Can Modern Graph Processing Engines Run Concurrent Queries Efficiently?

Matthias Hauck
Computer Engineering Group,
Ruprecht-Karls University of
Heidelberg / SAP SE
matthias.hauck@sap.com

Marcus Paradies
SAP SE
marcus.paradies@sap.com

Holger Fröning
Computer Engineering Group,
Ruprecht-Karls University of
Heidelberg
holger.froening@ziti.uni-heidelberg.de

ABSTRACT

Analytic graph processing has witnessed an ever-growing interest both in industry and academia with the focus on providing the most effective algorithm implementations to maximize single-query performance. In a complex application scenario, where multiple users issue concurrent queries to the analytic graph processing engine, the major performance metric is throughput rather than single-query elapsed time. As of today, there is no single-node graph engine that is designed for concurrent graph processing running multiple queries in parallel.

In this work, we analyze the single-node graph engine Galois and extend it to run multiple graph queries concurrently. We perform an extensive evaluation of Galois for various graph algorithms and data sets to gain a fundamental understanding of the performance bottlenecks of existing graph engines. Finally, we derive important insights and conclude that modern graph engines cannot be easily adapted to handle concurrent graph queries efficiently.

CCS CONCEPTS

•Information systems →Online analytical processing engines; •General and reference →*Performance;* **•Theory of computation →***Graph algorithms analysis;*

KEYWORDS

Concurrent query processing, Graph Processing, Experimental Study

ACM Reference format:
Matthias Hauck, Marcus Paradies, and Holger Fröning. 2017. Can Modern Graph Processing Engines Run Concurrent Queries Efficiently?. In *Proceedings of GRADES'17, Chicago, IL, USA, May 19, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3078447.3078452

1 INTRODUCTION

Analytic graph processing has witnessed a widespread adoption across multiple application domains, including social media, health care, transportation management, and telecommunication.

In practice, graph instances from these domains can consist of billions of edges and use hundreds of GB of main memory. Today even single computing machines can easily accommodate multiple hundreds of GB, even multiple TB, which makes a strong case to support such graph processing within one machine boundary. However, the sheer graph size requires analytic graph algorithms to be processed efficiently, while it is also necessary to utilize all available computing resources using efficient, parallelized implementations of these algorithms.

Especially when executed in parallel, efficient graph algorithm implementations have multiple interesting runtime properties. The most important one is the varying intra-algorithm parallelism. Due to the data-dependent execution behavior, the available parallelism during execution of a query can vary between one and multiple hundreds of concurrent activities [3]. Insufficient available parallelism can limit the potential system utilization especially in modern multi-core server systems. Additionally, there are also algorithmic properties that are sensitive to parallelism. For example, the positive effect of asynchronous execution can decrease due to an increasing number of parallel compute units. Limiting the parallel execution can increase algorithmic efficiency, while it also frees resources.

In classic relational enterprise database environments or web-scale NoSQL environments the systems must handle multiple queries at the same time. Multi-user setups are common, where users can issue queries concurrently to the same system. Additionally, these queries can expose independent parts, which can be executed independently and be issued as batches. There is no reason to assume that analytic graph engines will not be used similarly. A system could run different queries in batch or by different users on the complete graph, and in addition, analytic operations could run on different subgraphs.

Especially in interactive scenarios, users demand a low response time for the queries they issue. According to Little's law from queuing theory requires a low mean response time a high throughput. Typical approaches to improve throughput are to enhance system utilization by increasing the available parallelism or the efficiency of execution. More parallelism can be provided by introducing more intra-query parallelism within each query, or by increasing the inter-query parallelism through the concurrent execution of multiple queries. In contrast to increased system utilization, increased efficiency typically covers algorithmic improvements, optimized resource assignment or a limitation of intra-query parallelism to reduce overheads.

Both approaches are affected by the previously mentioned properties of graph algorithms. To the best of our knowledge, there are only a few publications about concurrent execution of analytic

GRADES'17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5038-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078452

graph queries like [11], especially none that analyses execution behavior. We fill this gap by an analysis of the current state of graph processing engines. As there are no systems built for concurrent query execution, we realize it by executing multiple instances of a state-of-the-art engine in parallel.

The main goal of this work is to give an overview, which implications on throughput arise from executing a state-of-the-art single-node graph processing engine in parallel. To get a first fundamental understanding of concurrent execution of graph queries, we focus on workloads where the input graphs and the executed algorithm are the same for all queries (homogeneous). We present our measurements for a concurrent execution of analytic graph queries with (1) different algorithms at different sizes, (2) with limited intra-query parallelism, and (3) with enhanced thread contention. Based on these measurements we provide an analysis of these three effects and identify issues related to concurrent execution.

2 BACKGROUND

In this study we choose Galois [3] as a representative for a modern graph engine. The first reason for this choice is clearly that it is one of the fastest graph processing engines available [7].

Galois has a programming model [5] that provides a rich tool set to formalize algorithms and to exploit parallelism. Algorithms are implemented as an operator and a strategy to schedule operators for execution. The range of schedulers include simple topology-centric, advanced asynchronous, and data driven schedulers that support priorities. In addition, some of these schedulers can exploit hardware properties like NUMA. The second reason we choose Galois is because this range of schedulers also allows to efficiently support a large range of algorithms.

Based on the insights from the work of Pingali et al. [5], we expect that this programming model exposes in contrast to for example BSP-style programming models a higher variability in the degree of concurrency. This variability in concurrency should result in similar resource usage. Then concurrent query execution could help to increase the overall performance by leveraging unused resources, which is our third reason.

The authors of Galois do not claim that Galois is made or suited for concurrent execution. In fact, the system is optimized for exclusive use of system resources, so it binds threads to CPU cores using `libnuma`[1]. Thread binding is realized by mapping deterministically threads for the same algorithm and identical thread numbers on the same system always to the same cores. This behavior is typically harmful for a concurrent query execution, as it causes a significant interference between different Galois instances.

3 METHODOLOGY

The overarching goal of this experimental analysis is to assess the feasibility of using existing graph processing engines for the concurrent execution of multiple analytic queries. A graph processing engine optimized for concurrent execution would be able to execute these concurrent queries at least without decreasing the throughput. However, one can anticipate that there are various reasons

[1]The use of libnuma make the use of tools like numactl not possible, because libnuma overwrites external bindings. So we are not able to enforce a collision-free thread binding (see also subsection 4.3).

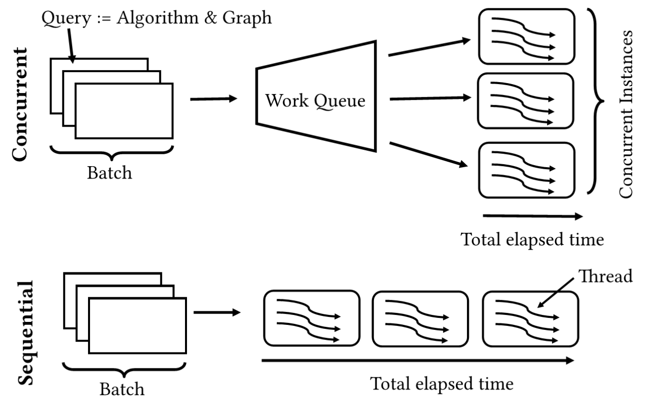

Figure 1: The components of the measurement process for concurrent and sequential query execution

both within the software and the hardware side that can decrease the throughput of a concurrent query execution.

Figure 1 gives an overview of our measuring approach and our terminology. In our definition, a query is an input *graph* and *algorithm* that is applied on the input *graph*. Graph query executions vary in their behavior based on these *algorithms* and input *graphs*. As a representative set of graph algorithm we select PageRank (PR), strongly connected component (SCC), and breadth-first search (BFS) and used Galois example implementations of them in the default variant, except for PR where we had to choose the pull variant.

We generate the input *graphs* using the RMAT graph generator implementation from the PBBS project [8] and use the default parameters ($a = 0.5$, $b = 0.1$, $c = 0.1$, $d = 0.3$). The generated data sets[2] are SMALL ($|V| = 0.13\,\text{M}$, $|E| = 1.92\,\text{M}$), MEDIUM ($|V| = 1.05\,\text{M}$, $|E| = 19.6\,\text{M}$) and LARGE ($|V| = 16.7\,\text{M}$, $|E| = 198\,\text{M}$). The sizes of the data sets are similar to the data size in productive environments.

We measure performance as throughput in terms of executed *queries* per second, but for readability we rather report the inverse of this metric, which is the elapsed time per *query*. To measure the elapsed time, we execute all queries for a combination of *algorithms* and input *graph* as a *batch*. So for *sequential* execution the elapsed time per *query* is equal to the mean time spent on one *query*, while for *concurrent* execution it is the calculated fraction of the *total elapsed time* spent on a *query* in a *batch*. The *total elapsed time* is measured from the start of the first query in a *batch* until the last query has completed. This means that the measurements include not only the *query algorithm* execution but also initialization, data load and shutdown of the engine. In our opinion this is also the typical usage scenario for such a system in a productive environment.

In our setup we have two types of parallelism: inter-query concurrency, due to *concurrent* execution of multiple Galois *instances*, and intra-query parallelism, as the query-internal parallel processing uses multiple *threads*. When multiple *queries* are executed *concurrently*, the *batch* size refers to the number of *concurrently* executed *instances*. A *concurrent work queue* (GNU parallel [9]) executes

[2]The number of vertices is automatically rounded up to the next power of two by the generator.

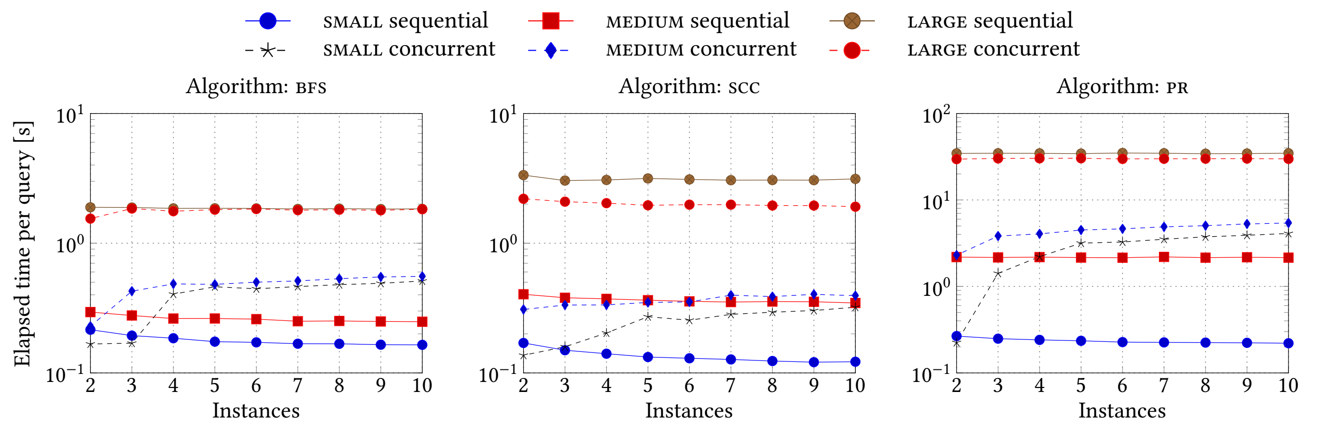

Figure 2: Elapsed time of a single query out of a batch using different algorithms on a homogeneous workload using all cores without thread binding.

the query *batches*. We evaluate 2 to 10 *concurrent instances*, but later report only a part of them, because the rest of the measurements give no additional insights. Apart from inter-query parallelism, which we control by the maximum number of *concurrent instances* the *work queue* can issue, we control intra-query parallelism with a *thread* limit that we pass to the *instances* as a command line argument. In our experiments we deactivate *thread* binding using an environment variable. We chose for BFS the same start node, so the work of the same configuration is always identical and it ensures that not an isolated vertex is selected.

3.1 System configuration

We use a dual-socket Linux-based system (SLES 11 SP 4) with Intel Xeon X5650 (Westmere) CPUs. Our system is equipped with 2×6 cores @2.66 GHz with Hyper-Threading enabled, 12 MB last-level cache for each socket, and 48 GB DDR3 RAM @1333 MHz. The CPU frequency scaling governor was set to “on demand”. We compile GALOIS with GCC 6.2 and Atlas 3.10.2 as a BLAS/Lapack library together with the optimization flags `-O3` and `-march=native` which enables the compiler to use the complete instruction set of the CPU. For time measurement we use the GNU tool `date` with nanosecond resolution.

4 OBSERVATIONS FOR HOMOGENEOUS GRAPH QUERY WORKLOADS

In this section we analyze the execution of homogeneous concurrent graph workloads where multiple queries of the same type and on the same input graph are executed concurrently. Even though, we believe that this is not the typical case of concurrent execution of graph queries, we also believe it to be important for a first fundamental understanding of interference effects.

4.1 Impact of input sizes and algorithms on concurrent throughput

In figure 2 we depict the experimental results for our analysis of homogeneous graph workloads for different graph algorithms and input graphs. We limit the number of threads each query can use to the number of CPU cores, so each of the concurrent instances could potentially use all available resources in the system. For each algorithm, we vary the input size and report the elapsed time per query for sequential and concurrent execution up to 10 concurrent instances.

In general, all three algorithms behave similar for different input sizes when executed concurrently, but the extent of performance (dis)advantage concerning their behavior varies. Our major observations are the following: 1) When executed in parallel, we see especially for small input sizes a substantial negative effect on the mean execution time. 2) With an increasing input size this performance disadvantage decreases and turns into an improvement. 3) With low concurrency degrees it is possible to achieve a throughput improvement. 4) With an increasing number of concurrent instances for smaller input sizes, the elapsed time increases until a plateau is reached. Additionally, for at least for SMALL and MEDIUM, this plateau is at a similar level for all algorithms. In detail, we observed the effects below.

Observation 1: We see for all algorithms for concurrent execution a penalty. For smaller input graph sizes the penalties are much higher than for larger. Algorithm BFS has a penalty between 118% and 210% for SMALL on 4 up to 10 instances, and between 54% and 124% for MEDIUM on 3 up to 10 instances. For PR the penalty is even larger, with 470% and 1760% for SMALL on 3 up to 10 instances, and between 77% and 152% for MEDIUM on 3 up to 10 instances. Algorithm SCC shows a behavior that favors concurrent execution: while for SMALL a concurrency penalty between 6% to 163% for 3 up to 10 concurrent instances exist, it changes to an improvement between -14% up to 24% for MEDIUM

Observation 2: At the largest input size, LARGE, the difference between sequential execution and concurrent execution diminishes. Algorithm BFS needs 1% and 5% less time on average for concurrent execution with an outlier of 18% for two instances, while PR has even an advantage in between 12% and 15% compared to a sequential execution. For SCC we see even an improvement of 31% up to 39%.

Observation 3: For a low degree of concurrency, we observe as a recurring pattern that for two concurrent instances, for BFS on SMALL also for three instances, the elapsed time is much better than

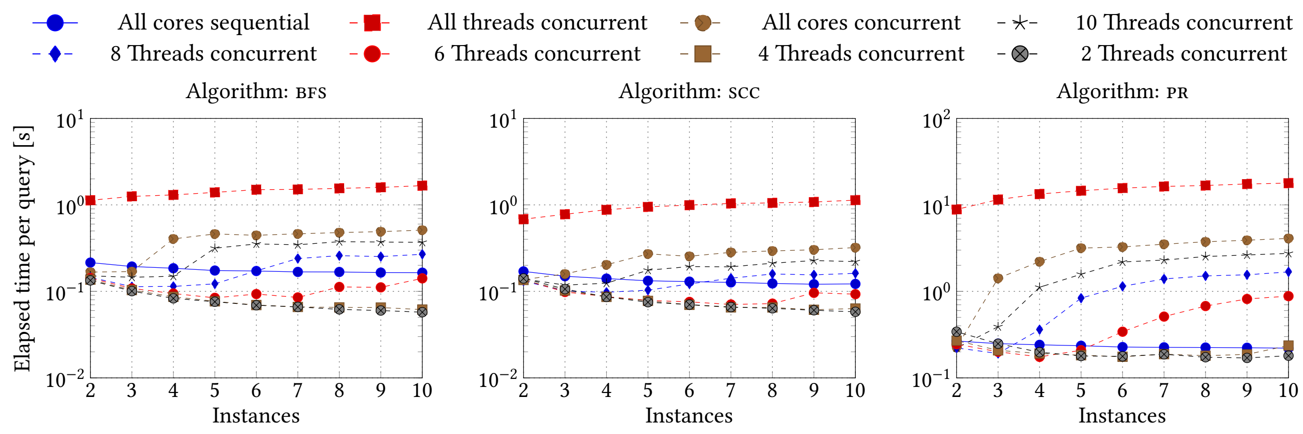

Figure 3: Elapsed time per query out of a batch on a SMALL graph with thread binding and an upper thread limit per query

for configurations with more concurrency. For the SMALL measurements this could be within the error of measurement, because of the small elapsed time for these experiments, but still this pattern appears for all algorithms and input sizes.

Observation 4: As already mentioned, we observe a plateau effect for concurrent execution. For BFS the time difference from SMALL to MEDIUM at 10 instances is only 8.5%, while the sequential difference is 51% and the difference from MEDIUM to LARGE is 229%. When PR is used, the relative differences are larger, but the effect remains. The difference from SMALL to MEDIUM is for 10 instances only 32%, while the sequential difference is 876% and the difference from MEDIUM to LARGE is 451%. The relative elapsed time differences for SCC are between BFS and PR.

4.2 Limiting intra query parallelism

In the previous experiment every query could use a number of threads that is equal to the number of available physical cores. Because the instances that execute the queries run independently from each other, they are not cooperative, which can lead to thread contention. In the measurements for figure 3 we limit the number of threads for concurrent execution to decrease the effect of the thread contention between different instances.

We observe that all three different algorithms show some common behavior. There is a significant penalty for concurrent query execution, when each query uses all available threads of the system (including Hyper-Threading threads). For lower thread bounds up to the number of physical cores the system behaves differently. Especially high thread counts show a mean elapsed time that constantly rises until it ends up in a plateau. If the thread count is lower, also the plateau is lower at a high degree of concurrency. At low thread counts we see also a falling slope for low degrees of inter-query parallelism, which might be caused by insufficient parallelism.

The point when the slope starts and when it reaches the plateau seems to be algorithm-dependent. We list the starts of the slope and plateau for BFS and PR in table 1. For SCC the start of the plateau is hard to see, because the slope is too shallow. It seems that the start of the slope and start of the plateau are related to algorithm-dependent thread numbers. The spread of the elapsed times at a given concurrency degree is also for PR larger than for BFS followed by SCC with the lowest spread.

4.3 Binding threads to physical cores

By default, GALOIS uses thread binding. For the previous measurements, we deactivated it, because the thread binding assigns the threads from concurrent queries to the same physical CPU threads. Our assumption is that this behavior causes interference between concurrent queries. Figure 4 illustrates the effect of thread binding.

For sequential execution the effect of thread binding is algorithm-dependent. In general, the positive effect of thread binding increases with the size of the graph. PR profits from thread binding in general (mean improvement MEDIUM: 12% LARGE: 19%) except for small graphs (mean penalty SMALL: 23%). SCC on the other side does not profit from thread binding (mean penalty SMALL: 43%, MEDIUM: 21%, LARGE: 0%)

For concurrent execution has thread binding obviously a negative effect as expected. In addition, we see for concurrent an input size behavior as for sequential execution. With increasing input size decreases the difference to the execution without thread binding. But for SCC the elapsed time gap remains even for LARGE, which is with a difference of 26% significantly larger than for the sequential execution case or PR with only 10%.

	Start of slope			Start of plateau			
	12 T	10 T	8 T	6 T	12 T	10 T	8 T
BFS	3 (36)	4 (40)	5 (40)	7 (42)	4 (48)	5 (50)	7 (56)
PR	2 (24)	2 (20)	3 (24)	4 (24)	5 (60)	6 (60)	8 (64)

Table 1: Points, where in figure 3 for a given number of threads (T) the elapsed time starts to increase (Slope), with a high rate and where this increase stops (plateau). The first number is the number of instances, the second is the number of total threads in the system.

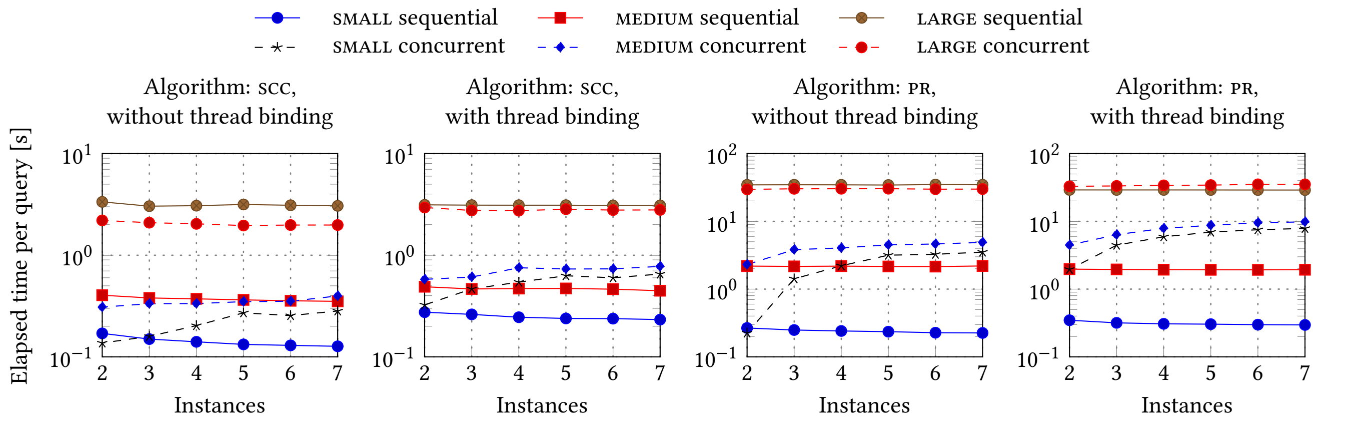

Figure 4: Elapsed time of a single query out of a batch using SCC or PR on a homogeneous workload using all cores with and without thread binding

5 DISCUSSION

We showed in section 4 the impact of concurrent graph query execution for different input sizes and algorithms. In addition, we also reported on effects when limiting intra-query parallelization and fostered thread contention.

The most significant observation we make is that we could not improve the throughput by executing queries concurrently, or even have a throughput comparable to sequential execution, when we use the engine with default parameters. As mentioned as observation 1 in section 4.1, we see for higher numbers of concurrent instances contention effects for all algorithms, but in particular for low and medium input sizes these contention effects are strong.

In section 4.2 we unveiled that this effect depends on the total number of active threads in the system. Up to a query specific system wide number of threads the elapsed time is low until it starts (slope start) to grow with the concurrent instances. Before this point, there can be a performance improvement through concurrent query execution. As we see in table 1 can for BFS at this point the system-wide accumulated thread limits be larger than the number of CPU threads, but for PR it is exactly the number of CPU threads.

The reason might be the different structure of these algorithms. PR is a topology-centric algorithm and can easily saturate all CPU threads, and when it uses more threads than the CPUs provide these threads would interfere through context switches. BFS on the other side is a data-driven algorithm, with a varying amount of work per thread during execution. In general, this behavior allow more worker threads than CPU threads to use without interference effects, because contention effects are small. In the scenario of section 4 there should be no such effect. All queries of a batch are completely equal, so all queries should have had the same time the same resource demand. Nonetheless, there is naturally some randomness in the execution that might allow the use of variable resource demand.

The increase of the mean elapsed time caused by thread contention ends at a larger total number of threads at an upper limit (plateau). After this point, there is no further increase of the elapsed time, so the maximum point of inefficiency is reached. This effect seems not to be bound directly to the number of threads otherwise there would be no plateau. As we saw in section 4.3, multiple threads executed on the same core always have a negative effect on concurrent execution. An explanation for this behavior could be that cache thrashing caused by thread context switches reach at this point its maximum that is limited by the time slice the threads get. Other publications [1, 4] already show that caching has a significant impact on graph algorithms, even when they are executed alone, so an enhanced effect through concurrent execution is not surprising.

Unexpected for us is the high elapsed time in figure 3, when in the concurrent case the queries can use Hyper-Threading cores. There is no similar behavior for sequential execution and for lower thread limits in situations with similar total thread counts. This behavior requires further analysis.

We also observe that the negative effect of concurrent execution decreases with increasing input data size for all algorithms (section 4.1 observation 2). This effect might be an artifact from the fact with increasing input size, the processing throughput becomes lower in general. In figure 2 the difference between SMALL and MEDIUM is smaller than the difference between MEDIUM and LARGE. This slowdown might be caused by a decreasing benefit of caching, because with an increasing size less of the graph and the data structures that are used by the algorithm fit in the cache. Concurrent execution of queries might enable the system to increase the utilization of the memory controller, what can increase the throughput.

Regardless what causes the effect, is it worth to look at 'small' graphs? In our opinion yes, because very often users are mainly interested in subgraphs. For example, a user that wants to analyze relationships from a subset of the users of a social network similar to LDBC SNB data sets [2] would only select the person-knows relationship that is significantly smaller than the complete graph. We expect that such kinds of selections are common on multi-user graph processing systems.

Nonetheless, the performance improvement we saw for large graphs and with an adaptation of runtime parameter (section 4.1 observation 2 & 3 and section 4.2) are promising.

6 RELATED WORK

There are only few publications about graph engines that are tailored to concurrent graph processing. In the relational database community concurrent query execution is common, especially in multi-user scenarios, and several approaches have been proposed to increase the query throughput in such scenarios.

Exemplary for the research on concurrent query execution in analytical workloads is the work by Psaroudakis et al. [6]. The authors compare two work sharing techniques, namely simultaneous pipelining and global query plans with shared operators. Simultaneous pipelining allows sharing common sub plans between queries and global query plans with shared operators allow sharing work between similar—but not necessarily equal—queries. The authors conclude that both techniques are orthogonal and can be gainfully combined except for low concurrency degree scenarios, where system resources are not fully saturated. In this case the sharing overhead is not outweighed by the potential performance improvements.

Seraph [11] is a distributed graph processing system tailored to concurrent query execution. The system addresses the problem of a high memory consumption of concurrent graph analytic queries by allowing multiple jobs that work on the same graph to share the graph representation. A copy-on-write approach allows the graph query to modify the graph while working on it—without any interference to other concurrently running graph queries. The reuse of the input graph allows increasing the resource utilization in a memory-constrained environment by executing a larger number of graph queries concurrently. In addition, Seraph contains a job scheduler that increases the resource utilization while limiting thread contention. Each job in Seraph consists of a set of work item bundles called *task*. The scheduler submits the task of a job to execution, with a rate limit based on a priority that increases with the age of the job.

While not explicitly designed for concurrent graph query execution, there are several approaches that could be beneficially used for concurrent graph query execution. One example is the multi-source breadth-first traversal (msbfs) by Then et al. [10], which shares the work of multiple BFS runs executed on the same graph. In practice, several graph centrality measures, such as closeness centrality, are based on multiple BFS runs from different start vertices. The ms-bfs algorithm accelerates such graph centrality computations by sharing the work between concurrent BFS runs. A generalization of msbfs to multiple concurrent graph queries could lead to similar results like operator sharing in the relational world.

7 CONCLUSION

In this work we analyzed the effect of concurrent graph query execution using the Galois system on homogeneous graph workloads. We report the elapsed time for the execution of different workloads using different algorithms and graphs of different size at several degrees of concurrency. In addition, we analyze combinations of different degress of intra-query and inter-query parallelism and the effects of enforced thread contention through thread binding.

From the measurements of this study we derive several insights. No throughput increase based on concurrent query execution can be achieved in most cases. Even worse, a concurrent execution of queries often harms throughput, especially when the input graph is small. Small input sizes could be the result of a subgraph selection, which occurs frequently in multi-user scenarios where users are interested only in parts of the complete graph. An adaption of configuration parameters, such as limiting the degree of intra-query parallelism, often exhibits significant performance improvements. Therefore, the performance degradation is not a problem of concurrency in general, but rather a problem of running multiple queries concurrently.

As a result of this study, we envision that a graph processing engine that wants to efficiently handle multiple concurrently running graph queries, the system architecture should be extended to cope with these new system requirements.

In practice, it is not feasible to force users to perform manual parameter tuning to avoid interference effects. A graph engine that is able to perform concurrent query execution needs to take into account effects that might be caused by caching, varying available parallelism or workload inhomogeneity, to achieve a throughput that is better than single query execution. Beyond optimization of simple concurrent executions, also advanced techniques are applicable, for instance similar to work sharing as it is used in the relational world.

We focus in this work on homogeneous graph query workloads, while in real-world scenarios inhomogeneous graph query workloads dominate. Therefore, we believe that further research about the interference for inhomogeneous workloads is necessary. An important part of this investigation could be the extension of graph processing benchmarks for concurrent query execution.

REFERENCES

[1] S. Beamer, K. Asanovic, and D. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015.

[2] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M. D. Pham, and P. A. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proc. SIGMOD'15*, 2015.

[3] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013.

[4] M. A. O'Neil and M. Burtscher. Microarchitectural performance characterization of irregular gpu kernels. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014.

[5] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46. ACM, 2011.

[6] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *Proceedings of the VLDB Endowment*, 6(9), 2013.

[7] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014.

[8] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012.

[9] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1), Feb 2011. doi: http://dx.doi.org/10.5281/zenodo.16303. URL http://www.gnu.org/s/parallel.

[10] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[11] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.