TPGen: A Self-stabilizing GPU-Based Method for Test and Prime Paths Generation

Ebrahim Fazli[1] and Ali Ebnenasir[2(✉)]

[1] Department of Computer Engineering, Zanjan Branch, Islamic Azad University, Zanjan, Iran
efazli@znu.ac.ir

[2] Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA
aebnenas@mtu.edu

Abstract. This paper presents a novel scalable GPU-based method for Test Paths (TPs) and Prime Paths (PPs) Generation, called TPGen, used in structural testing and in test data generation. TPGen outperforms existing methods for PPs and TPs generation in several orders of magnitude, both in time and space efficiency. Improving both time and space efficiency is made possible through devising a new non-contiguous and hierarchical memory allocation method, called Three-level Path Access Method (TPAM), that enables efficient storage of maximal simple paths in memory. In addition to its high time and space efficiency, a major significance of TPGen includes its self-stabilizing design where threads execute in a fully asynchronous and order-oblivious way without using any atomic instructions. TPGen can generate PPs and TPs of structurally complex programs that have an extremely high cyclomatic and/or Npath complexity.

Keywords: Prime Path · Test Path · GPU Programming

1 Introduction

This paper presents a scalable GPU-based method for the Generation of all Test Paths (TPs) and Prime Paths (PPs), called TPGen, for structural testing. Complete Path Coverage (CPC) is an ideal testing requirement where all execution paths in a program are tested. However, such coverage may be impossible because some execution paths may be infeasible, and the total number of program paths may be unbounded due to loops and recursion. Lowering expectations, one would resort to testing all simple paths, where no vertex is repeated in a simple path, but the Control Flow Graph (CFG) of even small programs may have an extremely large number of simple paths. Amman and Offutt [1] propose the notion of Prime Path Coverage (PPC), where a *prime path* is a *maximal* simple path; a simple path that is not included in any other simple

Published by Springer Nature Switzerland AG 2023
H. Hojjat and E. Ábrahám (Eds.): FSEN 2023, LNCS 14155, pp. 40–54, 2023.
https://doi.org/10.1007/978-3-031-42441-0_4

path. PP coverage is an important testing requirement as it subsumes other coverage criteria (e.g., branch coverage) in structural testing. As such, finding the set of all PPs of a program (1) expands the scope of path coverage, and (2) enables the generation of Test Paths (TPs), which are very important in test data generation. This paper presents a scalable approach for the generation of PPs and TPs in structurally complex programs.

Despite the crucial role of PPC in structural testing, there are a limited number of methods that offer effective and efficient algorithms for generating PPs and TPs for complex real-world programs. Amman and Offutt [2] propose a dynamic programming solution for extracting all PPs. Dwarakanath and Jankiti [6] utilize Max-Flow/Min-Cut algorithms to generate minimum number of TPs that cover all PPs. Hoseini and Jalili [10] use genetic algorithms to generate PPs/TPs of CFGs extracted from sequential programs. Sayyari and Emadi [14] exploit ant colony algorithms to generate TPs covering PPs. Sirvastava *et al.* [15] extract a Markov chain model and produce an optimal test set. Bidgoli *et al.* [4] apply swarm intelligence algorithms using a normalized fitness function to ensure the coverage of PPs. Lin and Yeh [11] and also Bueno and Jino [5] present methods based on genetic algorithm to cover PPs. Our previous work [8] generates PPs and TPs in a compositional fashion where we separately extract the PPs of each Strongly Connected Component (SCC) in a CFG, and then merge them towards generating the PPs of the CFG. Most aforementioned methods are applicable to simple programs and cannot be utilized for PP coverage of programs that have a high structural complexity; i.e., very large number of PPs. This paper exploits the power of GPUs in order to provide a time and space efficient parallel algorithm for the generation of all PPs.

Contributions: The major contributions of this paper are multi-fold. First, we present a novel high-performance GPU-based algorithm for PPs and TPs generation that works in a self-stabilizing fashion. The TPGen algorithm first generates the component graph of the input CFG on the CPU and then processes each vertex of the component graph (each SCC) in parallel on a GPU. TPGen is vertex-based in that each GPU thread T_i is mapped to a vertex v_i and a list l_i of partial paths is associated with v_i. Each thread extends the paths in l_i while ensuring their simplicity. The execution of threads is completely asynchronous. Thread T_i updates l_i based on the extension of the paths in the predecessors of v_i, and removes all covered simple paths from l_i. The experimental evaluations of TPGen show that it can generate all PPs of programs with extremely large cyclomatic [12] and Npath complexity [13] in a time and space efficient way. Cyclomatic Complexity (CC) captures the number of linearly independent execution paths in a program [12]. Npath complexity is a metric for the number of execution paths in a program while limiting the loops to at most one iteration [13]. TPGen outperforms existing sequential methods up to 3.5 orders of magnitude in terms of time efficiency and up to 2 orders of magnitude in space efficiency for a given benchmark. TPGen achieves such efficiency while ensuring data race-freedom without using 'atomic' statements in its design. Moreover, TPGen is self-stabilizing in the sense that the GPU threads start in any order.

Our notion of self-stabilization provides robustness against arbitrary initialization of TPGen where the order of execution of threads is arbitrary. This is different from traditional understanding of self-stabilization where an algorithm recovers if perturbed by transient faults. TPGen threads generate PPs without any kind of synchronization with each other, or with the CPU. Such lack of synchronization significantly improves time efficiency but is hard to design due to the risk of thread interference. As a result, we consider the design of TPGen as a model for other GPU-based algorithms, which by itself is a novel contribution. Second, we propose a non-contiguous and hierarchical memory allocation method, called Three-level Path Access Method (TPAM), that enables efficient storage of maximal simple paths. We also put forward a benchmark of synthetic programs for evaluating the structural complexity of programs and for experimental evaluation of PPs/TPs generation methods.

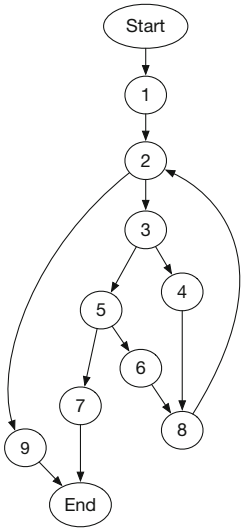
Organization. Section 2 defines some basic concepts. Section 3 states the PPs generation problem. Subsequently, Sect. 4 presents the TPAM method of memory allocation. Section 5 puts forward a highly time and space-efficient parallel algorithm implemented on GPU for PPs generation. Section 6 presents our experimental results. Section 7 discusses related work. Finally, Sect. 8 makes concluding remarks and discusses future extensions of this work.

2 Preliminaries

This section presents some graph-theoretic concepts that we utilize throughout this paper. A *directed graph* $G = (V, E)$ includes a set of vertices V and a set of arcs $(v_i, v_j) \in E$, where $v_i, v_j \in V$. A *simple path* p in G is a sequence of vertices $v_1, \cdots, v_k$, where each arc (v_i, v_{i+1}) belongs to E for $1 \leq i < k$ and $k > 0$, and no vertex appears more than once in p unless $v_1 = v_k$. A vertex v_j is *reachable* from another vertex v_i iff (if and only if) there is a simple path that emanates from v_i and terminates at v_j. A SCC in G is a sub-graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, and for any pair of vertices $v_i, v_j \in V'$, v_i and v_j are reachable from each other. Tarjan [16] presents a polynomial-time algorithm that finds the SCCs of the input graph and constructs its *component graph*. Each vertex of the input graph appears in exactly one of the SCCs. The result is a Directed Acyclic Graph (DAG) whose every vertex is an SCC. A Control Flow Graph (CFG) models the flow of execution control between the basic blocks in a program, where a *basic block* is a collection of program statements without any conditional or unconditional jumps. A CFG is a directed graph, $G = (V, E)$. Each vertex $v \in V$ corresponds to a basic block. Each edge/arc $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from block v_i to block v_j. A CFG often has a *start vertex* that captures the block of statement starting with the first instruction of the program, and has some *end vertices* representing the blocks of statements that end in a halt/exit/return instruction. (We use the terms 'arc' and 'edge' interchangeably throughout this paper.) Figure 1 illustrates an example method as well as its corresponding CFG (adopted from [3]) for a class in the Apache Commons library.

```
Start: private static int binarySearch0 (long[] a, int fromI, int toIndex, long key) {
1:   int low = fromIndex;
1:   int high = toIndex - 1;
2:   while (low <= high) {
3:        int mid = (low + high) >>> 1;
3:        long midVal = a[mid];
3:        if (midVal < key)
4:             low = mid + 1;
5:        else if (midVal > key)
6:             high = mid - 1;
7:        else
7:             return mid; // key found
8:   }
9:        return -(low + 1); // key not found.
End: }
```

(a) java.util.Arrays.binarySearch0()



(b) CFG for method (a)

Fig. 1. Example method and corresponding CFG

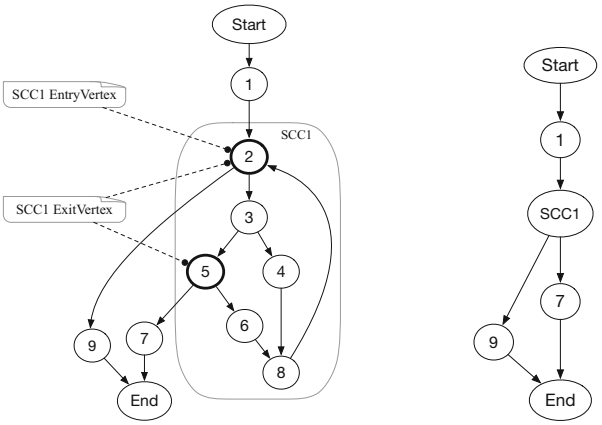
Definition 1 (PP). *A PP is a maximal simple path in a directed graph; i.e., a simple path that cannot be extended further without breaking its simplicity property (e.g., PP* $\langle 2, 3, 4, 8, 2 \rangle$ *in Fig. 1(b)).*

Definition 2 (TP). *A path p from* v_s *to* v_t *is a TP iff* v_s *is the Start vertex of G and* v_t *is an End vertex in G. (e.g., the path* $\langle Start, 1, 2, 3, 4, 8, 2, 9, End \rangle$ *in Fig. 1(b))*

Definition 3 (CompletePP). *A PP p from* v_s *to* v_t *is a CompletePP iff* v_s *is the Start vertex of G and* v_t *is an End vertex in G. (e.g., the PP* $\langle Start, 1, 2, 3, 5, 7, End \rangle$ *in Fig. 1(b))*

Definition 4 (Component Graph of CFGs). *The component graph of a CFG* $G = (V, E)$*, called CCFG, is a DAG whose vertices are the SCCs of G, and each arc* $(v_i, v_j) \in E$ *starts in an* SCC_i *and ends in a distinct* SCC_j *(see Fig. 2(b)).*

Since this paper presents a parallelized version of the method in [8], we represent a summary of the major steps of the algorithm of [8], illustrated in Fig. 3: (1) compute the component graph of the input CFG, denoted CCFG; (2) generate the set of PPs of CCFG and the set of PPs of each individual SCC in CCFG; (3) extract different types of intermediate paths of each SCC, and (4) merge the PPs of SCCs to generate all PPs of the original input CFG. Experimental evidence [8] indicates that the most time consuming step is the second one (i.e., PP generation) where we generate the internal PPs of each individual SCC. This is due to cyclic structure of SCCs. To resolve this bottleneck, we present an efficient parallel algorithm in Sect. 5.



(a) Extracted SCC (b) CCFG

Fig. 2. SCC and CCFG extracted from CFG for Fig. 1(b)

3 Problem Statement

Generating PPs and TPs of the control flow graphs related to real world programs with a large Npath complexity is an important problem in software structural testing. These types of graphs have a huge number of PPs and processing them under conventional algorithms on CPUs requires a lot of time. Thus, it is necessary to develop algorithms that address this problem and maintain the accuracy of the PP generation. In a graph-theoretic setting, the PPs generation problem can be formulated as follows:

Problem 1 (**PPs Generation).**

- **Input**: A graph $G = (V, E)$ that represents the CFG of a given program, a start vertex $s \in V$ and an end vertex $e \in V$.
- **Output**: The set of PPs finished at each vertex $v \in V$ and the set of TPs covering all PPs.

In principle, the number of PPs could be exponential. However, testers should ideally work with a minimum number of TPs that provide a complete PP coverage. Since finding the minimum number of TPs that provide complete PP coverage is hard, we focus on generating a small number of TPs, where each TP covers multiple PPs. For example, consider the second TP in the first column of Table 1 that covers six PPs in the second column of Table 1 (illustrated by the bold fonts). Notice that, this TP starts from the Start node (in Fig. 2(a)), iterates twice in the loop 2-3-4-8-2, and exits through the nodes 5, 7 and End. Figuring out that such a TP can cover six PPs by going through the loop 2-3-4-8-2 twice is non-trivial for human testers. Moreover, generating such TPs is impossible without extracting all PPs. Thus, it is important to efficiently solve Problem 1. We emphasize that testers generate test data only for TPs.

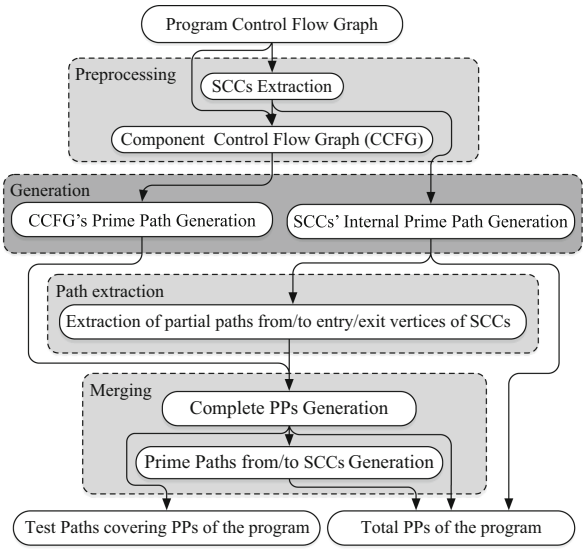


Fig. 3. Overview of the compositional method of [8].

Table 1. TPs and PPs generated for Fig. 1(b)

Test Paths	Prime Paths
{0,1,2,3,5,6,8,2,3,5,6,8,2,3,5,7,10}	**{8,2,3,4,8}{4,8,2,3,4}{2,3,4,8,2}**{4,8,2,3,5,6}
{0,1,2,3,4,8,2,3,4,8,2,3,5,7,10}	{5,6,8,2,3,5} {0,1,2,3,5,6,8} {3,5,6,8,2,9,10}
{0,1,2,3,5,6,8,2,3,4,8,2,9,10}	{2,3,5,6,8,2} **{4,8,2,3,5,7,10}** {3,4,8,2,9,10}
{0,1,2,3,4,8,2,3,5,6,8,2,9,10}	{6,8,2,3,5,7,10} {3,5,6,8,2,3} **{0,1,2,3,4,8}**
{0,1,2,3,5,7,10}	**{3,4,8,2,3}** {6,8,2,3,5,6} {8,2,3,5,6,8}
{0,1,2,9,10}	{5,6,8,2,3,4} {0,1,2,3,5,7,10} {0,1,2,9,10}

In practice, solving Problem 1 is more costly when the input graph is an SCC because every vertex is reachable from any other vertex in an SCC. For this reason, Sect. 5 proposes a parallel GPU-based algorithm that extracts the PPs of SCCs in a time and space efficient fashion. The in-degree of s is 0, and out-degree of e is 0. We focus on CFGs where all vertices $v \in V$ except e have a maximum out-degree of 2. Without loss of generality, we can convert a vertex v with an out-degree greater than 2 (i.e., switch-case structure) to vertices with out-degree 2 by adding some new intermediate vertices between v and its successor vertices. (See details in [9]).

4 Data Structures

In this section, we present a data structure for storing the input CFG (Sect. 4.1), a path data structure (Sect. 4.2), and a novel memory allocation method (Sect. 4.3) for storing the generated PPs.

4.1 CFG Data Structure

A matrix is usually stored as a two-dimensional array in memory. In the case of a sparse matrix, memory requirements can be significantly reduced by maintaining only non-zero entries. Depending on the number and distribution of non-zero entries, we can use different data structures. The Compressed Sparse Row (CSR, CRS or Yale format) [7] represents a matrix by a one-dimensional array that supports efficient access and matrix operations. We employ the CSR data structure (see Fig. 4) to maintain a directed graph in the global memory of GPUs, where vertices of the graph receive unique IDs in $\{0, 1, \cdots, |V| - 1\}$. To represent a graph in CSR format, we store end vertices and start vertices of arcs in two separate arrays $EndV$ and $StartV$ respectively (see Fig. 4). Each entry in EndV points to the starting index of its adjacency list in array $StartV$. We assign one thread to each vertex. That is, thread t is responsible for the vertex whose ID is stored in $EndV[t]$, where $\{0 \leq t < |V| - 1\}$ (see Fig. 4). For example, Fig. 4 illustrates the CSR representation of the graph of Fig. 1(b). Since the proposed algorithm computes all PPs ending in each vertex $v \in V$, maintaining the predecessor vertices is of particular importance. In CSR data structure, first the vertex itself and then its predecessor vertices are stored.

4.2 Path Structure

We utilize a set of flags to keep the status of each recorded path along with each vertex (see Fig. 5). Let v_i be a vertex and p be a path associated with v_i. The PathValidity flag ($p[0]$) indicates whether or not the recorded information represents a simple path. The PathExtension flag ($p[1]$) means that the current path is an extended path; hence not a PP. We assume each non-final vertex can have a maximum of two successor vertices. We use the LeftSuccessor ($p[2]$) and the RightSuccessor ($p[3]$) flags to indicate whether the thread of each corresponding successor has read the path ending in vertex v_i. Once one of those successor threads reads the path ending in v_i it will mark its flag. In each iteration of the algorithm, paths with marked extension and marked successor flags will be pruned. We set the CyclicPath flag ($p[4]$) if p is a cyclic path. If p is cyclic, then it will no longer be processed by the successor threads of v and is recorded as a PP at the v_i. (see Fig. 5).

Note: Since there is a unique thread associated to each vertex, we use the terms "successor" and "predecessor" for both vertices and threads.

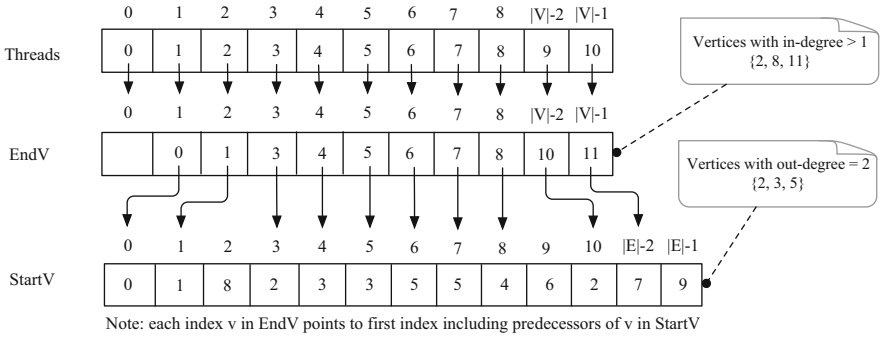


Fig. 4. Compressed Sparse Row (CSR) graph representation of Fig. 1(b)

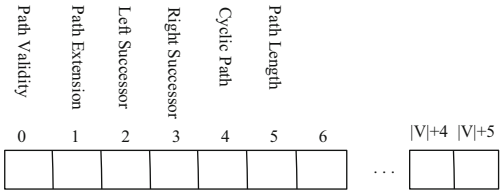


Fig. 5. Path structure

4.3 Three-Level Path Accessing Method (TPAM)

In each CFG, the extraction of the PPs is based on the generation of the simple paths terminated in each vertex $v_i \in V$. There is a list associated to each vertex v_i, denoted $v_i.list$ to record all generated PPs ending in v_i. To implement this idea, all acyclic paths ending in predecessors of v_i must be copied to the list of the vertex v_i. For a large CFG, the number of such paths could be enormous, which would incur a significant space cost on the algorithm. To mitigate this space complexity, we introduce a non-contiguous memory allocation method with a pointer-based Three-level Path Accessing Method (TPAM). TPAM is a path accessing scheme which consists of three levels of address tables in a hierarchical manner. The entries of Level 1 address table with length $|V|$ are pointers to each $v_i.list$ at Level 2 address tables. Level 2 address tables contain addresses of all paths stored in each $v_i.list$. The entries of the last level tables are actual paths information in memory (see Fig. 6).

All activities such as compare, copy, extend and delete are applied to the paths of each vertex. Let v_i be a vertex in V and p be a path in $v_i.list$. To access path p, the start address of the $v_i.list$ is discovered from the first array (i.e., $Path[v_i]$). The start address of path p is stored in Level 2 address table in Fig. 6 (i.e., $Path[v_i][p]$). The list of vertices of path p is in Level 3 path table, which according to path structure mentioned in Fig. 5, all activities can be done on the elements of the path (i.e., $Path[v_i][p][5]$ shows the length of the path p).

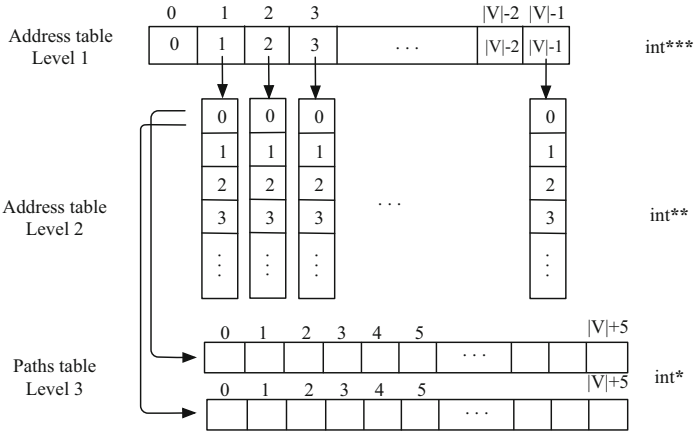


Fig. 6. Three-level Path Accessing Method (TPAM).

Instead of using malloc in allocating host memory, we call CUDA to create page-locked pinned host memory. Page-Locked Host Memory for CUDA (and other external hardware with DMA capability) is allocated on the physical memory of the host computer. This allocation is labeled as non-swappable (not-pageable) and non-transferable (locked, pinned). This memory can be accessed with the virtual address space of the kernel (device). This memory is also added to the virtual address space of the user process to allow the process to access it. Since the memory is directly accessible by the device (i.e., the GPU), the write and read speeds are high bandwidth. Excessive allocation of such memory can greatly reduce system performance as it reduces the amount of memory available for paging, but proper use of this memory allocation method provides a high performance data transfer scheme.

5 GPU-Based Prime Paths Generation

In order to scale up PPs generation, this section presents a parallel compositional method that provides better time efficiency in comparison with existing sequential methods for PPs generation. Specifically, we introduce a GPU-based PPs generation algorithm. The input of the PP generation algorithm (Algorithm 1) includes a CFG representing the Program Under Test (PUT). The output of Algorithm 1 is the set of all PPs finished at each vertex $v_i \in V$.

A GPU-based CUDA program has a CPU part and a GPU part. The CPU part is called the *host* and the GPU part is called the *kernel*, capturing an array of threads. The proposed algorithm includes one kernel. The host (i.e., CPU) initializes the $v_i.list$ of all v_i and an array of boolean flags, called $PublicFlag$, where $PublicFlag[v_i] = true$ indicates that the predecessors of vertex v_i have been updated and so the $v_i.list$ needs to be updated. One important objective is to design a self-stabilizing algorithm with no CPU-GPU communications, thus

the host launches the kernel Update-Vertex (i.e., Algorithm 1) only once. The proposed algorithm is implemented in such a way that there is no need for repeated calls to synchronize different threads. One of the major challenges in parallel applications that drastically reduces their efficiency is the use of atomic instructions. Atomic instructions are executed without any interruption, but greatly reduce the efficiency of parallel processing. The self-stabilizing device (i.e., GPU) code in this section is implemented without using atomic instructions.

Algorithm 1 forms the core of the kernel, and performs three kinds of processing on each vertex $v_i \in V$: pruning the extended paths in $v_i.list$, extending acyclic simple paths in the lists of predecessors of v_i, and examining the termination of all backward reachable vertices from v_i. **Lines 2 to 8** in Algorithm 1 remove extra paths from $v_i.list$. A path $p \in v_i.list$ is *extra* if it is extended by one of the v_i's successor(s) or covered by another path $p' \in v_i.list$.

Lines 9 to 19 extend eligible acyclic simple paths in the lists of all predecessor vertices of v_i. Suppose that $v_j \in V$ is one of the v_i's predecessor. A path $q \in v_j.list$ is an eligible path if q is not a cyclic path, and v_i is the start vertex of q in case v_i already appeared in q. The thread assigned to v_i runs a function called *ExtendPath* (in Algorithm 2) to append the new eligible path to the $v_i.list$. In **Lines 21 to 33**, the thread of v_i cannot be terminated if the vertex v_i is not the final vertex and has any unread paths in $v_i.list$ (Lines 21 to 25). Thread of v_i then examines the termination of all its backward reachable vertices by examining their *PublicFlag*. If all the ancestor vertices of v_i are terminated, then the vertex v_i will also set its *PublicFlag* to false and exit the while loop (Lines 28 to 32). In fact, self-stabilization is achieved through localizing path extension to each thread, but making sure that any change in ancestors of a vertex will eventually propagate to it.

We devise Algorithm 2 to append a new simple path to the list of a given vertex. This algorithm takes a path p as well as a specified vertex v as inputs. Algorithm 2 first adds the vertex v at the end of the path p and increments the length of p (Lines 2 and 3). Then, it checks the occurrence of vertex v as the first vertex of p. This property causes the new path p to be considered as a cycle in vertex v (Lines 4 to 6). Finally, Algorithm 2 sets *PathValidity* flag of the new path p to true and appends it to the end of $v.list$ (Lines 7 and 8).

Theorem 1. *Algorithm 1 terminates, is data race free and finds all PPs.*

Proof. Due to space constraints, we present a proof sketch and refer the readers to the complete proof in [9]. To **prove the termination** of Algorithm 1, we show that at some finite point in time, $v_i.LocalFlag$ and $v_i.PublicFlag$ will become false for every $v_i \in V$ and will remain false. As such, when *PublicFlag* of all vertices in $v_i.ReachedFrom$ are set to false, the thread assigned to v_i will eventually stop. When no more extensions occur for any vertex, Algorithm 1 terminates. To **prove data race freedom**, we show that neighboring threads cannot perform read and write operations on the same path simultaneously. Consider two arcs (v_j, v_i) and (v_i, v_k) in the input CFG. A data race could arise when the thread of v_k reads a path p in $v_i.list$ in Line 12 and at the same time the thread of v_i

Algorithm 1. Update-Vertex($v_i, G = (V, E)$)

```
1: while (v_i.PublicFlag = true) do
2:     for each path p ∈ v_i.list do
3:         if p has been read by both successors then
4:             if (p is an extended path) or (p is already included in some path p' ∈
   v_i.list) then
5:                 remove p;
6:             end if
7:         end if
8:     end for
9:     for each v_j where (v_j, v_i) ∈ E do // Read from predecessors.
10:        for each path q ∈ v_j.list do // PathValidity flag
11:            if q is not read by v_i then
12:                Label q as read by v_i; // Left or Right Successor flag
13:                if (q is not a cycle) and (v_i does not appear in q or v_i is the first
   vertex of q) then
14:                    ExtendPath (q,v_i);
15:                    Label q as an extended path; // PathExtension flag
16:                end if
17:            end if
18:        end for
19:    end for
20:    v_i.LocalFlag = false;
21:    for each path p ∈ v_i.list do
22:        if p is not read by both successors then
23:            v_i.LocalFlag = true;
24:        end if
25:    end for
26:    if v_i.LocalFlag = false then // all paths in v_i.list have been read by both
   v_i's successors
27:        v_i.PublicFlag = false;
28:        for each v_k where v_i is Reachable from v_k do
29:            if v_k.PublicFlag = true then
30:                v_i.PublicFlag = true;
31:            end if
32:        end for
33:    end if
34: end while
```

Algorithm 2. ExtendPath(Path[] p, Vertex v)

```
1: Path[] NewPath = p;
2: NewPath[5 + |p| + 1] = v;
3: NewPath[5] = p[5] + 1;
4: if v is the first vertex of p then
5:     NewPath[4] = 1; // CyclicPath flag
6: end if
7: NewPath[0] = 1; // PathValidity flag
8: append NewPath to v.list;
```

may be removing p in Line 5. However, this cannot occur because thread of v_i removes p if it has been read by both successors. That is, v_k must have read p before v_i can remove it. A similar conflict could occur when v_i extends a path p in $v_j.list$ in Line 14 and v_j wants to remove p in Line 5. This scenario is also impossible to occur because v_j can remove p only if it has already been read by v_i. We also show that if Algorithm 1 fails to find some prime path, then the list of some vertex must have been empty initially, which is contrary to initializing the list of every vertex with itself.

6 Experimental Results

This section presents the results of our experimental evaluations of the proposed GPU-based method for PPs and TPs generation compared to the CPU-based approach proposed in [8]. The experimental benchmark consists of a set of ten modified CFGs from [3] (which are taken from Apache Commons libraries). To increase the structural complexity of input CFGs, we synthetically include extra nested loops and a variety of conditional statements to create more SCCs. Our strategy for creating additional loops/SCCs is to include new arcs from the 'then' part of conditional statements back to their beginning. Table 2 presents the structure of these CFGs. Columns 3 to 9 of Table 2 provide the number of nodes, edges, and SCCs of each CFG. The total numbers of nodes and edges of all SCCs are mentioned as SccNodes and SccEdges, respectively. Columns 7 and 8 show the Cyclomatic Complexity (CC) [12] and Npath Complexity [13] of the input CFGs. The last column illustrates the number of prime paths produced with the GPU-based method. We compare the parallel and the sequential approaches with respect to their running time and memory consumption. The number of generated PPs for each CFG is provided in Column 9 of Table 2. We ran all the experiments on an Intel Core i7 machine with 3.6 GHz X 8 processors and 16 GB of memory running Ubuntu 17.01 with gcc version 5.4.1. The parallel approach is implemented on a Nvidia GTX graphical processing unit equipped with 4G RAM and 768 CUDA cores.

The bar graph of Fig. 7 illustrates the time efficiencies of the CPU-based and GPU-based approaches. (The reported timings for each approach is the average of twenty runs.) These values reflect the fact that the time costs of the CPU-based sequential method is less for smaller CFGs. Specifically, for the CFGs of the top five rows of Table 2, on average, the CPU-based method consumed 61% less time than the GPU-based method (due to the transfer overhead from CPU to GPU). However, for large CFGs at the bottom of Table 2, the parallel GPU-based method costs 39% less time than the sequential method. This time efficiency increases significantly with growing graph size. For example, the GPU-based time efficiency in the last graph is 71%. The recorded times indicate that by increasing the structural complexity, the GPU-based algorithm provides a better performance (assigning exactly one thread for each vertex). Thus, for real-world applications that have a large number of lines and complex structures, the GPU-based algorithm is expected to be highly efficient.

Table 2. Modified benchmark CFGs and their structural complexity

CFG	Original Functions	Graph structure after modification							PPs
		Nodes	Edges	SCC	SccNodes	SccEdges	CC	Npath	
1	AsmClassReaderAccept	180	214	18	78	83	35	2.1e7	35629
2	AsmClassWriterToByteArray	215	258	24	103	110	44	6.1e11	176481
3	SquareMesh2DcreateLinks	244	290	27	115	125	49	3.3e12	139684
4	PrivilizerAsmMethodWriter	355	431	38	160	173	68	4.5e22	253954
5	SingularValueDecomposition	486	567	47	223	244	104	1.1e23	643738
6	ListParserTokenManager	723	853	75	331	351	131	2.0e32	1016762
7	BOBYQAOptimizer	874	994	83	409	762	155	9.3e39	1477397
8	ParserParserTokenManager	963	1119	93	448	490	213	1.3e44	2573594
9	InternalXsltcCompilerCUP	1441	1713	149	626	712	273	4.1e68	4478382
10	XPathLexerNextToken	2160	2566	224	957	1073	404	8.4e97	9563583

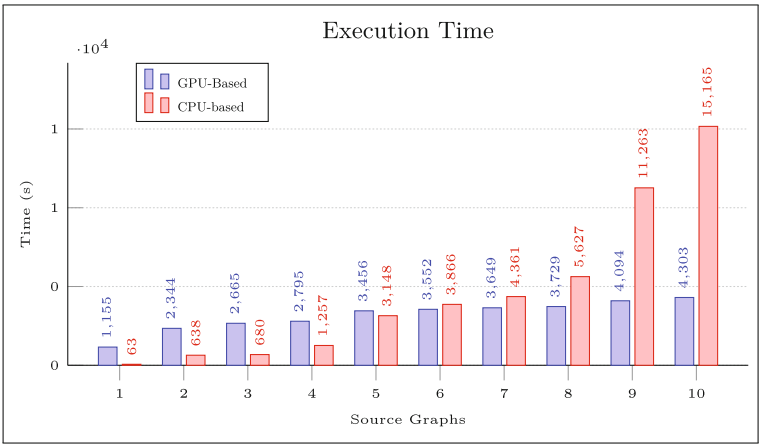


Fig. 7. Time cost of CPU-based vs. GPU-based method.

The bar graph of Fig. 8 illustrates the space efficiency of the CPU-based vs. the GPU-based approach. These values indicate that the GPU-based approach applying TPAM method has less memory costs than the CPU-based method. On average, the GPU-based approach consumes 62% less memory for the input CFGs. On the other hand, for more complex CFGs, the CPU-based method consumes a lot of memory due to the contiguous memory allocation.

7 Related Work

This section discusses related works on the prime and test paths coverage in model-based software testing context. There are two major categories of TPs generation/coverage. Static methods generate TPs of a given CFG. For example, Amman and Offutt [1] start with the longest PP and extend every PP to

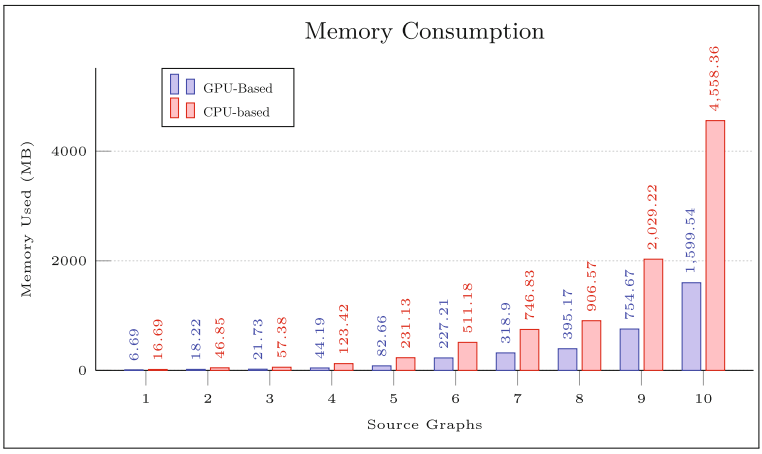


Fig. 8. Memory cost of CPU-based vs. GPU-based method.

visit the start and end vertices, thus forming a TPs. Their process continues with the remaining uncovered longest PPs. This algorithm does not attempt to minimize the number of TPs but is extremely simple. Fazli and Afsharchi [8] extract the set of SCC's entry-exit paths that cover all internal PPs of all SCCs. Then, they merge these paths using the complete paths of the component graph, thereby yielding complete TPs that cover all incomplete PPs. Dynamic methods instrument the PUT in order to analyze the coverage of a set of desired paths. For example, nature-inspired methods (e.g., genetic algorithms [10], ant colony [15], swarm intelligence [4]) provide dynamic methods for PPs and TPs coverage. TPGen, however, is a parallel self-stabilizing vertex-based algorithm that significantly scales up the PPs and TPs generation in a static fashion for structurally complex programs that are beyond the reach of existing methods.

8 Conclusions and Future Work

We presented a novel scalable GPU-based method, called TPGen, for the generation of all Test Paths (TPs) and Prime Paths (PPs) used in structural testing and in test data generation. TPGen outperforms existing methods for PPs and TPs generation in several orders of magnitude, both in time and space efficiency. To reduce TPGen's memory costs, we designed a non-contiguous and hierarchical memory allocation method, called Three-level Path Access Method (TPAM), that enables efficient storage of maximal simple paths in memory. TPGen does not use any synchronization primitives for the execution of the kernel threads on GPU, and starting from any execution order of threads, TPGen generates the PPs ending in any individual vertex; hence providing a fully asynchronous self-stabilizing GPU-based algorithm.

As an extension of this work, we plan to further improve the scalability of TPGen through execution on a network of GPUs. Moreover, we will integrate

PPs/TPs generation with constraint solvers towards generating test data for specific TPs. We will expand the proposed benchmark with more structurally complex programs. We also plan to develop tools that can calculate the structural complexity of a given CFG for different complexity measures (e.g., CC, Npath, PPs), and can compare two programs in terms of their structural complexity. An important application of such tools will be in program refactoring towards lowering structural complexity while preserving functional correctness.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
2. Ammann, P., Offutt, J., Xu, W., Li, N.: Graph coverage web applications (2008)
3. Bang, L., Aydin, A., Bultan, T.: Automatically computing path complexity of programs. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 61–72. ACM (2015). http://www.cs.ucsb.edu/~vlab/PAC/
4. Monemi Bidgoli, A., Haghighi, H., Zohdi Nasab, T., Sabouri, H.: Using swarm intelligence to generate test data for covering prime paths. In: Dastani, M., Sirjani, M. (eds.) FSEN 2017. LNCS, vol. 10522, pp. 132–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_9
5. Bueno, P.M.S., Jino, M.: Automatic test data generation for program paths using genetic algorithms. Int. J. Software Eng. Knowl. Eng. **12**(06), 691–709 (2002)
6. Dwarakanath, A., Jankiti, A.: Minimum number of test paths for prime path and other structural coverage criteria. In: Merayo, M.G., de Oca, E.M. (eds.) ICTSS 2014. LNCS, vol. 8763, pp. 63–79. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44857-1_5
7. Eisenstat, S.C., Schultz, M.H., Sherman, A.H.: Algorithms and data structures for sparse symmetric Gaussian elimination. SIAM J. Sci. Stat. Comput. **2**(2), 225–237 (1981)
8. Fazli, E., Afsharchi, M.: A time and space-efficient compositional method for prime and test paths generation. IEEE Access **7**, 134399–134410 (2019)
9. Fazli, E., Ebnenasir, A.: TPGen: a self-stabilizing GPU-based method for prime and test paths generation. arXiv preprint arXiv:2210.16998 (2022). https://arxiv.org/abs/2210.16998
10. Hoseini, B., Jalili, S.: Automatic test path generation from sequence diagram using genetic algorithm. In: 2014 7th International Symposium on Telecommunications (IST), pp. 106–111. IEEE (2014)
11. Lin, J.C., Yeh, P.L.: Automatic test data generation for path testing using gas. Inf. Sci. **131**(1–4), 47–64 (2001)
12. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)
13. Nejmeh, B.A.: NPATH: a measure of execution path complexity and its applications. Commun. ACM **31**(2), 188–200 (1988)
14. Sayyari, F., Emadi, S.: Automated generation of software testing path based on ant colony. In: 2015 International Congress on Technology, Communication and Knowledge (ICTCK), pp. 435–440. IEEE (2015)
15. Srivastava, P.R., Jose, N., Barade, S., Ghosh, D.: Optimized test sequence generation from usage models using ant colony optimization. Int. J. Softw. Eng. Appl. **2**(2), 14–28 (2010)
16. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)