# Shogun: A Task Scheduling Framework for Graph Mining Accelerators


Yibo Wu
School of Integrated Circuitrs, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
Beijing, China
wuyb21@mails.tsinghua.edu.cn

Jianfeng Zhu
School of Integrated Circuitrs, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
Beijing, China
jfzhu@tsinghua.edu.cn

Wenrui Wei
School of Integrated Circuitrs, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
Beijing, China
weiwr22@mails.tsinghua.edu.cn

Longlong Chen
School of Integrated Circuitrs, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
Beijing, China
cll17@mails.tsinghua.edu.cn

Liang Wang
School of Computer Science and Engineering, Beihang University
Beijing, China
lwang20@buaa.edu.cn

Shaojun Wei
School of Integrated Circuitrs, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
Beijing, China
wsj@tsinghua.edu.cn

Leibo Liu
School of Integrated Circuitrs, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
Beijing, China
liulb@tsinghua.edu.cn



## ABSTRACT

Graph mining is an emerging application of great importance to big data analytic. Graph mining algorithms are bottlenecked by both computation complexity and memory access, hence necessitating specialized hardware accelerators to improve the processing efficiency. Current accelerators have extensively exploited task-level and fine-grained parallelism in these algorithms. However, their task scheduling still has room for optimization. They use either breadth-first search, depth-first search or a combination of both, leading to either poor intermediate data locality, low parallelism or inter-depth barriers.

In this paper, two key insights on graph mining scheduling are gained, inspired by which a novel task scheduling framework named Shogun is proposed. First, task execution can be out-of-order to eliminate unnecessary barriers and improve PE utilization rate. Second, sacrificing intermediate data locality causes little harm to performance, when a locality monitoring mechanism is adopted to avoid severe locality loss. Hence, Shogun enables adaptive locality-aware out-of-order task scheduling by deploying a task tree to decouple the task generation and execution pipeline stages. Moreover, based on the flexible scheduling design, Shogun further develops accelerator optimizations including task tree splitting for load balance, and search tree merging to explore multiple search trees in parallel on one PE. Experimental results show that Shogun improves performance of a state-of-the-art accelerator by 63% on average with an area overhead of approximately 4%.



## CCS CONCEPTS

- **Hardware** → **Emerging technologies**.

## KEYWORDS

graph mining, scheduling, parallelism, locality

**ACM Reference Format:**
Yibo Wu, Jianfeng Zhu, Wenrui Wei, Longlong Chen, Liang Wang, Shaojun Wei, and Leibo Liu. 2023. Shogun: A Task Scheduling Framework for Graph Mining Accelerators. In *Proceedings of the*





*ISCA '23, June 17–21, 2023, Orlando, FL, USA*



ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589086

*50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589086

## 1 INTRODUCTION

Graph mining algorithms find subgraphs (a.k.a. embeddings) that are isomorphic to the given pattern in an input graph. They are widely used in domains such as bioinformatics [3, 4, 13, 40], chemoinformatics [16, 22, 44, 49], social network analysis [18, 20, 23, 51] and web spam detection [21, 33]. State-of-the-art pattern-aware graph mining algorithms [11, 27, 38, 39, 47] exploit the feature of the input pattern to build search trees and prune the automorphic subgraphs. Algorithm 1 gives an example of listing 4-cliques in a depth-first-search (DFS) manner, which heavily relies on set operations (including set intersection and subtraction) of neighbor sets. Graph mining algorithms are computation-bound due to these set operations [10, 12, 45]. They are also memory-bound due to the repeated neighbor set accesses and intermediate results storage [6, 15, 50].

Specialized hardware accelerators are proposed recently to address the computation and memory access inefficiencies of pattern-aware graph mining algorithms [6, 10, 12, 15, 45, 50]. The computation efficiency is improved with massive multithreading, set operation functional units (FUs) and fine-grained parallelism exploitation. The memory bottleneck is ameliorated by applying near-memory or in-memory computation, using new graph formats and avoiding unnecessary data movement.

Existing graph mining accelerators are not satisfactory in utilizing the memory and PE resources due to their task scheduling schemes. Graph mining algorithms such as Algorithm 1 extend partial results with new vertices computed from set operations at each search depth, effectively constructing search trees like the ones in Figure 1. Existing accelerators typically associate each node (vertex) in the search tree with a task for execution. The task scheduling of graph mining thus determines the exploration order of search trees. Early software frameworks use breadth-first-search (BFS) for scheduling [8, 52, 54, 55, 59], which improves parallelism but suffers from disastrous memory consumption explosion. Most accelerators use a DFS scheduling [12, 15, 50], so that the relationship between tasks can be easily represented with a task stack and the memory consumption is minimized. But DFS has a low parallelism. Besides, the serial search tree exploration in DFS reduces the locality in accessing intermediate results. As shown in Algorithm 1, tasks of the same $for$ loop could have shared the same intermediate results of previous depths as inputs (e.g., $S_0, S_1$). The locality loss reduces the cache efficiency.

Recently, some software frameworks [9, 53] and accelerater designs [10] adopt pseudo-DFS scheduling, which combines BFS and DFS and offers the best of two worlds. Rather than fetching one task (DFS) or all tasks (BFS) of a search depth for execution at a time, pseudo-DFS fetches a task group with a pre-configured group size. After the whole task group finishes execution, the first task in the group generates children tasks. Compared with DFS, the PE utilization rate is improved because multiple tasks of the task group are executed in parallel. Compared with BFS, the memory consumption can be well controlled by configuring the task group size.

Nonetheless, we observe that pseudo-DFS is still not ideal due to its inter-depth barriers. Tasks that complete execution earlier have to wait until the whole task group completes. Therefore, the time of a task group execution is bottlenecked by the slowest task. Graph mining algorithms are data-dependent and irregular, with a large variance in the task runtime. Such barriers leave PE resources idle and limit the potential performance gain.

To address the above issues in graph mining scheduling, this paper identifies two insights. The first one is that **the executions of tasks without parent-child relationships can be reordered and inter-depth barriers are unnecessary**. For example, for each search tree in Figure 1, tasks of different subtrees are independent and their execution can be out-of-order. Out-of-order task execution does not violate correctness, because pattern-aware graph mining algorithms ensure that, while extending a subgraph $E_{k-1}$ to subgraph $E_k$ with a vertex $v_k$, the task of vertex $v_k$ should only apply set operations on the neighbor sets of $E_{k-1}$'s vertices and the neighbor set of $v_k$ itself [39]. In other words, the task of vertex $v_k$ is independent of tasks outside the subgraph $E_{k-1}$ and can be reordered with them. However, existing graph mining accelerators mistakenly inherit the assumption of traditional graph algorithm accelerators, that inter-depth barriers are necessary to guarantee the correctness.

The second insight concerns about the intermediate result locality issue. **Sacrificing some intermediate result locality generally does little harm. But locality monitoring is still necessary in case a severe locality loss degrades the performance.** By scheduling independent tasks to execute together, out-of-order task scheduling reduces the locality like DFS. The second insight ensures that most of the time the benefit of a higher parallelism outweighs the locality loss. But it also necessitates that the scheduling scheme be locality-aware, and explains why other out-of-order task scheduling architectures cannot be directly applied in the context of graph mining [2, 14, 17, 28, 32, 58].

Leveraging the above insights, this paper proposes Shogun, which is a task **Sch**eduling framew**o**rk to improve **g**raph mining accelerator resource **u**tilizatio**n**. Shogun enables out-of-order task scheduling while being aware of data locality, by using a task tree design to decouple task generation and execution. Newly spawned children tasks are stored in the task tree. A scheduler in the task tree takes both parallelism and locality into consideration and decides the execution order of these tasks. The scheduler adaptively transitions to a conservative mode if locality loss severely harms the performance.

In summary, this paper makes the following contributions:

**Algorithm 1:** 4-Clique Listing

```
1  for u0 in V do
2      S0 = Nu0;
3      for u1 in S0 do
4          if u1 > u0 then
5              break;
6          end
7          S1 = Nu1 ∩ S0 = Nu1 ∩ Nu0;
8          for u2 in S1 do
9              if u2 > u1 then
10                 break;
11             end
12             S2 = Nu2 ∩ S1 = Nu2 ∩ Nu1 ∩ Nu0;
13             for u3 in S2 do
14                 if u3 > u2 then
15                     break;
16                 end
17                 output {u0, u1, u2, u3};
18             end
19         end
20     end
21 end
```

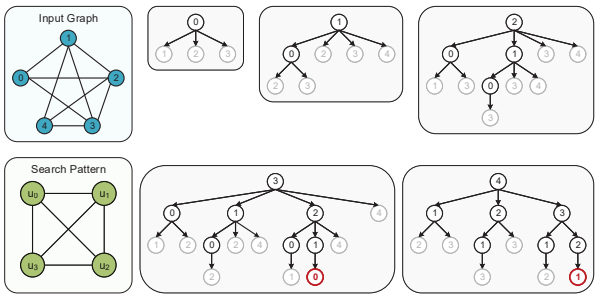



**Figure 1: An illustration of how the pattern-aware 4-clique listing algorithm (see in Algorithm 1) lists all isomorphic subgraphs of the input graph. The algorithm constructs 5 search trees as shown in the gray boxes. Each search tree roots at a unique vertex of the input graph. Gray circles in search trees are vertices pruned due to the symmetry breaking. Black circles are valid vertices after set operations and symmetry breaking. Red circles indicate a unique subgraph that is isomorphic to the pattern.**

- We analyze existing graph mining task scheduling schemes. Based on the analysis, we identify that the task scheduling is a considerable challenge in terms of memory access and PE utilization.
- We present Shogun, which is a locality-aware out-of-order task scheduling framework. A task tree structure is implemented to decouple task generation and execution.
- We propose two accelerator optimization methods based on the Shogun scheduling scheme. First, a task tree splitting scheme allows fine-grained workload balance among PEs. Second, a search tree merging scheme maps several search trees on one PE, further improving the accelerator resource utilization.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Graph Mining Algorithms and Accelerators

Given an input graph $G$ and a search pattern $P$, graph mining algorithms list or count all subgraphs $E$ that are isomorphic to $P$. Graph mining algorithms guarantees completeness and uniqueness [12], i.e., all subgraphs $E$ that are isomorphic to $P$ should be found once and only once. If a subgraph $E_i$ is automorphic to a previously found subgraph $E_j$, $E_i$ should be pruned by an automorphism test. Graph mining algorithms can solve many problems, e.g., triangle counting [24, 41, 42, 48], subgraph listing [7, 29, 30, 46] and frequent subgraph mining [1, 19, 25, 56].

A variety of algorithms are proposed to solve each specific graph mining problem. Graph mining software systems emerge to improve their programmability. Early graph mining systems are pattern-oblivious, i.e., they enumerate all subgraphs and then explicitly do isomorphism and automorphism tests [8, 17, 52, 54, 55, 59]. In contrast, pattern-aware systems exploit the pattern structure to prune the search tree and implicitly encode the isomorphism tests in search tree construction [11, 27, 38, 39, 47]. Pattern-aware systems achieve a far better performance by aggressively pruning the subgraph search space with the guidance of the pattern structure. Therefore, this paper targets at accelerators for pattern-aware algorithms.

Figure 1 shows how the pattern-aware 4-clique listing algorithm of Algorithm 1 lists all the 4-cliques in the input graph. For example, consider the search tree rooted at vertex 4 (bottom right of Figure 1). The depth-0 task corresponds to the root vertex 4 (line 1 in Algorithm 1). According to line 2 in Algorithm 1, the candidate vertex set for depth-1 tasks is the neighbor set of the depth-0 task. Thus, there are three depth-1 tasks that correspond to vertex 1, 2, 3. When the depth-1 task of vertex 1 spawns children tasks of depth-2, the children tasks are pruned due to the symmetry breaking of line 4 in Algorithm 1. Finally, only the subgraph with vertices $\{4, 3, 2, 1\}$ is determined as a unique isomorphic subgraph. One noticeable thing is the intermediate results reuse in the algorithm. As can be seen from both Figure 1 and Algorithm 1, tasks with the same parent task use the same intermediate results from previous depths as set operation inputs.

Graph mining algorithms are both computation-intensive and memory-intensive due to the set operations and repeated neighbor set accesses. Therefore, specialized accelerators are necessary for a higher execution efficiency. FlexMiner [12] is the first accelerator based on pattern-aware algorithms. It uses multiple processing elements (PEs) to exploit the coarse-grained parallelism among different search trees. FINGERS [10] extracts fine-grained parallelism within a search tree to

**Table 1: Qualitative comparison of existing task scheduling schemes**

| Scheduling Scheme | Memory Access | | PE Utilization | |
|---|---|---|---|---|
| | memory footprints | data locality | parallelization | inter-depth barriers |
| BFS | - | + | + | - |
| DFS | + | - | - | + |
| Pseudo-DFS | + | + | + | - |
| Shogun | + | + | + | + |

improve the single-PE performance. SparseCore [45] designs stream-based ISA extension for general purpose processors. SISA [6] proposes set-centric ISA extension and programming model for in-memory processing. DIMMining [15] and NDMiner [50] are in-DIMM designs. They propose optimizations in terms of graph formats, set operation FUs, symmetry breaking avoiding, memory request reordering, etc.

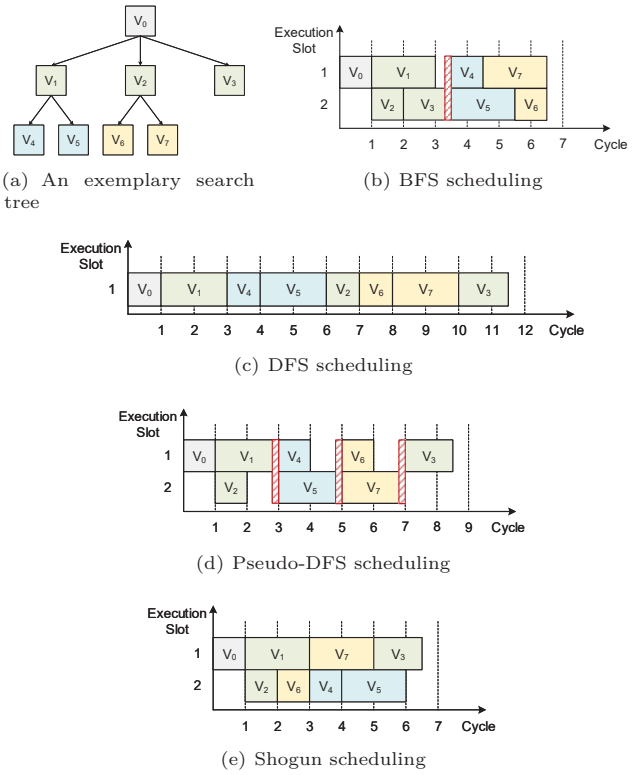
## 2.2 Task Scheduling in Graph Mining Accelerators

Graph mining constructs search trees from all vertices. Task scheduling in graph mining accelerators determines the search tree exploration order and thus substantially impacts the parallelism and performance. Apart from the ISA extension work that are oblivious to task scheduling [6, 45], the scheduling schemes of existing pattern-aware accelerators can be divided into three categories [10, 12, 15, 50].

**BFS:** BFS has only been adopted by some pattern-oblivious graph mining software systems [8, 52, 54, 55, 59], though it is still included here for comparison. Figure 2(b) shows a BFS scheduling example. After depth-0 task $V_0$ finishes execution, it spawns three children tasks ($V_1$, $V_2$, $V_3$), which are selected to execute together. Inter-depth barriers exist to keep the parent-child relationships between tasks. BFS has a high parallelism by executing multiple same-depth tasks together. Besides, these same-depth tasks have a high locality in accessing the intermediate results. Although BFS seems ideal according to Figure 2, it would suffers from memory consumption explosion when encountering high-degree vertices [53].

**DFS:** Most accelerators adopt the DFS scheduling [12, 15, 50], as shown in Figure 2(c). After task $V_1$ finishes execution, its children task $V_4$ has a higher execution priority over the same-depth tasks $V_2$ and $V_3$. DFS is friendly to task scheduler design because the tasks can be managed with a stack. It also minimizes memory footprints compared with BFS. However, its poor parallelism reduces the PE utilization rate. Moreover, DFS has a poor locality in accessing intermediate results because same-depth tasks are not executed together.

**Pseudo-DFS:** Pseudo-DFS has only been adopted by FINGERS [10] and some recent software frameworks [9, 53]. As shown in Figure 2(d), after task $V_0$ generates children tasks, task $V_1$ and task $V_2$ are selected as a task group for execution. After the whole task group completes, a new task group,




(a) An exemplary search tree

(b) BFS scheduling

(c) DFS scheduling

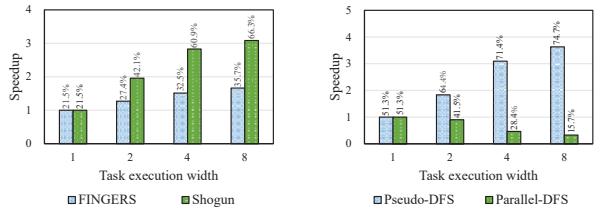
(d) Pseudo-DFS scheduling

(e) Shogun scheduling

**Figure 2: Scheduling schemes in graph mining accelerators. In Figure 2(a), each vertex is associated with a task. The PE can execute at most 2 tasks in parallel, although DFS only uses 1 execution slot. Red bars in BFS and pseudo-DFS are the barriers between different search depths. The blank space between barriers are the wasted PE resources. BFS prioritizes same-depth tasks over children tasks. DFS prioritizes children tasks over same-depth tasks. Pesudo-DFS fetches a task group (e.g., $V_1$ and $V_2$) for parallel task execution. After the whole group completes, pseudo-DFS prioritizes children tasks. Shogun removes inter-depth barriers so that different-depth tasks can be executed out-of-order to improve the PE utilization rate. In Figure 2(e), task $V_3$ is scheduled for execution after the subtree from task $V_2$ (including task $V_2$, $V_6$, $V_7$) completes in order to control memory consumption.**

which is composed of children tasks of task $V_1$, is selected for execution. Pseudo-DFS combines the benefits of both BFS and DFS. It controls memory footprints by configuring the task group size. It improves the parallelism and locality by scheduling multiple same-depth tasks to execute together. However, it inherits the inter-depth barrier from BFS scheduling. The barrier forces early-completed tasks to stall and wait until the last task in the group completes. Such idleness ceils the potential PE utilization rate.

As summarized in Table 1, existing scheduling schemes share weaknesses in either memory access or PE utilization. The analysis above reveals that the root cause lies in the scheduling order of tasks. For ease of discussion, this paper




(a) (b)

**Figure 3: Speedup comparisons between pseudo-DFS scheduling and parallel-DFS scheduling as the task execution width increases. Parallel-DFS explores multiple independent search trees on one PE and is an extreme case of out-of-order scheduling. Task execution width decides the maximum number of parallelly executing tasks. It is the task group size for pseudo-DFS and the number of independent search trees for parallel-DFS. In Figure 3(a), the input graph is AstroPh [36] and the search pattern is 4-clique. The numbers on the bars are the average FU utilization rates. In Figure 3(b), the input graph is Youtube [36] and the search pattern is tailed triangle. The numbers on the bars are the L1 cache hit rates.**

terms the result vertex set of a task where its children tasks are spawned from as **candidate set**, and same-depth tasks that are spawned from the same parent task/candidate set as **sibling tasks**. Prioritizing sibling tasks improves the intermediate result locality, whereas prioritizing different-depth tasks minimizes the impact of inter-depth barriers.

Shogun always tries to schedule abundant sibling tasks to occupy the whole PE execution width for higher locality. But Shogun is distinct from existing schemes by leveraging the runtime performance to judiciously schedule different-depth tasks for barrier-free out-of-order execution. Figure 2(e) shows a possible search tree exploration order of Shogun and the detailed scheduling scheme is in Section 3. Assuming a task execution width of 2, Shogun usually tries to execute sibling tasks in parallel, e.g., $V_1$ and $V_2$ are executed together. Shogun is barrier-free because after $V_2$ completes, $V_2$ does not wait for the slower $V_1$ to complete but generates children tasks instantly. Meanwhile, in order to improve the parallelism, Shogun selects the newly generated task $V_6$ for execution. Executing different-depth tasks (e.g., $V_1$ and $V_6$) out-of-order may incur locality loss, but Shogun incorporates a locality monitoring mechanism to avoid the performance degradation.

## 2.3 Shogun Insight

To resolve both the parallelism and locality issues, this paper identifies two insights which inspire the design of Shogun.

**Insight 1:** A task can be reordered with other tasks, so long as there is no parent-child relationship between them. Out-of-order task scheduling is necessary for further improving single PE utilization due to the removal of barriers.

To illustrate the first insight, Figure 3(a) introduces parallel-DFS scheduling and compares it to pseudo-DFS. Parallel-DFS explores multiple search trees on one PE in parallel. These search trees are independent from each other. Hence, parallel-DFS schedules tasks of different search trees out-of-order and is barrier-free. Parallel-DFS has not been adopted by any accelerator design but is compared here to highlight the benefits of out-of-order scheduling. Figure 3(a) shows that although increasing the task execution width is effective in application speedup, the upper limit of pseudo-DFS speedup and FU utilization rate is far from ideal, exposing a huge disparity with parallel-DFS. Barrier-free out-of-order scheduling is evidently effective in improving PE utilization rate.

**Table 2: Average number of input intermediate data cache lines requred by each task**

| Datasets | tc | tt | 4cl | 5cl | dia | 4cyc |
|---|---|---|---|---|---|---|
| Wiki-Vote [34] | 7.0 | 0.9 | 3.1 | 1.8 | 0.4 | 7.6 |
| AstroPh [36] | 2.8 | 0.4 | 2.2 | 2.3 | 0.1 | 4.9 |
| Youtube [36] | 2.4 | 11.3 | 1.9 | 1.5 | 0.7 | 20.1 |
| Patents [35] | 1.1 | 0.7 | 1.0 | 1.0 | 0.5 | 1.5 |
| LiveJournal [5] | 4.1 | 0.9 | 7.9 | 15.7 | 0.3 | 13.8 |
| Orkut [57] | 12.7 | 2.7 | 4.8 | 3.0 | 0.8 | 29.8 |

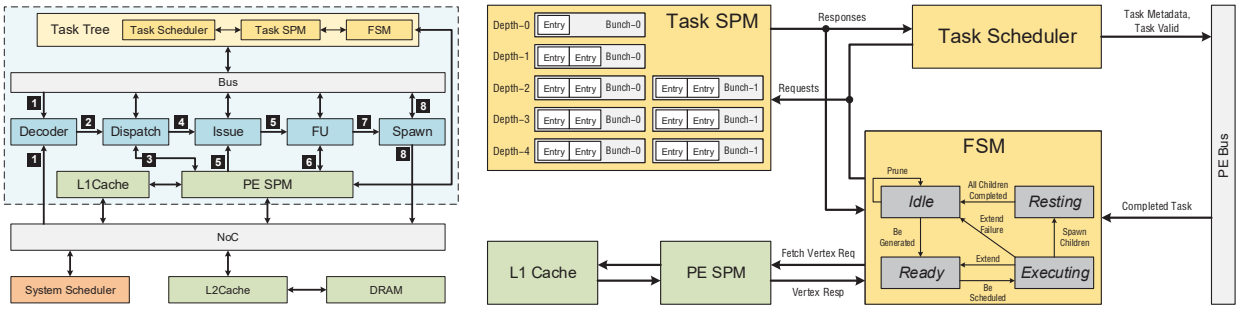
**Insight 2:** Generally speaking, intermediate data locality is not a concern for graph mining. But a locality-monitoring mechanism is necessary. Otherwise, fetching numerous intermediate results for independent tasks without any data locality could result in cache thrashing [26, 43], thereby seriously degrading the performance.

To demonstrate that locality is not a concern in common scenarios, Table 2 shows the average number of intermediate data cache lines required by each task in a recent graph mining system GraphPi [47]. Each cache line is 64-byte and consists of 16 integers. For a 32KB cache with 512 cache lines, the intermediate results required by multiple tasks can be simultaneously accommodated without incurring cache thrashing. Therefore, even if sibling tasks are not executed together as in BFS and pseudo-DFS, a later sibling task still has a high probability to find the required data in cache and the performance is not harmed by the locality loss.

Meanwhile, a sophisticated locality-monitoring mechanism is indispensable as highlighted by Figure 3(b). Figure 3(b) uses a larger input graph and a more memory-bound search pattern. As the task execution width increases, the poor locality of parallel-DFS incurs cache thrashing and a low hit rate, thus steeply degrading the performance.

# 3 SHOGUN TASK SCHEDULING

This section details the Shogun task scheduling implementation. Shogun is distinct from other designs by using a task tree structure to decouple task generation and execution. After a task completes execution and gets the candidate set for children tasks, it can directly spawn children tasks without being delayed by late tasks. Newly generated tasks are stored in the task tree. A scheduler in the task tree schedules new tasks out of order to remove inter-depth barriers. To improve the intermediate result locality, the task generation process produces bunches of sibling tasks, whose number equals the




(a) (b)

**Figure 4: Figure 4(a) shows the PE components, which are enclosed by the light blue dashed box. Each of the blue pipeline units include multiple stages. Figure 4(b) probes into the task tree structure and its connections with the other PE components. The task tree incorporates a spm, a fsm and a scheduler.**

PE execution width, i.e., these sibling tasks can cover the maximum number of parallelly running tasks on a PE. Correspondingly, the task scheduler tries to select sibling tasks to execute together and occupy the whole PE, thereby improving both the parallelism and locality. Section 3.1 gives the accelerator architecture and PE pipeline design. But Shogun is a general scheduling framework and can be implemented in other accelerator forms. Section 3.2 elaborates on the task tree design and scheduling scheme.

## 3.1 Accelerator Architecture and PE Pipeline

In the accelerator, a centralized system scheduler, multiple PEs and a shared L2 cache are connected with a network-on-chip (NoC). The system scheduler is responsible for dispatching root vertices of search trees to each PE. Each PE independently explores search trees constructed from the assigned root vertices and informs the system scheduler when the exploration completes.

Figure 4(a) shows the memory system with green boxes, which is composed of four types of components: scratchpad memory (spm), L1 cache, L2 cache and DRAM. Each PE owns a private L1 cache and spm. FUs in the PE directly interact with the spm for inputs and outputs of set operations. Following previous designs [10], intermediate results are stored in L1 cache and are evicted to L2 cache if replacement occurs. Graph data in the CSR form is stored only in L2 cache due to the streaming access pattern and relatively large quantity.

Figure 4(a) also shows the PE pipeline design enclosed in the light blue box. There are five pipeline units (blue boxes) and each unit includes several pipeline stages. The decoder unit fetches new tasks from the system scheduler or the task tree (❶), and decode them for dispatch (❷). The dispatch unit interacts with the PE spm to allocate cache lines (❸). After PE spm cache lines are allocated, the dispatch unit issues requests to fetch input data from the L1/L2 cache into the spm cache lines (❸), and forwards the task to the issue unit (❹). The task is issued once all input data are ready (❺). FUs take inputs from PE spm, conduct set operations of the issued task, and write the outputs into PE spm (❻). For large-degree vertices, the above pipeline stages may occur for multiple rounds [10]. After all computations are done (❼), the spawn unit interacts with the task tree for task generation or informs the system scheduler when the whole search tree has been explored (❽). Newly generated tasks are stored in the task tree. The task tree decouples the task generation (❽) and execution (❶). A scheduler in the task tree assesses the parallelism and locality, and determines the task execution order.

## 3.2 Task Tree and Task Scheduling Scheme

The task tree is the main structure that differentiates Shogun from previous designs [10, 12]. Previous designs adopt a task stack structure which merely stores the task information. Children tasks are executed once they are generated, so that the PE explores the search tree in a DFS or pseudo-DFS manner.

As a contrast, apart from recording the task information and relationship (Section 3.2.1), task tree handles two more jobs in a decoupling manner. First, the task tree manages out-of-order search tree exploration (Section 3.2.2). Second, the task scheduler determines the task execution order, taking both parallelism and locality into consideration (Section 3.2.3).

*3.2.1 Task tree structure.* As depicted in Figure 4(b), the task tree consists of three modules: a spm for task information storage, a finite-state machine (fsm) to control state transitions and actions, and a scheduler to decide the task execution order.

The task spm is statically arranged with two dimensions, *Depth* and *Bunch*. The *Depth* dimension corresponds to the search depth in the search tree. The *Bunch* dimension prepares groups of same-parent sibling tasks for scheduling. The number of entries per bunch is the same as the PE task execution width. In this way, the scheduler can ideally schedule as many sibling tasks as the PE task execution width. If there are not enough sibling tasks, non-sibling tasks from a different bunch are selected to improve the parallelism.

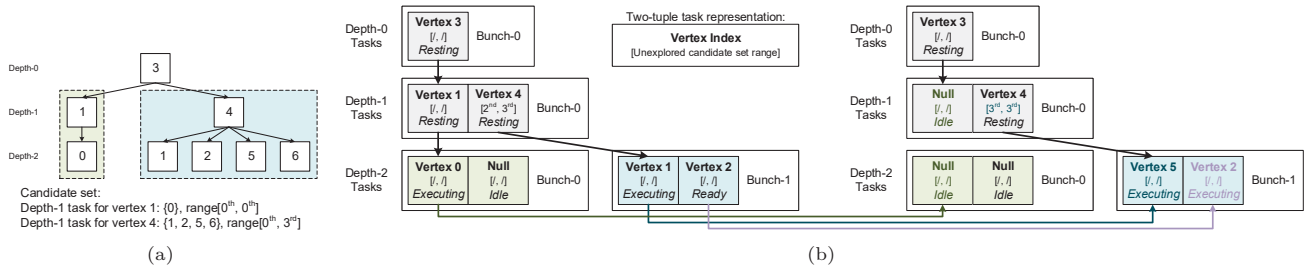



**Figure 5: A task tree management example. Figure 5(a) is the exemplary search tree. White blocks are tasks/vertices and the numbers are vertex indexes. The candidate sets of depth-1 tasks are given at the bottom. Figure 5(b) shows the corresponding task spm contents. Both the PE execution width and number of bunches per depth is 2 (except for depth-0). Tasks are stored with the two-tuple representation and each task entry has a state. Bold arrows between a task and a bunch represent the parent-child relationships. In the middle of Figure 5(b), two depth-2 tasks are in *Executing* state while the second depth-2 bunch-1 task is still in *Ready* state due to the execution width limit. From the middle of Figure 5(b) to the right, the subtree spawned from the first depth-1 task is completed, task spm entries involved in the subtree transition to *Idle* state. The first depth-2 bunch-1 task completes execution, then it extends to search on the next candidate vertex of its parent task, which is vertex 5. Because there is now an available task execution slot, the second depth-2 bunch-1 task also enters *Executing* state.**

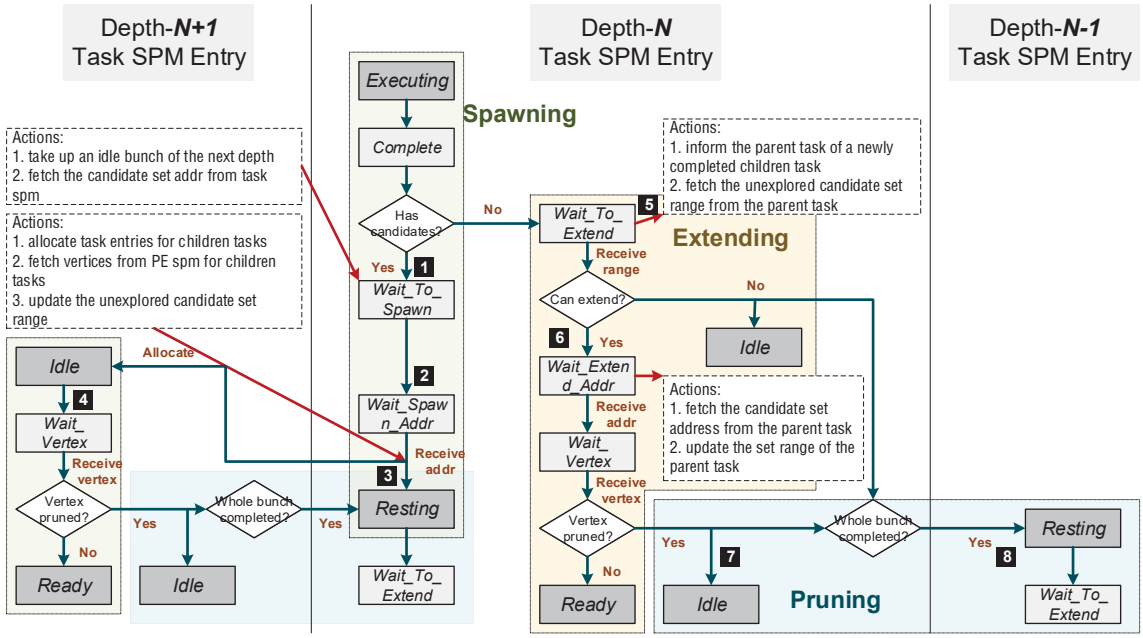



**Figure 6: Flow chart to illustrate state transitions in Shogun. The figure involves task spm entries from different depths, as highlighted at the top of the figure. State transitions on light green background are for the spawning process. State transitions on light yellow background are for the extending process. State transitions on light blue background are for the pruning process. Dark gray boxes are the four basic states and light gray boxes are the transient states.**

The task spm entry stores the task information with a two-tuple representation as shown in the middle of Figure 5(b). Each spm entry is also associated with a state. The task tree fsm controls the state transitions and actions of task spm entries. Figure 4(b) shows four basic states of the fsm:

**Idle**: A free task spm entry is in *Idle* state.

**Ready**: When a task is newly generated but has not been selected for execution, the entry is in *Ready* state.

**Executing**: The task is in the PE pipelines of Figure 4(a).

**Resting**: When a task has generated children tasks and is waiting for its candidate set to be completely explored, the entry is in *Resting* state. The entry cannot be recycled because its intermediate results might be used by children tasks.

In order to deal with cache and spm accesses, more transient states are actually used as detailed in following sections.

*3.2.2 Search tree exploration.* In exploring search trees such as the ones in Figure 1, the task tree should deal with three cases, task spawning, task extending and task pruning. Figure 5 illustrates what occurs in the task spm with an example. Figure 6 details the fsm state transitions.

**Task spawning:** In the middle of Figure 5(b), the depth-0 task generates two children tasks and transitions to *Resting* state. Its unexplored candidate set range becomes [/, /] because all its candidates (vertex 1 and 4) are being searched. Similarly, in the depth-1 bunch, both tasks have spawned depth-2 children tasks. The depth-1 task for vertex 4 has a range of [$2^{nd}$, $3^{rd}$] because it has only generated two children tasks from the four candidate vertices.

In Figure 6, state transitions on light green background are for the spawning process. The final goal is that the parent task transitions from *Executing* to *Resting* state (❸), while the generated children task occupies a new task spm entry, and transitions from *Idle* to *Ready* state (light green box in the depth-$N$ 1 task column).

When a task enters *Complete* state after finishing execution, if it has candidates, it enters $Wait\_To\_Spawn$ state for generating children tasks (❶ in Figure 6). The task tree takes up an idle bunch of the next search depth in order to store new tasks. Then the task fetches its candidate set address from the task spm, which is used when fetching vertices for children tasks. After these two actions, the task enters $Wait\_Spawn\_Addr$ state (❷). When the address is received, the task allocates task spm entries in the occupied bunch and fetches vertices for children tasks, updates the unexplored candidate set range, and transitions to *Resting* state (❸). Meanwhile, the states of children task entries transition from *Idle* to $Wait\_Vertex$ (❹). Finally, after receiving the vertices, children tasks transition to *Ready* state for scheduling.

**Task extending:** When a task completes execution, if the candidate set is empty or the task has reached the last depth, the task cannot spawn children tasks. Instead, the task should search on the next unexplored candidate vertex of its parent task. This process is termed as task extending. In Figure 5(b), when the depth-2 bunch-1 task for vertex 1 completes execution but cannot spawn, it extends and explores on vertex 5. The unexplored candidate set range of the parent task is updated from [$2^{nd}$, $3^{rd}$] to [$3^{rd}$, $3^{rd}$] because the $2^{nd}$ candidate is now being explored by the extended task.

In Figure 6, state transitions on light yellow background are for the extending process. When a task has no candidate to spawn children tasks, it enters $Wait\_To\_Extend$ state (❺). The task fetches the unexplored candidate set range from its parent task. If the range indicates that the parent task has some candidates left unexplored, the task is able to extend and transitions to $Wait\_Extend\_Addr$ state (❻), and further transitions to $Wait\_Vertex$ state after receiving the candidate set address from the parent task.

**Task pruning:** When the vertex being searched by a task does not satisfy the symmetry breaking conditions, the task should be pruned. Task pruning is necessary to ensure uniqueness in finding isomorphic subgraphs. Shogun conducts task pruning when fetching the vertex for a new task. During the task spawning or extending process, a task should fetch the corresponding vertex from the parent task's candidate set. If the fetched vertex satisfies the pruning conditions, the task is abandoned. Because this paper makes a common assumption that all vertex sets are sorted by ascending vertex index [10, 12, 47], the rest of the parent task's candidates will also satisfy the pruning condition and there is no need to explore them (Algorithm 1 uses the *break* statement in line 4, 7, 10 for the same reason).

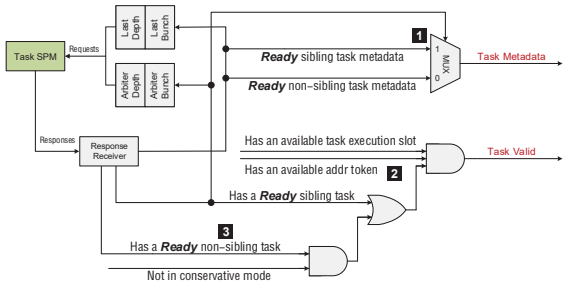



**Figure 7: Task tree scheduler design. The scheduler sends requests to the task spm to see if there are *Ready*-state sibling or non-sibling tasks. Based on the task spm responses, the scheduler outputs the selected task metadata and a Boolean value to indicate if there is a valid task being selected for execution.**

In Figure 6, state transitions on light blue background are for the extending process. When a child task receives a vertex but the vertex is pruned, the task transitions to *Idle* state (❼). Additionally, the fsm checks whether the whole bunch has become idle. If so, the bunch is recycled for future task spawning, and the parent task is informed that all its candidates have been explored (❽). The parent task should enter $Wait\_To\_Extend$ state for extending.

*3.2.3 Task scheduling.* As shown in Figure 7, the task tree scheduler generates two outputs. 'Task Valid' is a Boolean value indicating whether there is a valid task being selected for execution. 'Task Metadata' is the selected task information. When 'Task Valid' is true, the scheduler places the task information on the PE bus for the decoder unit to fetch.

When selecting a task for the task metadata output, the scheduler always prefers a sibling task, which is from the same bunch of the last selected task (❶ in Figure 7). However, when the number of sibling tasks in a bunch is not enough to completely cover the PE's execution width, Shogun flexibly allows non-sibling tasks to be executed together. The task tree maintains multiple bunches per depth to deal with such case. Although some locality is sacrificed due to the out-of-order scheduling, according to Section 2, the benefit of higher parallelism usually exceeds the performance degeneration resulting from locality loss. As for selecting a different bunch, this paper adopts a round-robin mechanism, although

other mechanisms (e.g., starting from the lowest depth) show negligible performance difference.

The task valid output decides if the selected task can be scheduled (❷). It depends on several conditions. The first one is that there is an available task execution slot so that the scheduled task can be executed later. The second condition is that there is an available address token for the new task to store intermediate results. This condition serves for controlling the memory footprints. Shogun preallocates empty vertex sets for each search depth before the application begins [37, 47]. Each vertex set is associated with a unique token, and tasks of the same search depth contend for a set of tokens. By default, the number of tokens assigned to each depth is equivalent to the PE execution width and the bunch size, so that ideally all sibling tasks of a bunch can execute together. But the number of tokens can be adjusted dynamically in order to reduce memory consumption.

The task valid output also depends on whether there is a *Ready*-state sibling/non-sibling task and the PE is in conservative mode (❸). If only non-sibling tasks exist and the PE is in conservative mode, the non-sibling task cannot be scheduled in case a severe locality loss induces cache thrashing and seriously degrades the performance. The conservative mode strictly disallows non-sibling tasks to execute together and can alleviate the cache thrashing.

Shogun transitions to the conservative mode when two conditions are satisfied. The first condition is that the L1 cache is suffering from cache thrashing. L1 cache only stores the intermediate results. If there is no thrashing, then performance is not degraded due to thrashing and improving the locality would not benefit the performance.. The L1 cache thrashing is judged by the average cache access latency. When thrashing occurs, a recently visited cache block is likely to be replaced soon, and the average cache latency can be extraordinarily high. The second condition is that when L1 cache thrashing occurs, the PE throughput should be low, which indicates that thrashing does degrade performance and improving the locality is necessary. The PE throughput is judged by the average FU utilization rate.

## 4 ACCELERATOR OPTIMIZATIONS

Based on the Shogun scheduling of Section 3, this section presents two accelerator optimizations, which prior accelerators cannot exploit due to their inflexible scheduling schemes.

### 4.1 Task tree splitting

In graph mining accelerators, although the runtime difference between independent search trees could be outstanding, the fact that each PE needs to explore numerous search trees veils the difference. However, load imbalance might still occur in certain cases and should be addressed. Existing accelerator designs lack load balance support because their DFS-style task scheduling cannot split a search tree onto multiple PEs.

Figure 8 illustrates the load balance procedures. The system scheduler is responsible for imbalance detection (❶). After dispatching all search trees, if most PEs have finished

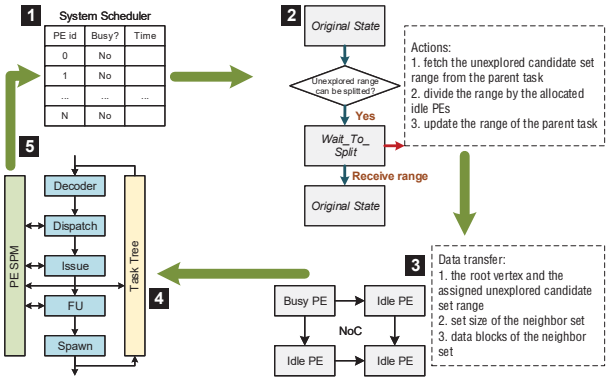



**Figure 8: An overview of load balance procedures. The system scheduler is responsible for detecting load imbalance occurrence. After detection, the scheduler instructs heavily-loaded PEs to split their task trees. These PEs transfer necessary data to idle PEs for execution. The load balance procedures can occur multiple times if load imbalance remains.**

all the assigned work while a few remain busy for a relatively long time, load imbalance is deemed to have occurred. Based on the number of idle PEs and busy PEs, the scheduler evenly apportions idle PEs to busy PEs for workload sharing.

When a busy PE receives a list of idle PEs for load balance, the PE tries to split its task tree into equivalent partitions (❷). Splitting the task tree for workload sharing should achieve a balance between the potential performance gain, the overhead of data transfer and the implementation complexity. Recognizing that pursuing a precise prediction is intractable and the data transfer overhead might offset the performance gain, this paper adopts a conservative but lightweight scheme that the task tree splitting occurs only at depth-1, i.e., the depth-0 root task is copied and deeper depths tasks are divided into several subtrees. This mechanism is based on two observations. First, splitting depth-1 tasks merely requires splitting the unexplored candidate set range of the depth-0 task, which is friendly to hardware implementation. Second, the intermediate results required by depth-1 tasks are indeed the neighbor set of the search tree root vertex, which could relieve the data transfer pressure. The task tree splitting is realized by adding a $Wait\_To\_Split$ state to the task tree fsm (❷). In this state, the task fetches the unexplored candidate set range of its parent task for partition.

After splitting the task tree, the busy PE transfers three types of partition message to the idle PEs (❸). The first type carries the root vertex and the assigned candidate set range so that the new idle PE can rebuild the task tree. The second type carries the set size of the root vertex neighbor set, which is used to allocate PE spm space in the idle PE. The third type involves the cache line data of the root vertex neighbor set. The generation and transferring of these messages reuse the existing PE spm and NoC structures and cause little hardware overhead. Although transferring the intermediate results may involve much NoC traffic, it is worthwhile compared with

other possible solutions. One possible solution is to make the original PE serve as a proxy for data access, i.e., the new PE sends memory access requests to the original PE. This solution requires repeated inter-PE traffic and is more expensive than the one adopted. Another possible solution is that the new PE directly fetches the intermediate results from L2 cache and DRAM, since the task tree splitting occurs only at depth-1 tasks and the intermediate results are merely neighbor sets of the search tree root vertex. However, the memory access to L2 cache and DRAM is more costly than the one-time data transfer between two PEs.

When all information of the partition has been transferred from the original PE to new PEs, the split task tree is executed on the new PEs as described in Section 3 (❹).

The load balance procedures can be conducted for multiple rounds to deal with the prediction uncertainty in load balance implementation (❺). For example, after splitting the original task tree onto multiple PEs, load imbalance might remain among the partitions. Such circumstance occurs due to the uncertainty of workload prediction. Another possible case is that the scheduler might allocate too many idle PEs to one busy PE. Therefore, the overhead of data transfer between these PEs offsets the performance gain of load balancing. This paper adopts a multi-round solution. The scheduler conservatively allocates at most 4 idle PEs to busy PEs for load balance. If load imbalance remains after all these procedures, the above procedures will be conducted again.

### 4.2 Search tree merging

In existing accelerator designs, a PE explores search trees one at a time in a serial manner. Such static relationship results in accelerator resource underutilization for certain graphs and patterns. On one hand, all of a PE's computing resources are exclusively manipulated by one search tree. On the other hand, the aggregated memory system bandwidth of the whole accelerator can be shared by at most $\#PEs$ search trees. For instance, it is observed that for the tailed triangle pattern, the computation density of Youtube [36] is more that 10 times over that of Patents [35]. Obviously existing accelerators cannot adapt to different characteristics of graphs and patterns.

To fully extract the potential for computing power and memory bandwidth under various application scenarios, this paper proposes a search tree merging scheme that allows a PE to process two search trees in parallel. In this manner, the accelerator resource can be shared by $\#PEs$ to $2 \times \#PEs$ search trees, yielding a better adaptivity.

Each PE makes independent decisions on whether to explore two search trees in parallel. There are three conditions to satisfy. First, the FU utilization rate is not high and there exists considerable performance improvement space. Second, the L1 cache is not suffering from thrashing, or as demonstrated in Figure 3(b), exploring independent search trees in parallel might drastically degrade the locality and performance. Third, the L2 cache and memory access latencies indicate that the memory system bandwidth has not been used up.

The implementation of exploring two search trees in parallel is quite simple with the task tree design of Section 3. In the original implementation of Figure 4(b), there is only one bunch for depth-0 and depth-1, because a search tree has one depth-0 root vertex and all the depth-1 tasks are sibling tasks generated from the root vertex. By adding one more bunch to both depth-0 and depth-1, the task tree can handle two search trees in parallel.

Finally, there is a recovery mechanism in case unexpected severe locality loss occurs due to the irregularity of graphs. This paper adopts a quiescing mechanism to stall the processing of one search tree for recovery. First, the PE should select a search tree to quiesce. Because both search trees occupy some bunches and vertex set tokens, quiescing either of them means leaving these occupied PE resources idle. To minimize the performance impact, the search tree that has a smaller maximum depth and occupies fewer bunches is selected. Second, when quiescing the selected search tree, task entries that are in *Resting*, *Ready* or *Complete* state are quiesced instantly. However, task entries that are in *Executing* state or transient state are not quiesced until they enter the *Resting*, *Ready* or *Complete* state. This is because for these tasks, there are pending memory system requests which are hard to track. If the task is instantly quiesced, these messages will be hanging in the accelerator, leading to deadlocks. After the selected search tree is completely quiesced and the other search tree completes exploration, the quiesced search tree is woken up for reexecution. The recovery process requires adding a *Quiesce* state to the task tree fsm and is easy to implement.

## 5 EVALUATION

### 5.1 Methodology

*5.1.1 Simulation configurations.* A cycle-level simulator is developed to evaluate Shogun. The main simulation configurations are given in Table 3. The basic computation fabric is similar to that of FINGERS [10]. Vertex sets are divided into fine-grained segments by dividers. Only paired segments will become inputs of set operations in intersection units (IUs).

In the task tree, the number of bunches per depth is 4 and the number of entries per bunch is 8. Both depth-0 and depth-1 have 2 bunches for search tree merging. The depth-0 bunches each has 1 entry and the depth-1 bunches each has 8 entries. The task execution width of both Shogun and FINGERS is 8. The task tree has 178 entries in total because the maximum depth is 6. GraphPi [47] is the only software system that manages to match 7-node patterns and we assume 6 is the largest depth possible.

After implementing the task tree and other necessary control logics in Verilog, and synthesizing using Synopsis Design Compiler under 28 $nm$ TSMC library, the area overhead of Shogun is approximately 0.037 $mm^2$ for a frequency of 1 GHz. We use the statistics of FINGERS [10] for the baseline, and Shogun incurs an area overhead of less than 4%.

**Table 3: Simulation configurations**

| | Simulator configurations |
|---|---|
| PEs | 10 PEs, task execution width 8, 178 task tree entries, 12 dividers, 24 IUs |
| Cache line size | 64 bytes |
| SPM | 16 KB per PE, 256 cache lines |
| L1 cache | 32 KB per PE, private, 4-way set associative |
| L2 cache | 4 MB, shared, 8-way set associative |
| Memory | Ramulator [31], 4 channels, DDR4-3200 |
| Search schedule | GraphPi [47] |
| Conservative mode transition conditions | (1) L1 cache average access latency $>$ 50 cycles (2) IU utilization rate $<$ 50% |

**Table 4: Evaluated graph datasets**

| Datasets | #Vertices | #Edges |
|---|---|---|
| Wiki-Vote (wi) [34] | 7.12 K | 100.37 K |
| AstroPh (as) [36] | 18.77 K | 198.11 K |
| Youtube (yo) [36] | 1.13 M | 2.99 M |
| Patents (pa) [35] | 3.77 M | 16.52 M |
| LiveJournal (lj) [5] | 4.00 M | 34.68 M |
| Orkut (or) [57] | 3.07 M | 117.19 M |

*5.1.2 Benchmarks.* Six commonly used patterns are evaluated: triangle (tc), tailed-triangle (tt), 4-clique (4cl), 5-clique (5cl), diamond (dia) and 4-cycle (4cyc). The graph mining schedules for these patterns are generated by GraphPi [47], although both Shogun and FINGERS are compatible with other optimized schedules. GraphPi is an edge-induced graph mining systems [39, 47]. Hence we modify GraphPi and also generate vertex-induced schedules for tt, dia and 4cyc. The suffixes '_e' and '_v' are used to clarify these two versions. The evaluated graph datasets are listed in Table 4. *wi* and *as* are small graphs and their graph data can be completely cached on chip. *yo* and *pa* are medium graphs with a very low average degree. But *yo* has a more significant degree variance than *pa*. *lj* and *or* are large graphs and they are more bottlenecked by the memory access to neighbor sets. Experiments that take longer than 4 days are excluded (lj-5cl, or-4cl, or-5cl, or-4cyc).

## 5.2 Evaluation of Shogun Scheduling Schemes

*5.2.1 Performance improvement.* Figure 9 shows the speedups that Shogun achieves over FINGERS. System optimizations in Section 4 are disabled in Shogun so as to evaluate only the scheduling scheme. Shogun improves the performance by 43% on average and up to 131% across the 47 cases. Figure 10 shows the average IU utilization rates of Shogun, which helps in understanding the effects of patterns and graphs. The IU utilization rate improvement of Shogun over FINGERS is the same as the performance improvement. Thus, the statistics of FINGERS are not given.

The attributes of each pattern and the corresponding search schedule plays a crucial role in the effectiveness of scheduling optimizations. Shogun is generally more suitable

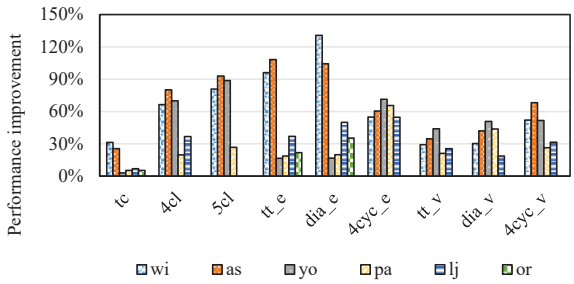



**Figure 9: Performance improvements of Shogun over FINGERS, all accelerator optimizations of Section 4 are disabled to compare the scheduling scheme.**

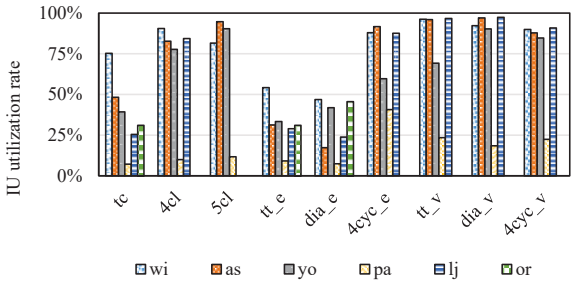



**Figure 10: Average IU utilization rates of Shogun. Dividing the utilization rates of Shogun by the speedups of Figure 9 can derive the utilization rates of FINGERS.**

for scenarios where the computation bottleneck is more outstanding than memory access bottleneck, and the pattern has more depths for extracting out-of-order parallelism.

**tc:** *tc* is the simplest pattern and almost all intermediate results can easily fit in L1 cache. Even for the largest evaluated graph orkut [57], the L1 cache hit rate exceeds 99%. Therefore, Shogun can fully enable out-of-order scheduling. However, because *tc* has a maximum depth of only 2, the exploitable parallelism is quite minimal and the computation is not intensive. Therefore, the speedups of applying Shogun and the IU utilization rates are modest.

**4cl, 5cl:** 4*cl* and 5*cl* are similar to *tc* as they all count size-$k$ cliques. All depths of tasks conduct set operations to materialize the intermediate results. Therefore, the quantity of intermediate results is small enough to be accommodated in L1 caches. However, 4*cl* and 5*cl* have more depths than *tc*. On one hand, 4*cl* and 5*cl* are more computation-intensive with higher IU utilization rates than other patterns. On the other hand, Shogun can extract more parallelism from out-of-order execution and achieve a better speedup.

Although both *yo* and *pa* are sparser compared with the others, their performance improvements polarize for 4*cl* and 5*cl*. This is due to the high skewness and low diameter of *yo*. Therefore, the exploitable parallelism matching 4*cl* and 5*cl* in *yo* is actually abundant and brings about a high speedup.

**tt_e, dia_e:** A prominent phenomenon for matching *tt_e* and *dia_e* is the low IU utilization rate. This is attributed to the low computation demands. *tt_e* and *dia_e* generate a large number of depth-2 tasks but only depth-1 tasks require intersection operations. In contrast, 4*cl* has set operations at every search depth, 4*cyc* has set operations for depth-2 tasks, whose number is the greatest among all depths.

As for the cases of matching *tt_e* and *dia_e* in *wi* and *as*, there is another factor that contributes to the low IU utilization rate. *wi* and *as* generate a tremendous number of tasks, whereas these tasks on average demand little computation or memory access. Therefore, *wi* and *as* spend most of the runtime in PE pipelines, e.g., accessing the task tree entries for task information. Although optimizing the PE pipeline design should ease this accelerator resource stall, this is outside the scope of this paper and we leave it for future work.

In contrast to the high performance improvements of matching 4*cl*, 5*cl* in *yo*, the improvements of $yo - tt\_e$ and $yo - dia\_e$ are marginal. In $yo - tt\_e$, depth-1 tasks fetch the neighbor set as the intermediate results used by depth-2 tasks. For highly skewed graphs such as *yo*, their large-degree vertices generate so many intermediate results that L1 cache trashing frequently occurs even though sibling tasks are executed together to improve the locality. $yo - tt\_e$ is thus bottlenecked by the access to intermediate results. In $yo - dia\_e$, due to the low average degree of *yo*, tasks on average demand little computation and each search tree does not generate enough tasks to fully utilize the PE on average. Nonetheless, the proposed search tree merging scheme will resolve this issue as validated later.

Due to the higher degrees, while matching *tt_e* and *dia_e*, *lj* and *or* also suffer the same intermediate result access bottlenecks as $yo - tt\_e$. However, the inherent parallelism of *lj* and *or* is also more abundant than *yo*. Therefore, *lj* and *or* exhibit higher performance improvements.

**4cyc_e, tt_v, dia_v, 4cyc_v:** These four patterns are very computation-intensive. 4*cyc_e* generates a huge number of depth-2 tasks from the neighbor sets of depth-1 vertices, and the intersection operation occurs just right in depth-2 tasks. *tt_v*, *dia_v* and 4*cyc_v* require intersection or subtraction at every search depth. These four patterns generate much larger memory footprints and the L1 cache miss rate is higher. Meanwhile, they exhibit higher IU utilization rates even for sparse graphs like *pa*.

In contrast, the speedup for *wi* and *as* is smaller compared with searching the other patterns. This is because FINGERS has already achieved a high IU rate due to the abundant fine-grained parallelism [10].

## 5.3 Evaluation of Accelerator Optimizations

*5.3.1 Task tree splitting evaluation.* Load imbalance is more likely to occur when two conditions are satisfied. First, the graph is highly skewed and the task runtime variance is significant. Second, the number of PEs is large, so that the number of independent search trees assigned to each PE is not large enough to tolerate the task runtime variance.

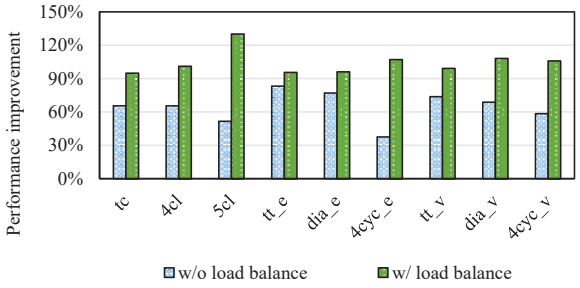



**Figure 11: Performance improvement comparison between Shogun with and without load balance. The input graph is Wiki-Vote [34] and the system has 20 PEs.**

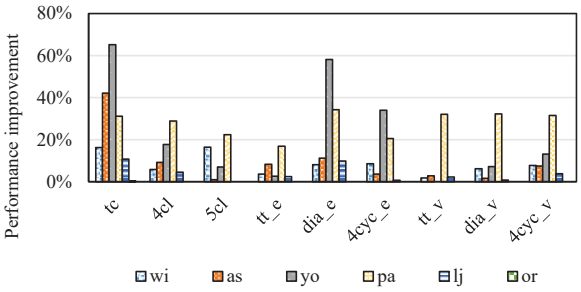
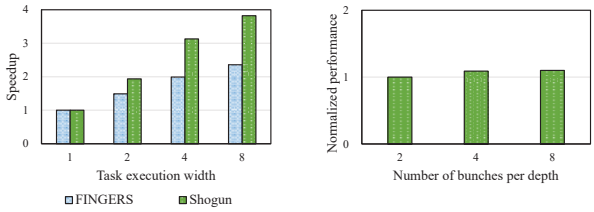



**Figure 12: Performance improvement comparison between Shogun with and without search tree merging.**



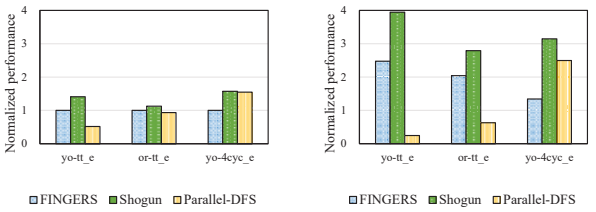

(a) Performance of Shogun and FINGERS as the task execution width increases

(b) Performance of Shogun as the number of bunches per depth increases

**Figure 13: Sensitivity study on the task execution width and the number of bunches per depth.**

Therefore, to illustrate the effectiveness of task tree splitting, Figure 11 compares the performance improvements of *wi* when the accelerator has 20 PEs. All the other experiments of Section 5 use 10 PEs and the load imbalance is almost negligible. Figure 11 shows that with load balance, the performance of Shogun can be further improved by 24%.

*5.3.2 Search tree merging evaluation.* Figure 12 shows the performance improvements when Shogun further supports search tree merging. *yo* and *pa* originally suffer from the poor parallelism due to their low degrees. Search tree merging is thus most effective for them. *wi* and *as* have limited




(a) Task execution width is 2, L1 cache size is 64 KB (b) Task execution width is 8, L1 cache size is 256 KB

**Figure 14: Comparison between Shogun, FINGERS and parallel-DFS. The performance are normalized to that of FINGERS when the task execution width is 2.**

improvements for *tt_e* and *dia_e* due to the accelerator resource bottleneck. *or* is barely improved because it already has abundant parallelism and has fully utilized memory bandwidth with neighbor set accessing. Combining the scheduling scheme with accelerator optimizations, the overall Shogun design finally gains a geomean speedup of 63%.

### 5.4 Sensitivity Study

Figure 13(a) conducts a sensitivity study on the task execution width (i.e., number of address tokens per depth). Shogun achieves a better scalability than FINGERS through the out-of-order scheduling. Figure 13(b) compares the Shogun performance with 2, 4, 8 bunches per depth. The average performance improvement increases by less 10%. Shogun is insensitive to the number of bunches because it is able to schedule tasks from different depths. Therefore, in this paper, the task spm entries/bunches are allocated to each depth in a static manner for hardware simplicity.

Figure 14 compares the performance of FINGERS, parallel-DFS and Shogun to demonstrate the necessity of locality monitoring. Only three cases are given because parallel-DFS and Shogun usually have a very similar performance. Since parallel-DFS suffers from L1 cache thrashing, we conservatively enlarge the L1 cache size to ameliorate the impact. In Figure 14(a), the task execution width is 2 and the L1 cache size is 64 KB. In In Figure 14(b), the task execution width is 8 and the L1 cache size is 256 KB. Even though L1 cache size is enlarged, parallel-DFS could still suffer from cache thrashing for troublesome graphs and patterns, whereas Shogun has a conservative mode to avoid cache thrashing.

## 6 CONCLUSION

Existing graph mining accelerator scheduling schemes are not satisfying in terms of memory access and PE utilization. This paper proposes Shogun, a locality-aware out-of-order task scheduling framework. Based on the scheduling design, Shogun further proposes task tree splitting and search tree merging. Shogun improves the performance by 63% on average with a hardware overhead of less than 4%.

## ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (Grant No. 61834002, No. 62104014), and in part by the National Key R&D Program of China (Grant No. 2021YFB2701201).

## REFERENCES

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 12 pages.

[2] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1247–1262.

[3] Balázs Adamcsek, Gergely Palla, Illés J. Farkas, Imre Derényi, and Tamás Vicsek. 2006. CFinder: Locating Cliques and Overlapping Modules in Biological Networks. *Bioinformatics* 22, 8 (2006), 1021–1023.

[4] 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.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 44–54.

[6] 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 *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 282–297.

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

[8] 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*.

[9] Jingji Chen and Xuehai Qian. 2021. Kudu: An Efficient and Scalable Distributed Graph Pattern Mining Engine.

[10] 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*. 43–55.

[11] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *Proceedings of the ACM International Conference on Supercomputing*. 378–391.

[12] 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)*. 581–594.

[13] Young-Rae Cho and Aidong Zhang. 2010. Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks. *IEEE Transactions on Information Technology in Biomedicine* 14, 1 (2010), 30–36.

[14] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 595–608.

[15] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. DIMMining: Pruning-Efficient and Parallel Graph Mining on near-Memory-Computing. In *Proceedings of the 49th Annual*

*International Symposium on Computer Architecture*. 130–145.
[16] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. 2005. Frequent Sub-Structure-Based Approaches for Classifying Chemical Compounds. *Knowledge and Data Engineering, IEEE Transactions on* 17 (2005), 1036– 1050.
[17] Vinicius Dias, Carlos H. C. 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.
[18] Alexandra Duma and Alexandru Topirceanu. 2014. A network motif based approach for classifying online social networks. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. 311–315. https://doi.org/10.1109/SACI.2014.6840083
[19] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.* (2014), 517–528.
[20] Katherine Faust. 2010. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks - SOC NETWORKS* 32 (2010), 221–233.
[21] Dima Feldman and Yuval Shavitt. 2008. Automatic Large Scale Generation of Internet PoP Level Maps. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*.
[22] Benoit Gaüzère, Luc Brun, and Didier Villemin. 2011. Two New Graph Kernels and Applications to Chemoinformatics. In *Proceedings of the 8th International Conference on Graph-Based Representations in Pattern Recognition*. 112–121.
[23] Mark Granovetter. 1983. The Strength of Weak Ties: A Network Theory Revisited. *Sociological Theory* 1, 1983 (1983), 201–233.
[24] Yang Hu, Hang Liu, and H. Howie Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 171–182.
[25] J Huan, Wei Wang, and Jan Prins. 2003. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. *Proceedings of the Third IEEE International Conference on Data Mining.IEEE Computer Society*, 549– 552.
[26] Aamer Jaleel, Kevin Theobald, Simon Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM Sigarch Computer Architecture News* 38 (2010), 60–71.
[27] 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*.
[28] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 228–241.
[29] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data*. 1231–1245.
[30] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. *SIGMOD '18: Proceedings of the 2018 International Conference on Management of Data*, 411–426.
[31] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49.
[32] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Chew. 2008. Scheduling strategies for optimistic parallel execution of irregular programs. *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 217–228.
[33] Yutaka I. Leon-Suematsu, Kentaro Inui, Sadao Kurohashi, and Yutaka Kidawara. 2011. Web Spam Detection by Exploring Densely Connected Subgraphs. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, Vol. 1. 124–129.
[34] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting Positive and Negative Links in Online Social Networks. In *Proceedings of the 19th International Conference on World Wide Web*. 641–650.
[35] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. 177–187.
[36] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection.
[37] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. 2021. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 289–303.
[38] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 21–37.
[39] 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.
[40] Tijana Milenković and Natasa Przulj. 2008. Uncovering biological network function via graphlet degree signatures. *Cancer Inform* (2008), 257–273.
[41] 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)*. 1–7.
[42] Roger Pearce, Trevor Steil, Benjamin W. Priest, and Geoffrey Sanders. 2019. One Quadrillion Triangles Queried on One Million Processors. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–5.
[43] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 381–391.
[44] Liva Ralaivola, Sanjay J. Swamidass, Hiroto Saigo, and Pierre Baldi. 2005. Graph kernels for chemical informatics. *Neural networks* 18, 8 (2005), 1093–1110.
[45] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. SparseCore: Stream ISA and Processor Specialization for Sparse Computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 186–199.
[46] 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.
[47] 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*. 1–14.
[48] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. *Proceedings - International Conference on Data Engineering* 2015 (2015), 149–160.
[49] Ichigaku Takigawa and Hiroshi Mamitsuka. 2012. Graph mining: Procedure, application to drug discovery and recent advances. *Drug discovery today* 18 (08 2012).
[50] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. NDMiner: Accelerating Graph Pattern Mining Using near Data Processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 146–159.
[51] Lei Tang and Huan Liu. 2010. *Graph Mining Applications to Social Network Analysis*. Vol. 40. 487–513.
[52] Carlos H. C. 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.
[53] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 209–224.
[54] 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 *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. 763–782.

[55] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, and John C. S. Lui. 2020. G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1369–1380.

[56] Xifeng Yan and Jiawei Han. 2002. GSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*. 721.

[57] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *2012 IEEE 12th International Conference on Data Mining*. 745–754.

[58] Yifan Yang, Zhaoshi Li, Yangdong Deng, Zhiwei Liu, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. GraphABCD: Scaling Out Graph Analytics with Asynchronous Block Coordinate Descent. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 419–432.

[59] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Xueqi Cheng. 2019. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine.