G-ThinkerCG: A Hybrid CPU-GPU Framework for Efficient Parallel Subgraph Finding

Akhlaque Ahmad, Da Yan, Lyuheng Yuan, Saugat Adhikari, Jiao Han, and Naman Nimbale
Indiana University Bloomington
Email: {akahmad, yanda, lyyuan, adhiksa, jiaohan, nnimbale}@iu.edu

***Abstract*—Given a large graph G, *subgraph finding* (SF) refers to a general category of problems that find all G's subgraphs meeting problem-specific criteria. SF is usually solved by searching a *subgraph enumeration tree* whose size grows exponentially with the graph size. While recent parallel solutions to SF on CPUs and GPUs have shown that the computation can finish in reasonable time for large graphs, there still lacks a unified framework that can efficiently leverage both CPU and GPU while maintaining load balancing and bounded GPU memory usage.**

In this paper, we introduce G-thinkerCG, a scalable subgraph-centric framework for SF problems that supports efficient CPU-GPU co-processing. G-thinkerCG shows that effective BFS-DFS hybrid strategy for subgraph extension, previously only explored in CPUs, is practical for GPU execution by introducing a novel *host subgraph buffering* scheme when GPU memory is insufficient. G-thinkerCG also inherits the efficient DFS-based subgraph extension model of G-thinker in CPUs, which is good at processing straggler subgraph-extension tasks that cannot saturate the massive threads of a GPU. Advanced dynamic work stealing is also supported within CPU threads and GPU warps, and between CPU and GPU. A user-friendly subgraph-centric programming interface is provided to facilitate the implementation of applications, unifying the subgraph-centric interfaces of previous CPU-only and GPU-only frameworks that are familiar to their users. Two representative applications are implemented on G-thinkerCG, and extensive experiments show that G-thinkerCG with hybrid CPU-GPU execution beats CPU- and GPU-only executions, and is more scalable than prior works.

***Index Terms*—graph, GPU, CPU-GPU, subgraph finding, maximal clique, subgraph matching.**

I. Introduction

Let $G = (V, E)$ be a graph where V and E denote the sets of vertices and edges, respectively. We consider the problem of finding (or counting) subgraphs of G that satisfy specific conditions—referred to as *subgraph finding* (SF). SF problems arise in various application domains such as social and biological network analysis [20]. However, SF is expensive to compute due to its huge search space, which is the power set of V in the worst case: for each subset $S \subseteq V$, one verifies whether the subgraph induced by S satisfies the given conditions. The problem is typically solved using a recursive backtracking algorithm that performs depth-first search (DFS) over a redundancy-free subgraph enumeration tree (see Section II-A), but each application has its own subgraph pruning techniques that make the overall computation tractable.

Existing GPU solutions to SF extend subgraphs by either breadth-first search (BFS) or DFS. While BFS allows coalesced memory access and hence a higher GPU processing throughput, all subgraphs with i vertices (or edges) need to be materialized and extended to the subgraphs with $(i + 1)$ vertices (or edges) at each iteration i, which can easily run out of the device memory (OOM). To avoid running OOM, a common approach is to extend subgraph enumeration subtrees for one chunk of initial vertices at a time. However, BFS from one chunk can still run OOM, so G^2-AIMD [55] proposes an adaptive chunk-size adjustment strategy to handle this issue.

In contrast, DFS approaches [47], [58] are memory-efficient since each computing unit (i.e., a thread block or a warp) performs DFS of its assigned subgraph enumeration subtrees using its preallocated stack, and can achieve competitive performance when using proper load balancing. However, since the stacks need to be preallocated in device memory, it is usually for SF problems like subgraph matching with a fixed depth of subgraph extension (decided by the query graph size), but not very viable for problems such as dense subgraph mining where the extension depth is not known beforehand.

A BFS-DFS hybrid solution to candidate extension has recently been explored in the CPU context for search problems [11], [57], where at the i^{th} level, we only extend a certain number of candidates, say, at most η, to be placed into the candidate buffer for the $(i+1)^{\text{th}}$ level, after which we proceed to extend those candidates at the $(i+1)^{\text{th}}$ level. The remaining candidates in the level-i buffer will be processed when the computation backtracks to level i again.

There are **two challenges in implementing this BFS-DFS hybrid solution in GPU:** (1) the solution needs to know the number of levels (denoted by L) beforehand, to preallocate L candidate buffers each with capacity η, but the extension depth is not known beforehand for many SF problems such as maximal clique finding; (2) a GPU usually has a much smaller device memory than the host memory, so deep levels of candidate extensions can cause OOM error. We address these issues in our GPU-based BFS-DFS solution (which is the first of its kind) by introducing a novel *host subgraph buffering* scheme when GPU memory is insufficient.

Another opportunity to speed up SF problems is by CPU-GPU co-processing, which has not been explored before. Note that CPU and GPU have **complementary** capabilities: (1) due to the power-law degree distribution of real graphs, most initial vertices can only be extended to a shallow number of levels, so each level can contain enough computing workloads to saturate the GPU resource (using BFS or BFS-DFS hybrid search); (2) however, some initial vertices can be extended

towards very deep levels, such as those high-degree ones in dense regions, which can be better handled by CPU threads by recursive depth-first search. Therefore, we can leverage the best of both worlds when candidates are properly assigned to either CPU or GPU based on their workload characteristics.

There are **two challenges in unifying CPU and GPU for co-processing. First,** although both CPU and GPU contexts use a subgraph-centric model, their programming models are different: (i) in a GPU kernel, we specify how a candidate subgraph in the level-i buffer can be extended into new candidates in the level-$(i+1)$ buffer, and the depth-first scheduling of such extension for memory saving is handled by the systems themselves; (ii) while the SOTA systems for CPU-parallel SF computing [35], [51], [52], [56] ask users to write a recursive *compute*(g) function as in the serial SF algorithm counterpart for depth-first set-enumeration tree search (where g is a candidate subgraph), and to only decompose g's computation into smaller tasks (corresponding to set-enumeration subtrees under g) that can be processed in parallel for straggler elimination when g has been processed for a too long time period.

The **second challenge** is the mismatch of load balancing schemes. As we explained, CPU-parallel SF systems use a timeout mechanism to decompose straggler subgraph DFS tasks, while GPU-parallel SF systems schedule subgraph-extension tasks themselves and handles load balancing by best efforts: if there are not enough candidates that get extended to level i, then GPU cannot saturate the massive threads when processing level-i candidate subgraphs (which are better handled by CPU threads, and should be stolen from GPU).

We unify the two different programming models by the concept of *subgraph-task*, which is the task of extending a (candidate) subgraph g to examine all its extended candidates. In this way, users can specify how a subgraph-task g can be extended into new subgraph-tasks in the next level in GPU programming model, and how g is recursively extended in CPU programming model. A subgraph-task is also the smallest unit of task stealing between CPUs and GPUs.

Based on the above ideas, we design our **G-thinkerCG** framework for SF computing, which effectively supports *CPU-GPU co-processing* and on the GPU side, *BFS-DFS* hybrid scheme for subgraph extension, both are the first of its kind. G-thinkerCG routes the initial subgraph-tasks to their proper processors, always tries to finish existing chunks of initial subgraph-tasks before taking a new chunk for processing to keep the memory cost bounded. G-thinkerCG conducts work stealing to ensure load balancing of subgraph-tasks among GPU and the CPU threads. In summary, our contributions are:

- A hybrid BFS-DFS approach to subgraph extension on a GPU is proposed that does not need to know the maximum extension depth beforehand, and supports subgraph spilling to host memory when device memory is low.
- A unified programming and work-stealing model is proposed based on the concept of subgraph-task, which allows G-thinkerCG to support efficient CPU-GPU co-processing, where each processor can focus on the tasks that it is good at. The programming model of G-thinkerCG keeps the subgraph-centric interfaces of prior CPU- and GPU-only SF systems familiar to their users.
- Two representative algorithms are implemented on G-thinkerCG, and extensive experiments are conducted to show its superior performance over baseline approaches.

The rest of this paper is organized as follows. Section II reviews the preliminaries and the related work. Then, Section III presents our hybrid BFS-DFS scheme for subgraph extension on a GPU, and Section IV describes the computation model, programming interface and system design of G-thinkerCG, and briefly introduces the two applications on top. Finally, Section V reports our experiments, and Section VI concludes this work and discusses future work.

II. Preliminaries and Related Work

We first review the background of GPU and SF problems, and then review the related work on parallel systems for SF.

A. Preliminaries

GPU Background. A GPU connects to the host CPU via the PCI-e bus. In CUDA, developers write device programs, known as *kernels*, which are explicitly launched by a CPU program to execute on the GPU. Each kernel runs in parallel across many threads organized into *blocks*. Blocks are scheduled to streaming multiprocessors (SMs) for independent execution, and each block is divided into *warps*—groups of 32 threads. A warp is the smallest scheduling unit in an SM with its 32 threads executing the same instruction simultaneously. If threads in a warp follow different control paths (e.g., branching), all execution paths are serialized using masking.

GPU threads cannot access host memory directly. Data must be explicitly transferred to device memory. CUDA Unified Memory provides a unified address space accessible by both CPU and GPU; space is allocated in host memory, and runtime transparently migrates pages to device memory as needed.

Given the GPU's memory hierarchy and warp-level parallelism, coalesced memory access is essential for performance. In GPU-parallel SF systems, the adjacency list of a vertex or the vertex set of a subgraph is processed collaboratively by an entire warp to ensure coalesced memory access. For instance, let $deg(v)$ be the degree of vertex v; a warp executes the following loop to process v's neighbors:

$$\text{for}(i = \text{LANE\_ID};\ \ i < deg(v);\ \ i \mathrel{+}= 32),$$

where LANE_ID is the ID of a thread in its warp. For example, thread 2 processes items at positions $i = 2, 34, 66, \cdots$.

Subgraph Finding (SF). Subgraph finding is typically solved using a recursive backtracking algorithm that performs DFS on a subgraph enumeration tree. We illustrate by considering the Bron–Kerbosch (BK) algorithm [7] for Maximal Clique Enumeration (MCE) as shown in Algorithm 1, which recursively enumerates all maximal cliques in a graph $G = (V, E)$.

Specifically, BK maintains three disjoint vertex sets: (1) R, containing vertices in the current subgraph; (2) P, the set of candidates that can extend R into a larger clique; and (3) X, the exclusion set used to ensure maximality. Figure 1 illustrates

Algorithm 1 Bron-Kerbosch Algorithm for Maximal Cliques

```
1: BronKerbosch(∅, V, ∅)
2: procedure BronKerbosch(R, P, X)
3:     if P = ∅ and X = ∅ then output R
4:     for all v ∈ P do
5:         BronKerbosch(R ∪ {v}, P ∩ N(v), X ∩ N(v))
6:         P ← P \ {v}
7:         X ← X ∪ {v}
```

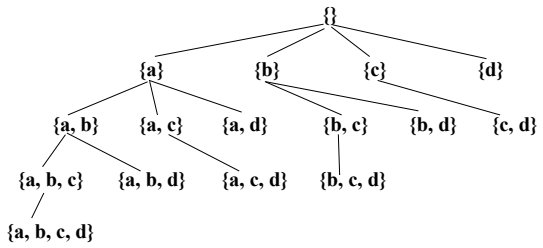


Fig. 1. Set-Enumeration Tree

the recursion tree (aka. subgraph enumeration tree) for a graph with four vertices $\{a, b, c, d\}$, where each node is annotated with the set R and corresponds to a subgraph of G induced by R. Let $N(v)$ denote the set of neighbors of vertex v. The algorithm ensures that every vertex in P and X is adjacent to all vertices in R, as enforced via Line 5 which prunes non-neighbors when a new vertex v is added to R. The exclusion set X tracks those candidates previously explored in sibling branches. For example, consider node $\{a, c\}$ in Figure 1: since b has been used to extend $\{a\}$ into its sibling $\{a, b\}$ on the left, Line 7 will add b to X for the branch rooted at $\{a, c\}$. Later, for example, if $\{a, c, d\}$ is found to be a clique but $b \in X$, then Line 3 will not output $\{a, c, d\}$ since adding b will produce a larger clique (already outputted by the branch under $\{a, b\}$). When P and X are both empty (Line 3), there are no further vertices that can be added to R, so R is a maximal clique.

Besides MCE, this set-enumeration process is also widely adopted by SF algorithms for finding maximum clique [51], [52], maximum k-plex [4], [8], [18], maximal k-plexes [13], [15], [45], maximum quasi-clique [10], maximal quasi-cliques [21], [27], [29], and size-bounded community [54].

Another representative SF problem is subgraph matching (SM), which enumerates all subgraphs of a data graph G that are isomorphic to a query graph G_q. Many SM algorithms have been proposed based on Ullmann's algorithm [42], which orders the query vertices and recursively extends the partial mapping by iterating over candidate data vertices for the next query vertex and pruning infeasible extensions. Note that this process essentially searches the subgraph enumeration tree over the data graph G. Please see details in Appendix A [3].

The Subgraph-Task Abstraction. From the above examples, we can see the following common characteristics of SF algorithms: (1) DFS is conducted over a subgraph enumeration tree, where each node represents a subgraph instance (for further extension) with a set of vertices already included. (2) The fanout at each node can vary significantly due to power-law degree distribution. (3) The subgraph enumeration tree is redundancy-free: each subgraph instance is generated exactly once. Let S be the vertex set of a node (i.e., a subgraph instance) in the subgraph enumeration tree $\mathcal{T}$, then we denote the **task** of examining S and extending larger subgraphs from S by $\boldsymbol{T_S}$. Note that T_S corresponds to the subgraph enumeration **subtree** rooted at node S in $\mathcal{T}$. This definition unifies the concepts of 'subgraphs' and 'tasks'.

B. Related Work

Graph Partitioning vs. Search Space Partitioning. Vertex-centric parallel systems such as Pregel [31] and its variants [14], [19], [30], [49] rely on graph partitioning to distribute vertices across machines for iterative computation by message passing. While GPU-accelerated vertex-centric systems exist [6], [17], [28], [34], [38], [44], [61], they are also for low-complexity iterative algorithms [50].

For SF problems where algorithms are recursive rather than iterative, a *subgraph-centric* model is more appropriate, where computations are performed at the subgraph level, and larger candidate subgraphs are extended from smaller ones. This strategy, termed *search space partitioning*, partitions the search space so that the searched subgraphs may overlap but there is no redundant computation (i.e., the same subgraph is checked at most once). For example, in Figure 1, the tasks (or, subtrees rooted at) $\{a\}$, $\{b\}$, $\{c\}$ and $\{d\}$ are redundancy-free and can be processed concurrently and independently, even though subgraphs in the subtrees under them share vertices. This holds recursively, for example, among all sibling nodes in the subgraph enumeration tree, such as $\{a, b\}$, $\{a, c\}$ and $\{a, d\}$.

General-Purpose Subgraph-Centric Systems for SF. Systems like Arabesque [41], RStream [43], Fractal [16], Pangolin [12], and GAMMA [23] adopt a general extend+filter programming model. These systems incrementally grow subgraphs by adding adjacent vertices/edges, and apply filtering to retain valid subgraphs. Among them, only Pangolin and GAMMA support GPU execution: Pangolin [12] struggles with large graphs due to BFS-style subgraph extensions since it does not handle OOM caused by a single chunk, while GAMMA [23] supports out-of-core processing (using host memory) for scalability but compromises efficiency due to the intensive use of joins that are expensive.

Although the serial SF algorithms naturally avoid redundancy in subgraph enumeration, earlier systems such as Arabesque [41], RStream [43] and Fractal [16] extend subgraphs naïvely, so they need to conduct graph isomorphism checks to remove redundant subgraphs that are isomorphic but extended from different smaller subgraphs. Pangolin [12] allows application developers to implement aggressive pruning strategies to reduce the enumeration search space, and to apply customized pattern classification methods to elide generic isomorphism tests. Peregrine [24] adopts a pattern-based programming model that treats graph patterns as first-class constructs and enables the extraction of pattern semantics to guide its exploration. However, these programming models are still not as straightforward to use as that of our G-thinkerCG framework, where parallel versions of the

redundancy-free SF algorithms can be implemented just like their serial counterparts without introducing any additional overheads caused by a new programming model.

G-thinker [20], [51], [52] and G-Miner [9] are pioneering distributed systems that allow users to specify backtracking algorithms as in their serial recursive algorithm counterparts to avoid materializing intermediate subgraphs as much as possible, where those vertices and their adjacency lists needed for task computation are dynamically requested and cached to each local machine for repeated visit to avoid communication. T-thinker [27] and G-thinkerQ [56] are the single-machine versions of G-thinker where the graph G is memory-resident, with G-thinkerQ [56] supporting online subgraph queries.

In the GPU context, G^2-AIMD [55] is a general-purpose subgraph-centric framework that conducts BFS-style subgraph extensions one chunk at a time, where each subgraph is extended by a warp. When a chunk runs out of the device memory, G^2-AIMD multiplicatively reduces the chunk size and retry again, while a successfully processed chunk increases the chunk size linearly. This is inspired by the AIMD algorithm in TCP congestion control, but the computation involved in a failed chunk is wasted. G^2-AIMD is the work closest to our current work, but our G-thinkerCG avoids wasted computation via a novel hybrid BFS-DFS approach for subgraph extension, and it additionally supports CPU-GPU co-processing. G-thinkerCG retains the user-friendly subgraph-centric programming interface of G^2-AIMD for GPU execution, with slight modifications to allow users to implement application programs with better load balancing.

Discussion. Except for G^2-AIMD [55] which targets general SF problems, existing GPU works focus on designing tailor-made GPU algorithms for a specific SF problem such as MCE or SM. Appendix B [3] reviews them in detail.

As a system paper, we simply reuse the application implementations from G^2-AIMD [55] on the GPU side, and those from G-thinkerQ [56] on the CPU side. Our goal is to show that the design of G-thinkerCG allows more efficient execution on a GPU as well as with CPU-GPU co-processing when running these applications, so application-specific tailor-made optimization techniques such as those in MCE-GPU [5] are currently not integrated into the application code (they may further improve performance if integrated). For a more detailed review of the systems for SF problems, please refer to [53].

III. Hybrid BFS-DFS Subgraph Extension on a GPU

Overview. Our GPU program runs a **GPU worker**, which is a CPU thread that calls kernels (i.e., GPU device functions) to perform the subgraph computations and extensions. To illustrate the idea of our BFS-DFS hybrid strategy for subgraph extension, we prepare a simplified example in Figure 2, where we assume: (1) subgraphs in the subgraph enumeration tree are denoted by A, B, $\cdots$; (2) a chunk of initial subgraphs $\{A, B\}$ are being extended; (3) each subgraph in levels 1 and 2 (denoted by L_1 and L_2) is extended to exactly three larger subgraphs; (4) each subgraph in L_3 does not produce new subgraphs, i.e., Level 3 is the last level; (5) subgraphs

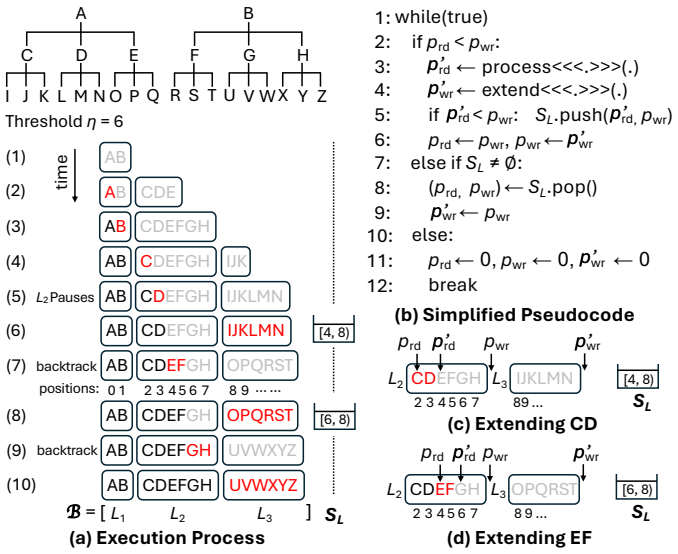


Fig. 2. Example of Hybrid BFS-DFS Subgraph Extension

are maintained in a **buffer** $\mathcal{B}$ that keeps the segment of subgraphs in L_1, followed by the subgraph segments L_2, L_3, etc.; (6) each subgraph occupies one position in $\mathcal{B}$ (it is a simplification and we will see the data structure in GPU later).

We request users to specify two device functions for an application: (1) **process(.)**, which computes the candidate elements (i.e., vertices or edges) to extend subgraphs at the current level L_i, and (2) **extend(.)**, which uses these candidate elements to generate new subgraphs at level L_{i+1}. Subgraphs are fetched from L_i by warps, which append newly extended subgraphs to L_{i+1}. At each level L_i, we extend at most η new subgraphs into L_{i+1} to bound the device memory consumption, where η is chosen to be much larger than the number of warps to saturate the GPU resource. Information of the remaining segment of subgraphs in L_i is pushed to a **stack** S_L, so that they will be processed by backtracking later when the subgraph enumeration subtrees of all current subgraphs in L_i have been fully extended.

Figure 2(a) shows the entire subgraph enumeration tree at the top, followed by 10 steps of BFS-DFS subgraph extension where we assume $\eta = 6$: (1) initially, L_1 contains the chunk $\{A, B\}$; (2) A is first extended which appends C, D, E to L_2; (3) B is then extended which appends F, G, H to L_2; (4) the processing then switches to L_2 where C is extended into I, J, K in L_3; (5) then D is extended into L, M, N, and since $\eta = 6$ subgraphs in L_3 have been appended, the processing switches to L_3; (6) since L_3 is the last level, I, J, K, L, M, N are fully processed and the processing then backtracks to L_2 to process the remaining subgraphs at position $[4, 8)$ in $\mathcal{B}$; (7) E, F are then extended to append $\eta = 6$ subgraphs in L_3; (8) these subgraphs in L_3 are then processed; (9) after backtracking, G, H are then extended; (10) finally, these extended subgraphs in L_3 are processed.

Figure 2(b) shows the pseudocode of the GPU worker's backtracking algorithm that we have illustrated with the example above, where each iteration handles the subgraph extension from the current level L_i to level L_{i+1}. Specifically,

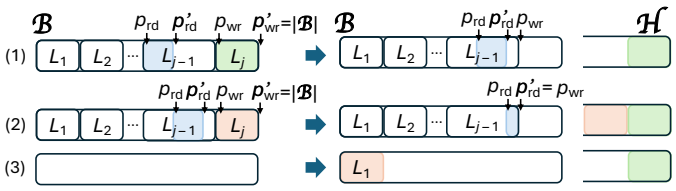


Fig. 3. Example of Host Buffering of Subgraphs

let $L_i = \mathcal{B}[p_{rd}, p_{wr})$ and $L_{i+1} = \mathcal{B}[p_{wr}, p'_{wr})$, where p_{rd}, p_{wr} and p'_{wr} are positions in $\mathcal{B}$. Additionally, p'_{rd} records the position of the next subgraph to process, i.e., subgraphs in $\mathcal{B}[p_{rd}, p'_{rd}) \in L_i$ have been processed. These position variables are illustrated in Figure 2(c)–(d).

In Line 2, we check if $L_i = \mathcal{B}[p_{rd}, p_{wr})$ is not empty. If so, in Line 3, kernel process(.) first checks the subgraphs from L_i to compute up to η candidate elements for extending these subgraphs (those candidates are usually stored in user-defined buffers in device memory), and p'_{rd} is updated so that $\mathcal{B}[p'_{rd}, p_{wr}) \in L_i$ keeps the remaining subgraphs in L_i to process. Then in Line 4, kernel extend(.) extends these subgraphs (i.e., $\mathcal{B}[p_{rd}, p'_{rd})$) into L_{i+1}, and p'_{wr} is updated to reflect the tail of L_{i+1}. Here, process(.) lets each warp read a subgraph at a time to compute candidate elements, while extend(.) lets each thread read a candidate element to extend its corresponding subgraph, so that if a subgraph has too many candidates, they can be extended by the threads of multiple warps for better load balancing. This is an improvement to G^2-AIMD's interface that only has extend(.) (called process-Subgraphs(.) in [55]) where after each warp reads a subgraph, it has to extend subgraphs for all candidates using only its 32 threads. Line 5 then pushes the positions $[p'_{rd}, p_{wr})$ of remaining subgraphs in L_i to the stack S_L for later processing when backtracking, and Line 6 updates p_{rd} to point to the beginning of L_{i+1}, and updates p_{wr} to point to the end of L_{i+1}, so that the next iteration will process level L_{i+1}.

Figure 2(c) shows an example where after subgraphs C, D are extended and L_3 reaches $\eta = 6$ subgraphs, $[4, 8)$ is pushed to S_L indicating that $\mathcal{B}[4, 8)$ are the remaining vertices to process when backtracking to L_2 again. Figure 2(d) shows an example where after E, F are extended, $[6, 8)$ is pushed to S_L indicating that $\mathcal{B}[6, 8)$ are the remaining vertices to process when backtracking to L_2 again.

If Line 2 finds that L_i is empty, then if the stack S_L is not empty (Line 7), Line 8 pops the previous level back as L_i and Line 9 clears L_{i+1} so that next iteration will process the remaining elements of L_i to extend new subgraphs into L_{i+1}.

On the other hand, if S_L is empty, then the backtracking algorithm has finished processing the current chunk $\{A, B\}$, so Line 11 clears the entire $\mathcal{B}$ and Line 12 breaks out of the while loop.

Stack-Based Host Buffering of Subgraphs. In CUDA implementation, we can preallocate a large space for $\mathcal{B}$ in the device memory using `cudaMalloc()`, but if a recursion path in the set-enumeration tree is deep, it is possible that the subgraphs in levels L_1, L_2, ..., L_j could exhaust the space of $\mathcal{B}$, even if there are more levels L_{j+1}, ... that need further extension. To support such cases, we need to maintain a host buffer $\mathcal{H}$ to keep those subgraphs that overflow from $\mathcal{B}$. The spilled subgraphs can be loaded back from $\mathcal{H}$ to $\mathcal{B}$ for processing later when $\mathcal{B}$ has space.

Figure 3 illustrates this process, where we assume overflow happens at level L_j. (1) When kernel extend(.) extends subgraphs in level L_{j-1} into L_j, if a warp finds that the next write-position p'_{wr} reaches $|\mathcal{B}|$ or beyond (which is possible since multiple warps are atomically forwarding p'_{wr}), it stops fetching more subgraphs from L_{j-1} for extending. When extend(.) returns, p'_{wr} is set to $|\mathcal{B}|$, and all subgraphs in L_j are pushed to the host stack $\mathcal{H}$. (2) The processing then backtracks to L_{j-1} again to extend the remaining subgraphs in L_{j-1} into L_j; if L_j causes $\mathcal{B}$ to overflow again, then the subgraphs in L_j is pushed to the host stack $\mathcal{H}$ again. Note that each time we extend at most so many subgraphs from L_{j-1} such that at most η subgraphs are extended to L_j, but if L_j reaches the end of $|\mathcal{B}|$, we stop extending subgraphs in L_{j-1} even if there are fewer than η subgraphs in L_j (if a subgraph is already extended, then the remaining extended subgraphs are pushed to $\mathcal{H}$ since $\mathcal{B}$ is full). Finally, (3) when the backtracking process for a chunk finishes, instead of taking a new chunk of initial subgraphs, we first move up to η subgraphs from $\mathcal{H}$ (if available) as a chunk for processing (note that this design is important in keeping the number of pending tasks in $\mathcal{H}$ as small as possible). In the last row of Figure 3, since all subgraphs fetched to L_1 of $\mathcal{B}$ were size-j subgraphs previously pushed to $\mathcal{H}$, we are basically extending the subgraphs from Level j to Level $(j+1)$ when extending from L_1 to L_2.

Finally, note that we choose stack rather than queue to implement $\mathcal{H}$, so that the search is near-DFS to process the subgraph enumeration subtrees of recently spilled subgraphs before those of their ancestors' sibling subgraphs. This, combined with our prioritizing of chunk fetching from subgraphs of $\mathcal{H}$ before new initial subgraphs, keeps the size of $\mathcal{H}$ as small as possible, which is important to avoid host OOM.

Subgraph Buffer Implementation. Recall that we simplified our discussion by assuming that each subgraph occupies exactly one position in $\mathcal{B}$ or $\mathcal{H}$, but in reality, it is represented by its vertex set, i.e., a segment of vertex IDs (using which one can get their adjacency lists from the graph G usually represented in CSR format). We implement both $\mathcal{B}$ and $\mathcal{H}$ using the same data structure `BufferBase` as summarized in Figure 4-❷. In this way, subgraphs are stored in the same format, so their movement between $\mathcal{B}$ and $\mathcal{H}$ can be automatically handled by the system without the need of calling user-specified functions. We introduce `BufferBase` next.

As Figure 4-❶ and ❷ shows, a `BufferBase` object is defined over two preallocated arrays: (1) **`vertices`**, which keeps the subgraph contents (usually vertex ID segments) one after another; and (2) **`offsets`**, where each element is a pair of integers `[start, end)` indicating that the subgraph is in `vertices[start, end)`. Figure 4-❶ shows three subgraphs in `vertices` along with their entries in `offsets` indicating their boundaries in `vertices`.

A BufferBase object actually tracks the segment of its subgraphs on these pre-allocated arrays (i.e., subarrays), so it is possible to define multiple BufferBase objects on the same arrays but pointing to different segments. As Figure 4-❶ and ❷ shows, three positions are tracked by a BufferBase object: (1) **ohead**, which is the beginning position (inclusive) of the buffer in offsets; (2) **otail**, which is the end position (exclusive) of the buffer in offsets; and (2) **vtail**, which is the end position (exclusive) of the buffer in vertices. In Figure 4-❶, the buffer contains exactly 3 subgraphs in vertices$[10, 50)$ as tracked by the 3 position trackers in offsets, highlighted in light gray. When this buffer is the current layer L_i in $\mathcal{B}$ to read subgraphs for extending, the dark gray prefix segment corresponds to previous layers that will be reached again when backtracking; while if this buffer is $\mathcal{H}$ which acts as a stack, there is no dark gray prefix segment since ohead is the bottom of the stack.

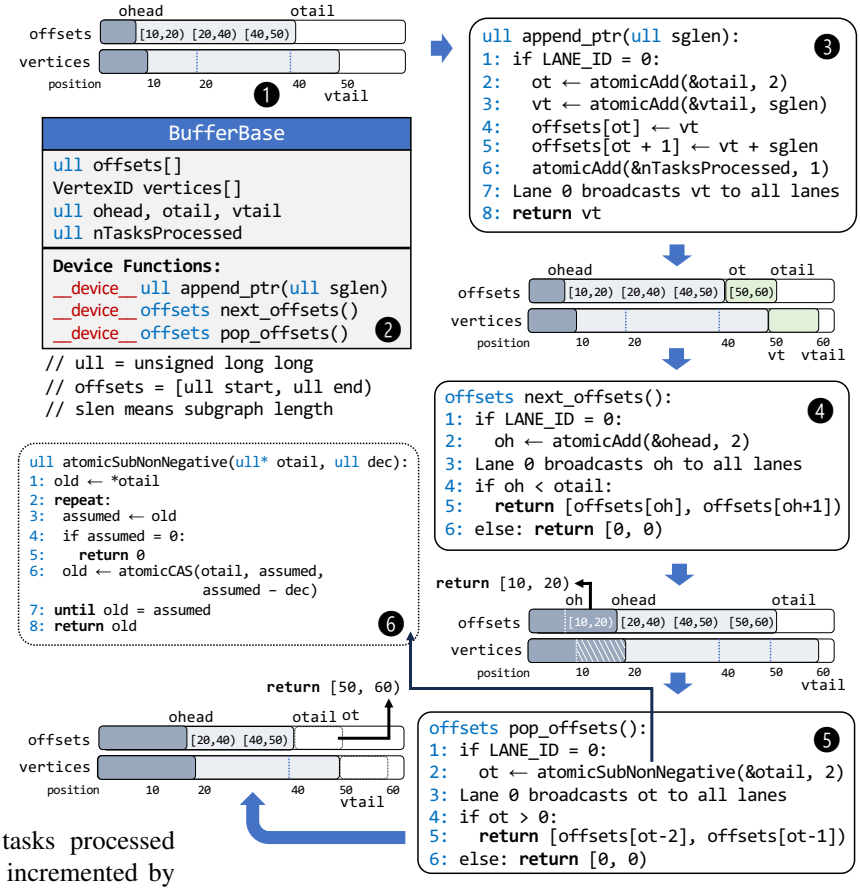


Fig. 4. BufferBase and Its Operations

As Figure 4-❷ shows, a BufferBase object also maintains a counter nTasksProcessed to track the number of tasks processed for the current iteration, which is atomically incremented by a warp when it appends a newly extended subgraph to the BufferBase object (see Figure 4-❸ Line 6). In user-defined functions (UDFs) such as extend(.), if a warp finds that nTasksProcessed $> \eta$, it exits the current iteration. Note that nTasksProcessed may be incremented beyond η by multiple warps, after they read nTasksProcessed and find it right below η; but all warps will exit the current iteration (e.g., for L_i) properly the first time they see nTasksProcessed $> \eta$. When the GPU worker proceeds to the next level L_{i+1}, or backtrack to the previous level L_{i-1} (when no subgraph is generated in L_{i+1}, and L_i is fully processed), it resets nTasksProcessed back to 0, so that the counter counts from 0 again when we process a new level.

Recall from Section II-A on Page 2 that given a segment indicated by start position s and the segment length len, the threads of an entire warp can read elements from or write elements to this segment in parallel using a for-loop that runs for $\lceil len/32 \rceil$ iterations. Therefore, to append a subgraph with $sglen$ elements to the BufferBase object L_{i+1} in a UDF, each thread of a warp calls the append_ptr(sglen) operation shown in Figure 4-❸, which properly updates trackers otail and vtail, and returns the start position vt for the warp to append the $sglen$ elements in parallel.

Specifically, in Lines 1–5, the first thread of a warp (LANE_ID $=$ 0) increments vtail by $sglen$ to reserve positions for the warp to copy the subgraph into vertices[.] (Line 3). Note that in CUDA, atomicAdd$(\&x, i)$ adds x by i and then returns the old value of x, so Line 3 returns v_t as the previous tail of L_{i+1} for appending a new subgraph, while vtail is updated to reflect the new tail after appending. To track this segment vertices[start, end) for the new subgraph, Line 2 also increments otail to reserve space for start and end, whose values are properly assigned in Lines 4 and 5, respectively. Finally, the first thread broadcasts the start position vt to all threads in the warp in Line 7 using __shfl_sync(.), which is then returned for the warp to copy subgraphs to vertices[vt, vt + sglen) in parallel. Note that while different warps are reserving space by atomically incrementing otail and vtail, the cost associated with this contention is minimal since atomic operations have been highly optimized by NVIDIA with native hardware support [36] as observed in [55] and [48].

To obtain a subgraph from the BufferBase object L_i in a UDF, each thread of a warp calls the next_offsets() operation shown in Figure 4-❹, which properly forwards ohead in Line 2, and returns the trackers [offsets[oh], offsets[oh+1]) for all threads of the warp to read the obtained subgraph from $\mathcal{B}$.vertices.

To pop a subgraph from the stack $\mathcal{H}$, each thread of a warp calls the pop_offsets(sglen) operation shown in Figure 4-❺, which properly decrements otail in Line 2, and returns the trackers [offsets[ot-2], ohead[ot-1]) for all threads of the warp to read the obtained subgraph from $\mathcal{H}$.vertices. Note that since we cannot decrement otail to be smaller than 0, we

cannot use `atomicSub(.)` since when multiple warps subtract 2 from `otail` consecutively, `otail` can become negative. We solve this issue by calling `atomicSubNonNegative(.)` as shown in Figure 4-❻, which is described in Appendix C [3].

In `BufferBase`, stack $\mathcal{H}$ is allocated by `cudaMallocManaged()` so both GPU kernels and CPU threads can access it. The latter is needed for G-thinkerCG to perform work stealing between CPU threads and the GPU. Position variables `ohead`, `otail` and `vtail` are also allocated using `cudaMallocManaged()`, allowing access from both user-defined kernels and the GPU worker — a CPU thread that runs the procedure in Figure 2(b) to process a chunk.

The GPU worker's backtracking algorithm in Figure 2(b) can be easily translated to use `BufferBase` for L_i and L_{i+1}, because for a level $L_i = \mathcal{B}[p_{rd}, p_{wr})$ in Figure 2, checking p_{rd} (resp. p_{wr}) is simply checking `ohead` (resp. `otail`), while updating p_{rd} (resp. p_{wr}) is simply updating `ohead` (resp. both `otail` and `vtail` together), and S_L now has entries of the format $\langle$`ohead`, `otail`, `vtail`$\rangle$. For example, Line 2 of Figure 2(b) checks if L_i is not empty, and we can now check the condition `ohead` $<$ `otail` instead of $p_{rd} < p_{wr}$.

IV. The G-thinkerCG System

This section introduces the computation model, programming interface, system design of G-thinkerCG, and the two applications on top.

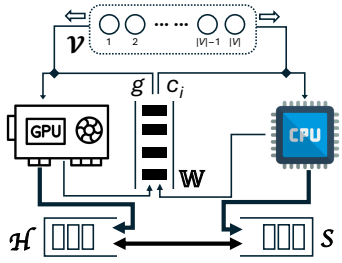


Fig. 5. System Overview

Overview. Figure 5 shows the system architecture of G-thinkerCG. Given a multi-core machine with a GPU, G-thinkerCG runs (1) a **master** thread that loads the graph in CSR format to both the host memory and the GPU device memory, creates the worker threads, and schedules their subgraph-centric computation; (2) a group of n_c CPU threads each operating as an independent **CPU worker**; and (3) a CPU thread called the **GPU worker** that calls GPU kernel functions for subgraph-centric computation. Idle CPU and GPU workers are placed in a queue $\mathbb{W}$ to be dequeued by the master (thread) for assigning (initial) vertex chunks or task chunks for computations.

The master initially loads the IDs of all vertices in G into a double-ended queue $\mathcal{V}$ from which vertex chunks are dequeued and assigned to the workers for processing, where each vertex corresponds to an initial subgraph that will be extended. It is common to sort the vertices in degeneracy order (i.e., order the vertices by iteratively removing the smallest-degree node, which is the peeling algorithm for k-core decomposition) [4], [13] to keep the candidate element size (for extending subgraphs) small, since smaller-degree neighbors have been considered and removed in previous branches. We can order vertices in $\mathcal{V}$ in degeneracy order, but the vertices at the tail still tend to have much deeper subgraph enumeration trees with very different depths, not suitable for level-by-level subgraph extension on a GPU. Therefore, as Figure 5 shows, when the master dequeues a worker w from $\mathbb{W}$, if w is a GPU worker, it dequeues a chunk of vertices from the front of $\mathcal{V}$; while if w is a CPU worker, it dequeues a chunk of vertices from the tail of $\mathcal{V}$. After a worker finishes its assigned chunk, it will enter $\mathbb{W}$ to wait for the master to assign new chunks. This waiting process has some overhead, so we need to set the vertex chunk size properly to amortize this cost. The chunk size is set as 50 for a CPU worker and 500K for a GPU worker, which we observe to work well.

Recall that each subgraph S is also a task T_S that searches the entire subgraph enumeration subtree rooted at S. We have seen the BFS-DFS hybrid computing model of the GPU worker that may spill subgraphs (i.e., tasks) to a stack $\mathcal{H}$ that uses host memory. We shall soon introduce the computing model of CPU workers after this overview, and we will see that a CPU worker processes each task (i.e., subtree) T_S but may decompose it into smaller tasks if the task runs beyond a timeout threshold τ_{time} ($\tau_{time} = 0.1$ second by default). Those tasks will be pushed to a global stack $\mathcal{S}$ dedicated for CPU-side spilled tasks, as shown in Figure 5, and they will be prioritized to be scheduled to idle CPU workers, and only when $\mathcal{S}$ is empty will a new chunk from $\mathcal{V}$ be dequeued for assigning. Note that like $\mathcal{H}$, we use stack for $\mathcal{S}$ to make the computation near-DFS to process the subgraph enumeration subtrees of recently spilled subgraphs before those of their ancestors' sibling subgraphs. Both the prioritized task scheduling from S and the stack choice of $\mathcal{S}$ help minimize the size of $\mathcal{S}$, which uses host memory. Finally, the spilled subgraph-tasks can be stolen between $\mathcal{S}$ and $\mathcal{H}$ for load balancing between CPU and GPU, as Figure 5 shows.

Figure 6 shows the more detailed system architecture, where we can see that each worker w maintains two buffers: (1) $\mathcal{L}_v$ which keeps a vertex chunk assigned by the master from $\mathcal{V}$ for processing (see ① and ②), and (2) $\mathcal{L}_t$ which keeps a group of tasks assigned by the master from $\mathcal{S}$ for processing (see ③ and ④). Note that ③ essentially steals tasks spilled from the CPU side to the GPU. In G-thinkerCG, the master assigns spilled tasks in $\mathcal{S}$ to the workers (see ③ and ④) since tasks in $\mathcal{S}$ and $\mathcal{L}_t$ are of the same type `TaskT`.

Master. Algorithm 2 shows the pseudocode of the master thread, where Line 1 creates CPU workers and the GPU worker and adds them to $\mathbb{W}$ to be scheduled for processing, and Line 2 initializes the stack $\mathcal{S}$ to be empty and initializes the vertex-chunk queue $\mathcal{V}$ to contain all vertices in G (with optional degeneracy ordering).

In G-thinkerCG, the master has a condition variable to allow it to wait by releasing its CPU core to avoid busy waiting, and any worker can use this condition variable to notify the master to wake up to resume execution. Similarly, each worker w has a condition variable through which the master can use to notify w to wake up. In Line 4, if the master finds $\mathbb{W}$ to be empty so there is no worker to schedule, it will wait and release its CPU core. Later if some worker enters $\mathbb{W}$, it will notify the master to wake up to continue execution from Line 5. In Line 5, if both $\mathcal{V}$

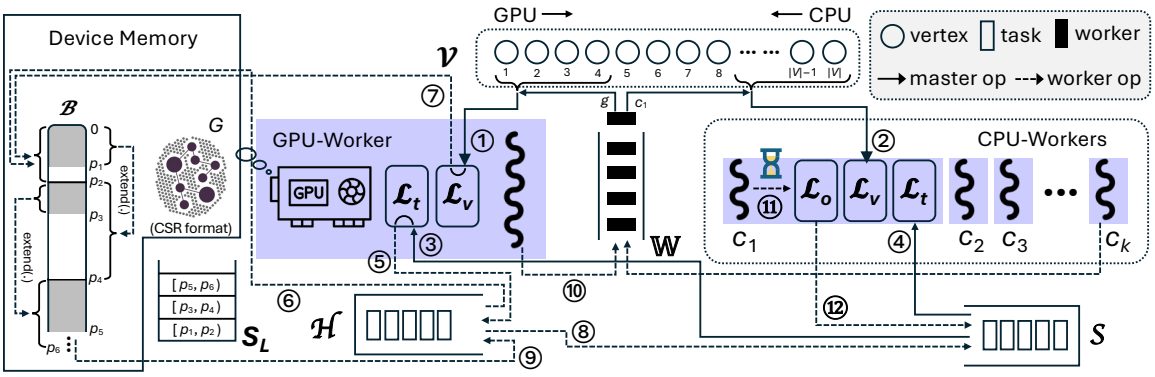


Fig. 6. Detailed G-ThinkerCG System Architecture

Algorithm 2 Pseudocode of the Master Thread

```
1:  Create CPU workers and the GPU worker and add them to 𝕎
2:  𝒮 ← ∅, 𝒱 ← V                    // note that G = (V, E)
3:  while |𝕎| ≠ n_c + 1 or 𝒱 ≠ ∅ or 𝒮 ≠ ∅ do
4:      if 𝕎 = ∅ then wait to be notified by a worker
5:      if 𝒱 = ∅ and 𝒮 = ∅ then continue
6:      w ← 𝕎.DEQUEUE()
7:      if w is a GPU-worker then
8:          if 𝒱 ≠ ∅ then w.ℒ_v ← 𝒱.POP_FRONT_BATCH()
9:          else w.ℒ_t ← 𝒮.POP_BATCH()
10:     else                          // w is a CPU-worker
11:         if 𝒮 ≠ ∅ then w.ℒ_t ← 𝒮.POP_BATCH()
12:         else w.ℒ_v ← 𝒱.POP_BACK_BATCH()
13:     notify w to wake up for processing
14: global_end_flag ← true, notify all workers (to terminate)
```

and $\mathcal{S}$ are empty, then all initial chunks have been assigned for processing and there is no spilled task to assign, so the master will busy wait (or sleep for a short time to avoid busy waiting) until either some tasks are spilled into $\mathcal{S}$ for assignment, or the job termination condition in Line 3 holds. Here, we terminate the job if (1) all (n_c+1) workers are idle and placed in $\mathbb{W}$ and (2) all initial chunks in $\mathcal{V}$ have been assigned and (3) there is no spilled task in $\mathcal{S}$ yet to process. When this happens, Line 14 sets the global end flag and notifies all workers waiting in $\mathbb{W}$ to wake up; these workers will then see the end flag being set, so they will terminate their execution properly.

In Line 6, it must hold that (1) $\mathbb{W} \neq \emptyset$ due to Line 4 and that (2) there are vertex chunks or spilled tasks to assign due to Line 5, so we dequeue a worker w from $\mathbb{W}$ to assign work to compute.

Let us assume $w = g$ is a GPU worker (Line 7). If there are still vertices (i.e., initial subgraphs) in $\mathcal{V}$, we assign a vertex chunk from the front of $\mathcal{V}$ to g's buffer $\mathcal{L}_v$ for processing (see ① in Figure 6). Otherwise, we pop a task chunk from $\mathcal{S}$ to g's buffer $\mathcal{L}_t$ for processing (see ③), i.e., CPU-to-GPU task stealing. The GPU worker g will then move them to $\mathcal{H}$ (see ⑤) to be scheduled for processing on the GPU (see ⑥). Recall that vertices in $\mathcal{V}$ is in degeneracy ordering, so we prioritize vertex chunks from the front of $\mathcal{V}$ for scheduling since they are more amenable for level-by-level GPU processing; but if $\mathcal{V}$ is exhausted, we let the idle GPU take some CPU-side spilled tasks in $\mathcal{S}$ to process for load balancing.

When $w = c_i$ is a CPU worker (Line 10), we schedule spilled tasks first (Line 11) before taking a new vertex chunk (Line 12), to support near-DFS exploration of the subgraph enumeration tree.

Finally in Line 13, the master notifies w to wake up and take over the filled $\mathcal{L}_v$ or $\mathcal{L}_t$ for processing by the worker.

GPU Worker. The GPU worker maintains a `GPUContext` object as shown in Figure 7, which contains an internal buffer `B` for $\mathcal{B} = [L_1, L_2, \ldots, L_i, L_{i+1}]$ on which two subgraph iterators are defined for use by UDFs: (1) `Brd` which corresponds to L_i for reading subgraphs for processing using `BufferT::next_offsets()` (recall Figure 4), and (2) `Bwr` which corresponds to L_{i+1} for appending extended subgraphs using `BufferT::append_ptr()`. Also, buffer H is defined for stack $\mathcal{H}$ where users can call `BufferT::append_ptr()` to push subgraphs, and the GPU worker calls `BufferT::pop_offsets()` to pop subgraphs for assignment. Finally, `source[]` is essentially referencing the vertex chunk buffer $\mathcal{L}_v$ of the GPU worker.

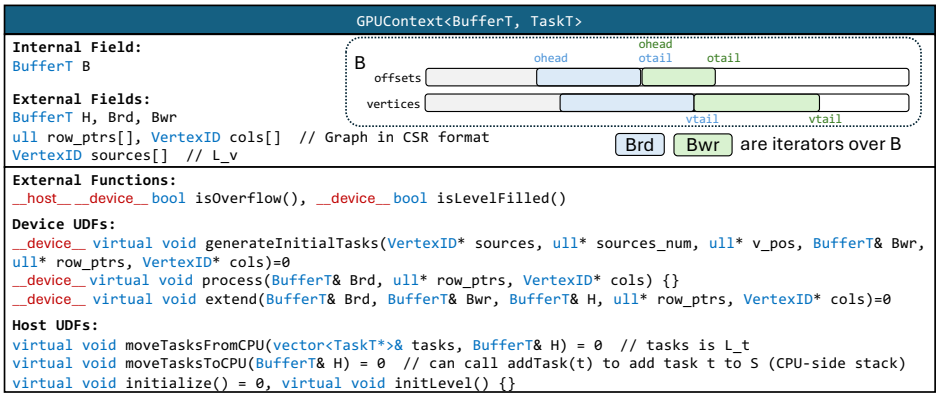


Fig. 7. The GPUContext Class with UDFs

GPUContext provides **3 user-defined kernel functions** for the GPU workers to call, where row_ptrs and cols are the CSR representation of the input graph G (i.e., row_ptrs$[i]$ tracks the start position of vertex i's adjacency list in cols) and are passed to all the 3 functions for users to access the adjacency lists. **(1) UDF generateInitialTasks(.)** creates initial subgraphs into Bwr $= L_1$ from the vertex chunk in GPU worker's $\mathcal{L}_v$ (see ⑦ in Figure 6). Among the input arguments, sources and source_num are the start position and size of the vertex chunk in $\mathcal{L}_v$. Since the vertex chunk's size can be larger than η, and we only allow η vertices to generate initial subgraphs into L_1 for extension at a time, v_pos is used to track the next vertex in the chunk buffer sources[] to process. In generateInitialTasks(.), users read up to η vertices in sources[] from position v_pos to generate initial subgraphs into Bwr $= L_1$, and v_pos is properly forwarded to prepare for the next call of generateInitialTasks(.) when backtracking. **(2) UDF process(.)** is optional for implementation, and when it is used, users read each subgraph g from Brd and compute the candidate elements to extend g to be appended to an intermediate buffer, denoted by $\mathcal{Q}$ which users can preallocate in the device memory using the **host UDF initialize()** called at the beginning of the job. Recall from Figure 2(b) on Page 4 that in each iteration, the GPU worker calls UDFs process(.) and extend(.) in Lines 3–4; since they use $\mathcal{Q}$, users need to clear $\mathcal{Q}$ (i.e., reset its tail back to 0) in the **host UDF initLevel()** which the GPU worker calls right before Line 3. Finally, **(3) UDF extend(.)** lets users read subgraphs from Brd, and append extended subgraphs to Bwr, or H (i.e., $\mathcal{H}$) if $\mathcal{B}$ overflows. If process(.) is used to write candidates to $\mathcal{Q}$, each thread in extend(.) can read a candidate in $\mathcal{Q}$ and use it to extend its subgraph, which gives a better load balancing performance than letting each warp extend a subgraph from Brd for all its candidates.

As Figure 7 shows, **isOverflow(.)** is both a host function and a device one. It judges $\mathcal{B}$ as overflowing if more than 95% capacity of offsets[] or vertices[] have been used, which can be checked using Bwr.otail and Bwr.vtail (since Bwr $= L_{i+1}$ is the last level in $\mathcal{B}$). Note that the space of $\mathcal{B}$ is allocated to be huge, so even if all warps find that $\mathcal{B}$ does not overflow but right at 95% capacity, and they append new subgraphs simultaneously, they will not cause the capacity of $\mathcal{B}$ to go beyond 100% (which we use assertion to catch in our code to avoid system crash), so all warps will see that $\mathcal{B}$ overflows afterwards and stop appending new subgraphs.

In Figure 2(b) after Line 4 has called extend(.), the GPU worker calls isOverflow(.) (as a host function) and if it returns *true*, Bwr $= L_{i+1}$ is pushed to stack $\mathcal{H}$ as we have illustrated in Figure 3.

Device function isOverflow(.) is also used in the device function **isLevelFilled(.)**, which returns *true* when either Bwr.nTasksProcessed $> \eta$ or isOverflow(.) returns *true*. In extend(.), we call isLevelFilled(.) and if it returns *true*, we stop reading more subgraphs from Brd for extension into Bwr. In the case when we use process(.) to generate candidate elements into intermediate buffer $\mathcal{Q}$ first, in extend(.), for each candidate $q \in \mathcal{Q}$ we need to call isOverflow(.) to check if $\mathcal{B}$ overflows; if so, we append the subgraph extended with q to H rather than to Bwr (see ⑨).

Finally, **host UDF moveTasksFromCPU($\mathcal{L}_t$, $\mathcal{H}$)** specifies how to convert each task object (of type TaskT) in the GPU worker's $\mathcal{L}_t$ (which holds tasks stolen from $\mathcal{S}$) to $\mathcal{H}$ of type BufferT for GPU processing (see ⑤ of Figure 6); while **host UDF moveTasksToCPU($\mathcal{H}$)** specifies how to convert each subgraph from $\mathcal{H}$ (of type BufferT) to a task object (of type TaskT) in stack $\mathcal{S}$ (see ⑧ of Figure 6).

Algorithm 3 Pseudocode of the GPU Worker

```
1:  while true do
2:      wait to be notified by master
3:      if global_end_flag = true then break
4:      if L_t ≠ ∅ then H ← moveTasksFromCPU(L_t)   // ⑤
5:      v_pos ← 0
6:      while true do
7:          if H ≠ ∅ then
8:              pop up to η tasks from H to add to Bwr (= L_1) // ⑥
9:              if |W| > τ_c then S ← moveTasksToCPU(H)   //⑧
10:         else if v_pos < |L_v| then
11:             Bwr, v_pos ← generateInitialTasks <<<>>> (L_v, v_pos)   //⑦
12:         else break
13:         Brd ← Bwr (i.e., L_1) and update Bwr as L_2
14:         run the algorithm in Figure 2(b) to extend Brd = L_1
15:     L_v ← ∅, L_t ← ∅
16:     append itself to W and notify the master   // ⑩
```

Algorithm 3 shows the pseudocode of the GPU worker, which is awakened by the master in Line 2 to process the assigned tasks (see ①), after which it waits in $\mathbb{W}$ in Line 16 (see ⑩) to be awakened again. If the job terminates, Line 3 will terminate the processing.

In each iteration, we first check if the master has stolen some tasks from $\mathcal{S}$ on the CPU side to $\mathcal{L}_t$ (see ③) and if so, UDF moveTasksFromCPU(.) is called in Line 4 to move tasks from $\mathcal{L}_t$ to $\mathcal{H}$ (see ⑤), so that they will be scheduled for GPU processing (see Lines 7–8). Afterwards the while loop at Line 6 processes up to η tasks from $\mathcal{L}_t$ (with a higher priority) or $\mathcal{L}_v$ in each iteration until all assigned tasks are completed. **Case 1:** if $\mathcal{H}$ is not empty (Line 7), then Line 8 pops up to η tasks to L_1 in $\mathcal{B}$ (see ⑥) for processing by Lines 13 and 14. Moreover, Line 9 checks if a significant number of CPU threads are idle ($\tau_c = n_c/2$ by default) and if so, UDF moveTasksToCPU(.) is called to move tasks from $\mathcal{H}$ to $\mathcal{S}$ (see ⑧). We do not trigger GPU-to-CPU work stealing if there are no more than τ_c idle CPU workers to avoid task thrashing, since some tasks may be spilled to $\mathcal{S}$ to be assigned to them soon. **Case 2:** otherwise, Line 11 processes the vertex chunk in $\mathcal{L}_v$ for up to η vertices at an iteration (for processing by Lines 13–14) and forwards v_{pos} properly (see ⑦) which is initialized as 0 in Line 5. When all vertices in $\mathcal{L}_v$ are processed (i.e., Line 10 gives *false*), Line 12 exits the processing. The GPU worker finally clears $\mathcal{L}_t$ and $\mathcal{L}_v$ to collect future tasks (Line 15) before waiting in $\mathbb{W}$ (Line 16). Note that Line 14 runs our BFS-DFS subgraph extension and may spill subgraphs to $\mathcal{H}$, which will be caught later in Line 7

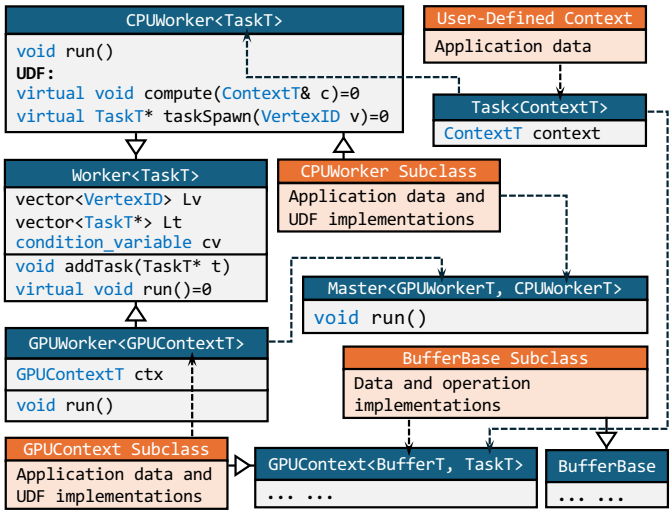


Fig. 8. Programming Interface

for near-DFS processing.

CPU Workers and Overall Programming Interface. Figure 8 shows the C++ programming interface of G-thinkerCG, where orange classes are those that need to be specified by users for their applications. Note that `BufferBase` (resp. `GPUContext`) has been described in Figure 4 (resp. Figure 7). Specifically, `GPUWorker` implements the GPU worker, and `CPUWorker` implements a CPU worker; both classes are subclasses of the base class `Worker` that maintains $\mathcal{L}_v$ and $\mathcal{L}_t$ to take the assigned tasks from the master, and a condition variable for waiting and for master to wake it up.

On the GPU side, `GPUWorker` contains a GPUContext object ctx, whose type is specified using the class `GPUContext` that takes the user-defined `BufferT` type (used by $\mathcal{B}$ and $\mathcal{H}$) and the CPU-side task type `TaskT` (used by work stealing UDFs). Here, `BufferT` can be `BufferBase` or another implementation that users revise on top of `BufferBase` to include additional information into subgraphs. Users specify a subclass of `GPUContext<BufferT, TaskT>` that defines additional application data and implements the UDFs, and pass it to `GPUWorker` to define the application-specific GPU worker.

On the CPU side, users need to define task data as a type `ContextT` (e.g., it can be $\langle R, P, X\rangle$ for Algorithm 1), which is passed to the class `Task` to create the application-specific task type `TaskT` to pass to `CPUWorker`; this `CPUWorker` base class is inherited by a user-defined subclass that implements all its UDFs to define the application-specific CPU worker. Finally, both the user-defined GPU and CPU worker types are passed to `Master` (which creates and manages these workers), and we then call `run()` of the `Master` object to start the SF computation. Note that we cannot directly move the UDFs of `GPUContext` to `GPUWorker` and let its subclass implement these UDFs to pass to `Master`, since CUDA does not support virtual functions or polymorphism in device code.

We use the task-based model of G-thinkerQ [27], [56] for CPU threads, where users need to implement two UDFs for `CPUWorker`: (1) `taskSpawn`(v) which indicates how to spawn an initial subgraph from v, and (2) `compute`(c) which specifies how a task processes and extends its subgraph based on its context object c. For example, to implement Algorithm 1, a task context maintains (R, P, X), `taskSpawn`(v) creates $(R, P, X) = (\{v\}, \{u \in N(v) \,|\, u > v\}, \{u \in N(v) \,|\, u < v\})$, and `compute`($c$) implements the recursive backtracking algorithm as in Algorithm 1 on its (R, P, X).

To eliminate straggler tasks, if a task runs for more than τ_{time} (= 0.1 second by default), we regard it as timing out and decompose it into smaller tasks to be added back to $\mathcal{S}$ by calling `Worker::addTask(.)`, to be scheduled on other idle workers. These tasks are actually first appended to a buffer $\mathcal{L}_o$ (see ① in Figure 6), and when $\mathcal{L}_o$ becomes full, tasks therein are flushed to $\mathcal{S}$ (which is mutex-protected) with one lock operation to reduce contention (see ②). Appendix D [3] illustrates how to implement timeout mechanism for Algorithm 1.

Applications. We implement MCE and SM, the two applications of G^2-AIMD [55], on G-thinkerCG. The GPU application code is almost the same. For the CPU application code, we follow G-thinkerQ [56]. For MCE, we directly use `BufferBase` but maintain an additional buffer `labels[]` of the same size as `vertices[]` to indicate whether each vertex is in R or P or X so that a GPU thread can operate on it properly. In [55], each warp in `extend(.)` directly reads a subgraph from `Brd` and appends all extended subgraphs to `Bwr`. We instead initialize a large buffer $\mathcal{Q}$ in device memory, and let each warp in `process(.)` append (s, e, c) to $\mathcal{Q}$ where `vertices`$[s, e)$ tracks the subgraph g, and c is the candidate to extend g. Later in `extend(.)`, each thread reads (s, e, c) to access $g =$ `vertices`$[s, e)$ for extending with c, and appends the extended subgraph to `Bwr`, allowing better load balancing. For SM, we follow the *prefix* mode (aka. super-subgraph scheme) of [55] to implement `BufferT`'s `offsets[]` array entry as (s, m, e), where `vertices`$[s, m)$ keeps the subgraph vertices and `vertices`$[m, e)$ keeps the candidates to extend with.

V. Experiments

We now report our experimental results. Our code has been released at https://github.com/akhlaqueak/G-ThinkerCG.

Experimental Setup. We conduct experiments on a machine with an NVIDIA Tesla P100 GPU (16 GB device memory), an Intel Xeon E5-2680 v4 CPU (14 cores, 28 threads with hyper-threading), and 256 GB DDR4 RAM. We run G-thinkerCG in three modes: (1) **CPU-Only** which runs with 28 CPU workers, (2) **GPU-Only** where each GPU kernel is configured with 56 blocks (since P100 has 56 streaming multiprocessors) and 1024 threads per block, and (3) **Hybrid** which runs CPU-GPU co-processing. For the device memory, $\mathcal{B}$ is allocated with all the leftover memory after storing the graph and application-specific warp buffers for keeping intermediate results (e.g., for the BK algorithm, each warp reserves a buffer with a capacity to hold 200K vertices and R/P/X labels, and the $\mathcal{Q}$ buffer has a capacity to hold 100M vertices). For the host memory, $\mathcal{H}$ and $\mathcal{S}$ each is allocated with half of the memory.

We adopt the following default parameters in G-thinkerCG: (1) the bound on subgraph appending to `Bwr` is set as $\eta =$

TABLE I
PERFORMANCE COMPARISON ON MAXIMAL CLIQUE ENUMERATION (MCE)

G	$\|V\|$	$\|E\|$	d_{max}	L1+IP	L1+IPX	G²-AIMD	CPU-Only	GPU-Only	Hybrid	Gain^{H2G}	Gain^{H2C}	Loss^{P2H}	Loss^{PX2H}
soc-sinaweibo	58.65M	522.6M	278K	11.1	10.8	108.2	205.6	150.1	109.2	1.4	1.9	9.9	10.1
trackers	27.66M	281.2M	11,571K	OOM	OOM	51.1	28.8	47.8	30.5	1.6	0.9	–	–
wikipedia_link_en	25.92M	1086M	4271K	21.9	OOM	430.5	334.2	446.4	253.0	1.8	1.3	11.5	–
edit-cebwiki	8.482M	23.58M	8421K	OOM	OOM	9.9	1.3	7.8	1.2	6.3	1.0	–	–
wikipedia_link_ceb	7.891M	127.8M	3636K	0.8	OOM	17.6	4.5	15.1	2.4	6.2	1.9	3.0	–
edit-svwiki	7.570M	43.88M	5669K	OOM	OOM	17.4	6.4	14.2	3.8	3.8	1.7	–	–
edit-enwiktionary	5.812M	54.23M	3504K	0.8	OOM	11.5	13.1	10.0	5.8	1.7	2.3	6.8	–
edit-zhwiki	5.005M	33.08M	1208K	0.5	0.5	6.1	7.8	5.1	3.9	1.3	2.0	7.4	7.9
edit-itwiki	4.799M	63.84M	1076K	2.0	OOM	17.8	30.9	15.0	11.8	1.3	2.6	6.0	–
edit-shwiki	4.579M	12.53M	4043K	0.7	OOM	5.7	1.3	4.3	1.3	3.3	1.0	1.8	–
edit-mgwiktionary	4.063M	17.65M	4059K	OOM	OOM	0.7	0.3	1.5	0.2	9.5	1.9	–	–
soc-livejournal	4.033M	55.86M	2651	0.5	0.4	5.3	11.6	6.8	4.3	1.6	2.7	9.0	11.4
edit-viwiki	3.496M	27.30M	2187K	0.1	OOM	3.0	1.0	2.4	0.8	3.0	1.3	6.1	–
edit-frwiktionary	3.411M	26.85M	2487K	0.1	OOM	4.5	1.3	3.6	1.2	3.0	1.1	9.4	–
wikipedia_link_sr	3.175M	206.6M	369K	OOM	OOM	1279.8	725.9	1097.6	462.7	2.4	1.6	–	–
edit-jawiki	2.981M	42.83M	385K	0.2	0.3	3.5	7.0	3.0	2.4	1.3	2.9	9.9	9.0
socfb-B-anon	2.937M	41.91M	4356	0.2	0.2	2.5	5.0	2.3	1.8	1.3	2.7	8.9	10.7
edit-plwiki	2.646M	42.43M	850K	1.2	1.2	11.9	19.6	9.9	7.6	1.3	2.6	6.2	6.6
edit-ukwiki	2.132M	17.44M	306K	0.1	0.1	1.8	2.5	1.6	1.2	1.3	2.0	13.5	10.7
soc-pokec	1.632M	44.60M	14.8K	0.1	0.1	1.7	3.2	1.7	1.0	1.7	3.3	6.9	8.4

$1000 \times \{\text{\# of warps}\}$ (where there are $56 \times 32 = 1792$ warps), (2) the threshold on the number of idle CPU workers to trigger GPU-to-CPU work stealing (see Line 9 of Algorithm 3) is set as $\tau_c = n_c/2$ (where $n_c = 28$ is the number of CPU workers), (3) each CPU chunk contains 50 vertices and each GPU chunk contains 500K vertices, (4) when work stealing happens, `moveTasksFromCPU(.)` moves 100K tasks from $\mathcal{S}$ to $\mathcal{H}$, while `moveTasksToCPU(.)` moves 1M tasks from $\mathcal{H}$ to $\mathcal{S}$.

This default parameter configuration is carefully tuned, and observed to work generally well on various graphs. Notably, since CPU threads are much more powerful, even though a CPU chunk has only 50 vertices, a lot of chunks will have been processed between two consecutive triggering of GPU-to-CPU work stealing, so we move 1M tasks at each tiggering. Moreover, we let Line 9 of Algorithm 3 also check if $|\mathcal{S}|$ is fewer than 1M, and only steal tasks if so since otherwise, $\mathcal{S}$ has enough tasks to keep CPU workers busy before the next triggering of GPU-to-CPU work stealing.

Performance Comparison on Maximal Clique Enumeration (MCE). Table I shows the MCE running time (unit: second) on 20 large real graph datasets for the 3 modes of G-thinkerCG, our major baseline G^2-AIMD [55], and two variants of MCE-GPU [5] that integrates additional tailor-made optimizations such as binary encoding, sub-warp partitioning, partial induced subgraphs and a compact representation of exclusion set X (see Appendix B [3] for details of those optimizations). The BK algorithm variant with pivoting [55] is implemented by all the systems for the MCE application. The graphs are all from KONECT [1] and Network Repository [2], selected to be large enough so that MCE is challenging but can be solved in reasonable time. Note that the maximum degree d_{max} is huge for most graphs, making MCE quite challenging.

We can see that GPU-Only beats G^2-AIMD on 15 out of the 20 graphs, indicating that our BFS-DFS hybrid strategy is superior than the chunk-adaptive BFS solution of G^2-AIMD which may incur wasted chunk computation due to OOM.

More importantly, Hybrid is consistently faster than both CPU-only and GPU-only, showing that the CPU-GPU co-processing is very effective. In particular, Gain^{H2G} shows the running time ratio for GPU-only v.s. Hybrid which is up to 9.5 (i.e., Hybrid is up to 9.5× faster than GPU-only), and Gain^{H2C} shows the running time ratio for CPU-only v.s. Hybrid which is up to 3.3.

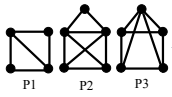
Recall that MCE-GPU integrates additional optimizations such as binary encoding, sub-warp partitioning, partial induced subgraphs and a compact representation of exclusion set X, which we currently do not integrate in our MCE application. These optimizations trade space for efficiency, hence are not scalable. We consider two variants for MCE-GPU: L1+IP and L1+IPX. Since MCE-GPU uses `__nanosleep()` not supported by the Pascal architecture of P100, and is memory-hungry, we have to find another machine with an A100 GPU (80 GB device memory) to run MCE-GPU.

Regarding the names of the MCE-GPU variants, L1 means that top-level subtrees are distributed to thread blocks for parallel processing, while L2 means that the second-level subtrees (i.e., more fine-grained) are distributed. Since Table IV of [5] shows that L2 has no clear advantage and L1 wins in most cases, we only compare with L1 variants. Also, IP refers to the use of partial induced subgraphs (i.e., induced on P only) that is more space-efficient, while IPX refers to the use of full induced subgraphs (i.e., induced on P and X) that uses more space but is more time efficient. However, both of them need to materialize a subgraph for every top-level task, thus they frequently run out of memory (OOM, on A100 with 80 GB memory) as shown in Table I, an issue that does not occur in G-thinkerCG and G^2-AIMD (on P100 with 16 GB memory). However, for graphs where MCE-GPU can complete, it is indeed up to an order of magnitude faster even than Hybrid. Specifically, Loss^{P2H} shows the running time ratio for Hybrid v.s. L1+IP, and Loss^{PX2H} shows the ratio for Hybrid v.s. L1+IPX. This shows that tailor-made GPU solutions with application-specific optimizations have the potential to beat applications implemented on a general system like our G-thinkerCG, although we can also integrate these optimizations with some implementation efforts (which are future works).

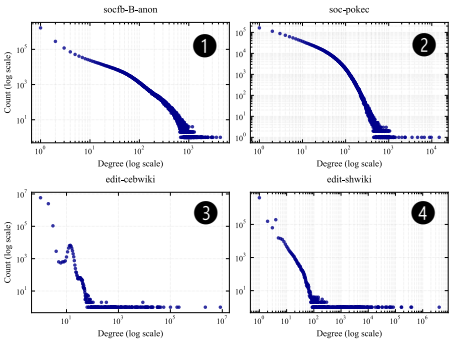
Performance Comparison on Subgraph Matching (SM). Table II(c) shows the SM running time (unit: second) on 7 of

TABLE II
PERFORMANCE COMPARISON ON SUBGRAPH MATCHING (SM)

(a) Query Graphs:



(b) Degree Distributions:



(c) Time:

G	Query	G²-AIMD	CPU-Only	GPU-Only	Hybrid	Gain^{H2G}	Gain^{H2C}	STMatch	EGSM
soc-sinaweibo	P1	35.5	OOT	58.4	59.2	1.0	–	Error	OOM
	P2	292.1	OOT	327.7	322.1	1.0	–	Error	OOM
	P3	470.7	OOT	431.6	436.7	1.0	–	Error	OOM
edit-cebwiki ❸	P1	222.2	65.7	5.3*	5.3*	1.0	12.50	633.9	35.7
	P2	361.0	OOT	26.0*	26.0*	1.0	–	OOT	71.6
	P3	OOT	OOT	115.2	115.0	1.0	–	OOT	Error
edit-shwiki ❹	P1	439.2	24.3	4.5*	4.5*	1.0	5.40	106.7	65.2
	P2	728.1	18.9	0.9*	0.9*	1.0	21.52	151.8	9.7
	P3	OOT	OOT	1165.2	1162.8	1.0	–	198.4	OOM
edit-mgwiktionary	P1	0.6	141.5	0.3	0.7	0.5	204.74	OOT	11.8
	P2	0.8	30.2	0.3	0.5	0.6	64.02	710.1	4.6
	P3	10.0	213.3	0.5	6.9	0.1	30.92	OOT	13.1
soc-livejournal	P1	2.2	8.3	2.0	2.3	0.9	3.59	26.9	OOM
	P2	61.3	476.8	51.7	49.9	1.0	9.56	OOT	OOM
	P3	102.3	767.5	94.6	94.5	1.0	8.13	OOT	OOM
socfb-B-anon ❶	P1	1.7	5.7	1.5	1.7	0.9	3.28	7.4	22.6
	P2	14.1	34.2	10.2	10.2	1.0	3.35	32.9	OOM
	P3	31.5	157.2	22.2	22.6	1.0	6.96	31.7	OOT
soc-pokec ❷	P1	2.0	2.8	1.8	1.0	1.7	2.68	13.1	12.1
	P2	6.5	9.5	6.0	4.0	1.5	2.38	9.1	196.7
	P3	13.8	53.2	7.7	7.9	1.0	6.70	11.2	OOM

the 20 graphs used in Table I for 3 representative query graphs shown in Table II(a). The programs run out of our maximum allowed time threshold of 1800 seconds (denoted by OOT) on the other 13 graphs.

Note that while we use the *prefix* mode of [55] by default which lets a warp extend a subgraph g with all its candidates so that the set intersection results on g can be shared, this leads to load balancing problems on *edit-cebwiki* and *edit-shwiki* where many vertices have very high fanouts (see Table II(b)-❸❹) – unlike the other power-law graphs (see Table II(b)-❶❷) – which leads to extreme load imbalance where a few warps need to extend with many candidates while all others have finished much earlier, leading to much slower running time on GPU-Only. Therefore, we switch to the *expand* mode of [55] (where each warp extends with one candidate) on both graphs for P1 and P2 which reduces the running time significantly. These numbers have been highlighted with asterisks in Table II(c).

We can see that GPU-Only is faster than G^2-AIMD in most cases. However, Hybrid does not improve GPU-Only in most cases other than *soc-pokec*, and there is even a slowdown on *edit-mgwiktionary*. This is because unlike in MCE where CPU handles the few deep subgraph enumeration trees while GPU handles the massive shallow trees, all trees are of the same height (equal to the query subgraph size) in SM, so CPU is not as complementary; tasks assigned for CPU processing lag far behind GPU processing, bottlenecking Hybrid's throughput. Based on this observation, we recommend not to enable CPU processing in G-ThinkerCG for the SM application.

We also include the two latest GPU solutions to SM for comparison, STMatch [47] and EGSM [40], the results of which are shown in the last two columns of Table II(c). We can see that both systems are much slower than GPU-Only and Hybrid, which shows the superiority of our BFS-DFS hybrid strategy for subgraph extension on a GPU compared with the methods in STMatch and EGSM. Note that EGSM frequently runs out of memory (OOM), while STMatch frequently runs out of the maximum time threshold of 1800 seconds (OOT). STMatch also encounters the error of illegal memory access on *soc-sinaweibo*, so its implementation is not robust enough.

TABLE III
EFFECT OF SYSTEM PARAMETERS

Parameter	Value	soc-sinaweibo	wikipedia_link_sr
η / #{warps}	100	203.8	OOT
	500	113.9	542.3
	1000	**109.6**	**465.2**
	2000	231.0	948.0
CPU Chunk	1	117.1	357.6
	10	108.9	**316.1**
	50	**91.6**	465.2
	100	110.8	564.9
GPU Chunk	100K	143.3	511.9
	500K	**112.1**	**315.9**
	1M	159.4	1085.5
τ_c	1	220.3	567.3
	n_c / 2	**111.4**	**471.9**
	n_c / 4	112.5	476.7

System Parameter Study. We next present the effect of various system parameters on the running time of Hybrid using two representative datasets *soc-sinaweibo* and *wikipedia_link_sr*. As shown in Table III (where OOT means running out of the maximum allowed time threshold of 1800 seconds), our default parameters generally achieve the best performance. Note that a tradeoff exists for each parameter: when η and vertex chunk sizes are too large, the task granularity is too coarse which hampers load balancing; while when η and GPU chunk size are too small, the computing power of GPU threads cannot be saturated, and when CPU chunk size is too small, the cost for CPU workers to wait in $\mathbb{W}$ increases. As for τ_c, if it is too small, GPU-to-CPU task stealing would be too aggressive and could lead to task thrashing if some CPU worker encounters a task timeout and adds many new tasks back to $\mathcal{S}$; while if it is too large, surplus GPU tasks cannot be timely rescheduled to CPU workers for processing. Our default parameters achieve a good tradeoff.

VI. CONCLUSIONS AND FUTURE WORK

We proposed G-thinkerCG, a programming framework for general-purpose subgraph finding (SF) problems with CPU-GPU co-programming, where GPU uses a novel BFS-DFS hybrid execution scheme. Regarding future works, we plan to implement more SF applications such as quasi-cliques and k-plexes, and support multi-GPU and distributed processing.

AI-Generated Content Acknowledgement

The authors declare that no generative artificial intelligence (AI) tools were used in the creation of this article's content, including text, figures, tables, or code.

References

[1] Network Datasets from KONECT. http://konect.cc/networks/.

[2] Network repository. https://networkrepository.com.

[3] Online technical report. https://github.com/akhlaqueak/G-ThinkerCG/blob/main/techical-report.pdf.

[4] A. Ahmad, D. Yan, X. Chen, L. Yuan, Q. Zhang, and S. Adhikari. Maximum k-plex finding: Choices of pruning techniques matter! *Proc. VLDB Endow.*, 18(9):2928–2940, 2025.

[5] M. Almasri, Y. Chang, I. E. Hajj, R. Nagi, J. Xiong, and W. W. Hwu. Parallelizing maximal clique enumeration on gpus. In *PACT*, pages 162–175. IEEE, 2023.

[6] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *PPoPP*, pages 235–248. ACM, 2017.

[7] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[8] L. Chang, M. Xu, and D. Strash. Efficient maximum k-plex computation over large sparse graphs. *Proc. VLDB Endow.*, 16(2):127–139, 2022.

[9] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12. ACM, 2018.

[10] J. Chen, S. Cai, S. Pan, Y. Wang, Q. Lin, M. Zhao, and M. Yin. Nuqclq: An effective local search algorithm for maximum quasi-clique problem. In *AAAI*, pages 12258–12266. AAAI Press, 2021.

[11] J. Chen and X. Qian. Khuzdul: Efficient and scalable distributed graph pattern mining engine. In T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, editors, *ASPLOS*, pages 413–426. ACM, 2023.

[12] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.*, 13(8):1190–1205, 2020.

[13] Q. Cheng, D. Yan, T. Wu, L. Yuan, J. Cheng, Z. Huang, and Y. Zhou. Efficient enumeration of large maximal k-plexes. In A. Simitsis, B. Kemme, A. Queralt, O. Romero, and P. Jovanovic, editors, *EDBT*, pages 53–65. OpenProceedings.org, 2025.

[14] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, 2015.

[15] Q. Dai, R. Li, H. Qin, M. Liao, and G. Wang. Scaling up maximal k-plex enumeration. In M. A. Hasan and L. Xiong, editors, *CIKM*, pages 345–354. ACM, 2022.

[16] V. V. dos Santos Dias, C. H. C. Teixeira, D. O. Guedes, W. M. Jr., and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *SIGMOD*, pages 1357–1374. ACM, 2019.

[17] Z. Fu, B. B. Thompson, and M. Personick. Mapgraph: A high level API for fast development of high performance graph analytics on gpus. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, pages 2:1–2:6. CWI/ACM, 2014.

[18] S. Gao, K. Yu, S. Liu, and C. Long. Maximum k-plex search: An alternated reduction-and-bound method. *Proc. VLDB Endow.*, 18(2):363–376, 2024.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In C. Thekkath and A. Vahdat, editors, *OSDI*, pages 17–30. USENIX Association, 2012.

[20] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, 2020.

[21] G. Guo, D. Yan, L. Yuan, J. Khalil, C. Long, Z. Jiang, and Y. Zhou. Maximal directed quasi-clique mining. In *ICDE*, pages 1900–1913. IEEE, 2022.

[22] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*, pages 1067–1082. ACM, 2020.

[23] L. Hu, L. Zou, and M. T. Özsu. Gamma: A graph pattern mining framework for large graphs on gpu. In *ICDE*. IEEE, 2023.

[24] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *EuroSys*, pages 13:1–13:16. ACM, 2020.

[25] J. Jenkins, I. Arkatkar, J. D. Owens, A. N. Choudhary, and N. F. Samatova. Lessons learned from exploring the backtracking paradigm on the GPU. In *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, volume 6853 of *Lecture Notes in Computer Science*, pages 425–437. Springer, 2011.

[26] G. Jiang, C. Q. Zhou, T. Jin, B. Li, Y. Zhao, Y. Li, and J. Cheng. Vsgm: View-based gpu-accelerated subgraph matching on large graphs. In *SC*, pages 739–753. IEEE Computer Society, 2022.

[27] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *VLDB J.*, 31(4):649–674, 2022.

[28] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In B. Plale, M. Ripeanu, F. Cappello, and D. Xu, editors, *HPDC*, pages 239–252. ACM, 2014.

[29] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In *PKDD*, volume 5212 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2008.

[30] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, 2014.

[31] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2014.

[32] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu. Graphzero: A high-performance subgraph matching system. *ACM SIGOPS Oper. Syst. Rev.*, 55(1):21–37, 2021.

[33] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *SOSP*, pages 509–523. ACM, 2019.

[34] K. Meng, J. Li, G. Tan, and N. Sun. A pattern based algorithmic autotuner for graph processing on gpus. In *PPoPP*, pages 201–213. ACM, 2019.

[35] L. Meng, Y. Shao, L. Yuan, L. Lai, P. Cheng, X. Li, W. Yu, W. Zhang, X. Lin, and J. Zhou. Revisiting graph analytics benchmark. *Proc. ACM Manag. Data*, 3(3):208:1–208:28, 2025.

[36] T. NVIDIA. Nvidia® tesla® p100–the most advanced data center accelerator ever built. Technical report, Technical Report WP-08019-001, 2017.

[37] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. In *SC*, page 100. IEEE/ACM, 2020.

[38] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Trans. Knowl. Data Eng.*, 30(1):29–42, 2018.

[39] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098, 2020.

[40] X. Sun and Q. Luo. Efficient gpu-accelerated subgraph matching. *Proc. ACM Manag. Data*, 1(2):181:1–181:26, 2023.

[41] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440. ACM, 2015.

[42] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[43] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*, pages 763–782. USENIX Association, 2018.

[44] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In R. Asenjo and T. Harris, editors, *PPoPP*, pages 11:1–11:12. ACM, 2016.

[45] Z. Wang, Y. Zhou, M. Xiao, and B. Khoussainov. Listing maximal k-plexes in large real-world graphs. In F. Laforest, R. Troncy, E. Simperl, D. Agarwal, A. Gionis, I. Herman, and L. Médini, editors, *WWW*, pages 1517–1527. ACM, 2022.

[46] Y. Wei, W. Chen, and H. Tsai. Accelerating the bron-kerbosch algorithm for maximal clique enumeration using gpus. *IEEE Trans. Parallel Distributed Syst.*, 32(9):2352–2366, 2021.

[47] Y. Wei and P. Jiang. Stmatch: Accelerating graph pattern matching on GPU with stack-based loop optimizations. In F. Wolf, S. Shende, C. Culhane, S. R. Alam, and H. Jagode, editors, *SC*, pages 53:1–53:13. IEEE, 2022.

[48] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam. cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure. In *SC*, pages 69:1–69:14. ACM, 2021.

[49] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Found. Trends Databases*, 7(1-2):1–195, 2017.

[50] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proc. VLDB Endow.*, 7(14):1821–1832, 2014.

[51] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *ICDE*, pages 1369–1380. IEEE, 2020.

[52] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *VLDB J.*, 31(2):287–320, 2022.

[53] D. Yan, L. Yuan, A. Ahmad, C. Zheng, H. Chen, and J. Cheng. Systems for scalable graph analytics and machine learning: Trends and methods. In R. Baeza-Yates and F. Bonchi, editors, *KDD*, pages 6627–6632. ACM, 2024.

[54] K. Yao and L. Chang. Efficient size-bounded community search over large networks. *Proc. VLDB Endow.*, 14(8):1441–1453, 2021.

[55] L. Yuan, A. Ahmad, D. Yan, J. Han, S. Adhikari, X. Yu, and Y. Zhou. G^2-aimd: A memory-efficient subgraph-centric framework for efficient subgraph finding on gpus. In *ICDE*, pages 3164–3177. IEEE, 2024.

[56] L. Yuan, G. Guo, D. Yan, S. Adhikari, J. Khalil, C. Long, and L. Zou. G-thinkerq: A general subgraph querying system with a unified task-based programming model. *IEEE Trans. Knowl. Data Eng.*, 37(6):3429–3444, 2025.

[57] L. Yuan, D. Yan, A. Ahmad, J. Han, S. Adhikari, and Y. Zhou. Out-of-core parallel spatial join outperforming in-memory systems: A BFS-DFS hybrid approach. In R. W. Wisniewski and I. Brandic, editors, *HPDC*, pages 23:1–23:14. ACM, 2025.

[58] L. Yuan, D. Yan, J. Han, A. Ahmad, Y. Zhou, and Z. Jiang. Faster depth-first subgraph matching on gpus. In *ICDE*, pages 3151–3163. IEEE, 2024.

[59] L. Zeng, L. Zou, and M. T. Özsu. Sgsi–a scalable gpu-friendly subgraph isomorphism algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[60] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. GSI: gpu-friendly subgraph isomorphism. In *ICDE*, pages 1249–1260. IEEE, 2020.

[61] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distributed Syst.*, 25(6):1543–1552, 2014.

Algorithm 4 Ullmann's Subgraph Matching Algorithm

1: **Input:** query graph $G_q = (V_q, E_q)$, data graph $G = (V, E)$
2: generate matching order $\pi = [u_1, u_2, \ldots, u_{|V_q|}]$ $(u_i \in V_q)$
3: ENUMERATE$(\emptyset, 1, \pi)$
4: **procedure** ENUMERATE(S, i, π)
5: $C_S(u_i) \leftarrow$ viable vertex candidates in G to match u_i
6: **for all** $v \in C_S(u_i)$ **do**
7: append S with (u_i, v)
8: **if** $|S| = k$ **then output** S
9: **else** ENUMERATE$(S, i + 1, \pi)$
10: pop (u_i, v) from S

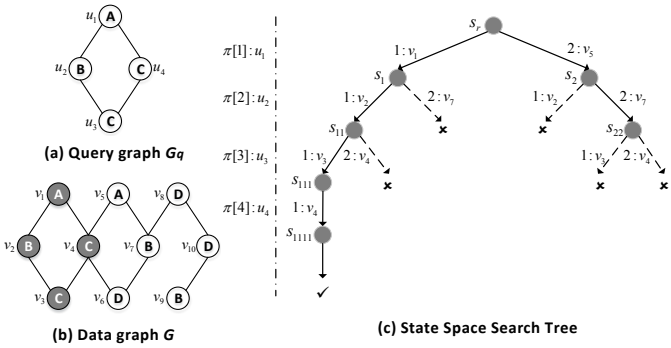


Fig. 9. Illustration of Subgraph Matching

APPENDIX A
SUBGRAPH MATCHING

Subgraph matching (SM) is a representative SF problem, which enumerates all subgraphs of a data graph G that are isomorphic to a query graph G_q. Many SM algorithms [39], [56] have been proposed based on Ullmann's algorithm [42], which orders the query vertices and recursively extends the partial mapping by iterating over candidate data vertices for the next query vertex and pruning infeasible extensions.

We now briefly describe the basic Ullmann's algorithm [42] as shown in Algorithm 4. The algorithm incrementally matches data vertices to query vertices $u_1, \ldots, u_{|V_q|}$, maintaining the partial mapping in S. This search is conducted over a state-space tree as illustrated in Figure 9(c). For instance, the path $S = [(u_1, v_5), (u_2, v_7), (u_3, v_4)]$ fails due to the absence of edge (v_7, v_4) corresponding to query edge (u_2, u_3). Various enhancements have been proposed by various SM algorithms [39], such as optimizing the matching order (Line 2) and deriving tighter candidate sets $C_S(u_i)$ (Line 5).

Note that the state-space tree is also a subgraph enumeration tree over G: a node at Level i corresponds to a subgraph in G matching the query vertex prefix $[u_1, \ldots, u_i]$. Since the programming model of G-thinkerCG is designed based on the subgraph enumeration tree and subgraph-task abstractions, it can implement the CPU-GPU co-processing programs for all SM algorithms by properly adapting their serial algorithm counterparts based on the interface of G-thinkerCG.

APPENDIX B
GPU ALGORITHMS FOR SF PROBLEMS

Except for G^2-AIMD, existing GPU works mainly target a specific SF problem for acceleration, using either BFS- or DFS-based subgraph extension. BFS-based solutions compute one chunk at a time, while DFS-based solutions assign independent subgraph enumeration subtrees to different computing units (thread blocks or warps), each of which maintains its own stack in the global memory for backtracking.

For the application of MCE, [25] is a DFS-based approach that assigns different subtrees to thread blocks, while within a subtree, warps parallelize operations like filtering adjacency lists. The work finds that their GPU implementations only achieved about 1.4–2.25× the performance of a single CPU core. GBK [46] implements the BK algorithm using BFS-based approach but it does not address the issue of subgraph number explosion. Like [25], MCE-GPU [5] also employs a DFS-based approach to implement BK but it implements several effective performance optimizations: (1) binary encoding that represents adjacency lists as bit vectors to allow fast intersections; (2) sub-warp partitioning that divides thread blocks into groups smaller than a warp to resolve the warp over-provisioning issue for small adjacency lists; (3) space-efficient data representations including partial induced subgraphs and a compact representation of the exclusion set X. This tailor-made solution to MCE does achieve remarkable performance, but we find that it runs OOM on many graphs, because technique (3) needs to materialize and maintain an adjacency matrix for each top-level subtree – even though the space cost has been optimized to $O(\Delta + d^2)$ where Δ is the maximum degree and d is the graph degeneracy – which is costly in space. Moreover, all three techniques are orthogonal to the design of G-thinkerCG and can be integrated into our MCE application with some implementation efforts, though we do not implement them currently which makes our MCE program more scalable (since we read adjacency lists from a global graph rather than materialize many adjacency matrices).

For the application of SM, Ullmann's algorithm (recall Algorithm 4) is used to extend partially matched data graph instances into fully matched ones, where each subgraph instance is extended by a warp. Among them, GSI [60] and cuTS [48] are BFS-based and for in-device processing, while PBE [22], VSGM [26] and SGSI [59] explore methods to partition a large input graph so that only one partition needs to be loaded to a GPU for processing at each time. This out-of-core processing technique is orthogonal to the design of our G-thinkerCG, and we can directly integrate VSGM's subgraph k-means-based binning approach to construct graph partitions if the input graph is too large to fit into the device memory of a GPU. More recently, DFS-based approaches emerge such as STMatch [47] and EGSM [40], which assign independent subtrees to warps for backtracking search, each maintaining its own stack in GPU global memory, with load balancing via task splitting and work stealing. While STMatch reports a good performance, EGSM shows that its DFS execution is nearly two orders of

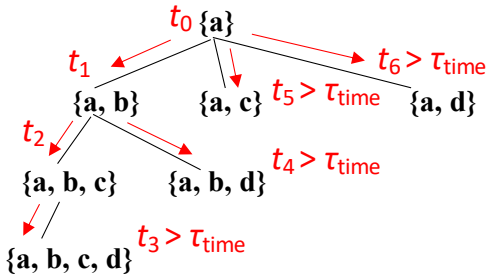


Fig. 10. Timeout Example

magnitude slower than its BFS execution as well as a hybrid BFS-DFS matching strategy, which divides a query graph into groups and extends a group of vertices at a time (BFS), while within each group, subgraph matching is performed in the DFS order. Note that the hybrid BFS-DFS strategy of EGSM is only slightly faster than their BFS strategy, and it is based on query graph partitioning and is quite different from our novel BFS-DFS strategy to be presented in Section III. Last but not the least, GraphPi [37], GraphZero [32] and AutoMine [33] further propose a compilation-based approach to generate efficient subgraph enumeration code with a favorable vertex matching order.

Appendix C
The Algorithm of `atomicSubNonNegative(.)`

To pop a subgraph from the stack $\mathcal{H}$, each thread of a warp calls the `pop_offsets(sglen)` operation shown in Figure 4-❺, which properly decrements `otail` in Line 2, and returns the trackers [`offsets[ot-2], ohead[ot-1]`) for all threads of the warp to read the obtained subgraph from $\mathcal{H}$.`vertices`. Note that since we cannot decrement `otail` to be smaller than 0, we cannot use `atomicSub(.)` since when multiple warps subtract 2 from `otail` consecutively, `otail` can become negative. We solve this issue by calling `atomicSubNonNegative(.)` as shown in Figure 4-❻.

Specifically, Line 1 reads the value of `otail` into variable `old`, which is then read into `assumed` in Line 3. If this value is already 0 (Line 4), `otail` cannot be further decremented so Line 5 returns 0 directly as the original value of `otail`. Otherwise, Line 6 atomically checks if `otail`'s previously retrieved value (i.e., `old`) still equals `assumed`: (1) if so, no other warp has changed `otail` since Line 3, so the decremented value (`assumed` − `dec`) is written to `otail`; (2) while otherwise, `otail` has been updated by another warp, so we cannot update `otail` due to the atomicity requirement of subtraction, and `old` gets the latest value of `otail` in Line 6 and tries again until an atomic subtraction is successfully done (Line 7), after which the old value of `otail` before subtraction is returned in Line 8.

Appendix D
Implementation of Timeout Mechanism

To implement the timeout mechanism for Algorithm 1, we let a task note down its creation time t_0, and in `compute`(c) that runs Algorithm 1, we change Line 5 to check if current time $t_{cur} - t_0 > \tau_{time}$; if so, instead of recursively process the subgraph enumeration subtree, the worker creates a task $t = (R \cup \{v\}, P \cup N(v), X \cup N(v))$ and calls `addTask`(t) to add t to stack $\mathcal{S}$. Figure 10 shows this process for the task with $R = \{a\}$, where timeout occurs before processing $\{a, b, c, d\}$ so it becomes a new task along with $\{a, b, d\}$, $\{a, c\}$ and $\{a, d\}$ during backtracking that are pushed to $\mathcal{S}$.