VF2-PS: Parallel and Scalable Subgraph Monomorphism in Arachne

Mohammad Dindoost, Oliver Alvarado Rodriguez, Sounak Bagchi[+], Palina Pauliuchenka, Zhihui Du, and David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
{md724,oaa9,pp272,zd4,bader}@njit.edu

***Abstract*—This paper introduces a novel, parallel, and scalable implementation of the VF2 algorithm for subgraph monomorphism developed in the high-productivity language Chapel. Efficient graph analysis in large and complex network datasets is crucial across numerous scientific domains. We address this need through our enhanced VF2 implementation, widely utilized in subgraph matching, and integrating it into Arachne—a Python-accessible, open-source, large-scale graph analysis framework. Leveraging the parallel computing capabilities of modern hardware architectures, our implementation achieves significant performance improvements. Benchmarks on synthetic and real-world datasets, including social, communication, and neuroscience networks, demonstrate speedups of up to 97X on 128 cores, compared to existing Python-based tools like NetworkX and DotMotif, which do not exploit parallelization. Our results on large-scale graphs demonstrate scalability and efficiency, establishing it as a viable tool for subgraph monomorphism, the backbone of numerous graph analytics such as motif counting and enumeration. Arachne, including our VF2 implementation, can be found on GitHub: https://github.com/Bears-R-Us/arkouda-njit.**

I. Introduction

Subgraph monomorphism is a general case of the subgraph isomorphism problem, which is known to be NP-complete [1]. Various heuristics have been developed to enhance its practical performance, including pruning rules, strategic searching orders, preprocessing steps, and the utilization of auxiliary data structures. In the pursuit of advancing graph monomorphism tools, we identified a pivotal limitation in the usage of the Python NetworkX library within various scientific communities. Despite its simplicity and widespread adoption, this tool is sequential and cannot exploit the ubiquitous parallel processing capabilities of hardware architectures.

Addressing this computational inefficiency, we developed a parallel version of the subgraph monomorphism algorithm, VF2, in the high-level parallel Chapel language [2], [3]—a decision motivated by Chapel's ability to leverage concurrent hardware resources productively. By implementing the core computations in Chapel while preserving the user interface in Python, we bridge the strength of both worlds: the ease of use of Python and the parallel execution prowess of Chapel.

[+]Sounak Bagchi attends Edison Academy Magnet School, Edison, NJ

Furthermore, existing domain-specific tools, such as DotMotif [4], while proficient within its niche of brain connectome analysis in neuroscience, lack the generality required for broad scientific applications. Our tool transcends this limitation by providing a versatile solution that is not confined to a specific field or dataset type.

The sequential nature of currently available algorithms like those in DotMotif and NetworkX results in large execution times, even for analyses of modest complexity. By parallelizing the VF2 algorithm, we have realized performance enhancements by up to 97X using 128 cores. Such an improvement not only substantiates the effectiveness of parallel processing for graph monomorphism problems but also democratizes access to high-speed computational tools across diverse research domains. Our implementation is a significant step toward enabling comprehensive graph-theoretical studies on a scale and at a speed previously unattainable with existing sequential methods.

We integrated our parallel subgraph monomorphism algorithm into the open-source graph framework Arachne [5], which is an extension to the existing library, Arkouda [6], [7], that provides Pandas- and NumPy-like operations at a supercomputing scale. Arachne fits into typical exploratory data analytics workflows in Arkouda by providing methods to automatically transform tabular data into graphs, allowing for graph-like operations and queries at scale. It is on its way to becoming a tool to provide graph analytical capabilities at a supercomputing scale by harnessing the full power of the systems it runs on [5], [8]–[13].

Arachne automatically executes graph analytical kernels on both shared-memory and distributed-memory, transferring data as needed between compute nodes and their processors. Arachne can be used to process massive, distributed graphs, into workable chunks for shared-memory algorithms such as VF2 or triangle counting. This hybridization allows Arachne to be used everywhere, from single devices to compute clusters, fully exploiting the computing power and architecture of each device.

The main contributions in this paper are as follows:

1) **Parallel and Optimized VF2 Implementation**: We present a parallel and optimized implementation of the

VF2 algorithm for subgraph monomorphism. This implementation was developed using Chapel and integrated into Arachne.

2) **Handling efficiently Large Graphs**: Our implementation has been tailored to handle large graphs efficiently. By leveraging Chapel's capabilities, we have modified the VF2 algorithm to optimize its performance specifically for large graphs across various domains. This includes enhancements that significantly improve its scalability and speed.
3) **Experimental Results**: We provide comprehensive experimental results on both synthetic and real-world graphs. These results demonstrate that our subgraph monomorphism method outperforms the widely-used, Python-based graph packages, DotMotif and NetworkX, in terms of speed. Our approach is particularly effective for large-scale graph data, showcasing substantial performance improvements.

II. Problem Formulation

Subgraph isomorphism is a fundamental problem in graph theory and computational science found in extensive applications across various domains, including chemistry [14], bioinformatics [15], pattern recognition [16], [17], neuron science [18], [19] and network analysis [20]. In addition, solving the subgraph monomorphism problem yields solutions to other well-known graph problems, including the graph monomorphism problem [21], k-truss counting [22], the Hamiltonian Cycle problem [23], and the maximum clique [24] problem.

For the purposes of this paper, we focus on a more general case of subgraph isomorphism, namely subgraph monomorphism. A subgraph monomorphism between two graphs exists if there is a mapping from the vertices of one graph to the vertices of another graph that preserves vertex adjacency. Unlike isomorphism, this mapping does not need to be bijective; it only needs to be injective. That is, the number of edges between the matched vertices does not need to be exact, only that at least one edge exists.

Let $G_1 = (V_1, E_1)$ be a graph (host graph) with $n_1 = |V_1|$ vertices and $m_1 = |E_1|$ edges. Define G_2 similarly as (V_2, E_2). Moreover, let vertices and edges be labeled through $\lambda(v)$ and $\lambda(u, v)$, respectively. A *subgraph monomorphism* is an injective function $f : V_2 \to V_1$ satisfying the following properties:

- $\forall u, v \in V_2, (u, v) \in E_2 \implies (f(u), f(v)) \in E_1$
- $\forall u \in V_2$, $\lambda(u) = \lambda(f(u))$
- $\forall (u, v) \in E_2$, $\lambda(u, v) = \lambda(f(u), f(v))$

We say that G_2 is monomorphic to a subgraph of G_1 under the mapping function f. Generally, the host graph and subgraph can be either directed or undirected, but for this paper, we focus on directed graphs. Our implementation can be easily expanded to work for undirected graphs. Given two graphs, the problem involves determining whether one graph can be embedded within the other as a subgraph, preserving both vertex labels and edge relationships. Despite its theoretical significance, subgraph isomorphism, and transitively subgraph monomorphism, is recognized as an NP-complete problem [25], implying that finding an exact solution for larger graphs becomes increasingly challenging and computationally intractable.

III. Parallel Algorithm Implementation

In Algorithm 1, we show our parallel subgraph monomorphism algorithm. Our primary improvement and optimization methods are two-fold. Firstly, we reduce the amount of space utilized by the VF2 implementation by restructuring the state data structure. In the original VF2 data structure, two vectors, $core\_1$ and $core\_2$ are used to keep the current mapping. However, iterating over $core\_1$ is both time-consuming and inefficient, particularly for large graphs, leading to wasted time and storage. To address this, we just use $core_2[n_2] = n_1$ to keep the current mapping (n_1, n_2), which means that the vertex n_1 in the host graph G_1 is mapped to n_2 in the subgraph G_2. This restructuring not only saves space but also makes the search for unmapped vertices easier. Based on the simplified state data structure, it suffices to check the value of $core_2[i], 0 \leq i \leq |V_2|-1$, to know if a vertex has been mapped. If it retains the original value of -1, this indicates that vertex i has not been mapped. Otherwise, it indicates that i in G_2 is mapped to $core_2[i]$ in G_1. By eliminating the need to iterate over $core\_1$, we make the algorithm more efficient in terms of both storage and parallel performance.

Secondly, we leverage the highly efficient and dynamic parallelization capabilities of Chapel, which automatically generates parallel tasks and assigns them to available threads based on the current system load. The optimal point to spawn these tasks is immediately after generating $candidates$ (line 9), ensuring that task execution is distributed effectively. As seen on Algorithm 1, by carefully structuring our algorithm to eliminate data dependencies, we allow for seamless parallel execution. As a result, tasks are dynamically allocated to available threads, maximizing resource utilization. Some tasks may be completed earlier than others, particularly if none of the candidates are feasible. In such instances, freed threads can be promptly reassigned to subsequent recursive calls, provided they are available when execution reaches one of the nested $forall$ loops. This approach ensures efficient use of computational resources, enhancing both performance and scalability.

Now, we will discuss each extra method within Algorithm 1. We will also discuss how we formulated our $State$ data structure and maintain our final mappings in a list M. We utilize the typical definition of a list, where ours is based off the Chapel language standard library list that allows for parallel-safe appends (`pushBack`) and removals from the end of the list (`popBack`) to prevent race conditions.

A. *State Architecture*

States that are crucial for the matching process and can be described as a state space representation (SSR) [26]. Each state generated is stored inside of a class named $State$ that stores the sizes of the graph and subgraph with integer variables n_1

Algorithm 1 Parallel *VF2* algorithm that generates the mappings of vertices u from the host graph that are mapped to vertices v from the subgraph.

Input: A state $S_{current}$ with the current mapping information for a given recursive depth d.

Output: Mappings M of all host graph and subgraph pairs that induce a monomorphism.

```
1: $M = list(int)$                                   ▷ Parallel-safe list.
2: if $d == n_2$ then                                ▷ $n_2$ is the size of the subgraph.
3:     for $v \in S_{current}.core$ do
4:         $M.pushBack(v)$
5:     end for
6:     return $M$
7: end if
8: $candidates = getCandidatePairs(S_{current})$
9: for all $(u, v) \in candidates$ do
10:    if $isFeasible(u, v, S_{current})$ then
11:        $S_{clone} = S_{current}.clone()$
12:        $addToTinTout(u, v, S_{clone})$
13:        $M_{new} = VF2(S_{clone}, d + 1)$
14:        for $m \in M_{new}$ do
15:            $M.pushBack(m)$
16:        end for
17:    end if
18: end for all
19: return $M$
```

and n_2, respectively. Further, a single *core* is maintained to keep partial mapping, where the indices of *core* correspond to the indices of the subgraph vertices. Then, the successors and predecessors of each group of mapped vertices from the host graph and subgraph are maintained (terminal sets) to quickly generate the next candidates in the next depth $d + 1$ as seen in line 13 of Algorithm 1. Lastly, each state contains methods to quickly return if a given vertex from the host graph or subgraph is mapped. This is discussed at the beginning of Section III.

B. Checking Feasibility of (u, v) Mapped Pairs

The most important part of subgraph monomorphism is confirming that a candidate pair is feasible because we need to ensure that we generate and maintain only consistent states. This procedure is responsible for both syntactic and semantic checks. This step is illustrated in line 10 of Algorithm 1. It involves checking five different cases where u is the vertex from the host graph being mapped to some vertex v in the subgraph. The first two cases, known as core rules [27], check the consistency of the partial mapping. The remaining cases help us distinguish and prune the search trees.

1) **Rpred** Ensures that the mappings of the predecessor vertices of u and v are consistent with the existing partial mapping in the given state. This is equivalent to a check of the topological consistency of the graph.
2) **Rsucc** Ensures that the structural relationship of successor vertices of u and v is preserved in the mapping. This rule is essential for pruning the search space and ensuring the integrity of the subgraph monomorphism being found.
3) **Rin** Ensures that the candidate pair (u, v) being considered for the mapping does not violate the in-degree constraints. This is done by comparing the number of predecessor vertices of u and v that are not already part of the partial mapping. The rule enforces that $u \in G$ should not have more predecessors outside the current mapping than $v \in H$. This condition helps in pruning the search space effectively.
4) **Rout** ensures that the candidate pair does not violate out-degree constraints in the VF2 algorithm. It checks that the number of connections u and v have with vertices not yet included in the mapping is consistent and follows the subgraph monomorphism rules. The rule ensures that the vertex $u \in G$ should not have more successors outside the current mapping than $v \in H$. This condition helps in pruning the search space effectively as with Rin.
5) **Rnew** ensures that the candidate pair (u, v) being considered for inclusion in the mapping is compatible in terms of their relationships with vertices not yet included in the mapping. This rule looks at both predecessor and successor vertices of u and v, comparing their counts with the vertices outside the current mapping. In essence, this rule checks whether the potential addition of (u, v) to the mapping would maintain the consistency of the graph structure, particularly with respect to nodes that are yet to be mapped. It ensures that u does not have more external (outside the current mapping) predecessor or successor connections than v. This constraint helps reduce the search space by pruning candidate pairs that cannot be part of a valid monomorphism, thus enhancing the efficiency of our VF2 algorithm.

C. Generating Candidate Pairs

Using the optimized way of checking if vertices from the host graph and subgraph have not been mapped, as discussed in Section III-A, the candidates are generated by the method in line 8 of Algorithm 1 by looping over out-neighbors of the state at a given point of the subgraph and adding them as candidates with the vertices that make up the out-neighbors of the state from the host graph. This is done only as long as there are out-neighbors available for both the host graph and subgraph mapped vertices at a given state. If not, then the same checks are applied to the in-neighbors. If there are no in-neighbors generated for either of the mapped vertices for the host graph and subgraph, then both unmapped vertices are returned for the host graph and main graph.

D. Generating T_{in} and T_{out}

Line 12 of Algorithm 1 generates the successors and predecessors (terminal sets) of the current generated mappings and adds them to a given state. This is straightforward as it checks to ensure the successors and predecessors have not yet been mapped, and if not, they are added to T_{in} and T_{out} for both the host graph and subgraph. The vertices invoking this method, the candidate pair (u, v) where u is from the host graph, and v from the subgraph, are also removed.

IV. Experimental Study

A. Datasets

We carried out our experiments on both a suite of synthetic graphs that were crafted explicitly for benchmarking and real-world datasets. The synthetic directed graphs were derived from standard random graph models, including Erdős–Rényi [28], Watts–Strogatz [29], and Barabási–Albert [30], which are frequently used in network analysis studies.

The real-world datasets used include the Hemibrain v.1.2 dataset [31], the Enron email network, and the Math Overflow temporal network [32]–[36] whose specifications are noted in Table. I. To demonstrate the efficiency and performance of Arachne, we conducted systematic comparisons between our implementation and those from well-established and widely used Python libraries such as NetworkX and DotMotif.

B. Experimental Configuration

1) Hardware Configuration: Experiments were computed on a server that contains two AMD EPYC 7713 (Milan) @ 2.0GHz CPUs with 64 cores per CPU and 1TB RAM.

2) Software Configuration: Arachne is set up to work in the client-server model, where the client is usually a Python script or Jupyter notebook and the server is the Arkouda server usually running on an HPC (high-performance computing) system. For our experiments, two compute nodes were allocated, one to behave as the server and the other as the client, with the Arkouda server running on the server node and the Python script executing on the client node. Each experiment had its own Arkouda server running since currently, Arkouda servers only allow one connected instance.

C. Performance Comparisons

The first set of experiments is based on Erdős–Rényi random graphs. To encompass the full spectrum of graph types one might encounter in practice, we assembled a diverse collection of Erdős–Rényi graphs. The generation of these graphs was centered on varying the probability of an existing edge between a pair of vertices, ranging from a sparsity of $P = 0.0005$ to a fully connected state of $P = 1$. For the sake of space, we only present the results for the two ends of the probability spectrum. These results can be seen in Figs. 1 and 2.

The diagrams provided show that Arachne's computation time remains the lowest across all probabilities, maintaining a significant lead even as the number of edges reaches millions. When $P = 0.0005$ (Fig. 2), the graphs depict a gradual (almost linear) increase in computation time for all three libraries as the edge counts increase, but Arachne's slope is remarkably flatter. The most telling is the graph with $P = 1$ (Fig. 1) where every possible edge is present, making the graph fully connected. Here, Arachne significantly outperforms the competing libraries, underlining its robustness in handling dense networks with millions of found monomorphisms (monos). Another perspective to consider is the scale of the largest graphs in these figures. The largest graph in Fig. 1 contains almost 94 million monos, while the largest one in Fig. 2 contains nearly 13 million monos. This indicates that in Fig. 1, there is substantial backtracking due to the density of connections. In contrast, despite the large number of nodes and edges in Fig. 2, Arachne's efficient handling of large graphs minimizes the performance impact. This demonstrates Arachne's capability to manage extensive graph structures effectively, showcasing its superior performance in both sparse and dense graph scenarios.

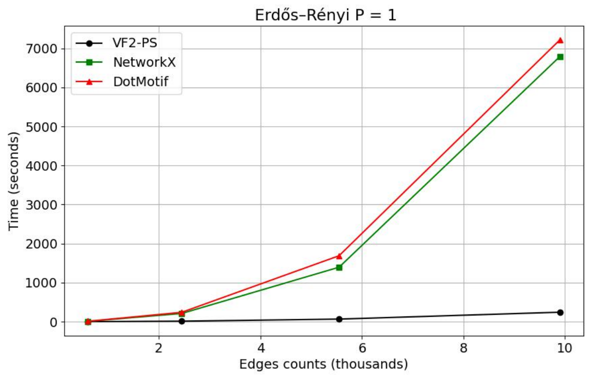

Fig. 1. Execution time comparison on random graphs with p = 1.

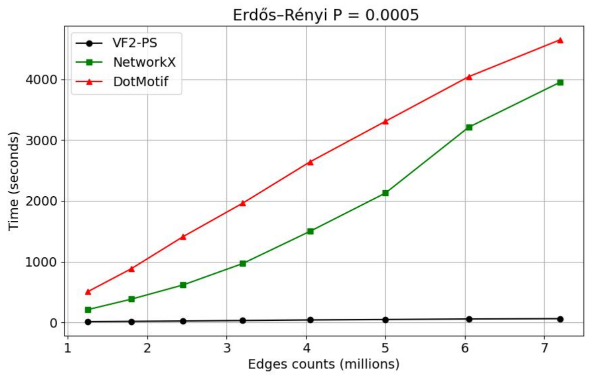

Fig. 2. Execution time comparison on random graphs with p = 0.0005.

The next series of experiments utilizes Barabási–Albert graph generation models, which are also known as "scale-free networks" due to their characteristic power-law degree distribution. In this study, we generated scale-free graphs based on different configurations of the Barabási-Albert model parameters to showcase the spectrum of possible network structures within the scale-free paradigm, from a highly clustered network structure with a significant focus on vertices with high in-degrees, indicative of a centralized connectivity pattern, to a star-like topology with influential central vertices. Fig 3 shows one of the many generated configurations, characterized by parameters $\alpha = 0.41$, $\beta = 0.54$, and $\gamma = 0.05$,

TABLE I

REAL-WORLD DATASETS USED FOR EXPERIMENTATION, SORTED BY THE NUMBER OF EDGES.

Dataset	Number of vertices	Number of edges	Density	Field
Enron Emails	36,692	183,831	0.0001	Communication network
Math Overflow	24,818	506,550	0.0008	Social network
Hemibrain v1.2	21,739	3,550,403	0.0075	Neuroscience

produces a network with moderate preferential attachment and a higher propensity for internal connections, resulting in moderate clustering and balanced degree distribution.

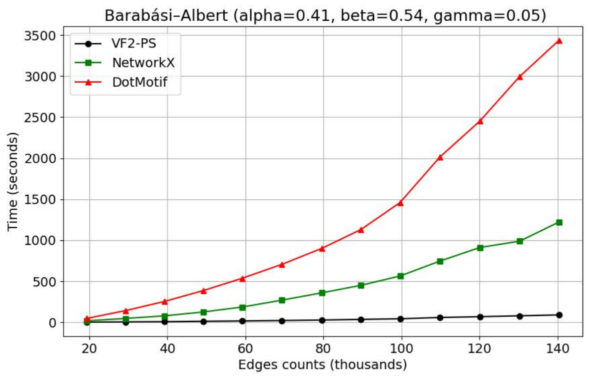

Fig. 3. Execution time comparison on scale-free graphs.

For the third category of random graphs, we employed the Watts-Strogatz model to generate networks with a variety of configurations to illustrate the variability in small-world network characteristics. These configurations ranged from networks with a high degree of rewiring, introducing significant randomness and resulting in attributes similar to random graphs, characterized by relatively low clustering and shorter path lengths. Conversely, some configurations preserved much of the original ring lattice structure, resulting in networks with high clustering and longer path lengths, typical of regular lattices.

These contrasting setups demonstrate the impact of the rewiring probability in transitioning from highly structured networks to those exhibiting more random properties, highlighting the flexible nature of the Watts-Strogatz model in exploring the continuum between regular and random networks. The results consistently reveal that our implementation can efficiently handle the intricate structures of small-world graphs with millions of edges, which are common in many real-world scenarios. For the sake of conciseness, we present only one of the diagrams as a sample of the results, as VF2-PS consistently outperformed the other algorithms across all configurations, as shown in Fig. 4.

To provide a rigorous evaluation of Arachne's performance and accuracy, we conducted an extensive series of tests examining subgraph monomorphism across various structures. These structures, including three-node, four-node, and fully connected cliques of increasing sizes, were carefully selected for their relevance and frequent occurrence in numerous do-

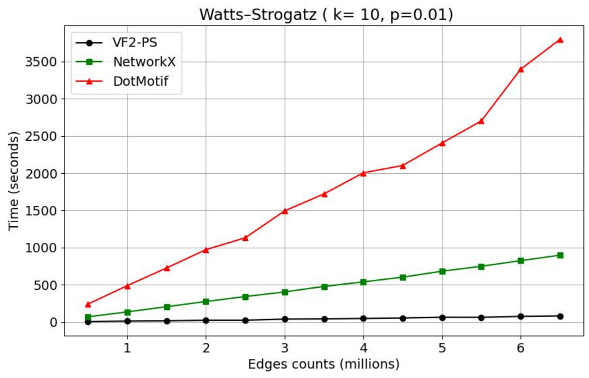

Fig. 4. Execution time comparison on small-world graphs with $k = 10$ and $p = 0.01$.

mains, such as motif counting, where subgraph search tasks are prevalent. In crafting our experimental graphs, we generated 300 distinct directed Erdős–Rényi graphs. The vertex counts for these graphs were uniformly distributed within a range of 100 to 300, and their edge densities were uniformly sampled from a continuum spanning 0.05 to 0.1. This thorough approach enabled a comprehensive assessment of Arachne, measuring its efficiency and precision against a broad spectrum of graph densities and configurations that one might encounter in real-world scenarios.

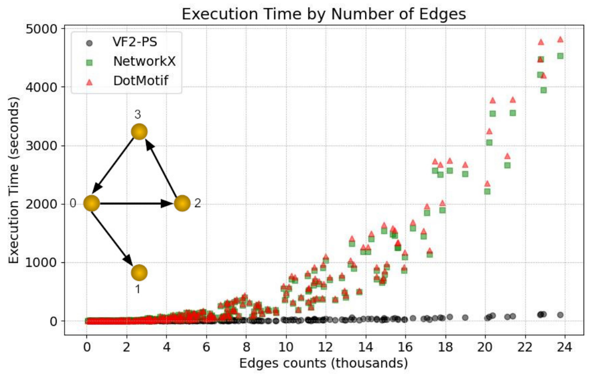

Fig. 5. Execution time comparison for 300 random graphs with density (0.05 to 0.1).

These tests were carefully designed to examine the effect of both network size and subgraph size and structure. By employing the mentioned configurations, we ensured a robust evaluation of VF2-PS's capabilities. Fig. 5 reflects the aggregation of data from 300 individual graph generations using one of the most challenging subgraphs to stress and test all parts of our implementation. This provides a robust statistical basis for our conclusions.

D. *Performance Results on Real-World Graphs*

In order to expand our understanding of subgraph monomorphism and validate the practical effectiveness of Arachne, we extended our analysis to encompass real-world datasets. These include directed three-vertex, four-vertex and five-vertex subgraphs from the Hemibrain v1.2 dataset in neuroscience, the Math Overflow temporal network in social science, and the Enron email network in communication science. The results of three-vertex, illustrated in Figures 6-8, reveal VF2-PS's capability to efficiently process complex and diverse graph structures. Three-vertex motifs, specifically "fan-in," "fan-out," and "path-2," are prevalent across various domains and are characterized by unique connectivity patterns between vertices. Our analysis indicates that the number of monomorphisms detected and the orientation of the edges significantly influence the performance of subgraph monomorphism tools. For the Enron email network, Arachne achieved an impressive speedup of 81.516 times compared to the widely used NetworkX. Similarly, in the Math Overflow dataset, VF2-PS facilitated a speedup of 72.780 times, and for the Hemibrain dataset, the speedup reached 97.023 times. These metrics highlight Arachne's robust performance and precision in motif finding tasks.

This consistent performance across varied datasets highlights VF2-PS's adaptability and efficiency as an analytical tool, making it particularly suitable for fields like computational biology and social network analysis, where the directional characteristics of relationships are fundamental to network dynamics.

E. *Scalable Performance*

To demonstrate how our parallel implementation can leverage cores to improve performance, we used different numbers of parallel threads to run the same task on the same graph. In Chapel, we can update the value of the environmental variable `CHPL_RT_NUM_THREADS_PER_LOCALE` to vary the number of threads. For this category of experiments, we generated five different graphs with a total number of vertices ranging from 250 to 4,000 and P values ranging from 0.02 to 0.05. Our hardware configuration could support up to 128 threads, so we tested each graph with 1 to 128 threads. The results shown in Fig. 9, belongs to this category with P sets to 0.03.

When the graph is large enough, we can see that the execution time will decrease almost linearly with the total number of threads (the graph with 4,000 vertices). This means that when we increase the number of parallel threads, the total execution time will decrease linearly. However, if the graph size is not large enough when we increase the number of parallel threads, the parallel efficiency will decrease (for graphs with 1,000 and 2,000 vertices), and even worse, the total execution time will increase (for graphs with 250 and 500 vertices). This trend

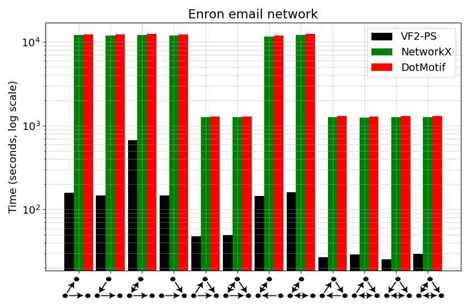

Fig. 6. Execution times for 3-vertex motifs for the Enron email network.

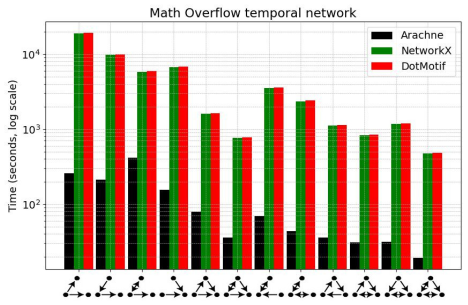

Fig. 7. Execution times for 3-vertex motifs for the Math Overflow temporal network.

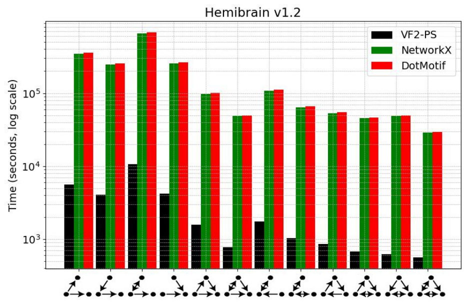

Fig. 8. Execution times for 3-vertex motifs for the Hemibrain dataset.

can be seen in graphs with a small size. The reason for this is that when the graph size is small, there is not enough work to be assigned to each parallel thread. However, employing more parallel threads will have additional overhead. When the time savings of parallel execution cannot compensate for the additional parallel overhead, the total execution time will increase instead of decrease. Our experimental results in Fig. 9 show that our method can achieve scalable performance on large graphs.

V. Related Work

Subgraph monomorphism can be formulated as a constraint programming problem. The backtracking approach is widely used to solve subgraph monomorphism. The constraint programming approach was introduced by McGregor [37] to solve the problem and improved by Ullmann [38]. Significantly, Ullmann proposed an algorithm called Focus-Search, which

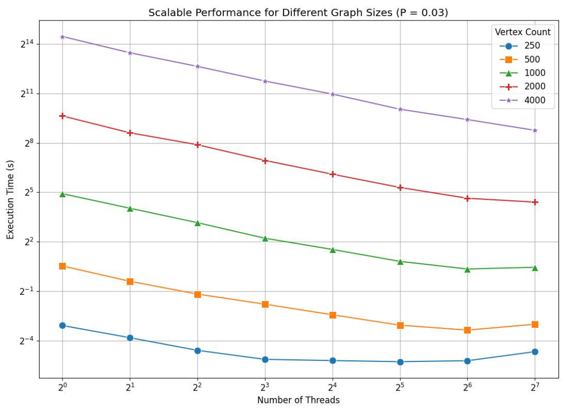

Fig. 9. Execution time comparison for random graphs with the same probability $P = 0.03$ but different number of vertices and edges when different parallel threads are used.

applies a bit-vector domain reduction to each step, to improve his early version [39].

State space representation is another popular data structure for solving the subgraph isomorphism problem. In this approach, the search space (also called state space) is conceptually defined as a tree of states, where each state corresponds to a partial mapping of the pattern vertices onto target vertices. For example, a typical VF2 method [40] contains two main phases: search and refinement. The first step is the same as Ullmann's algorithm. The main difference is in the refinement phase. The algorithm initially deals with the first vertex, selects a vertex connected with the already matched query vertices, searches for a subgraph match, and backtracks if not. The real innovation of VF2 is it brings in feasibility rules to prune in advance. VF2++ [41], VF3 [42], and VF3-Light [43] are different variants of the VF algorithm family.

QuickSI [44] identifies a sequence of query graph nodes to match by using the node frequency information that is pre-computed from the data graph and tries to access nodes having infrequent vertex labels and infrequent adjacent edge labels as early as possible. SPath [45] uses a path-based indexing technique as patterns of comparison in the data graph and neighborhood signatures to minimize searching space. BoostIso [46] defines four types of vertex relationships to reduce duplicate computation. CFL-Match [47] presents the core-forest-leaf decomposition of the query graph, and the compact path-based index aims to postpone cartesian products. CECI [48] proposes the compact embedding cluster index and divides the data graph into multiple mappings clusters for parallel processing. These algorithms employ different heuristic methods to improve their performance.

Several works [49]–[52] have been conducted to compare the performance of different algorithms or even the efficiency of different heuristic methods. Their results show that no algorithm can always be the best for all cases. How to employ different heuristic methods for different scenarios can have very different effects.

Parallel technology is another important method to improve practical performance. PGX.ISO [53] employed a parallel in-memory method to achieve a significant performance boost over an existing secondary storage-based RDF database. STwig [54] works in a distributed environment. Graphflow [55] can execute with multiple threads on CPUs. CECI [48] and Glasgow [56], [57] can run in parallel on both a single machine and multiple machines. VF3P [58] is the parallel version of VF3.

GpSM [59] and GunrockSM [60] outperform previous works by leveraging breadth-first search favoring GPU parallelism. PBE [61] divides the graph into several partitions, each of which can be placed in GPU memory to support a large data graph beyond the capability of GPU memory. Zeng *et al.* [62] avoids joining twice and employs fine-grained load balance strategies to improve their algorithm on GPUs.

Currently, there are useful packages that scientists can use for research. In the field of connectomes, DotMotif [4] is developed to reduce the expertise and time required to analyze biological graphs of any size. Conversely, data scientists also use the NetworkX Python library [63].

VI. Conclusion

Large graph analytics, particularly for NP-complete problems, present significant challenges for data scientists. Providing data scientists with high-performance, easy-to-use tools for handling large graphs can significantly enhance their productivity and efficiency. This paper presents a novel parallelized implementation of the VF2 subgraph monomorphism algorithm within the open-source Arachne graph framework. Arachne supports large-scale graph analytics by leveraging supercomputers or cloud resources through a popular Python interface. Comprehensive experimental results demonstrate that Arachne outperforms other Python-based subgraph monomorphism implementations, such as NetworkX and DotMotif.

For future work, we plan to expand our evaluation to existing parallel frameworks focusing on distributed-memory parallelism.

Acknowledgment

We thank the Chapel and Arkouda communities for their support and guidance and the NSF for funding this project through grant CCF-2109988.

References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, 2023, pp. 143–152.

[2] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[3] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu *et al.*, "Chapel comes of age: Making scalable programming productive," *Cray User Group*, 2018.

[4] J. K. Matelsky, E. P. Reilly, E. C. Johnson, J. Stiso, D. S. Bassett, B. A. Wester, and W. Gray-Roncal, "DotMotif: an open-source tool for connectome subgraph isomorphism search and graph queries," *Scientific Reports*, vol. 11, no. 1, p. 13045, Jun. 2021, number: 1 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/s41598-021-91025-5

[5] O. A. Rodriguez, Z. Du, J. Patchett, F. Li, and D. A. Bader, "Arachne: An Arkouda Package for Large-Scale Graph Analytics," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7.

[6] M. Merrill, W. Reus, and T. Neumann, "Arkouda: Interactive Data Exploration Backed by Chapel," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, 2019, pp. 28–28.

[7] W. Reus, "2020 Keynote Arkouda: Chapel-Powered, Interactive Supercomputing for Data Science," in *Chapel Implementers and Users Workshop (CHIUW) 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 650–650.

[8] Z. Du, O. Alvarado Rodriguez, J. Patchett, and D. A. Bader, "Interactive Graph Stream Analytics in Arkouda," *Algorithms*, vol. 14, no. 8, 2021. [Online]. Available: https://www.mdpi.com/1999-4893/14/8/221

[9] O. A. Rodriguez, F. V. Buschmann, Z. Du, and D. A. Bader, "Property Graphs in Arachne," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–7.

[10] Z. Du, O. A. Rodriguez, F. Li, M. Dindoost, and D. A. Bader, "Contour algorithm for connectivity," in *The 30th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), India, December 18-21, 2023.*, 2023.

[11] Z. Du, J. Patchett, O. A. Rodriguez, F. Li, and D. A. Bader, "High-performance truss analysis in Arkouda," in *The 29th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), India, December 18-21, 2022.*, 2022.

[12] O. A. Rodriguez, Z. Du, J. T. Patchett, F. Li, and D. A. Bader, "Arachne: An Arkouda package for large-scale graph analytics," in *The 26th Annual IEEE High Performance Extreme Computing Conference (HPEC), Virtual, September 19-23, 2022*, 2022.

[13] Z. Du, O. A. Rodriguez, and D. A. Bader, "Enabling exploratory large scale graph analytics through Arkouda," in *The 25th Annual IEEE High Performance Extreme Computing Conference (HPEC), Virtual, September 20-24, 2021*, 2021.

[14] V. Lacroix, C. G. Fernandes, and M.-F. Sagot, "Motif Search in Graphs: Application to Metabolic Networks," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 3, no. 4, pp. 360–368, 2006.

[15] V. Carletti, P. Foggia, and M. Vento, "Performance comparison of five exact graph matching algorithms on biological databases," in *New Trends in Image Analysis and Processing–ICIAP 2013: ICIAP 2013 International Workshops, Naples, Italy, September 9-13, 2013. Proceedings 17*. Springer, 2013, pp. 409–417.

[16] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[17] P. Foggia, G. Percannella, and M. Vento, "Graph matching and learning in pattern recognition in the last 10 years," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 28, no. 01, p. 1450001, 2014.

[18] O. Sporns, "Structure and function of complex brain networks," *Dialogues in clinical neuroscience*, vol. 15, no. 3, pp. 247–262, 2013.

[19] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[20] V. Lyzinski, D. E. Fishkind, M. Fiori, J. T. Vogelstein, C. E. Priebe, and G. Sapiro, "Graph matching: Relax at your own risk," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 1, pp. 60–73, 2015.

[21] S. Fortin, "The Graph Isomorphism Problem," 1996.

[22] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, no. 3, pp. 157–166, 2017.

[23] V. S. Borkar, V. Ejov, J. A. Filar, and G. T. Nguyen, *Hamiltonian cycle problem and Markov chains*. Springer Science & Business Media, 2012, vol. 171.

[24] Q. Wu and J.-K. Hao, "A review on algorithms for maximum clique problems," *European Journal of Operational Research*, vol. 242, no. 3, pp. 693–709, 2015.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990.

[26] N. J. Nilsson, *Principles of artificial intelligence*. Springer Science & Business Media, 1982.

[27] V. Carletti, P. Foggia, and M. Vento, "Vf2 plus: An improved version of vf2 for biological graphs," in *Graph-Based Representations in Pattern Recognition: 10th IAPR-TC-15 International Workshop, GbRPR 2015, Beijing, China, May 13-15, 2015. Proceedings 10*. Springer, 2015, pp. 168–177.

[28] O. ERD and A. Renyi, "On random graphs," *Publ. Math*, vol. 6, pp. 290–297, 1959.

[29] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[30] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[31] C. S. Xu, M. Januszewski, Z. Lu, S.-y. Takemura, K. J. Hayworth, G. Huang, K. Shinomiya, J. Maitin-Shepard, D. Ackerman, S. Berg *et al.*, "A connectome of the adult drosophila central brain," *BioRxiv*, pp. 2020–01, 2020.

[32] S. J. Cook, T. A. Jarrell, C. A. Brittin, Y. Wang, A. E. Bloniarz, M. A. Yakovlev, K. C. Nguyen, L. T.-H. Tang, E. A. Bayer, J. S. Duerr *et al.*, "Whole-animal connectomes of both caenorhabditis elegans sexes," *Nature*, vol. 571, no. 7763, pp. 63–71, 2019.

[33] J. G. White, E. Southgate, J. N. Thomson, S. Brenner *et al.*, "The structure of the nervous system of the nematode caenorhabditis elegans," *Philos Trans R Soc Lond B Biol Sci*, vol. 314, no. 1165, pp. 1–340, 1986.

[34] T. A. Jarrell, Y. Wang, A. E. Bloniarz, C. A. Brittin, M. Xu, J. N. Thomson, D. G. Albertson, D. H. Hall, and S. W. Emmons, "The connectome of a decision-making neural network," *Science*, vol. 337, no. 6093, pp. 437–444, 2012.

[35] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[36] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM 2017. ACM, Feb. 2017. [Online]. Available: http://dx.doi.org/10.1145/3018661.3018731

[37] J. J. McGregor, "Relational consistency algorithms and their application in finding subgraph and graph isomorphisms," *Information Sciences*, vol. 19, no. 3, pp. 229–250, 1979.

[38] J. R. Ullmann, "Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism," *ACM J. Exp. Algorithmics*, vol. 15, feb 2011. [Online]. Available: https://doi.org/10.1145/1671970.1921702

[39] J. Ullmann, "An Algorithm for Subgraph Isomorphism," *J. ACM*, vol. 23, no. 1, p. 31–42, jan 1976. [Online]. Available: https://doi.org/10.1145/321921.321925

[40] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[41] A. Jüttner and P. Madarasi, "VF2++—An improved subgraph isomorphism algorithm," *Discrete Applied Mathematics*, vol. 242, pp. 69–81, 2018.

[42] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 804–818, 2017.

[43] V. Carletti, P. Foggia, A. Greco, A. Saggese, and M. Vento, "The VF3-light subgraph isomorphism algorithm: when doing less is more effective," in *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshop, S+ SSPR 2018, Beijing, China, August 17–19, 2018, Proceedings 9*. Springer, 2018, pp. 315–325.

[44] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[45] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[46] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[47] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1199–1214.

[48] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1447–1462.

[49] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An In-Depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases," *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 133–144, 2012.

[50] T. Ma, S. Yu, J. Cao, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan, "A comparative study of subgraph matching isomorphic methods in social networks," *IEEE Access*, vol. 6, pp. 66 621–66 631, 2018.

[51] C. Solnon, "Experimental evaluation of subgraph isomorphism solvers," in *Graph-Based Representations in Pattern Recognition: 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19–21, 2019, Proceedings 12*. Springer, 2019, pp. 1–13.

[52] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1083–1098.

[53] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee, "PGX.ISO: parallel and efficient in-memory engine for subgraph isomorphism," in *Proceedings of Workshop on Graph Data management Experiences and Systems*, 2014, pp. 1–6.

[54] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, p. 788–799, may 2012. [Online]. Available: https://doi.org/10.14778/2311906.2311907

[55] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1695–1698.

[56] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble, "Sequential and parallel solution-biased search for subgraph algorithms," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2019, pp. 20–38.

[57] C. McCreesh and P. Prosser, "A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs," in *International conference on principles and practice of constraint programming*. Springer, 2015, pp. 295–312.

[58] V. Carletti, P. Foggia, P. Ritrovato, M. Vento, and V. Vigilante, "A parallel algorithm for subgraph isomorphism," in *Graph-Based Representations in Pattern Recognition: 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19–21, 2019, Proceedings 12*. Springer, 2019, pp. 141–151.

[59] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Database Systems for Advanced Applications*, M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, Eds. Cham: Springer International Publishing, 2015, pp. 299–315.

[60] X. Sun and Q. Luo, "Efficient GPU-Accelerated Subgraph Matching," *Proc. ACM Manag. Data*, vol. 1, no. 2, jun 2023. [Online]. Available: https://doi.org/10.1145/3589326

[61] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "GPU-Accelerated Subgraph Enumeration on Partitioned Graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1067–1082. [Online]. Available: https://doi.org/10.1145/3318464.3389699

[62] L. Zeng, L. Zou, and M. T. Özsu, "SGSI–A Scalable GPU-friendly Subgraph Isomorphism Algorithm," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[63] A. Hagberg and D. Conway, "NetworkX: Network analysis with python," *URL: https://networkx.github.io*, 2020.