Scaling Peregrine

Project Milestone 6

Yonas Kelemework(DavidPatterson)
Simon Fraser University
ygk1@sfu.ca(301472405)

1. Introduction

Peregrine (Jamshidi et al. 2020) is a pattern-aware single machine graph mining framework that explores sub-graphs of interest, while avoiding unnecessary exploration of sub-graphs. Peregrine provides a pattern based programming models where the user can specify patterns of interests to be explored. Peregrine also introduces Anti-edge and Anti-vertex, special structural constraints on patterns for advanced matching tasks. Peregrine starts exploration tasks at each data vertex and utilizes the generated exploration plan for the pattern being mined. In this project, I used CAF(C++ Actor Framework)(Charousset et al. 2013) and adopted a Client-Server design to scale Peregrine from a single machine framework to a distributed framework. The client sends pattern matching tasks to the server, and compiles results from the servers forming the final output. I also used CAF's native handlers to implement a fault tolerance solution. The implementation is evaluated using widely accepted network datasets from the SNAP group(Leskovec and Krevl 2014).

2. Design and Implementation

The implementation consists of a client process which runs on a local machine and remote(distributed) process running on remote nodes as shown in Figure 1 As shown in Figure 2, for counting, enumerate and existence query application, the client connects to the remote servers and sends the remote data graph directory, the pattern name to be searched, the number of threads per node and starting index of the tasks assigned to each remote server. The remote(distributed) actors receive the data graph name, pattern name to search, number of threads, number of processes and start task id. With the above inputs the remote actors load the data graph,

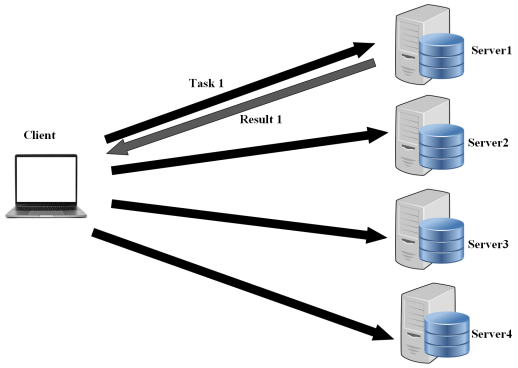

Figure 1. Distributed Model

generate the patterns to be matched or counted and, carry out a multi-threaded search. Each server actor uses the task ID to identify the starting task index, and the number of processes to fetch the next task which will be the **current task + number of processes**. For existence queries if one server

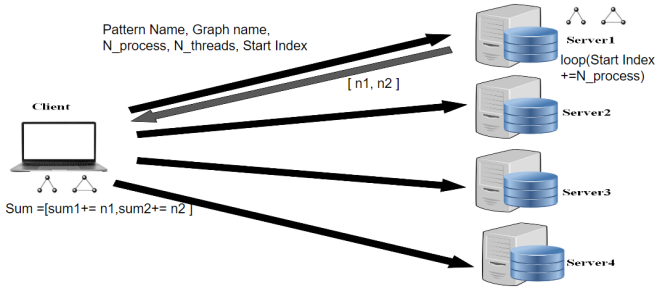

Figure 2. Distributed model for Count, Enumerate and Existence Query Applications

finds the pattern earlier than the others the client will not wait for other servers and returns immediately. As for the servers we can either send them exit messages to shut them down or keep them alive for subsequent requests depending on how we want to run our service. The above task partitioning method although very naive is effective since it maintains the dynamic work balancing feature of the original multi-threaded implementation. To reduce communication overhead for counting, enumeration and existence query applica-

tions the current implementation only sends the pattern name and leaves the pattern generation task to the server to reduce the communication overhead of sending the patterns. The servers returns only a vector of count values which are added up and matched with the patterns generated in the client process to report the final result. FSM follows the same client-

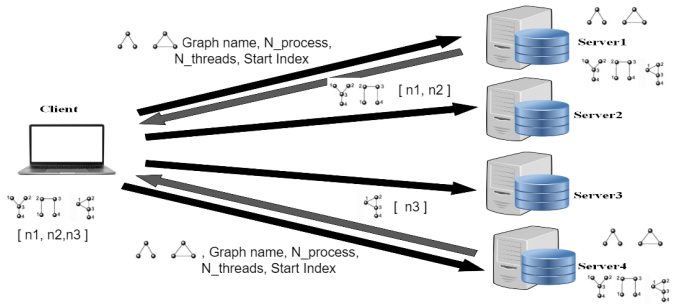

Figure 3. Distributed model for Frequent Subgraph mining(FSM) Application

server model as the above solutions, Figure 3 , however the task mapping method and data transferred between the client and servers differs from the applications above. The applications above divide the task at the granularity of individual vertex tasks. FSM allocates tasks to the servers at the granularity of patterns. For a set of patterns to be explored we divide the patterns among servers. Hence a given pattern is explored and it's mappings aggregated at a single server. This implementation has the cost of not dividing the work evenly, since exploring some pattterns might take longer than others. Initially FSM was designed to distribute tasks based on vertex tasks(similar to counting, enumeration and existence query applications),but the implementation suffered from a correctness bug due to not being able to move the aggegation step from the server side to the client side. In terms of communications the distributed FSM application initially sends a 2 star pattern from the client to the servers. Since we only have one pattern in the first step only one server will do the mining task. Next the server checks the support for the returned patterns and sends only the patterns that pass the support check to the client. The client compiles the frequent patterns sent by each server and sends the compiled pattern vector to the servers for the next exploration. Upon receiving the compiled frequent patterns from the client, each server extends the frequent patterns, takes its portion of the extended patterns and carries on with the exploration. The above steps take place step by step until we reach the exploration step limit.

Fault Tolerance: CAF provides default down message handlers which are invoked whenever a server goes down. I modified the down message handler to provide fault tolerance for our application. The client always saves a record of which task is assigned to each server. When the down message handler is invoked it matches the server which failed to the task record and re-assigns the task to a different server and update the work record. When a job is done it's record is cleared from the record list. This relay of work continues until all servers are down, in which case we print out the error message and exit.

3. Evaluation

Correctness was evaluated by comparing the output of my implementation with the vanilla Peregrine. To the study scaleability, I used citeseer, orkut and patents data-sets as data graphs and run motif, clique and pattern searches. The client process was running on a local machine while the server processes were running as instances on a remote server, with 96 cores and 500GB memory. The Node and single machine times reported are the execution times for the graph mining tasks. Total time for the distributed framework measures the total time, comprised of connection, synchronization and computation times. The Node and single machine times reported are the execution times for the graph mining tasks only. Total time for the distributed framework(Total distributed in the Figures) measures the total time, comprised of connection, synchronization, computation times and initialization overheads.

Table 1. 3-motifs counting, for orkut data graph execution times in(s)

Threads	Node 1	Node 2	Total	Single machine
1	81.87	85.09	86.21	162.67
2	41.12	40.97	42.14	80.71
4	20.63	20.22	21.6	39.91
8	12.04	11.13	13.21	21.407
16	6.97	7.03	8.01	11.14

Table 2. 3-motifs counting, for Patents data graph execution times in(s)

Threads	Node 1	Node 2	Total	Single machine
1	5.03	5.09	5.52	9.99
2	2.64	2.63	2.97	6.21
4	1.32	1.33	1.64	2.84
8	0.946	0.862	1.26	1.47
16	0.46	0.488	0.903	0.897

Table 3. 3-motifs enumerating, for orkut data graph execution times in(s)

Threads	Node 1	Node 2	Total	Single machine
1	360.991	359.996	361.89	719.974
2	186.393	180.309	187.306	357.548
4	89.6778	89.2898	90.6011	178.736
8	44.8422	44.8218	45.8561	91.2974
16	22.7535	23.0511	24.0321	45.9559

As we can see from Figures Tables 2, 1, 3, 4,5, 6, the time taken by each node is close enough to avoid one node

Table 4. 3-motifs enumerating, for patents data graph execution times in(s)

Threads	Node 1	Node 2	Total	Single machine
1	13.2802	13.1959	13.5783	26.0559
2	7.70045	7.09683	8.00097	13.3042
4	3.39916	3.37573	3.69357	6.72858
8	1.70797	1.69881	2.00822	3.7057
16	1.0938	1.07214	1.3956	1.92452

Table 5. 14-clique existence, for orkut data graph execution times in(s)

Threads	Node 1	Node 2	Total	Single machine
1	0.017	0.01	1.22	0.038
2	0.0090	0.012	1.23	0.024
4	0.0068	0.0051	1.23	0.017
8	0.0032	0.0031	1.22	0.0071
16	0.0033	0.0033	1.23	0.0034

Table 6. 14-clique existence, for patents data graph execution times in(s)

Threads	Node 1	Node 2	Total	Single machine
1	11.61	11.01	11.99	20.51
2	5.87	5.57	6.31	10.39
4	2.83	2.83	3.29	5.23
8	1.27	1.66	2.09	3.04
16	0.94	0.88	1.49	1.56

Table 7. 3-FSM 25K Support with 16 thread per node(N) runs, for patents data graph execution times in(s)

N	1	2	3	4	Total time
1	1891.27	0	0	0	1893.17
2	775.51	732.74	0	0	777.75
4	371.02	389.9	350.03	321.44	401.28

waiting for the other to finish, hence our task distribution avoids being bottle-necked by a critical path node. The total time is higher(1 second) than the node times mainly due to the initialization overhead of generating the patterns to match with the vector of count values we get from the server as well as time taken in accumulating the results received from the servers. Other synchronization and communication overheads were negligible and are measured to be in the order of milliseconds. As shown in Tables 2,6 and more obviously Table 5, the initialization overhead affects scaleability when its value becomes a significant portion of the total time. However, since the overhead is only about a second, it becomes negligible for real application that involve billions of vertices and edges.

As shown in Table 7 due to pattern based task division FSM execution time per node is not as fair as the other applications. We can improve the task allocation by moving the aggregation thread to the client side and using vertex based task mapping. However this apporach comes at a cost of sending a larger table to the client which introduces more network overhead.

Scaleability: To evalute scalability I evaluated execution time of our 4 applications(Count, Enumeration, Existence QUery and FSM) with 1,2,4 and 8 nodes, where each node runs a multithreaded(16 threads) instance. The runs are then normalized against the vanilla single machine version of Peregrine.

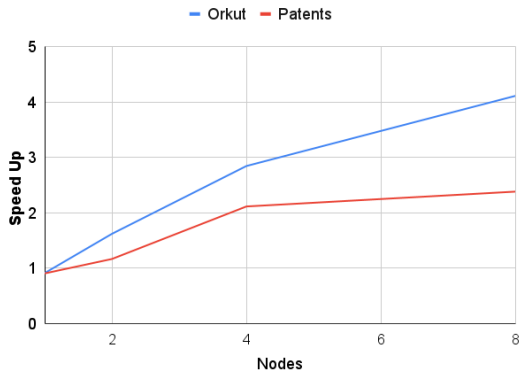

Figure 4. 3-motifs counting performance speedup normalized against single threaded single machine run on a orkut and patents data graph

As shown in Figure 4 and 5 the speedup gradually flattens with more nodes, because as the number of nodes increases the overall runtime decreased quickly and the majority of the runtime becomes dominated by constant initialization, synchronization and communication overheads. Orkut's speedup is more than patents because for patents the execution time is small, hence with more nodes the mining task's time becomes < 2 seconds and other overheads mentioned above will increasingly account for the majority of the runtime. As shown in figure As shown in Figure 6 for Orkut the speedup is 1 because the runtime is $< 1sec$ and the mining task only accounts for 0.1% of the runtime. For Patents the speedup flattens beyond 4 nodes because of a similar reason as above where the actual mining task becomes a small portion of the overall runtime, hence allocating more resource will not improve overall performance. As shown in Figure 7 the speedup scales linearly because the runtime is dominated by the mining task ($< 99\%$), hence allocating more resources will leads to better speedups. Since we also do pattern based work partitioning, as the number of nodes increases the number of pattern-support pairs sent from the server to the client also decreases which reduced the communication overhead as well.

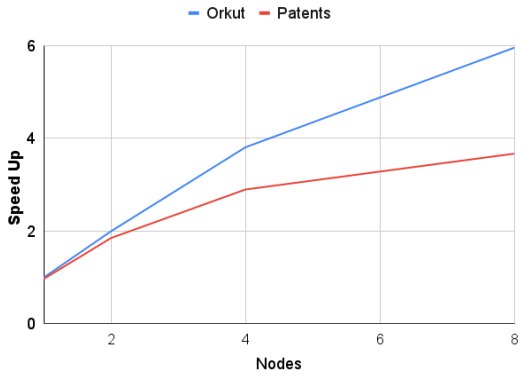

Figure 5. 3-motifs enumerate performance speedup normalized against single threaded single machine run on a orkut and patents data graph

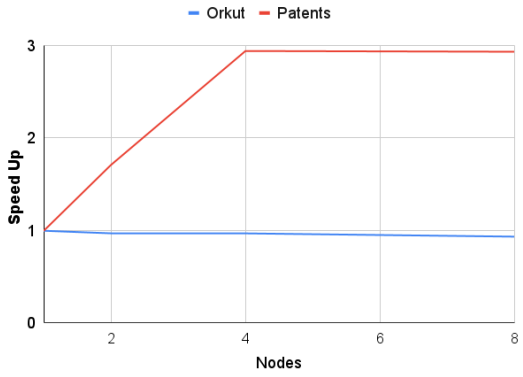

Figure 6. 14-clique existence query performance speedup normalized against single threaded single machine run on a orkut and patents data graph

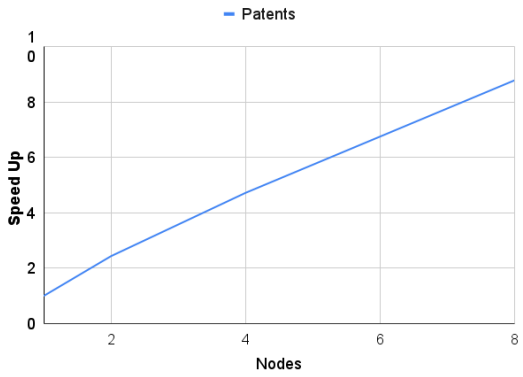

Figure 7. 3-FSM 25K Support performance speedup normalized against single threaded single machine run on patents data graph

4. Limitations

Our current implementation does not partition the data graph among the servers, hence each server needs to have a copy of the data graph. Secondly, the current implementation is a bit bulky because we added the distribution logic in the application codes, which affects the programmablity of peregrine. The code can be cleaned up and we can add a level of abstraction for distributing any user defined application, not only the above 4 application.

5. Conclusion

In this project, I scaled Peregrine, a pattern aware graph mininig system, by using an actor framework,CAF. I mapped tasks to remote nodes using either vertex or pattern granularity based on the application. I was able to observe promising scalability results which indicate Peregrine's ability to scale from a single machine implementation to a multi-node distributed solution. I was also able to used CAF's built-in down message handlers to provide fault tolerance against crash failures. The current implementation can be better optimized and added as an abstraction layer to Peregrine, to enable users to easily build any distributed mining application without worrying about the underlying details.

Source Code

Presentation Video

References

D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!*, pages 87–96, New York, NY, USA, Oct. 2013. ACM.

K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387548. URL `https://doi.org/10.1145/3342195.3387548`.

J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.