An Algorithm for the Parallelization of Large-Scale Attack and Compliance Graph Generation Using Message-Passing Interface

NOAH L. SCHRICK *Member, IEEE,*, AND PETER J. HAWRYLAK, *Senior Member, IEEE,*

***Abstract*—Large-scale attack and compliance graphs can be used for detecting, preventing, and correcting cybersecurity or compliance violations with a system or set of systems. However, as modern-day networks expand in size, and as the number of possible exploits and regulation mandates increase, large-scale attack and compliance graphs can seldom be feasibly generated through serial means. This work presents a parallelized generation process that leverages Message-Passing Interface (MPI) for distributed computing. A task parallelism approach was implemented that includes compatibility for a hybrid MPI-OpenMPI graph generation. This approach was deployed on a High-Performance Computing (HPC) system where a large amount of performance data was collected to capture and conduct a comprehensive analysis on the approach. This work discusses the need for this approach, describes the design process and experimental setup, and illustrates the success that was obtained through speedup and efficiency metrics.**

***Index Terms*—Attack Graph; Compliance Graph; MPI; High-Performance Computing; Cybersecurity; Compliance and Regulation; Speedup; Parallelism;**

I. Introduction

As the size of computer networks continues to grow, cybersecurity analysts are tasked to mitigate risk with increasing difficulty. The authors of [1], [2], and [3] discuss how the rapidly expanding network sizes bring about drastic changes along with the requirement to shift and refocus to accommodate the expansion. This includes presenting novel architectures to support the ever-growing IPTV networks, examinations of computer viruses through epidemiology modeling, and evaluations of new routing schemes. In recent years, a greater usage of cyber-physical systems and a growing adoption of the Internet of Things (IoT) also contributes to an increased need for risk mitigation across varying types of networks, as discussed by the authors of [4], [5], and [6]. One approach for analyzing the large number of hosts (assets) and growing lists of exploits is to automate the generation of attack or compliance graphs for later use. Attack and compliance graphs are directed acyclic graphs (DAGs) that typically represent one or many systems as nodes in a graph, and any changes that could be made to them as edges. The automation of these graphs has been used and presented by authors such as [7], [8], and [9]. The graph generators will take system information and exploits to check for as input, and will exhaustively draw all possible ways that the systems may be at risk of a cybersecurity attack or at risk of violating a compliance regulation or mandate. If a system is able to be modified through a setting change (regardless of intent), have its compliance standing altered, or have a policy updated, an edge is drawn from that node to a new node with the changed system properties. This process is repeated until all possible alterations are identified and represented in the resulting attack or compliance graph.

The authors are affiliated with the Tandy School of Computer Science, Department of Computer Science, University of Tulsa, Tulsa, OK 74104 USA (e-mail: noah-schrick@utulsa.edu, peter-hawrylak@utulsa.edu).

Due to the expansion in network size, and with the inclusion of IoT and cyber-physical devices, the generation of attack and compliance graph quickly becomes difficult with the large number of assets needed to be processed. In addition, the number of regulatory and compliance checks, the large number of exploit and vulnerability entries available, and any custom internal standard checks or zero-day scripting causes a state space explosion in the graph generation process. As a result, real-world graphs become infeasible to generate and process serially.

The attack and compliance graph generation is a viable process to parallelize and deploy on High-Performance Computing (HPC) environments, and related parallel and speedup works are discussed in Section II. This work presents an extension to RAGE (RAGE Attack Graph Engine) [10] to function on distributed computing environments to take advantage of the increased computing power using message-passing. As mentioned by the author of [11], MPI is a widely used message-passing API, and one goal of this work was to utilize an API that was not only familiar and accessible, but versatile and powerful for parallelizing RAGE for distributed computing platforms. This work discusses a task parallelism approach for the generation process, and uses OpenMPI for the MPI implementation.

II. Related Works

For architectural and hardware techniques for general graph generation improvement, the authors of [12] discuss the high cache miss rate, and how general prefetching does not increase the prediction rate due to nonsequential graph structures and data-dependent access patterns. However, the authors continue to discuss that generation algorithms are known in advance, so explicit tuning of the hardware prefetcher to follow the traversal order pattern can lead to better performance. The authors were able to achieve over 2x performance improvement of a breadth-first search approach with this method. Another hardware approach is to make use of accelerators. The authors of [13] present an approach for minimizing the slowdown caused by the underlying graph atomic functions. By using the atomic function patterns, the authors utilized pipeline stages where vertex updates can be processed in parallel dynamically. Other works, such as those by the authors of [14] and [15], leverage field-programmable gate arrays (FPGAs) for graph generation in the HPC space through various means. This includes reducing memory strain, storing repeatedly accessed lists, storing results, or other storage through the on-chip block RAM, or even leveraging Hybrid Memory Cubes for optimizing parallel access.

From a data structure standpoint, the authors of [16] describe the infeasibility of adjacency matrices in large-scale graphs, and this work and other works such as those by the authors of [17] and [18] discuss the appeal of distributing a graph representation across systems. The author of [18] discusses the usage of distributed adjacency lists for assigning vertices to workers. The authors of [18] and [19] present other techniques for minimizing communication costs by achieving high compression ratios while maintaining a low compression cost. The Boost Graph Library and the Parallel Boost Graph Library both provide appealing features for working with graphs, with the latter library notably having interoperability with MPI, Graphviz, and METIS [20], [21].

There have also been numerous approaches at generation improvement specific to attack graphs. As a means of improving scalability of attack graphs, the authors of [7] present a new representation scheme. Traditional attack graphs encode the entire network at each state, but the representation presented by the authors uses logical statements to represent a portion of the network at each node. This is called a logical attack graph. This approach led to the reduction of the generation process to quadratic time and reduced the number of nodes in the resulting graph to $\mathcal{O}(n^2)$. However, this approach does require more analysis for identifying attack vectors. Another approach presented by the authors of [22] represents a description of systems and their qualities and topologies as a state, with a queue of unexplored states. This work was continued by the authors of [23] by implementing a hash table among other features. Each of these works demonstrates an improvement in scalability through refining the desirable information output.

Another approach for generation improvement is through parallelization. The authors of [23] leverage OpenMP to parallelize the exploration of a FIFO queue. This parallelization also includes the utilization of OpenMP's dynamic scheduling. In this approach, each thread receives a state to explore, where a critical section is employed to handle the atomic functions of merging new state information while avoiding collisions, race conditions, or stale data usage. The authors measured a 10x speedup over the serial algorithm. The authors of [24] present a parallel generation approach using CUDA, where speedup is obtained through a large number of CUDA cores. For a distributed approach, the authors of [25] present a technique for utilizing reachability hyper-graph partitioning and a virtual shared memory abstraction to prevent duplicate work by multiple nodes. This work had promising results in terms of speedup and in limiting the state-space explosion as the number of network hosts increases.

III. Necessary Components

A. Serialization

In order to distribute workloads across nodes in a distributed system, various types of data will need to be sent and received. Support and mechanisms vary based on the MPI implementation, but most fundamental data types such as integers, doubles, characters, and Booleans are incorporated into the MPI implementation. While this does simplify some of the messages that need to be sent and received in the MPI approaches of attack and compliance graph generation, it does not cover the vast majority of them when using RAGE.

RAGE implements many custom classes and structs that are used throughout the generation process. Qualities, topologies, network states, and exploits are a few such examples. Rather than breaking each of these down into fundamental types manually, serialization functions are leveraged to handle most of this. RAGE already incorporates Boost graph libraries for auxiliary support, so this work extended this further to utilize the serialization libraries also provided by Boost. These libraries also include support for serializing all STL classes, and many of the RAGE classes have members that make use of the STL classes. One additional advantage of the Boost library approach is that many of the RAGE classes are nested. For example, the NetworkState class has a member vector of Quality classes, and the Quality class has a Keyvalue class as a member. When serializing the NetworkState class, Boost will recursively serialize all members, including the custom class members, assuming they also have serialization functions.

When using the serialization libraries, this work opted to use the intrusive route, where the class instances are altered directly. This was preferable to the non-intrusive approach, since the class instances were able to be altered with relative ease, and many of the class instances did not expose enough information for the non-intrusive approach to be viable.

IV. Implementation of the Tasking Approach

The high-level overview of the attack and compliance graph generation process can be broken down into six main tasks. These tasks are described in Figure 1. Prior works such as that seen by the authors of [23], [24], and [25] work to parallelize the graph generation using OpenMP, CUDA, and hyper-graph partitioning. This approach, however, utilizes Message Passing Interface (MPI) to distribute the six identified tasks of RAGE to examine the effect on speedup, efficiency, and scalability for attack and compliance graph generation.

A. Algorithm Design

The design of the tasking approach is to leverage a pipeline structure with the six tasks and MPI nodes. After its completion, each stage of the pipeline will pass the necessary data to the next stage through various MPI messages, where the next stage's nodes will receive the data and execute their tasks. The pipeline is considered fully saturated when each task has a dedicated node solely for executing work for that

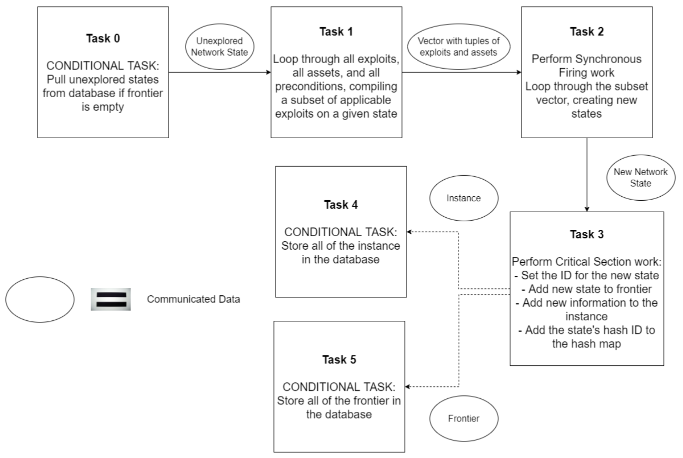

Task 0
CONDITIONAL TASK: Pull unexplored states from database if frontier is empty
Unexplored Network State
Task 1
Loop through all exploits, all assets, and all preconditions, compiling a subset of applicable exploits on a given state
Vector with tuples of exploits and assets
Task 2
Perform Synchronous Firing work
Loop through the subset vector, creating new states
New Network State
Task 3
Perform Critical Section work:
- Set the ID for the new state
- Add new state to frontier
- Add new information to the instance
- Add the state's hash ID to the hash map
Instance
Task 4
CONDITIONAL TASK: Store all of the instance in the database
Task 5
CONDITIONAL TASK: Store all of the frontier in the database
Frontier
Communicated Data

Fig. 1. Overview of the Tasking Pipeline for the Attack and Compliance Graph Generation Process

Task	Node Rank(s) Allocated
0	0
1	$[1, n_1]$
2	$[(n_1 + 1), n_2]$
3	0
4	n_3
5	n_4

$$n_1 = \begin{cases} 1, & world.size() \leq num\_tasks \\ 1 + \lceil \frac{world.size() - num\_tasks}{2} \rceil, & otherwise \end{cases}$$

$$n_2 = \begin{cases} 2n_1, & world.size()\%2 = 0 \\ 2n_1 - 1, & otherwise \end{cases}$$

$$n_3 = n_2 + 1$$

$$n_4 = n_3 + 1$$

Fig. 2. Node Allocation for each Task

task. When there are less nodes than tasks, some nodes will process multiple tasks. When there are more nodes than tasks, additional nodes will be assigned to Tasks 1 and 2. Timings were collected in the serial approach for various networks that displayed more time requirements for Tasks 1 and 2, with larger network sizes requiring vastly more time to be taken in Tasks 1 and 2. As a result, additional nodes are assigned to Tasks 1 and 2. Node allocation can be seen in Figure 2; where "world.size()" is an integer value representing the number of nodes used in the program, and "num_tasks" is an integer value representing the number of tasks used in the pipeline. By using a variable for the number of tasks, it allows for modular usage of the pipeline, where tasks can be added and removed without needing to change any allocation logic work; only communication between tasks may need to be modified, and the allocation can be adjusted relatively simply to include new tasks.

For determining which tasks should be handled by the root note, a few considerations were made, where minimizing communication cost and avoiding unnecessary complexity were the main two considerations. In the serial approach, the frontier queue was the primary data structure for the majority of the generation process. Rather than using a distributed queue or passing multiple sub-queues between nodes, the minimum cost option is to pass states individually. This approach also assists in reducing the complexity. Managing multiple frontier queues would require duplication checks, multiple nodes requesting data from and storing data into the database, and devising a strategy to maintain proper queue ordering, all of which would also increase the communication cost. As a result, the root node will be dedicated to Tasks 0 and 3.

B. Communication Structure

The underlying communication structure for the tasking approach relies on a pseudo-ring structure. As seen in Figure 2, nodes n_2, n_3, and n_4 are derived from the previous task's greatest node rank. To keep the development abstract, a custom send function checks the world size ("world.size()") before sending. If the rank of the node that would receive a message is greater than the world size and therefore does not exist, the rank would then be "looped around" and corrected to fit within the world size constraints. After the rank correction, the MPI Send function was then invoked with the proper node rank.

C. Task Breakdown

1) Task 0: Task 0 is performed by the root node, and is a conditional task; it is not guaranteed to be executed at each pipeline iteration. Task 0 is only executed when the frontier is empty, but the database still holds unexplored states. This occurs when there are memory constraints, and database storage is performed during execution to offload the demand. Additional detail is discussed in Section VI-C. After the completion of Task 0, the frontier has a state popped, and the root node sends the state to n_1. If the frontier is empty, the root node sends the finalize signal to all nodes.

2) Task 1: Task 1 begins by distributing the workload between nodes based on the local task communicator rank. Rather than splitting the exploit list at the root node and sending sub-lists to each node allocated to Task 1, each node checks its local communicator rank and performs a modulo operation with the number of nodes allocated to determine whether it should proceed with the current iteration of the exploit loop. Since the exploit list is static, each node has the exploit list initialized prior to the generation process, and communication cost can be avoided from sending sub-lists to each node. Each node in Task 1 works to compile a reduced exploit list that is applicable to the current network state. A breakdown of the Task 1 workload distribution can be seen in Figure 3.

Once the computation work of Task 1 is completed, each node must send their compiled applicable exploit list to Task 2. Rather than merging all lists and splitting them back out in Task 2, each node in Task 1 will send an applicable exploit list to at most one node allocated to Task 2. Based on the

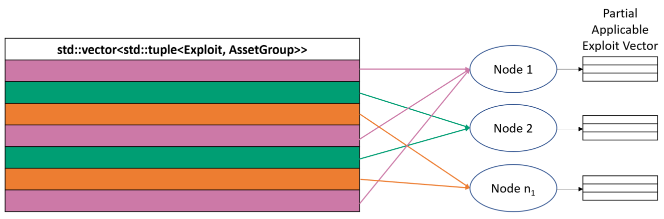

std::vector<std::tuple<Exploit, AssetGroup>>
Node 1
Node 2
Node n_1
Partial Applicable Exploit Vector

Fig. 3. Data Distribution of Task One

allocation of nodes seen in Figure 2, there are 2 potential cases: the number of nodes allocated to Task 1 is equal to the number of nodes allocated to Task 2, or the number of nodes allocated to Task 1 is one greater than the number of nodes allocated to Task 2. For the first case, each node in Task 1 sends the applicable exploit list to its global rank+n_1. This case can be seen in Figure 4. For the second case, since there are more nodes allocated to Task 1 than Task 2, node n_1 scatters its partial applicable exploit list in the local Task 1 communicator, and all other Task 1 nodes follow the same pattern seen in the first case. This second case can be seen in Figure 5.

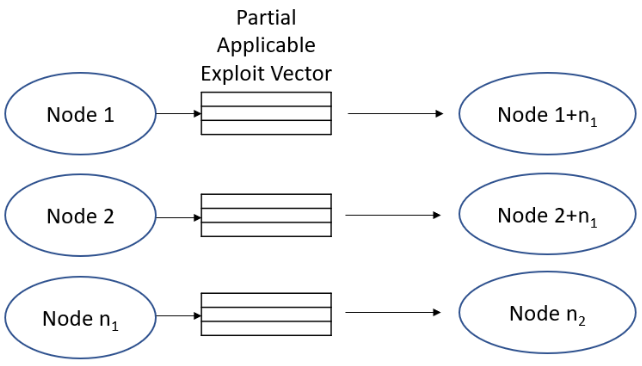


Fig. 4. Communication From Task 1 to Task 2 when the Number of Nodes Allocated is Equal

3) Task 2: Each node in Task 2 iterates through the received partial applicable exploit list and creates new states with edges to the current state. Part of Task 2's workload is to handle a feature called synchronous firing. This feature allows for a grouping of assets. Rather than an exploit firing separately across all assets, synchronous firing allows for an exploit to fire one time, simultaneously across a group of assets. Syncing multiple exploits that could be distributed across multiple nodes leads to additional overhead and complexity. To prevent these difficulties, each node checks its partial applicable exploit list for exploits that are part of a group, removes these exploits from its list, and sends the exploits belonging to a group to the Task 2 local communicator root. Since the Task 2 local root now contains all group exploits, it can execute the synchronous firing work without additional

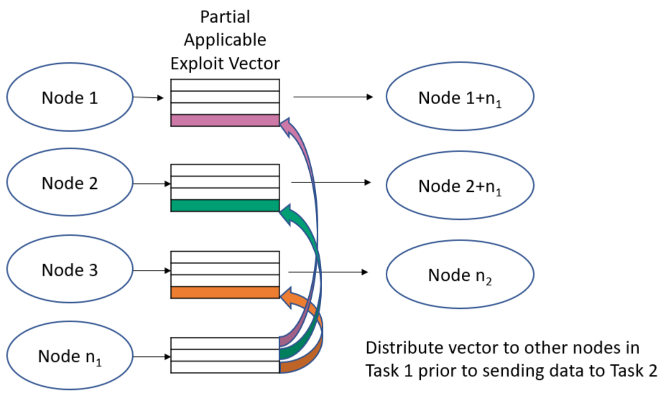

Partial Applicable Exploit Vector
Node 1
Node 2
Node 3
Node n_1
Node 1+n_1
Node 2+n_1
Node n_2
Distribute vector to other nodes in Task 1 prior to sending data to Task 2

Fig. 5. Communication From Task 1 to Task 2 when Task 1 Has More Nodes Allocated

communication or synchronization between other MPI nodes in the Task 2 stage. Other than the additional setup steps required for synchronous firing for the local root, all work performed during this task by all MPI nodes is that seen from the synchronous firing figure (Figure 6).

4) Task 3: Task 3 is performed only by the root node, and no division of work is necessary. The root node will continuously check for new states until the Task 2 finalize signal is detected. This task consists of setting the new state's ID, adding it to the frontier, adding its information to the instance, and inserting information into the hash map. When the root node has processed all states and has received the Task 2 finalize signal, it will complete Task 3 by sending the instance and/or frontier to Task 4 and/or 5, respectively if applicable, then proceed to Task 0.

5) Task 4 and Task 5: Intermediate database operations, though not frequent and may never occur for small graphs, are lengthy and time-consuming when they do occur. As discussed in Section VI-C, the two main memory consumers are the frontier and the instance, both of which are contained by the root node's memory. Since the database storage requests are blocking, the pipeline would halt for a lengthy period of time while waiting for the root node to finish potentially two large storages. Tasks 4 and 5 work to alleviate the stall by executing independently of the regular pipeline execution flow. Since Tasks 4 and 5 do not send any data, no other tasks must wait for these tasks to complete. The root node can then asynchronously send the frontier and instance to the appropriate nodes as needed, clear its memory, and continue execution without delay. After initial testing, it was determined that the communication cost of the asynchronous sending of data for Tasks 4 and 5 is less than the time requirement of a database storage operation if performed by the root node.

D. MPI Tags

To ensure that the intended message is received by each node, the MPI message envelopes have their tag fields specified. When a node sends a message, it specifies a tag that corresponds with the data and intent for which it is sent.

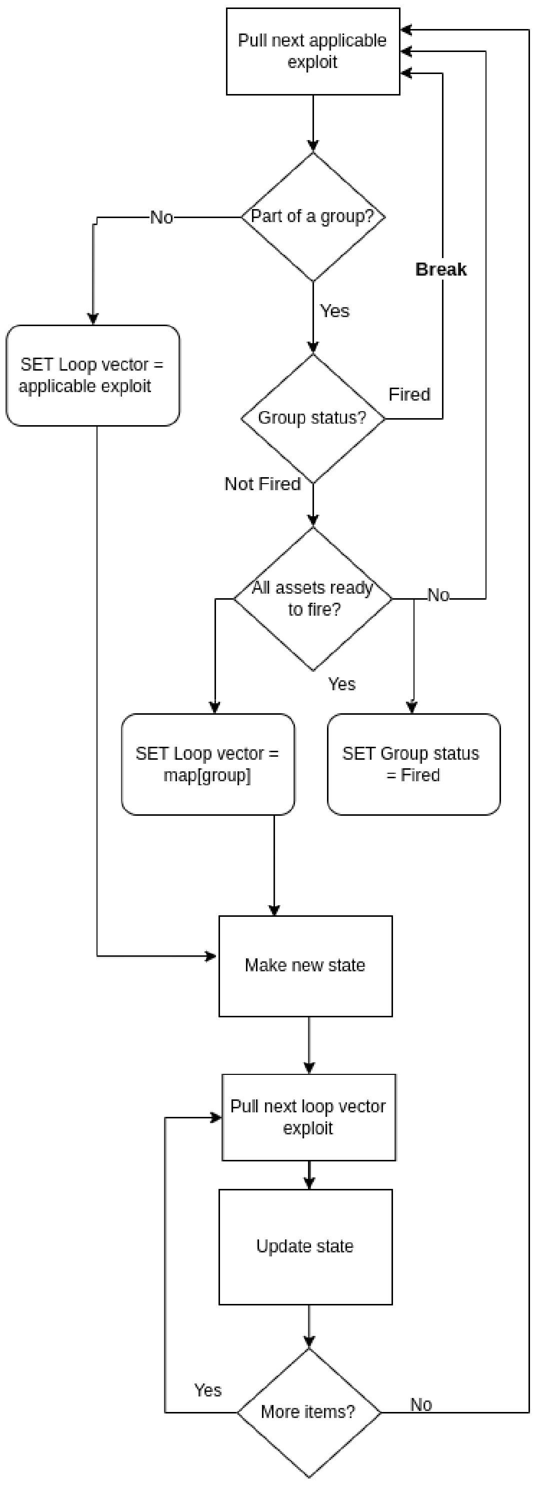

Pull next applicable exploit
Part of a group?
No
Yes
Break
SET Loop vector = applicable exploit
Group status?
Fired
Not Fired
All assets ready to fire?
No
Yes
SET Loop vector = map[group]
SET Group status = Fired
Make new state
Pull next loop vector exploit
Update state
Yes
More items?
No

Fig. 6. Program Flow of Synchronous Firing in Task 2

Tag	Description
2	Task 2 Finalize Signal
3	Fact for Hash Map Update
4	NetworkState for Hash Map Update
5	NetworkState to be Added to the Frontier
6	Current NetworkState Reference for Edge Creation
7	Factbases for Task 4
8	Edges for Task 4
9	Group Exploit Vectors for Local Root in Task 2
10	Exploit Reference for Task 3 Work
11	AssetGroup Reference for Task 3 Work
14	Continue Signal
15	Finalize Signal
20	Current NetworkState Reference for Task 1
21	Applicable Exploit Vector Scatter for Task 1 Case 2
30	Applicable Exploit Vector Send to Task 2
40	NetworkState Send to Task 2
50	NetworkState to Store in Task 5

TABLE I
MPI TAGS FOR THE MPI TASKING APPROACH

The tag values were arbitrarily chosen, and tags can be added to the existing list or removed as desired. When receiving a message, a node can specify to only look for messages that have an envelope with a matching tag field. Not only do tags ensure that nodes are receiving the correct messages, they also reduce complexity for program design. Table I displays the list of tags used for the MPI Tasking approach.

V. PERFORMANCE EXPECTATIONS AND USE CASES

Due to the amount of communication between nodes to distribute the necessary data through all stages of the tasking pipeline, this approach is not expected to outperform the serial approach in all cases. This tasking approach was specifically designed to reduce the computation time when the generation of each individual state increases in time. This approach does not offer any guarantees of processing through the frontier at an increased rate; it's main objective is to distribute the workload of individual state generation. As discussed in Section I, the amount of entries in the National Vulnerability database and any custom vulnerability testing to ensure adequate examination of all assets in the network sums to large number of exploits in the exploit list. Likewise for compliance graphs and compliance examinations, Section I also discussed that the number of compliance checks for SOX, HIPAA, GDPR, PCI DSS, and/or any other regulatory compliance also sums to a large number of compliance checks in the exploit list. Since the generation of each state is largely dependent on the number of exploits present in the exploit list, this approach is best-suited for when the exploit list grows in size. As will be later discussed, it is also hypothesized that this approach is well-suited when many database operations occur.

VI. EXPERIMENTAL SETUP

In order to capture a comprehensive image of the tasking approach's impact on performance, a number of parameters were altered and the generation properties were examined. Table II presents each task and the parameters that affect the performance of each task. Generating larger graphs

Task	Shortened Description	Performance Affected By
0	Retrieve Next State	Database Load
1	Compile List of Applicable Exploits	Number of Exploits
2	Loop through List of Applicable Exploits	Number of Applicable Exploits
3	Bookkeeping	Number of States
4	C/R and/or memory clear of graph instance	Database Load
5	C/R and/or memory clear of frontier	Database Load

TABLE II
TASK DESCRIPTIONS AND PERFORMANCE NOTES

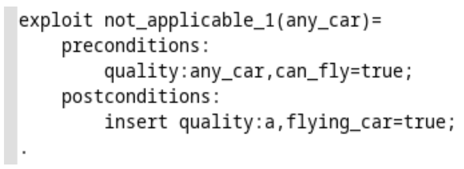
would increase the runtime, but does not necessarily stress each task or provide a consistent, reliable way to draw conclusions regarding the tasking approach. In order to ensure consistency across the experimental testing and minimize the possibility of introducing bias, all tests generated the exact same graph. All tests would generate the same graph with identical numbers of states, identical numbers of edges, identical labeling, and identical inner workings and underlying properties. The following subsections describe the altered parameters, the manner in which they were altered, and how data integrity of the resulting graph was preserved. The parameter alteration process focused on avoiding artificial inflation of the performance metrics, and each subsection emphasizes the practicality of each altered parameter.

A. Number of Exploits

Task 1 loops through the number of exploits and checks each exploit against the list of assets to see if an exploit is applicable at the current state. As the number of exploits grows, the time taken for Task 1 will increase accordingly. The exploit list used by Task 1 does not need to be applicable to the current asset or state, or even to any asset or any state. Regardless of if the exploit is applicable or not, Task 1 still loops through the entirety of the exploit list to check if any exploit may be applicable. Therefore, to prevent state-space explosion but still gather valid results, each exploit list in the tests contained a set of exploits that could be applicable, and all remaining exploits were not applicable. The not applicable exploits were created in a fashion similar to that seen in Figure 7. By creating a multitude of not applicable exploits, the exploit list is able to be artificially increased, which ensures that the resulting graph maintains the same number of states, edges, and identical properties. For the experimental setup, the original exploit list begins with a size of 6, and artificially doubles in size until a final set of graphs is generated using an exploit size of 49,152 exploits. A Python script was used to generate the exploit lists.

B. Applicability of Exploits

When the number of exploits is artificially increased, the runtime for the overall generation process also increases. However, solely increasing the number of exploits adds a strain on only Task 1; Tasks 0, 2, 3, 4, and 5 are not adequately stress tested through the number of exploits alone. As a result,



```
exploit not_applicable_1(any_car)=
    preconditions:
        quality:any_car,can_fly=true;
    postconditions:
        insert quality:a,flying_car=true;
.
```

Fig. 7. Example of a Not Applicable Exploit for the MPI Tasking Testing

additional parameters will need to be altered to capture a thorough image of the tasking performance.

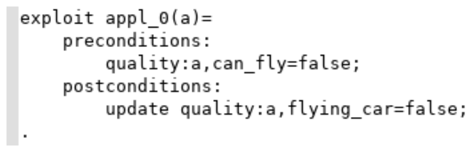
One parameter that can be carefully altered without affecting the resulting graph is the applicability of exploits. As the number of exploits applicable to any state grows, the runtime for Task 2 similarly increases since it must process all applicable exploits and generate new states and edges from the current state. In order for an exploit to be applicable and to not change the resulting graph, the exploit needs to have a precondition that is universally true, with a postcondition that has no effect. For the automobile example, an alteration to the "not applicable" exploit seen in Figure 7 can be performed. The new, artificially applicable exploit can be seen in Figure 8. These artificial exploits will be applicable for any asset at any state in the test network, since no car in this example will ever posses a quality that allows it to fly. Likewise, though the exploit will be processed, the postcondition updates the car quality to match the quality it already contains ("flying_car=false" is instantiated in the input network model). The update keyword in the postcondition still triggers the update function, even if no change is actually made. By updating the car quality in this manner, it is ensured that no change to the resulting graph is made, while still gathering accurate timing data and not skipping any functions called in Task 2.

In RAGE, when an applicable exploit is processed, a new state is always created. The new state is hashed, and its hash is compared to the known hashes. If the hash already exists, the state is discarded and program flow continues. If the hash had not been seen, then the state is added to the instance and frontier. Due to this behavior, it is ensured that the approach for the artificially applicable exploits can capture realistic performance data. In the case of the artificially applicable exploit, the new state is still created and hashed, timing data is captured, and the new state is then discarded along with its edges.

The applicability of exploits was tested by using percentages of overall exploits, excluding the 6 base exploits. The artificial exploits were generated with a Python script based on the example seen in Figure 8. As an illustration, in the case were there are 12 total exploits, the applicability of exploits tests the performance when a percentage of the total exploits were applicable, following the example shown below:

- 0% (floor(0.00 * (12-6 base exploits)) = 0 exploits)
- 25% (floor(0.25 * (12-6 base exploits)) = 1 exploit)
- 50% (floor(0.50 * (12-6 base exploits)) = 3 exploits)
- 75% (floor(0.75 * (12-6 base exploits)) = 4 exploits)

- 100% (floor(1.00 * (12-6 base exploits)) = 6 exploits)



```
exploit appl_0(a)=
    preconditions:
        quality:a,can_fly=false;
    postconditions:
        update quality:a,flying_car=false;
.
```

Fig. 8. Example of an Artificially Applicable Exploit for the MPI Tasking Testing

C. Database Load

The database load parameter is a parameter passed to RAGE to determine when to offload data. The generation of large-scale attack and compliance graphs often faces challenges with scalability and state space explosion. For these large-scale graphs, as the generation process progresses, the resulting graph and the queue of unexplored states begins to consume too much memory for most systems, and the process either needs to offload the data or run the risk of its process being killed for constraints on memory consumption. Works by the authors of [10], [23], and [26] strive for maximum performance of the generation process, which involves making full use of system memory. Since network operations, database operations, and reading and writing from disks slows the generation, it is often preferred to store all data in memory. As a result, there is a balance between performance and system memory consumption.

RAGE has the option for automatically offloading to a PostgreSQL database based on its memory consumption. The database load parameter can either be a float between 0 and 1, or can be an integer greater than 1. If the parameter is a float, RAGE will automatically offload its graph instance or frontier if the memory consumed by either exceeds a percentage of total system memory corresponding to the float value. If the parameter is an integer greater than 1, RAGE wil automatically offload its graph instance or frontier if the number of items in either is greater than the parameter value.

Since the goal of the stress tests is to generate identical graphs for all tests while still stressing each task, the load parameter was carefully altered. The total number of states generated by RAGE is known in advance since the resulting graph has already been generated, and will be constant for all stress tests. For the automobile example being tested, the total number of states is 394. In order to test the database load, it is preferable to use an integer value for the load parameter rather than a float. By using an integer value, it is possible to specifically target how often the offloading process should occur. If a float value was used, additional work would be needed to give RAGE only a certain amount of system memory per test. Though possible, there is more simplicity with passing in static integer values since the graph is known in advance.

The database load parameter was changed based on percentage of the total resulting graph size, as follows:

- 0% Load (Do not ever write to the database) - DBLoad = 395
- 25% Load (Write to the database when 25% of the total resulting graph size is in memory) - DBLoad = 296
- 50% Load (Write to the database when 50% of the total resulting graph size is in memory) - DBLoad = 197
- 75% Load (Write to the database when 75% of the total resulting graph size is in memory) - DBLoad = 79
- 100% Load (Write to the database on every new state) - DBLoad = 1

The database load parameter stresses Tasks 0, 4, and 5. Task 4 will be stress tested on all load parameters, except for when the load is 0% (size 395), which serves as the control. Task 4 will experience the greatest workload when the load parameter is 100% (size 1), since as soon as new states are discovered in previous tasks, Task 4 will begin. Task 0 and Task 5 will experience stress at the same intervals. When the queue of unexplored states increases to a size greater than the load parameter, Task 5 will empty the queue, and Task 0 will be forced to pull new states from the database.

D. Testing Platform

All data was collected on a 13 node cluster, with 12 nodes serving as dedicated compute nodes, and 1 node serving as the login node. Each compute node has a configuration as follows:

- OS: CentOS release 6.9
- CPU: Two Intel Xeon E5-2620 v3
- Two Intel Xeon Phi Co-Processors
- One FPGA (Nallatech PCIE-385n A7 Altera Stratix V)
- Memory: 64318MiB

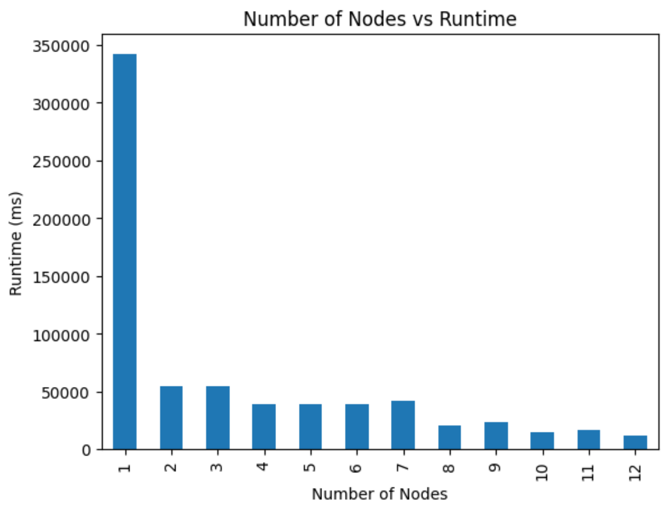
All nodes are connected with a 10Gbps Infiniband interconnect.

E. Testing Process

Each parameter discussed in this section was individually changed until all permutations of parameters were explored. In addition to changing the parameters, all tests were conducted on a varying number of nodes. All permutations of parameters were examined on 1 compute node (serially) through 12 compute nodes. A bash script for looping through parameters was created on the distributed computing testing platform, with jobs sent to Slurm Workload Manager [27]. When a job is completed with Slurm, the bash script would use grep on the output file to extract the necessary data, and add it to a CSV file that was used for the data analysis.

VII. Analysis and Results

Due to a limited amount of compute time, only preliminary results were gathered, instead of deploying the entire testing benchmark suite. For the local University cluster, job allocations had a fixed amount of allowed runtime prior to job scheduler termination. Users were also permitted only a limited amount of concurrent job allocations. With a limited amount of concurrent jobs and a limited amount of job runtime, the full benchmark suite would need to be broken up into several sequences of jobs that would span across a great length of time, since the next jobs could only begin once the previous sequence ended.



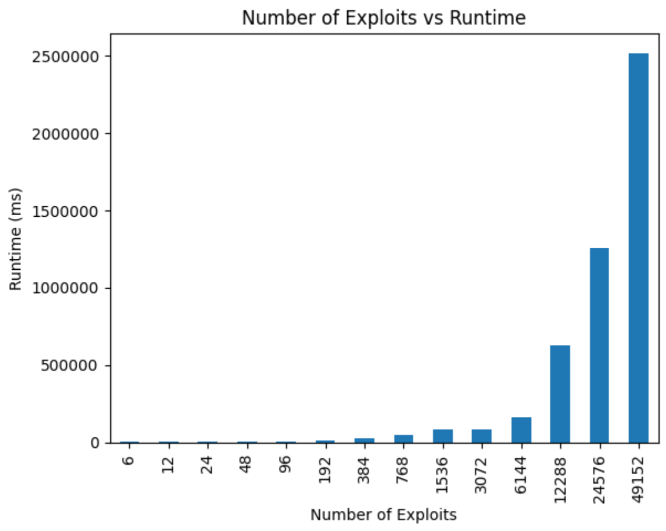
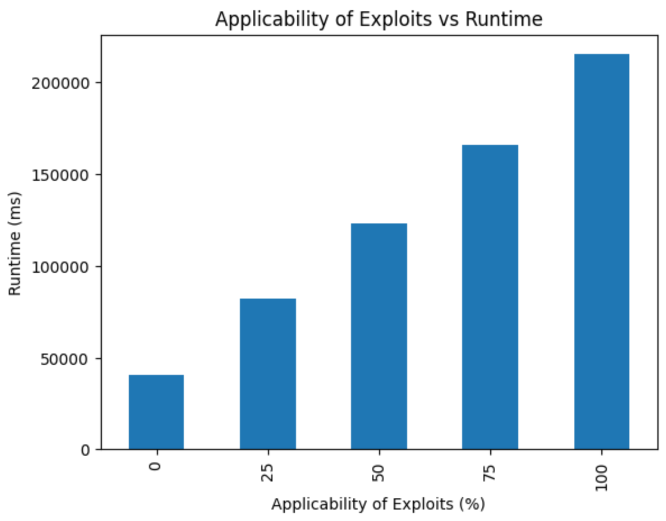


Fig. 9. Number of Nodes and Number of Exploits (Averaged) vs. Runtime (ms), Combining and Averaging Across All Other Parameters



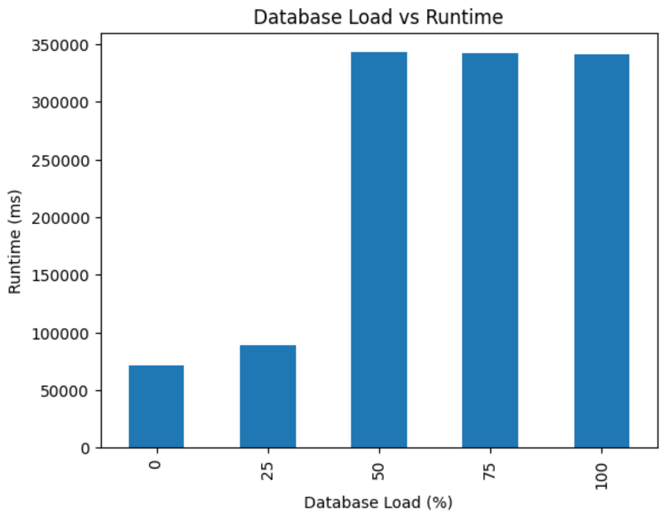


Fig. 10. Applicability of Exploits (%) and Database Load (%) (Averaged) vs. Runtime (ms), Combining and Averaging Across All Other Parameters

The preliminary results gathered were intended to be the "slowest" tests that would have the lowest speedup. The "worst-case", minimum-bound data was collected to determine potential success of this approach. If the slowest tests still yielded promising speedups or efficiencies, then this approach would be viable and appealing for future, in-depth testing that could stress each component of the generation process.

Exploratory data analysis was performed on the resulting data using Python to ascertain data relationships. Due to the multivariate nature of the data, there is difficulty visualizing all four independent variables (parameters) and the outcome (runtime) simultaneously. Using pivot tables, Figures 9 and 10 show the runtime as they relate to the average of each individual parameter. These figures display the expected outcome: as the number of nodes increase, the runtime decreases, and as the number of exploits, applicability of exploits, and database load increases, the runtime likewise increases.

In terms of speedup, when the number of entries in the exploit list is small, the serial approach has better performance. As discussed in Section V, this is expected due to the time elapsed for the communication cost exceeding the time taken to generate a state. However, as the number of items in the exploit list increase, the Tasking Approach quickly begins to outperform the serial approach. It is notable that even when the tasking pipeline is not fully saturated (when there are less compute nodes assigned than tasks), the performance is still approximately equal to that of the serial approach. The other noticeable feature is that as more compute nodes are assigned, the speedup continues to increase.

Figure 11 displays the overall minimum, maximum, and mean of speedup across all problem sizes. All parameters are combined and averaged, which leads to the high-magnitude drops in outcome variables. This effect is made more noticeable since the minimum-bound data was collected, where the large majority of data was collected using only a few nodes. It is observable through the mean and maximum bars that as other problem size parameters increase, the speedup of the Tasking Approach also increases. Since database load, applicability of exploits, and number of exploits all affect the runtime, increasing the problem size through any of these parameters showcases the viability of the parallelized approach. At the same time, it is worth noting that the parallelized approach is not strictly better. The minimum speedups shown in Figure 11 demonstrate that for small problem sizes, the serial approach performs better due to the communication costs.

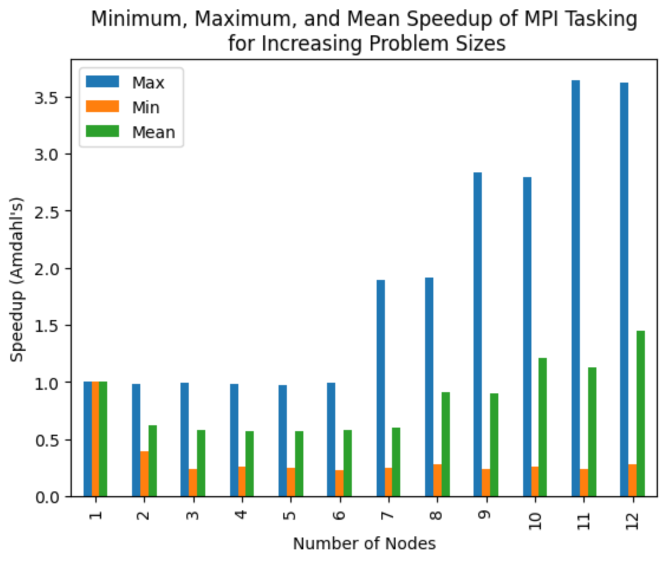


Fig. 11. Minimum, Maximum, and Mean Speedup of MPI Tasking Across All Problem Sizes, Combining and Averaging Across All Parameters

Figure 12 displays the overall minimum, maximum, and mean of efficiency across all problem sizes. All parameters are combined and averaged, which leads to the high-magnitude drop in outcome variables. This effect is made more noticeable since the minimum-bound data was collected, where the large majority of data was collected using only a few nodes. In terms of efficiency, 2 compute nodes offer the greatest value. While the 2 compute node configuration does offer the greatest efficiency, it does not provide a speedup greater than 1.0 on any of the testing cases conducted. The results also demonstrate that an odd number of compute nodes in a fully saturated pipeline has better efficiency that an even number of compute nodes. When referring to Figure 2, when there is an odd number number of compute nodes, Task 1 is allocated more nodes than Task 2. Task 1 was responsible for iterating through an increased size of the exploit list, so more nodes is advantageous in distributing the workload. However, when many exploits were not applicable, Task 2 had a lower workload. Some test cases only had 6 applicable exploits, which is a substantially lower workload for Task 2 compared to cases where Task 1 had upwards of 49,000 exploits. As the applicability of exploits increases, the disparity in efficiency for odd and even number of nodes is not present.

Speedups and efficiencies were also computed across each parameter. Using pivot tables, mean speedups and mean efficiencies were computed for a parameter across all node configurations. Figures 13 and 14 display the speedups and

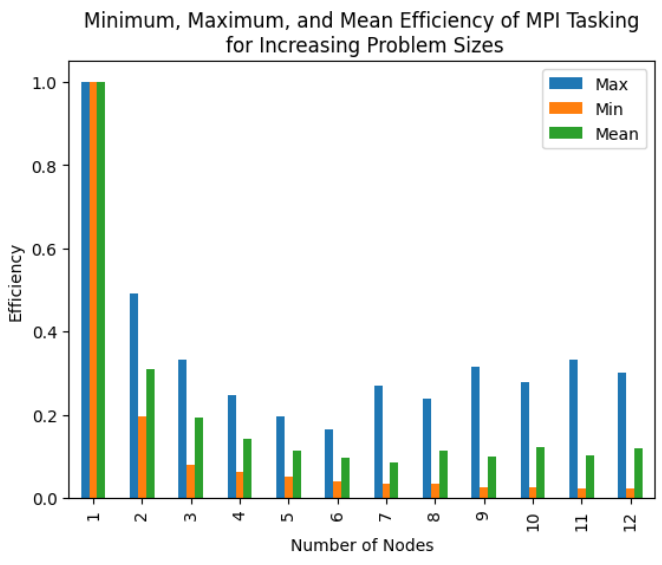


Fig. 12. Minimum, Maximum, and Mean Efficiency of MPI Tasking Across All Problem Sizes, Combining and Averaging Across All Parameters

efficiencies of the exploit parameter and the applicability of exploits parameter, respectively. The number of nodes has the largest impact on the exploit parameter, and Figure 13 illustrates that even when fewer nodes are used, speedup can still be obtained as the exploit list grows in size. Figure 14 demonstrates that though Task 2 has less of an impact on overall runtime and contribution to speedup, speedup is still achievable as more compute nodes are added and as the applicability of exploits increase. Though database load was not a parameter to easily include in preliminary testing, speedup is expected as this parameter changes. By dedicating nodes to solely handle database operations, the tasking pipeline is able to move to new state generation without the need to wait for all preceding database operations to complete.

VIII. Conclusion and Future Work

This work presents a task parallelism approach for large-scale attack and compliance graphs. This approach is a distributed approach rather than a shared-memory approach, allowing for the generation of these large-scale graphs to be deployed on HPC systems. Tasks were identified in the generation process, with parallelization of the tasks clearly identified and incorporated into the tasking algorithm. The results of this approach highlighted its success, showing speedups as generation parameters and the number of nodes increased. Efficiencies were also computed, with various figures illustrating the efficiency across the generation process as a whole, as well as efficiencies across individual parameters.

Though the results presented in this work were preliminary, they still highlight the viability of this approach. Despite each Task having limited stress during the generation process, speedups of over 3.5x can still be obtained. Exploit applicability and database load parameters were

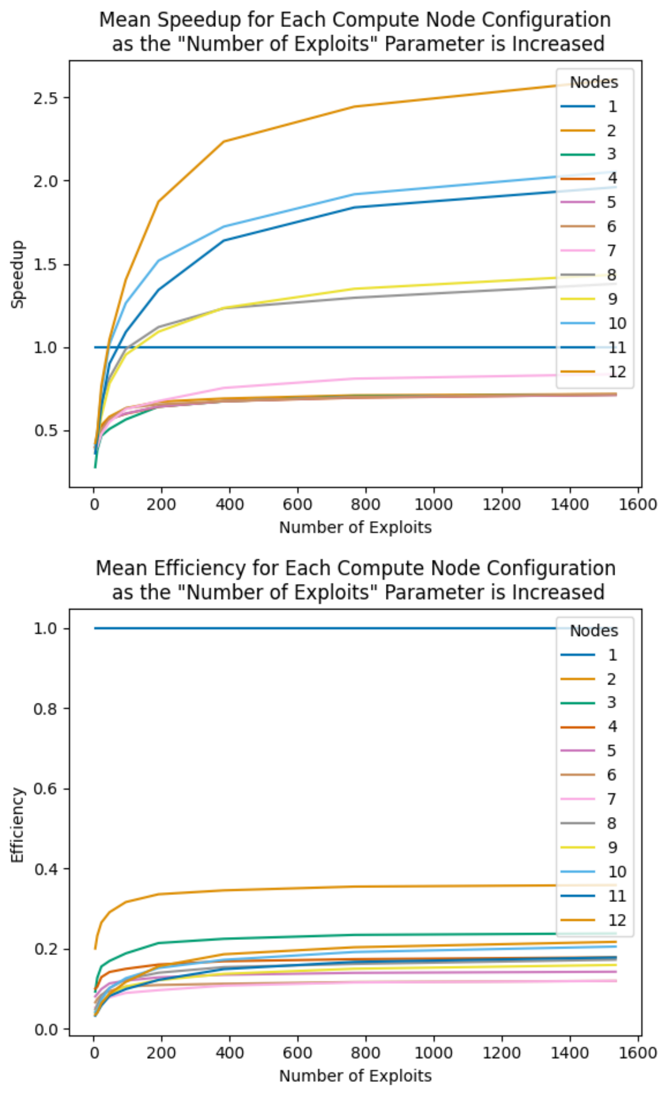


Fig. 13. Mean Speedup and Efficiency for the Exploit Parameter Across the Number of Compute Nodes, Combining and Averaging Across All Other Parameters

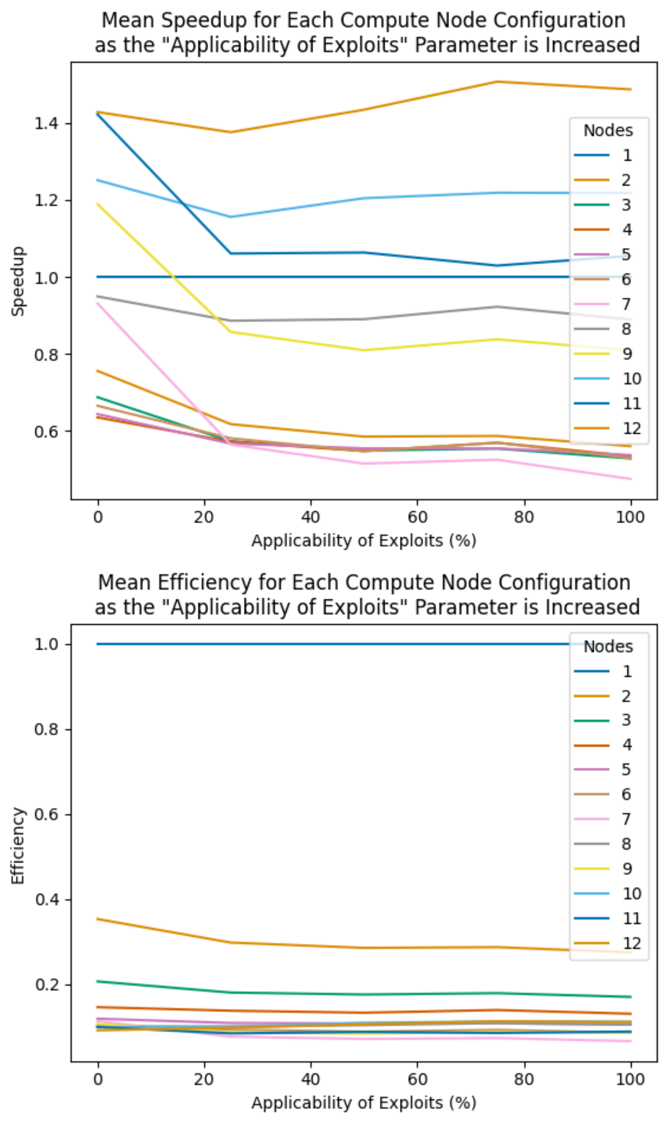


Fig. 14. Mean Speedup and Efficiency for the Applicability of Exploit Parameter Across the Number of Compute Nodes, Combining and Averaging Across All Other Parameters

almost entirely unexplored during the preliminary result benchmarking, and speedup is still achievable. As exploit applicability and database load are introduced into the generation process, both speedup and efficiency are expected to increase in future testing. With more compute time, the approach of this work can be deployed on a testing platform to examine how speedups and efficiencies change as the complexity of the generation process increases.

Future work can be performed to investigate and improve the method of this work. This work focused on a distributed approach to large-scale graph generation, but leaves room for additional parallelism at the node-level. For example, after work for Tasks 1 or 2 has been distributed to a node, the node can then leverage OpenMP for additional parallelism. Results can be obtained to show how the additional parallelism affects the speedup and efficiency of the approach.

Additional work can be performed to investigate long-term speedup and efficiency of the approach. This work made use of a local HPC cluster with a limited number of compute nodes. The generation algorithm can be deployed to larger clusters to measure speedup and efficiency as more nodes are added to the tasking pipeline. Scalability can likewise be reexamined as the number of nodes increases.

The analysis portion of this work also has room for additional investigations. This work measured speedup according to Amdahl [28]. Since tasks are clearly defined and timing data is collected for each task, other speedup metrics can be used. Both Gustafson's Law [29] and Sun and Ni's Law [30] can be leveraged to obtain other results regarding the speedup of the tasking approach.

References

[1] N. Dakhno, O. Leshchenko, Y. Kravchenko, A. Dudnik, O. Trush, and V. Khankishiev, "Dynamic model of the spread of viruses in a computer network using differential equations," in *2021 IEEE 3rd International Conference on Advanced Trends in Information Theory (ATIT)*, pp. 111–115, 2021.

[2] M. Kwon, J. Kwon, B. Park, and H. Park, "An architecture of iptv networks based on network coding," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 462–464, 2017.

[3] X. Bai, M. Liang, and S. Zhu, "A new routing scheme for large-scale computer network," in *2018 14th IEEE International Conference on Signal Processing (ICSP)*, pp. 1019–1023, 2018.

[4] N. Baloyi and P. Kotzé, "Guidelines for Data Privacy Compliance: A Focus on Cyberphysical Systems and Internet of Things," in *SAICSIT '19: Proceedings of the South African Institute of Computer Scientists and Information Technologists 2019*, (Skukuza South Africa), Association for Computing Machinery, 2019.

[5] E. Allman, "Complying with Compliance: Blowing it off is not an option.," *ACM Queue*, vol. 4, no. 7, 2006.

[6] J. Hale, P. Hawrylak, and M. Papa, "Compliance Method for a Cyber-Physical System." U.S. Patent Number 9,471,789, Oct. 18, 2016.

[7] X. Ou, W. F. Boyer, and M. A. Mcqueen, "A Scalable Approach to Attack Graph Generation," *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pp. 336–345, 2006.

[8] A. T. Al Ghazo, M. Ibrahim, H. Ren, and R. Kumar, "A2g2v: Automated attack graph generator and visualizer," in *Proceedings of the 1st ACM MobiHoc Workshop on Mobile IoT Sensing, Security, and Privacy*, Mobile IoT SSP'18, (New York, NY, USA), Association for Computing Machinery, 2018.

[9] M. Li, P. Hawrylak, and J. Hale, "Strategies for practical hybrid attack graph generation and analysis," *Digital Threats*, oct 2021.

[10] K. Cook, "RAGE: The Rage Attack Graph Engine," Master's thesis, The University of Tulsa, 2018.

[11] P. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, print ed., 2011.

[12] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," *Proceedings of the International Conference on Supercomputing*, vol. 01-03-June, 2016.

[13] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 2018.

[14] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search," *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 207–216, 2017.

[15] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on FPGA: A case study of breadth-first search," *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 105–110, 2016.

[16] S. Arifuzzaman and M. Khan, "Fast parallel conversion of edge list to adjacency list for large-scale graphs," in *HPC '15: Proceedings of the Symposium on High Performance Computing*, pp. 17–24, Apr. 2015.

[17] X. Yu, W. Chen, J. Miao, J. Chen, H. Mao, Q. Luo, and L. Gu, "The Construction of Large Graph Data Structures in a Scalable Distributed Message System," in *HPCCT 2018: Proceedings of the 2018 2nd High Performance Computing and Cluster Technologies Conference*, pp. 6–10, June 2018.

[18] P. Liakos, K. Papakonstantinopoulou, and A. Delis, "Memory-Optimized Distributed Graph Processing through Novel Compression Techniques," in *CIKM '16: Proceedings of the 25th ACM International Conference on Information and Knowledge Management*, pp. 2317–2322, Oct. 2016.

[19] J. Balaji and R. Sunderraman, "Graph Topology Abstraction for Distributed Path Queries," in *HPGP '16: Proceedings of the ACM Workshop on High Performance Graph Processing*, pp. 27–34, May 2016.

[20] K. Lab, "Parmetis - parallel graph partitioning and fill-reducing matrix ordering." http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[21] J. Siek, L.-Q. Lee, and A. Lumsdaine, "The boost graph library, vers. 1.75.0." https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/index.html.

[22] K. Cook, T. Shaw, J. Hale, and P. Hawrylak, "Scalable attack graph generation," *Proceedings of the 11th Annual Cyber and Information Security Research Conference, CISRC 2016*, 2016.

[23] M. Li, P. Hawrylak, and J. Hale, "Concurrency Strategies for Attack Graph Generation," *Proceedings - 2019 2nd International Conference on Data Intelligence and Security, ICDIS 2019*, pp. 174–179, 2019.

[24] M. Li, P. J. Hawrylak, and J. Hale, "Implementing an attack graph generator in cuda," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 730–738, 2020.

[25] K. Kaynar and F. Sivrikaya, "Distributed attack graph generation," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 519–532, 2016.

[26] M. Li, P. Hawrylak, and J. Hale, "Combining OpenCL and MPI to support heterogeneous computing on a cluster," *ACM International Conference Proceeding Series*, 2019.

[27] SchedMD, "Slurm Workload Manager." https://slurm.schedmd.com/overview.html, Apr. 2023. Version 23.02.

[28] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), p. 483–485, Association for Computing Machinery, 1967.

[29] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, p. 532–533, may 1988.

[30] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pp. 324–333, 1990.

Noah L. Schrick is a fourth-year PhD student in Computer Science at the University of Tulsa. He received his Bachelor of Science in Electrical and Computer Engineering and his Master of Science in Computer Science at the University of Tulsa.

His research focus is on cybersecurity and compliance, where he is working on the analysis of large-scale attack and compliance graphs to detect, correct, and predict violations in regulations or mandates. He has additional research interests in high-performance computing, research computing, platform engineering, and scientific software development. Noah L. Schrick is a TU-Cyber Fellow at the University of Tulsa, where he focuses on the innovation and growth of industry-applicable research.

Peter J. Hawrylak is an Associate Professor in the Department of Electrical and Computer Engineering, with joint appointments in the Tandy School for Computer Science and the School of Cyber Studies, at The University of Tulsa, Tulsa, OK, USA. He has published more than 60 publications and holds 15 patents in the radio frequency identification (RFID), energy harvesting, and cyber-security areas. His research interests include RFID, security for low-power wireless devices, Internet of Things applications, critical infrastructure security, high-performance computing, and digital design. Dr. Hawrylak's research has been supported by NASA, DOD, the U.S. Army, DOE, Argonne National Laboratory, DOT, EPA, CDC, NSF, and OCAST.

Dr. Hawrylak is a senior member of the IEEE and IEEE Computer Society, has served as Secretary of the Tulsa Section of the IEEE 2015-2018, Vice-Chair of the Tulsa Section of the IEEE 2019-2020, and is currently serving as Chair of the Tulsa Section of the IEEE (2020-Present) leading the Section through the COVID-19 pandemic. He served as chair of the RFID Experts Group (REG) of Association for Automatic Identification and Mobility (AIM) in 2012-2013. Peter received AIM Inc.'s Ted Williams Award in 2015 for his contributions to the RFID industry. Dr. Hawrylak serves on the Organizing Committee of the International IEEE RFID Conference, and served two terms as the Editor-in-Chief of the IEEE RFID Virtual Journal (2016-2019) and also as the Editor-in-Chief of the International Journal of Radio Frequency Identification Technology and Applications (IJRFITA) journal published by InderScience Publishers, which focuses on the application and development of RFID technology.

Peter is a senior member (M'05-SM'17) of IEEE and the IEEE Computer Society, is a member of IEEE-HKN, and is a member of Tau Beta Pi. He has served as the faculty advisor for the IEEE-HKN chapter at The University of Tulsa (Zeta Nu chapter) from Aug. 2010 to Dec. 2020.