Partitioning Strategy Selection for In-Memory Graph Pattern Matching on Multiprocessor Systems

Alexander Krause, Thomas Kissinger, Dirk Habich, Hannes Voigt,
Wolfgang Lehner
{firstname.lastname}@tu-dresden.de

Technische Universität Dresden
Database Systems Group
Dresden, Germany

Abstract. Pattern matching on large graphs is the foundation for a variety of application domains. The continuously increasing size of the underlying graphs requires 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, a fine-grained graph partitioning becomes increasingly important. Hence, we present a classification of graph partitioning strategies and evaluate representative algorithms on medium and large-scale 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. Our in-depth evaluation reveals that the optimal partitioning strategy depends on a variety of factors and consequently, we derive a set of indicators for selecting the optimal partitioning strategy suitable for a given graph and workload.

1 Introduction

Recognizing comprehensive patterns on large graph-structured data is a prerequisite for a variety of application domains such as fraud detection [11], biomolecular engineering [8], scientific computing [13], or social network analytics [9]. 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 [12], Galois [7] or, Green-Marl [5]) tend to 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).*

For instance, on a multiprocessor system each processor maintains at least one separate memory domain that is accessible for other processors via a communication network. However, efficient data processing on those systems faces several issues such as the increased latency and the decreased bandwidth when accessing remote memory domains. To further scale up on those NUMA systems, pattern matching on graphs needs to carefully consider these issues as well as the limited scalability of synchronization primitives such as atomic instructions [18].

To scale up *pattern matching* on those NUMA systems, 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. [10] and Kissinger et al. [6]. This architecture is characterized by implicitly partitioning data into small partitions that are pinned to a NUMA node to preserve a local memory access. In contrast to the bulk synchronous parallel (BSP) processing model [15], which is often used for graph processing, the data partitions are processed by local worker threads that communicate asynchronously via a high-throughput message passing layer. Hence, the overall performance of the *pattern matching* mainly depends on the graph partitioning.

In this paper, we systematically evaluate the influence of different graph partitioning strategies on the performance of *pattern matching* using a data-oriented architecture. Therefore, we introduce a novel classification of graph partitioning strategies and evaluate performance aspects of representative partitioning algorithms for each class. Our exhaustive evaluation on medium (4 sockets) and large-scale (64 sockets) NUMA systems reveals that the selection of the appropriate partitioning strategy depends on a multitude of factors such as graph characteristics, query pattern, the number of partitions, and worker threads. Thus, we argue that there is no one-size-fits-all strategy for partitioning graphs within a NUMA system and identify key features that shall guide partitioning strategy selection process.

Contributions. Our contributions are summarized as follows:

(1) We present a graph pattern matching processing model that is based on a fine-grained *data-oriented architecture* that is designed to operate on large scale-up NUMA systems (Section 2).
(2) We provide a classification of graph partitioning strategies that arranges the individual strategies based on a *partitioning criterion* and a *balancing criterion*. Moreover, we describe instances of the respective classes that we consider for our evaluations (Section 3).
(3) We exhaustively evaluate our identified partitioning strategies for different graphs and patterns on a medium and large-scale NUMA system and reason about the results. Our investigations show that the optimal partition strategy depends on a variety of factors (Section 4).
(4) Based on our evaluations, we derive a set of indicators that should be considered in the process of selecting the optimal partitioning strategy for pattern matching on graphs (Section 4.3).

Finally, we will give an overview of the related work (Section 5) and conclude the paper (Section 6).

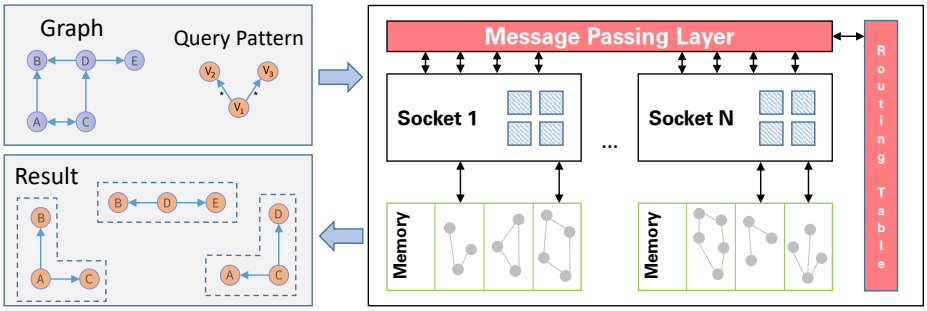


Fig. 1. Scalable graph pattern matching based on a Data-Oriented Architecture [6, 10].

2 Graph Pattern Matching on NUMA Systems

Within this paper, we focus on *edge-labeled multigraphs* as a general and widely employed graph data model [8,9,11]. 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. Hence, edge-labeled multigraphs allow any number of labeled edges between a pair of vertices. A prominent example for edge-labeled multigraphs is RDF [4].

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 [14]. For instance, the *query pattern* depicted on the left hand side of Figure 1 searches for all vertices that have two outgoing edges resulting in three matching subgraphs for the given underlying graph. A well-studied mechanism for expressing such query patterns are *conjunctive queries (CQ)* [17], 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 is decomposed into the conjunctive query $\{(\boldsymbol{V_1}, *, V_2), (\boldsymbol{V_1}, *, V_3)\}$.

To scale up graph pattern matching on large multiprocessor systems, we employ an approach that is based on a *data-oriented architecture (DORA)* [10], which is known for its superior scalability on NUMA systems [6]. As illustrated on the right hand side of Figure 1, the graph is implicitly split into a set of disjoint partitions. Each partition is placed in the local memory of a specific processor that runs *workers* on its local hardware threads. These workers are limited to operate exclusively on local graph partitions and leverage a high-throughput message passing layer for the inevitable communication. Only one worker is allowed to access a partition at a time to avoid costly fine-grained lockings of the data structures. Consequently, the number of workers is limited to the available local hardware threads and the number of local partitions can be chosen arbitrarily. An integral part of the message passing layer is the *routing table*, which keeps track of the partitioning and thus, maps the *partitioning criteria* (cf., Section 3) to the corresponding partition using a hash table. The overall goal of this architecture is (1) to restrict the access of threads to data structures in the local main memory,

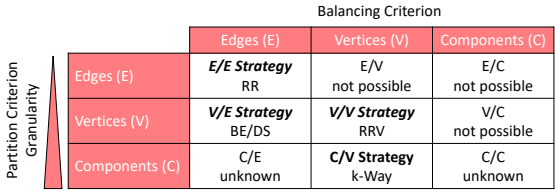


Fig. 2. Classification of graph partitioning strategies and representative algorithms.

(2) to reduce the necessity of locks or atomic instructions, and (3) to hide remote memory latency using the high-throughput message passing layer.

To actually process *conjunctive queries* on such a data-oriented architecture, the *edge predicates* – CQs are consisting of – are evaluated one after another. Every time an edge predicate matches within a partition, a new message is generated by the worker thread to evaluate the successive edge predicate unless the predicate was the last one of the CQ. These messages are either sent to a single partition (unicast) or to all partitions (broadcast) depending on the edge predicate and partitioning criterion. Due to the topology-driven nature of pattern matching and the comprehensive structure of graphs, the appropriate selection of a *partitioning strategy* for a specific combination of query pattern and underlying graph is crucial for such an architecture as we will show throughout this paper.

3 Graph Partitioning Strategies

In this section, we provide a classification of known graph partitioning strategies and detail on our heuristic implementations of the individual strategies that we consider for further evaluation. We restrict our considerations to partitioning strategies that generate a disjoint set of graph partitions and leave redundancy for future work. As shown in Figure 2, our classification spans two dimensions:

(1) The *partitioning criterion* that denominates the basic unit of the graph a partitioning strategy is operating on.
(2) The *balancing criterion* describing the unit of the graph that is balanced to achieve an equal utilization of the parallel compute resources.

For both dimensions those units are either fine-grained *edges (E)*, *vertices (V)*, or coarse-grained *components (C)* naming a connected set of vertices. Hence, a *partitioning strategy* is a combination of a *partitioning criterion* and a *balancing criterion*. Partitioning a graph at a specific granularity implies that more coarse-grained balancing criteria are not applicable (i.e., E/V, E/C, and V/C strategy). To the best of our knowledge, there are no known viable representatives for the C/E and C/C strategy. In the following, we detail on the feasible strategies and describe our heuristic implementations that we use for our evaluation:

E/E Strategy. This partitioning strategy works on the most fine-grained level. We implemented this strategy using the *round-robin (RR)* algorithm, which evenly distributes edges to partitions in a lightweight fashion. This strategy

is likely to distribute many or all outgoing edges of one vertex to multiple partitions. This decomposition leads to the necessity of broadcasts for the evaluation of all *edge predicates*.

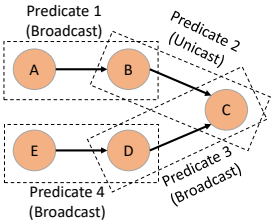
V/V Strategy. This strategy partitions a graph by its vertices and balances the amount of vertices per partition. Hence, our *round-robin vertices (RRV)* algorithm is a specific implementation of this strategy and distributes every vertex and all of its outgoing edges to the partitions using the lightweight round-robin method. The advantage with regard to our pattern matching processing model (cf., Section 2) is that all outgoing edges of a vertex belong to a single partition being listed in the routing table. Thus, each *edge predicate* with a known source vertex can be routed to a single partition (unicast).

V/E Strategy. Similar to the *RRV* strategy, the graph is partitioned by its vertices. However, this partitioning strategy balances the number of edges. We consider two specific algorithms as implementation of this strategy: *balanced edges (BE)* and *distributed skew (DS)*. Both algorithms sort the vertices by the number of outgoing edges in a descending order. The BE algorithm iterates over this sorted list and assigns each vertex and all of its outgoing edges to the currently smallest partition to greedily balance the edges across the partitions. Thus, all outgoing edges of a vertex belong to the same partition, which once again results in a unicast for *edge predicates* with a known source vertex. The DS algorithm is a state-of-the-art approximation for handling skewed data in distributed joins [3] and extends the BE algorithm. To relieve highly connected vertices, DS equally distributes the edges of vertices that have significantly more outgoing edges compared to the average vertex across all partitions. Nevertheless, *edge predicates* aiming at those source vertices require a broadcast to all partitions. Because most real world graphs exhibit a non-uniform edge per vertex distribution, all vertex-oriented partitioning strategies (RRV, BE and DS) lead to different partitioning results.

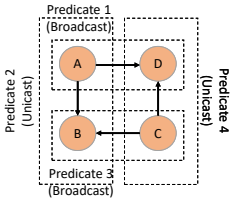
C/V Strategy. The goal of a component-oriented strategy is to preserve locality by storing strongly connected vertices within the same partition. We leverage the well-known state-of-the-art *multilevel k-Way* algorithm [2] as representative, which tries to balance the vertices among the partitions. Similar to the vertex-oriented strategies, we store all outgoing edges of a vertex in the same partition to avoid broadcasts during the pattern matching process.

4 Experimental Evaluation

To investigate the influence of the *partitioning strategies* (c.f, Section 3) on the *pattern matching* performance, we conducted an exhaustive evaluation on a medium and large-scale multiprocessor system. We use four test data graphs of varying application domains that are generated with graph benchmark framework gMark [1]. Additionally, we defined two *conjunctive queries* as depicted in Figure 3: (1) the *V* query shapes a V with five vertices and four edges and (2) the *Quad* query is a rectangle, which consists of four vertices and four edges. For both queries,



(a) V Query



(b) *Quad* Query

Graph	#Vertices	#Edges	Average out Degree	Median out Degree
Biblio	546k	780k	2.85	2
Shop	53k	3M	112.56	20
Social	91k	232k	5.07	3
Uniprot	57k	339k	11.90	7

(c) Graph meta information

Fig. 3. Query patterns and test graphs for the medium-scale system.

four *edge predicate* evaluations are necessary. Based on the query semantics, the evaluation of the edge predicates happens as follows:

V Query. The first edge predicate evaluation is broadcasted to all partitions, because only the edge label is known and not the source vertex. The intermediate result is a set of end vertices, which are used as source vertices for the second request. Depending on the partitioning strategy, the second edge request is evaluated using either unicast or broadcast messages (cf., Section 3). The intermediate result is a set of destination vertices, which are destination vertices for the third edge predicate. Hence, the third requests needs to be broadcasted to all partitions, because the source vertex is unknown. The same applies for the fourth edge predicate evaluation.

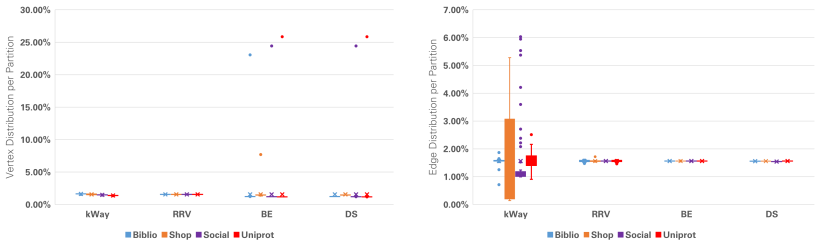
Quad Query. The edge predicate evaluation of the *Quad* query is similar to the one of the V query with the difference that the evaluation of the fourth edge predicate depends on the partitioning strategy. Thus, this predicate can mostly be evaluated without the need of a broadcast.

As the edge predicate evaluation of our two queries suggests, pattern matching is a combination of unicasts and broadcasts within a partitioned environment. On the one hand, broadcasts distribute the evaluation of edge predicates to all partitions favoring edge-balanced partitions for an efficient execution. On the other hand, unicast messages assign edge predicate evaluations to single partitions, which – in contrast – favors vertex-balanced partitions.

For all of our experiments, we loaded the graph-under-test into main memory; partitioned it; and evenly distributed the partitions across the sockets and executed both pattern queries for all partitioning strategies and all possible *system configurations (SC)*. In our case a system configuration denominates a combination of the active workers and the total number of partitions.

4.1 Evaluation on a Medium-Scale Multiprocessor System

Our medium-scale multiprocessor system consists of 4 sockets each equipped with an Intel Xeon CPU E7-4830 – resulting in 32 physical cores and 64 hardware threads – and 128 GB of main memory. For this system, we use the graphs with the characteristics listed in Figure 3(c).



(a) Vertex distribution. (b) Edge distribution.

Fig. 4. Partitioning results for 64 partitions.

Partitioning Results. Figure 4 shows an overview of partitioning results for the different strategies and our test graphs. Since we have 64 hardware threads, we split the graphs into 64 partitions. The plots show the distribution of vertices and edges over the 64 partitions using box plots. From these plots and our experiments with varying number of partitions, we can derive the following observations:

(1) The partitioning and balancing criteria of the respective strategies are fulfilled independently of the graphs. For instance, our *round-robin vertices (RRV)* algorithm partitions the graphs by vertices and ideally balances the vertices among the 64 partitions, i.e., the vertices are evenly distributed over the partitions as depicted in Figure 4(a). The same applies for *balanced edges (BE)* and *distributed skew (DS)* which perfectly balance the edges among the partitions, as shown in Figure 4(b).

(2) Depending on the strategy, balancing is done either by vertices or edges. This can lead to an imbalance on the non-balancing criterion depending on the underlying graph. For instance, *BE* and *DS* balance the partitions on edges. However, there are few partitions with a much higher number of vertices than the others (illustrated as single dots in Figure 4(a)). These outliers depend on the graph data. For *DS* outlier, partitions exist for Uniprot and Social, but not for Biblio and Shop. The same issue is observable for *RRV*, however the imbalance on the edges over the partitions is not as remarkable.

(3) The k-Way algorithm partitions graphs by components and balances the vertices. On the one hand, this leads to an even distribution of the vertices over the partitions for our test graphs as shown in Figure 4(a). This potentially leads to an imbalanced number of edges per partition and this imbalance is very different for the four test graphs, as visible in Figure 4(b).

(4) The E/E strategy performs worst. The round-robin distribution of the edges among all partitions leads to the necessity of broadcasts during all edge predicate evaluations, which massively inhibits the system. Therefore, we omit the E/E results henceforward.

To summarize, each partitioning strategy is able to successfully maintain its respective balancing criterion while partitioning the graph into the considered

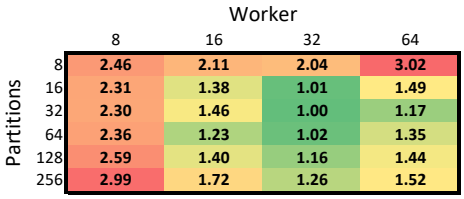


Fig. 5. System configuration heat map for RRV. V query on Biblio graph.

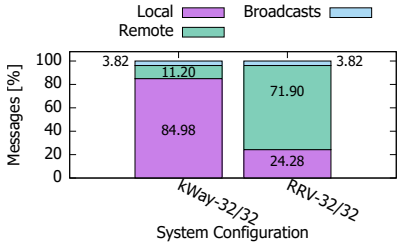
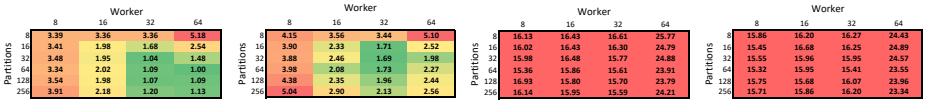


Fig. 6. Messages per partitioning algorithm. V query on Biblio graph.



(a) C/C: k-Way (b) V/V: RRV (c) V/E: BE (d) V/E: DS

Fig. 7. System configuration heat map. V query on Biblio graph. Color shadings relative to the global optimum (k-Way 64/64).

number of partitions. However, the quality of the result is different for each case. Depending on the graph, there are partitions that vary greatly from the majority.

Number of Partitions and Workers. If we compare the partitioning results of Figure 4 for the Biblio graph, we find that the V/V strategy (RRV) achieves the best partitioning result in terms of balanced partitions for both vertices and edges. Generally, such partitioning is very beneficial for our pattern matching.

In the first set of experiments, we use that setting to investigate the influence of the system configuration on the pattern matching performance for the V query. Thus, we varied the number of active workers between 8 and 64 and used 8 to 256 partitions. The heat map from Figure 5 shows the slowdown factors compared to the optimal configuration. The optimal configuration uses 32 partitions and 32 workers. Generally, the pattern matching scales well for physical hardware threads, which is indicated by the coloring trend from orange to green between the columns for 8 and 32 workers. In this case, 64 workers are not beneficial, because the V query employs two broadcasting requests at the end and the hyper-threads do not provide as much performance as their physical siblings.

Partitioning Strategies. After examining the query performance for a single graph partitioning strategy, we conducted the same experiments with the remaining strategies to show the influence of the different partitioning strategies in detail. The resulting heat maps are depicted in Figure 7. From these heat maps, we derive the following three facts:

(1) The V/E strategy, represented by the *BE* and *DS* algorithms, performs comparatively bad. This happens because the query massively hits the vertex outlier partition, which can be seen in Figure 4(a). Hence, this partition becomes a bottleneck for the second edge predicate of the V query.
(2) The k-Way partitioning results in a better query performance and utilizes the whole system with its optimal system configuration being 64 partitions by 64 workers. The advantage of k-Way is the partitioning and balancing of components. For the Biblio graph this results in even distribution of vertices and an almost even distribution of edges among the partitions. Furthermore, connected vertices are partitioned together, which is not necessarily the case for RRV as illustrated in Figure 6. For the k-Way partitioning, the system creates mostly socket local messages and only a few remote messages, whereas the V/V strategy results in many remote messages as connected vertices are distributed among partitions on remote sockets.

From these results, we can conclude that the C/V partitioning strategy results in partition population that allows the system to scale up to its full potential.

Varying Graphs. After thoroughly examining the influences of different partitioning strategies on one graph, we conducted the same experiments for all other graphs from Figure 3(c). Figure 8 presents the best system configurations per partitioning strategy and highlights the overall optimum. We showed that the C/C strategy performs best for the V query on the Biblio graph by utilizing the whole system and should be the strategy of choice. However, when querying the Shop graph with a k-Way partitioning, the query performance drops by a factor of 2.3 while employing 32/32 as its optimal system configuration. The slowdown can be explained by the massive imbalance of edges within the partitions of k-Way as shown in Figure 4(b). The other strategies show well balanced edges per partition, therefore all of them result in equal query performance. The same holds for the Social graph. The Uniprot graph is special in terms of the intermediate results, which are shown in Figure 11. Compared to the Biblio graph, the V query produces a huge number of broadcasts for the Uniprot graph in the third edge predicate (c.f. Figure 3(a)), which inhibts the system from scaling well, and therefore yields better performance for less workers. We conclude that the behavior of the query is strongly tied to the underlying graph.

	Biblio		Shop		Social		Uniprot	
Strategy	SC	ms	SC	ms	SC	ms	SC	Ms
V/V: RRV	32/32	65	**32/32**	**11790**	**32/32**	**665**	8/8	884
V/E: BE	32/128	838	32/32	12387	16/16	666	**8/8**	**878**
V/E: DS	8/16	849	32/32	11964	32/32	673	8/8	890
C/V: k-Way	**64/64**	**48**	32/32	27376	32/32	864	8/8	885

Fig. 8. Optimal system configurations per graph and partitioning strategy for the V query.

	Biblio		Shop		Social		Uniprot	
Strategy	SC	ms	SC	ms	SC	ms	SC	Ms
V/V: RRV	32/32	2663	**32/64**	**5773**	32/32	102	32/32	22
V/E: BE	32/32	2617	32/64	5850	16/16	132	**32/32**	**21**
V/E: DS	32/32	2682	32/64	5982	**32/32**	**94**	32/32	22
C/V: k-Way	**32/32**	**2254**	64/128	15217	32/64	304	32/32	24

Fig. 9. Optimal system configurations per graph and partitioning strategy for the Quad query.

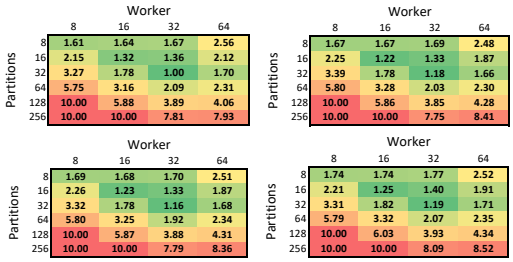


Fig. 10. System configuration heat maps. Quad query on Biblio graph.

Messages per Edge Request	Biblio	Uniprot
1	299,488	971
2	117	970
3	267	294,932
4	837	10.320

Unicast	Broadcast	Final result

Fig. 11. Intermediate results for each edge predicate of the V query

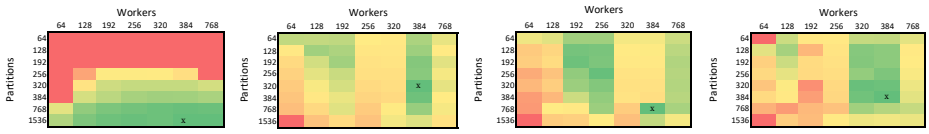
Varying Queries The previous paragraph concluded our test series for the V query. Now we want to show the performance implications of all considered influence factors for a second query type, namely the *Quad* query from Figure 3(b). The results for all system configurations, graphs and partitioning strategies are shown in the heat maps of Figure 9 and Figure 10. The optimal configurations are now always tied to 32 Workers with a varying number of partitions. We see the same run time behavior as for the V query, except for the V/E strategy. The *Quad* query does not hit the vertex outlier partitions (c.f. Figure 4(a)), which enables the BE and DS partitionings to compete with RRV and k-Way. The Shop and Social graphs show an equal slowdown for C/V, compared to the other strategies. However, the Uniprot graph now scales well with the hardware threads, since there are more intermediate results in the Unicast edge predicate.

4.2 Evaluation on a Large-Scale Multiprocessor System

Our large multiprocessor system is an SGI UV 3000 with 64 sockets each equipped with an Intel Xeon CPU E5-4655 v3 and a total of 8 TB main memory. We conducted the same experiments as for Section 4.1 and used gMark to scale up all graphs from Figure 3(c) by a factor of 10 while preserving all other graph properties. All in all, we found that the entirety of our experiments on the large-scale system confirmed our observations from the medium-scale system. Figure 12 illustrates the heat maps for the *Quad* query on the *Social* graph for the SGI system. As for the medium-scale system, we see that using the hyper-threads is also not feasible on the SGI system. However, utilizing all physical cores leads to optimal performance in many cases, which underlines that our processing scales well with the employed hardware. In contrast to the medium-scale system, we see more variations in the heat maps, which is explained by the bigger number of sockets and the increasing influence of the NUMA effect on query performance.

4.3 Lessons Learned

Employing an optimal partitioning strategy is crucial for query performance. To find out the best strategy for a given query, we found that weighing the amount of broadcasts against unicasts, which result from the query pattern, is important.



(a) C/V: k-Way (b) V/V: RRV (c) V/E: BE (d) V/E: DS

Fig. 12. System configuration heat maps. Quad query on Social graph.

Dominant Unicasts It is desirable to partition the graph using a strategy which balances both edges and vertices. We argue that employing the C/V strategy is beneficial, even if there is a minor edge imbalance, since the unicast part of the query will benefit from the locality property of adjacent graph partitions. However, if the edge imbalance exceeeds a certain limit, we suggest switching to the V/V strategy.

Dominant Broadcasts Each partitioning strategy performs well. However it is desirable to achieve a balanced amount of edges between the partitions. As edges represent the amount of data records per partition, balancing them results in a more evenly distributed work in the system. All of the evaluated partition strategies have proven to be viable for graph pattern matching on a data-oriented architecture, except for the E/E strategy because of its broadcast-only nature.

The challenge is to adequately estimate the influences of broadcasts and unicasts due to their dependency on the underlying graph. Our experiments showed, that the optimal system configuration varies among the different workloads. As a rule of thumb, we found that it is beneficial to not use hyper threads in most cases and directly mapping the number of graph partitions to the number of workers.

5 Related Work

Graph processing on NUMA systems is considered by a broad community. There are many studies towards optimizing the data layout for and the execution of a Breadth First Search (BFS) on a NUMA machine, such as Yasui et al. show in [18]. We prove that, unlikely for BFS, it is not the best practice to always utilize the maximum number of available cores, depending on the executed query.

Verma et al. [16] examine different graph partitioning strategies of existing systems and give an suggestion of which strategy to use for specific analytical algorithms. In contrast to the authors, we generally categorize graph partitioning strategies based on their partitioning and balancing criterion. Also, we don't evaluate specific algorithms but whole graph partitioning categories with respect to their influence on the query performance.

Many graph systems like Ligra [12] or Galois [7] often only state that the data will be partitioned and how, but not why. We have shown that using one partitioning scheme for all graphs is not the optimal solution and may result in huge slowdown factors, compared to the possibly best system configuration.

6 Conclusions and Future Work

In this paper, we could show a plethora of dependencies for graph partitioning and processing on NUMA systems. We have proven that there is no one-size-fits-all strategy in terms of a good combination of a system configuration and graph partitioning algorithm out of the box. As outlined in Section 3, we see a need to examine the effects of optimization measures such as vertex or edge replication.

References

1. G. Bagan et al. Generating flexible workloads for graph databases. *PVLDB*, 2016.
2. A. Buluç et al. Recent advances in graph partitioning. *CoRR*, 2013.
3. L. Cheng et al. Efficiently handling skew in outer joins on distributed systems. CCGrid, 2014.
4. S. Decker et al. The semantic web: the roles of xml and rdf. *IEEE*, 2000.
5. S. Hong et al. Green-Marl: a DSL for easy and efficient graph analysis. *ASPLOS*, 2012.
6. T. Kissinger et al. ERIS: A numa-aware in-memory storage engine for analytical workload. *ADMS*, 2014.
7. D. Nguyen et al. A lightweight infrastructure for graph analytics. *SIGOPS*, 2013.
8. H. Ogata et al. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. *Nucleic Acids Research*, 2000.
9. E. Otte et al. Social network analysis: a powerful strategy, also for the information sciences. *Journal of Information Science*, 2002.
10. I. Pandis et al. Data-oriented transaction execution. *PVLDB*, 2010.
11. S. Pandit et al. Netprobe: A fast and scalable system for fraud detection in online auction networks. WWW, 2007.
12. J. Shun et al. Ligra: a lightweight graph processing framework for shared memory. *SIGPLAN*, 2013.
13. M. K. Tas et al. Greed is good: Optimistic algorithms for bipartite-graph partial coloring on multicore architectures. *CoRR*, 2017.
14. T. Tran et al. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. ICDE, 2009.
15. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 1990.
16. S. Verma et al. An experimental comparsion of partitioning strategies in distributed graph processing. *VLDB Endowment*, 2017.
17. P. T. Wood. Query languages for graph databases. *SIGMOD*, 2012.
18. Y. Yasui et al. Numa-aware scalable graph traversal on SGI UV systems. *HPGP*, 2016.