

Parallelization of Large-Scale Attack and Compliance Graph Generation Using Message-Passing Interface

Noah L. Schrick
Tandy School of Computer Science
The University of Tulsa
Tulsa, USA
noah-schrick@utulsa.edu

Peter J. Hawrylak
Tandy School of Computer Science
The University of Tulsa
Tulsa, USA
peter-hawrylak@utulsa.edu

Abstract—

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

I. INTRODUCTION

II. RELATED WORKS

Previous works for graph generation, and specifically for attack graph generation, have seen promising results as discussed in Sections ?? and ?. This work attempts to further those efforts and extend RAGE to function on distributed computing environments to take advantage of the increased computing power using message-passing. As mentioned by the author of [1], MPI is the most widely used message-passing API, and this work intended to utilize an API that was not only familiar and accessible, but versatile and powerful for parallelizing RAGE for distributed computing platforms. This Chapter discusses two approaches for parallelism: task parallelism in Section IV, and data parallelism in Section ?. All approaches in this work utilize OpenMPI for the MPI implementation.

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. 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 [2], [3], and [4] 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. When there are less nodes than tasks, some nodes will process multiple tasks. When there are more nodes than tasks,

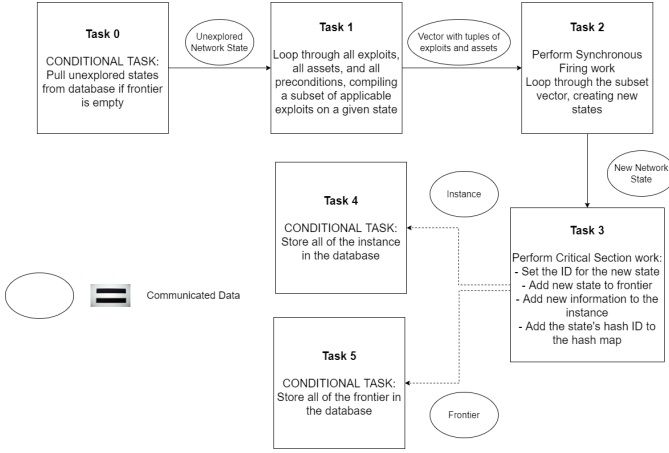


Fig. 1. Task Overview of the Attack and Compliance Graph Generation Process

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. In this Figure, “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 node, 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.

1) 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

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

$$n_1 = \begin{cases} 1, & \text{world.size() } \leq \text{num_tasks} \\ 1 + \lfloor \frac{\text{world.size()} - \text{num_tasks}}{2} \rfloor, & \text{otherwise} \end{cases}$$

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

$$n_3 = n_2 + 1$$

$$n_4 = n_3 + 1$$

Fig. 2. Node Allocation for each Task

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.

2) 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, as discussed in Section ???. 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.

3) 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 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 allocation of nodes seen in Figure 2, there are 2 potential

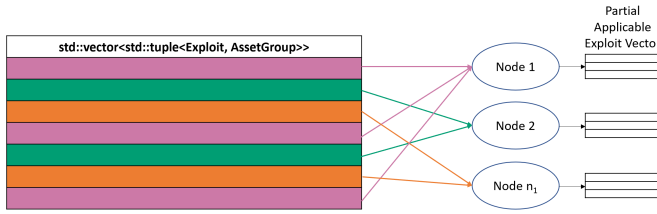


Fig. 3. Data Distribution of Task One

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.

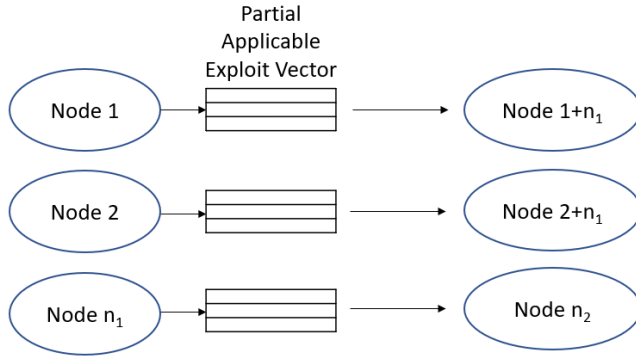


Fig. 4. Communication From Task 1 to Task 2 when the Number of Nodes Allocated is Equal

4) *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. However, synchronous firing work is performed during this task, and 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 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,

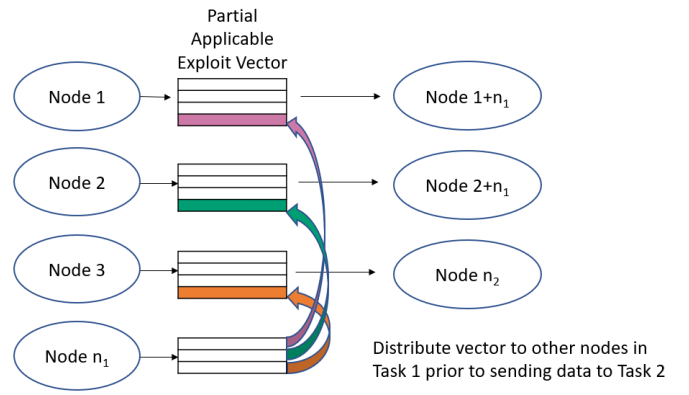


Fig. 5. Communication From Task 1 to Task 2 when Task 1 Has More Nodes Allocated

all work performed during this task by all MPI nodes is that seen from the Synchronous Firing figure (Figure ??).

5) *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.

6) *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 ??, 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.

7) *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. 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

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

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 ??, 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 ?? 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

A. Number of Exploits

B. Database Load

C. Testing Platform

Hammer

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	
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

Looping through MPI Nodes, Number of Exploits, Applicability of Exploits, Database Loads Then compute Speedup with Amdahl and Gufs, then Eff for both

VII. RESULTS

A series of tests were conducted on the platform described at the beginning of Section ??, and results were collected in regards to the effect of the MPI Tasking approach on increasing sizes of exploit lists for a varying number of nodes. The exploit list initially began with 6 items, and each test scaled the number of exploits by a factor of 2. The final test was with an exploit list with 49,512 entries. If all of the items in these exploit lists were applicable, the runtime would be too great for feasible testing due to the state space explosion. To prevent state-space explosion but still gather valid results, each exploit list in the tests contained 6 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 6. By creating a multitude of not applicable exploits, testing can safely be conducted by ensuring state space explosion would not occur while still observing the effectiveness of the tasking approach.

The results of the Tasking Approach can be seen in Figure 7. In terms of speedup, when the number of entries in the exploit list is small, the serial approach has better performance. This is expected due to the communication cost requiring more time than it does to generate a state, as discussed in Section V. 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.

In terms of efficiency, 2 compute nodes offer the greatest value since the speedup using 2 compute nodes is approximately 1.0 as the exploit list size increases. While the 2 compute node option 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 than an even number of compute nodes.

When referring to Figure 2, when there is an odd number of compute nodes, Task 1 is allocated more nodes than Task 2. In the testing conducted, Task 1 was responsible for iterating through an increased size of the exploit list, so more nodes is advantageous in distributing the workload. However, since many exploits were not applicable, Task 2 had a lower workload where only 6 exploits could be applicable. This will be further elaborated upon in Section ??, but it is expected that efficiency will increase for real networks, since nodes in Task 2 will see a realistic workload.

Figures 8, 9, and 10 display the results of the tasking approach for runtime in milliseconds, speedup, and efficiency respectively in table format.

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

Fig. 6. Example of a Not Applicable Exploit for the MPI Tasking Testing

VIII. ANALYSIS

IX. CONCLUSION

ACKNOWLEDGMENT

REFERENCES

REFERENCES

- [1] P. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, print ed., 2011.
- [2] 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.
- [3] 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.
- [4] K. Kaynar and F. Sivrikaya, "Distributed attack graph generation," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 519–532, 2016.

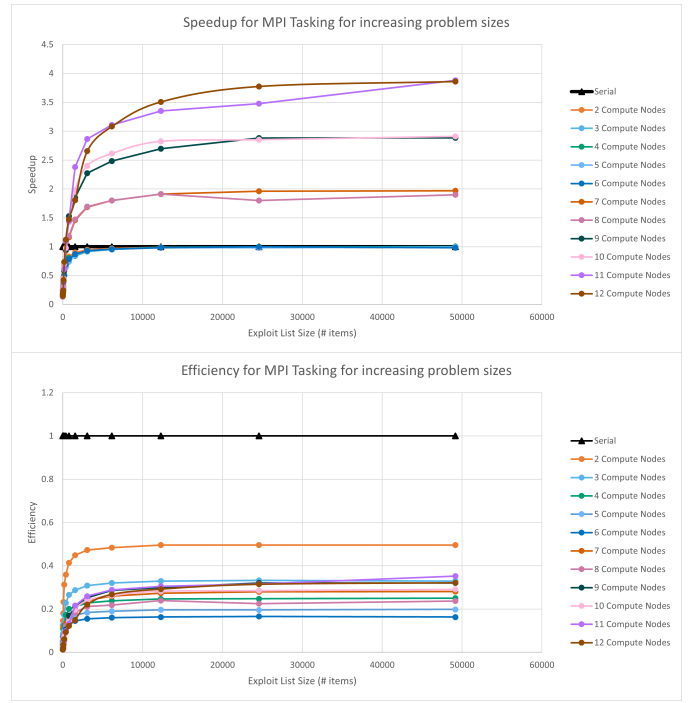


Fig. 7. Speedup and Efficiency of the MPI Tasking Approach for a Varying Number of Compute Nodes with an Increasing Problem Size

Exploit List Size #		Runtime (ms)											
		1	2	3	4	5	6	7	8	9	10	11	12
6	355.02	1596.66	2287.35	2276.78	2225.82	2138.12	2672.84	1992.90	2172.18	1982.38	2586.11	2391.48	
12	419.82	1654.51	1897.94	2345.01	2322.80	2582.81	2529.24	2316.80	2508.23	2410.59	2167.11	2464.40	
24	513.54	1756.40	1892.16	2403.20	2346.26	2365.43	2717.41	2462.34	2542.14	2542.14	2187.01	2519.76	
48	700.21	1946.70	2623.23	2181.01	2457.20	2544.08	2563.71	2078.25	2763.28	2582.10	2696.78	2910.59	
96	1078.65	2307.96	2756.93	2557.62	2612.13	2571.85	2484.83	2597.73	2713.97	2419.33	2615.57	2558.72	
192	1811.31	2895.17	3501.44	3653.90	3584.37	3582.88	3094.95	2635.66	3135.77	2539.62	2988.98	2467.72	
384	3278.96	4560.61	4778.55	4749.58	5069.14	4742.43	3517.16	3375.91	3469.36	3348.37	2996.82	2921.66	
6144	6200.71	7475.54	7769.95	7701.17	8371.15	7886.30	5329.62	5230.78	4055.31	4357.08	4289.35	4216.63	
1536	12042.06	13407.58	13968.34	14293.45	14088.51	13828.52	8264.50	8175.76	6514.99	6090.04	5058.62	6680.81	
3072	52293.52	54362.74	54779.59	54991.61	55058.52	54889.47	28849.18	28824.93	20789.65	19780.57	16633.22	16815.77	
6144	73919.25	74840.32	73185.22	72340.69	75443.52	75369.98	39141.52	39149.52	27926.98	24623.84	22420.52	21883.25	
12288	94544.98	95317.90	95590.85	95689.76	96028.53	96150.43	49433.86	49474.11	35064.31	33471.10	28207.82	26950.73	
24576	187671.44	189204.82	188283.40	188982.20	191532.71	188323.69	95659.96	104213.57	65096.51	65764.78	53931.01	49699.13	
49152	373057.48	375956.92	377786.02	372479.17	374725.03	380541.38	189340.71	196541.05	129281.00	128175.73	96110.95	96614.48	

Fig. 8. Results for the MPI Tasking Approach in Terms of Runtime in Milliseconds

Exploit List Size #		Speedup											
		1	2	3	4	5	6	7	8	9	10	11	12
6	1.00	0.22	0.16	0.16	0.16	0.17	0.13	0.18	0.16	0.18	0.14	0.15	
12	1.00	0.25	0.22	0.18	0.18	0.16	0.17	0.17	0.17	0.17	0.15	0.17	
24	1.00	0.29	0.26	0.21	0.22	0.22	0.19	0.21	0.20	0.25	0.23	0.20	
48	1.00	0.36	0.27	0.32	0.28	0.27	0.27	0.34	0.25	0.27	0.26	0.24	
96	1.00	0.47	0.39	0.42	0.41	0.42	0.45	0.42	0.40	0.45	0.41	0.42	
192	1.00	0.63	0.52	0.50	0.51	0.51	0.59	0.69	0.58	0.71	0.61	0.73	
384	1.00	0.72	0.69	0.69	0.65	0.69	0.93	0.97	0.94	0.98	1.09	1.12	
768	1.00	0.83	0.80	0.81	0.74	0.79	1.16	1.19	1.53	1.42	1.45	1.47	
1536	1.00	0.90	0.86	0.84	0.85	0.87	1.46	1.47	1.85	1.98	2.38	1.80	
3072	1.00	0.95	0.91	0.92	0.92	0.93	1.68	1.69	2.27	2.40	2.87	2.66	
6144	1.00	0.97	0.96	0.95	0.95	0.96	1.80	1.80	2.48	2.61	3.11	3.08	
12288	1.00	0.99	0.99	0.99	0.98	0.98	1.91	1.91	2.70	2.82	3.35	3.51	
24576	1.00	0.99	1.00	0.99	0.98	1.00	1.96	1.80	2.88	2.85	3.48	3.78	
49152	1.00	0.99	0.99	1.00	1.00	0.98	1.97	1.90	2.89	2.91	3.88	3.86	

Fig. 9. Results for the MPI Tasking Approach in Terms of Speedup

Exploit List Size #		Efficiency											
		1	2	3	4	5	6	7	8	9	10	11	12
6	1.00	0.11	0.05	0.04	0.03	0.03	0.03	0.02	0.02	0.02	0.02	0.01	0.01
12	1.00	0.13	0.07	0.04	0.04	0.03	0.03	0.02	0.02	0.02	0.02	0.02	0.01
24	1.00	0.15	0.09	0.05	0.04	0.04	0.03	0.03	0.03	0.03	0.02	0.02	0.01
48	1.00	0.18	0.09	0.08	0.06	0.05	0.04	0.04	0.03	0.03	0.02	0.02	0.01
96	1.00	0.23	0.13	0.11	0.08	0.07	0.06	0.05	0.04	0.04	0.04	0.04	0.04
192	1.00	0.31	0.17	0.12	0.10	0.08	0.08	0.09	0.06	0.07	0.06	0.06	0.06
384	1.00	0.36	0.22	0.17	0.13	0.12	0.12	0.12	0.12	0.10	0.10	0.10	0.09
768	1.00	0.41	0.27	0.20	0.15	0.13	0.17	0.15	0.17	0.14	0.13	0.12	
1536	1.00	0.45	0.29	0.21	0.17	0.15	0.21	0.18	0.21	0.20	0.22	0.15	
3072	1.00	0.47	0.31	0.23	0.18	0.15	0.24	0.21	0.25	0.24	0.26	0.22	
6144	1.00	0.48	0.32	0.24	0.19	0.16	0.26	0.22	0.29	0.26	0.29	0.27	
12288	1.00	0.50	0.33	0.25	0.20	0.16	0.27	0.24	0.30	0.28	0.30	0.29	
24576	1.00	0.50	0.33	0.25	0.20	0.17	0.28	0.23	0.32	0.29	0.32	0.31	
49152	1.00	0.50	0.33	0.25	0.20	0.16	0.28	0.24	0.32	0.29	0.35	0.32	

Fig. 10. Results for the MPI Tasking Approach in Terms of Efficiency