Asynchronous Graph Pattern Matching on Multiprocessor Systems

Alexander Krause(✉), Annett Ungethüm, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner

Database Systems Group, Technische Universität Dresden, Dresden, Germany
{Alexander.Krause,Annett.Ungethuem,Thomas.Kissinger,Dirk.Habich, Wolfgang.Lehner}@tu-dresden.de

Abstract. Pattern matching on large graphs is the foundation for a variety of application domains. Strict latency requirements and continuously increasing graph sizes demand the usage of highly parallel in-memory graph processing engines that need to consider non-uniform memory access (NUMA) and concurrency issues to scale up on modern multiprocessor systems. To tackle these aspects, graph partitioning becomes increasingly important. Hence, we present a technique to process graph pattern matching on NUMA systems in this paper. As a scalable pattern matching processing infrastructure, we leverage a data-oriented architecture that preserves data locality and minimizes concurrency-related bottlenecks on NUMA systems. We show in detail, how graph pattern matching can be asynchronously processed on a multiprocessor system.

1 Introduction

Recognizing comprehensive patterns on large graph-structured data is a prerequisite for a variety of application domains such as biomolecular engineering [11], scientific computing [17], or social network analytics [12]. Due to the ever-growing size and complexity of the patterns and underlying graphs, *pattern matching* algorithms need to leverage an increasing amount of available compute resources in parallel to deliver results with an acceptable latency. Since modern hardware systems feature main memory capacities of several terabytes, state-of-the-art graph processing systems (e.g., Ligra [16] or Galois [10]) store and process graphs entirely in main memory, which significantly improves scalability, because hardware threads are not limited by disk accesses anymore. To reach such high memory capacities and to provide enough bandwidth for the compute cores, modern servers contain an increasing number of memory domains resulting in a *non-uniform memory access (NUMA)*. To further scale up on those NUMA systems, pattern matching on graphs needs to carefully consider issues such as the increased latency and the decreased bandwidth when accessing remote memory domains, as well as the limited scalability of synchronization primitives such as atomic instructions [21].

The widely employed *bulk synchronous parallel (BSP)* processing model [19], which is often used for graph processing, does not naturally align with pattern

M. Kirikova et al. (Eds.): ADBIS 2017, CCIS 767, pp. 45–53, 2017.
DOI: 10.1007/978-3-319-67162-8_6

matching algorithms [3]. That is because a high number of intermediate results is generated and need to materialized and transferred within the communication phase. Therefore we argue for an asynchronous processing model that neither requires a full materialization nor limits the communication to a distinct global phase. For efficient *pattern matching* on a single NUMA system, we employ a fine-grained *data-oriented architecture (DORA)* in this paper, which turned out to exhibit a superior scalability behavior on large-scale NUMA systems as shown by Pandis et al. [13] and Kissinger et al. [6]. This architecture is characterized by implicitly partitioning data into small partitions that are explicitly pinned to a NUMA node to preserve a local memory access.

Contributions. Following to a discussion of the foundations of graph pattern matching in Sect. 2, the contributions of the paper are as follows:

(1) We adapt the data-oriented architecture for scale-up graph pattern matching and identify the *partitioning strategy* as well as the design of the *routing table* as the most crucial components within such an infrastructure (Sect. 3).
(2) We describe an asynchronous query processing model for graph pattern matching and present the individual operators a query is composed of. Based on the operator characteristics, we identify *redundancy* in terms of partitioning as an additional critical issue for our approach (Sect. 4).
(3) We thoroughly evaluate our graph pattern matching approach on multiple graph datasets and queries with regard to scalability on NUMA systems. Within our evaluation, we focus on different options for the partitioning strategy, routing table, and redundancy as our key challenges (Sect. 5).

Finally, we discuss the related work in Sect. 6 and conclude the paper in Sect. 7 including promising directions for future work.

2 Foundations of Graph Pattern Matching

Within this paper, we focus on pattern matching for *edge-labeled multigraphs* as a general and widely employed graph data model [12,14]. An edge-labeled multigraph $G(V, E, \rho, \Sigma, \lambda)$ consists of a set of vertices V, a set of edges E, an incidence function $\rho : E \rightarrow V \times V$, and a labeling function $\lambda : E \rightarrow \Sigma$ that assigns a label to each edge, according to which edge-labeled multigraphs allow any number of labeled edges between a pair of vertices. A prominent example for edge-labeled multigraphs is RDF [2].

Pattern matching is a declarative topology-based querying mechanism where the query is given as a graph-shaped pattern and the result is a set of matching subgraphs [18]. For instance, the *query pattern* depicted in Fig. 1 searches for a vertex V_1, that has two outgoing edges targeting V_2 and V_3. Additionally, the query pattern seeks a fourth vertex V_4 which also has two outgoing edges to the same target vertices. The query pattern forms a rectangle with four vertices and four edges of which we search for all matching subgraphs in a graph. A well-studied mechanism for expressing such query patterns are *conjunctive queries*

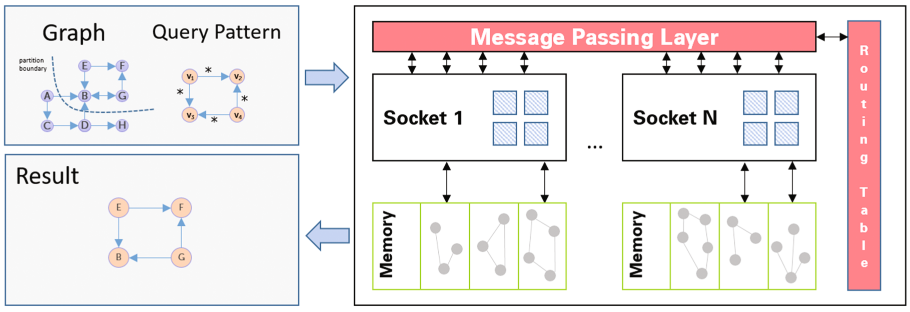


Fig. 1. Scalable graph pattern matching based on a data-oriented architecture [6,13].

(CQ) [20], which decompose the pattern into a set of *edge predicates* each consisting of a pair of vertices and an edge label. Assuming a wildcard label, the exemplary query pattern in Fig. 1 can be decomposed into the conjunctive query $\{(\boldsymbol{V_1}, *, V_2), (\boldsymbol{V_1}, *, V_3), (\boldsymbol{V_4}, *, V_3), (\boldsymbol{V_4}, *, V_2)\}$, where the bold vertices represent the source vertex of an edge. These four edge predicate requests form a sequence, that is processed by starting at each vertex in the data graph, because the query pattern does not specify a specific starting vertex.

3 Scalable Graph Pattern Matching Architecture

In this section, we briefly describe our target architecture and refer to the extended version of our paper [7] for more details. Figure 1 illustrates a NUMA system with N sockets which can run multiple worker threads concurrently, based on the underlying hardware. A graph of the form described in Sect. 2 can be distributed among the main memory regions, which are attached to one of the sockets. The distribution of the graph among these memory regions inherently demands graph partitioning and an appropriate partitioning strategy.

Partitioning Strategy. However, partitioning a graph will most likely lead to edges, which span over multiple partitions, like the edges $A \rightarrow B$ and $D \rightarrow B$ on the left hand side of Fig. 1. For instance, if vertex A is considered as a potential match for a query pattern, the system needs to lookup vertex B in another partition. Moving to another partition requires that the complete matching state needs to be transferred to another worker, which requires communicational efforts between the two responsible workers. Hence, the selection of the *partitioning strategy* is crucial when adapting the data-oriented architecture for graph pattern matching, because locality in terms of the graph topology is important [8].

Routing Table. Because one partition can not always contain all the necessary information for one query, it is inevitable to communicate intermediate results between workers. The communication is handled by a high-throughput message passing layer, which hides the latency of the communication network, as depicted in Fig. 1. The system stores the target socket and partition information in a crucial data structure, the *routing table*. The routing table determines the target partition as well as the target NUMA node per vertex. Thus, the routing table needs to be carefully designed, because real world graphs often feature millions of vertices and billions of edges.

Since *routing table* and *partitioning strategy* depend on each other, we consider the following three design options for our discussion and evaluation:

Compute Design. This design uses a hash function to calculate the target partition of a vertex based on its identifier on-the-fly and stores no data at all. Nevertheless, due to the simplicity of the routing table, the partitioning strategy can not take any topology-based locality information into account.

Lookup Design. The lookup design consists of a hash map, which stores a precomputed graph partitioning, i.e. one partition entry per vertex in the graph, thus this design doubles the memory footprint, since the graph is stored once as graph data and once in the routing table as topology data. As partitioning strategy, we use the well known *multilevel k-Way* partitioning to create a disjoint set of partitions. This heuristical approach creates partitions with high locality and tries to minimize the edge cut of the partitioning [5].

Hybrid Design. We created this design to combine the advantages of the two previous approaches, i.e. a small and locality preserving routing table. To enable this combination, we employ a dictionary as auxiliary data structure that maps virtual vertex ids to the original vertex ids of the locality aware graph partitioning. The dictionary is only used for converting the vertex ids of final query results. This range-based routing table maps dense ranges of virtual ids to the respective partition and has very low memory footprint such that the routing table easily fits into the cache of the multiprocessors.

4 Graph Pattern Matching Processing Model

The architecture introduced in Sect. 3 needs specific operators for pattern matching on NUMA systems. We identified three logical operators, which are necessary to model *conjunctive queries* as described in Sect. 2:

Unbound Operator. The unbound operator performs a parallel vertex scan over all partitions and returns edges matching the specified label. The unbound operator is always the first operator in the pattern matching process.

Vertex-Bound Operator. The vertex-bound operator takes an intermediate matching result as input and tries to match a new vertex in the query pattern.

Edge-Bound Operator. The edge-bound operator ensures the existence of additional edge predicates between vertices which are matching candidates for certain vertices of the query pattern. It performs a data lookup with a given source and target vertex as well as a given edge label. If the lookup fails, both vertices are eliminated from the matching candidates. Otherwise the matching state is passed to the next operator or is returned as final result.

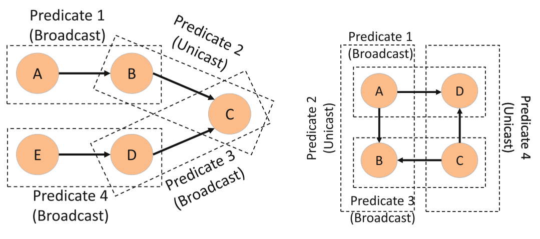
To actually compose a *query execution plan (QEP)*, the query compiler sequentially iterates over the *edge predicates* of the *conjunctive query*. For each edge predicate, the query compiler determines whether source and/or target vertex are bound and selects the appropriate operator for the respective edge predicate. For the example query pattern in Fig. 1, the resulting operator assignments of the QEP are shown in Fig. 2(c).

Each operator is asynchronously processed in parallel and generates new messages that invoke the next operator in the QEP. Hence, different worker threads can process different operators of the same query at the same point in time. Based on the operator and its parametrization, we distinguish two ways of addressing a message that are related to the *routing table*:

Unicast. A unicast addresses a single graph partition and requires that the source vertex is known respectively bound by the operator. This case occurs for the *vertex-bound operator* if the source vertex is bound and for the *edge-bound* operator.

Broadcast. A broadcast targets all partitions of a graph, which increases the pressure on the message passing layer and requires the message to be processed on all graph partitions and thus, negatively affects the scalability. Additionally, *vertex-bound operators* that bound the target vertex require a broadcast.

Broadcasts generated by *vertex-bound operators* significantly hurt the scalability of our approach. The cause of this problem is inherently given by the data-oriented architecture, because a graph can either be partitioned by the source or the target vertex of the edges. Hence, we identify *redundancy* in terms of partitioning as an additional challenge for our approach. To reduce the need for broadcasts to the initial *unbound operator*, we need to redundantly store the graph partitioned by source vertex and partitioned by target vertex. However, the need for redundancy depends on the query pattern as well as on the graph itself as we will show within our evaluation.



(a) *V* Query (b) *Quad* Query

Edge predicate	Bound variable	Unbound variable	Operator
$v_1, *, v_2$	$\emptyset$	$\{v_1, v_2\}$	Unbound
$v_1, *, v_3$	$\{v_1\}$	$\{v_3\}$	Vertex-Bound
$v_4, *, v_3$	$\{v_3\}$	$\{v_4\}$	Vertex-Bound
$v_4, *, v_2$	$\{v_2, v_4\}$	$\emptyset$	Edge-Bound

(c) Query execution plan example.

Fig. 2. Query patterns and example operators for the query from Fig. 1.

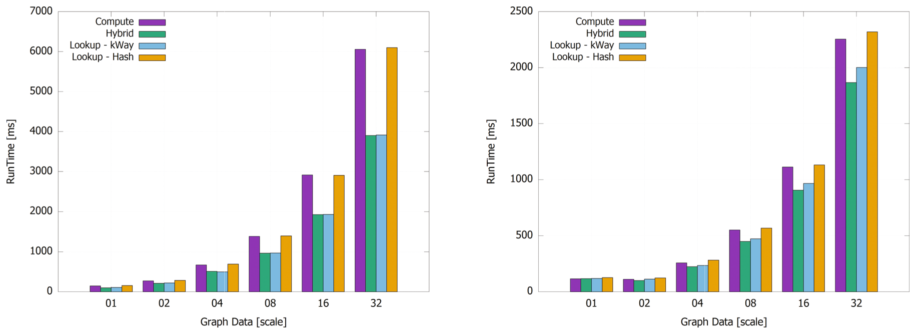
5 Evaluation

In this section, we briefly describe our findings and refer to our extended version for an in depth explanation of the individual results [7]. We used a bibliographical like graph, which we call *biblio* for the remainder of this paper. The *biblio* graph was generated using the graph benchmark framework gMark [1] and has 546 k vertices, 780 k edges and an average out degree of 2.85 per vertex. Our NUMA system consists of four sockets equipped with an Intel Xeon E7-4830 CPU and a total of 128 GB of main memory. We defined two queries which are shaped like shown in Figs. 2(a) and (b) and ran them on the *biblio* graph.

Routing Table and Partitioning Strategy. Based on Fig. 3 we examine the infleunce of the routing table on the query performance. The figure shows the query runtime for the V query on the *biblio* graph, which we scaled up from factor 1 to factor 32. On the left hand side, we show the sole influence of the routing table and on the right hand side of the figure we show the query runtime per routing table design, if redundancy is used. In Fig. 3(a) we can see that our hybrid design marginally outperforms the memory intensive lookup design with a k-Way partitioning. The compute design and the lookup design which uses a hash function perform equally in terms of query performance. The advantage of our hybrid design and the k-Way based lookup design stems from the better graph partitioning algorithm, because neighborhood locality of adjacent vertices is considered. Our experiments showed, that the compute design results in the lowest time spent in the routing table per worker, which is not surprising. However, our hybrid design almost reaches the same routing table time due its the small size.

Avoiding Broadcasts with Redundancy. In Sect. 4 we mentioned that broadcasts hurt the scalability of a system. This issue is depicted in Fig. 4. The figure shows the scalability of our systems for both query types from Fig. 2. On the right hand side, we see that the *Quad* query suffers more from broadcasts. The reason is, that many tuples are matched for predicate 2 (c.f. Fig. 2(b)), which leads to a high number of broadcasts during the evaluation of predicate 3. For the V query on the left hand side of the Figure, we can see that the employment of redundancy still decreases the query runtime, but not as much as for the *Quad* query, because the broadcasting predicates are not dominant for this specific query instance.

Combining Redundancy and Routing Table Optimizations. Aside from testing both optimization techniques individually, we combined them to examine their synergy. In Fig. 3(b), we demonstrate the query performance of the *V* query on the *biblio* graph, again scaled up to factor 32. By adding redundancy to the query execution, all routing table designs greatly benefit in terms of query performance. However, we can now see a bigger advantage of our hybrid design, compared to the lookup k-Way design.



(a) Query runtime without redundancy (b) Query runtime with redundancy

Fig. 3. *V* query on the *biblio* graph using different scale factors.

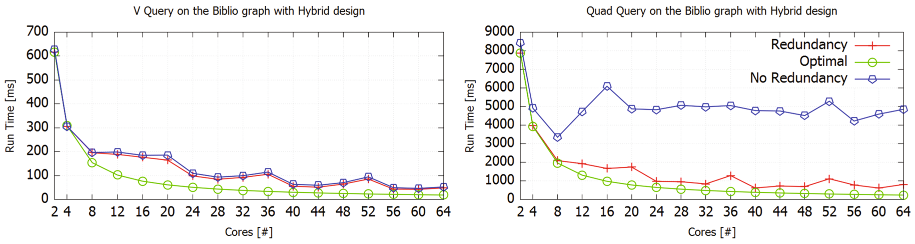


Fig. 4. Impact of redundancy, both queries on the *biblio* graph.

6 Related Work

Graph analytics is a widely studied field, as the survey from McCune et al. [9] shows. Many systems leverage the increased compute performance of scale-up or scale-out systems to compute graph metrics like PageRank and the counting of triangles [15], Single Source Shortest Path [15] or Connected Components [15]. Many of the available systems are inspired by the Bulk Synchrones Processing Model [19], which features local processing phases which are synchronized by a global superstep. A general implementation is the Gather-Apply-Scatter paradigm, as described in [4]. Despite working on NUMA systems, these processing engines are globally synchronized and lack the scalability of a lock-free architecture. We improve this issue by leveraging a high throughput message passing layer for asynchronous communication between the worker threads. However, in contrast to the systems mentioned above, we are calculating graph pattern matching and not graph metrics, like for instance GraphLab which is the only asynchronous graph processing engine according to [9].

7 Conclusions

In this paper, we showed that the performance of graph pattern matching on a multiprocessor system is determined by the communication behavior, the employed routing table design and the partitioning strategy. Our *Hybrid* routing table design implementation allows the system to leverage both the advantages from a *Compute* design and a *Lookup* design. Because of an intermediate step, the underlying graph partitioning algorithm is interchangable and can thus be adapted to specific partitioning requirements. Furthermore we could show that avoiding broadcasts is equally important. This issue was mitigated by introducing redundancy in the system. The added memory footprint can be mitigated with the positive influence of our *Hybrid* design, since it scales directly with the number of data partitions in the system.

Acknowledgments. This work is partly funded within the DFG-CRC 912 (HAEC).

References

1. Bagan, G., et al.: Generating flexible workloads for graph databases. PVLDB **9**, 1457–1460 (2016)
2. Decker, S., et al.: The semantic web: the roles of xml and rdf. IEEE **4**, 63–73 (2000)
3. Fard, A., et al.: A distributed vertex-centric approach for pattern matching in massive graphs. In: 2013 IEEE International Conference on Big Data (Oct 2013)
4. Gonzalez, J.E., et al.: Powergraph: Distributed graph-parallel computation on natural graphs. In: OSDI (2012)
5. Karypis, G., et al.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
6. Kissinger, T., et al.: ERIS: A numa-aware in-memory storage engine for analytical workload. In: ADMS (2014)
7. Krause, A., et al.: Asynchronous graph pattern matching on multiprocessor systems (2017). https://arxiv.org/abs/1706.03968
8. Krause, A., et al.: Partitioning Strategy Selection for In-Memory Graph Pattern Matching on Multiprocessor Systems (2017). http://wwwdb.inf.tu-dresden.de/europar2017/. Accepted at Euro-Par 2017
9. McCune, R.R., et al.: Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. **48**(2), 25:1–25:39 (2015)
10. Nguyen, D., et al.: A lightweight infrastructure for graph analytics. In: SIGOPS (2013)
11. Ogata, H., et al.: A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. Nucleic Acids Res. **28**, 4021–4028 (2000)
12. Otte, E., et al.: Social network analysis: a powerful strategy, also for the information sciences. J. Inf. Sci. **28**, 441–453 (2002)
13. Pandis, I., et al.: Data-oriented transaction execution. PVLDB **2**, 928–939 (2010)
14. Pandit, S., et al.: Netprobe: A fast and scalable system for fraud detection in online auction networks. In: WWW (2007)
15. Seo, J., et al.: Distributed socialite: A datalog-based language for large-scale graph analysis. PVLDB **6**, 1906–1917 (2013)
16. Shun, J., et al.: Ligra: a lightweight graph processing framework for shared memory. IN: SIGPLAN (2013)
17. Tas, M.K., et al.: Greed is good: Optimistic algorithms for bipartite-graph partial coloring on multicore architectures. CoRR (2017)

18. Tran, T., et al.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: ICDE (2009)
19. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**, 103–111 (1990)
20. Wood, P.T.: Query languages for graph databases. SIGMOD **41**, 50–60 (2012)
21. Yasui, Y., et al.: Numa-aware scalable graph traversal on SGI UV systems. IN: HPGP (2016)