana Milenkovi¢ and Nata²a Pr°ulj. 2008. Uncovering Biological Network Function via Graphlet Degree Signatures. Cancer Informatics 6 (2008), CIN S680.

[42] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. Pagerank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab.

[43] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In 2019 IEEE High Performance Extreme Computing Conference (HPEC).

[44] Roger Pearce, Trevor Steil, Benjamin W Priest, and Geo rey Sanders. 2019. One Quadrillion Triangles Queried on One Million Processors. In 2019 IEEE High Performance Extreme Computing Conference (HPEC).

[45] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: E ciently Counting All 5-vertex Subgraphs. In 26th International Conference on World Wide Web (WWW). 1431 1440.

[46] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. 2005. Graph Kernels for Chemical Informatics. Neural Networks 18, 8 (2005), 1093 1110.

[47] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2020. IntersectX: An E cient Accelerator for Graph Mining. arXiv preprint arXiv:2012.10848 (2020).

[48] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In 24th ACM Symposium on Operating Systems Principles (SOSP). 472 488.

[49] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel Subgraph Listing in a Large-Scale Graph. In 2014 International Conference on Management of Data (SIGMOD). 625 636.

[50] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching Through E ective Redundancy Elimination. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 1 14.

[51] Julian Shun and Guy E Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). 135 146.

[52] Shixuan Sun and Qiong Luo. 2019. Scaling Up Subgraph Query Processing with E cient Subgraph Matching. In 35th International Conference on Data Engineering (ICDE). 220 231.

[53] Carlos HC Teixeira, Alexandre J Fonseca, Marco Sera ni, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In 25th ACM Symposium on Operating Systems Principles (SOSP). 425 440.

[54] Vasileios Trigonakis, Jean-Pierre Lozi, Tomá² Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Cha . 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In USENIX Annual Technical Conference (ATC). 209 224.

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

[56] 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 36th International Conference on Data Engineering (ICDE). 1369 1380.

[57] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A Locality-Aware Energy-E cient Accelerator for Graph Mining Applications. In 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 895 907.

[58] Olaf Zorzi. 2019. Granovetter (1983): The Strength of Weak Ties: A Network Theory Revisited. Springer Fachmedien Wiesbaden, 243 246.