Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines

Peitian Pan and Chao Li
Department of Computer Science and Engineering, Shanghai Jiao Tong University
{ptpan2014, chaol}@sjtu.edu.cn

***Abstract*—Parallel graph processing has been routinely used to solve problems in a wide range of real life applications today. Efficiently handling many concurrent graph processing queries in a multi-user environment is highly desirable as we enter a world full of edge device oriented services. Existing research, however, primarily focuses on processing a single, large graph dataset and leaves the efficient handling of multiple mid-sized graph queries an interesting yet challenging open problem.**

In this work, we investigate the management of multiple graph processing queries on shared-memory machines. We carefully analyze the performance of several graph algorithms and find some of them do not scale well with more CPU cores (more threads) while others have diminishing returns as the number of CPU cores grow. Further, we identify the trade-off relationship between the increasing cost of atomic operations and the growing available memory bandwidth with more CPU cores. Motivated by the above observations, we propose *Congra*, a dynamic graph scheduler that intelligently manages multiple concurrent graph queries for better system throughput and resource efficiency. *Congra* collects the memory bandwidth consumption and atomic operations characteristics of graph queries via offline profiling and decides which subset of CPU cores for the query to run, as long as the query is not blocked due to heavy resources contention. We implement *Congra* in C++ on top of the Ligra graph processing framework and test it with judiciously selected graph processing query combinations. Our results show the *Congra* improves query throughput by 60% compared to existing designs. It exhibits much better quality of service and readily supports a scale-out computing environment.

Keywords— Concurrent graph processing, query scheduling, memory bandwidth, efficiency, throughput

I. INTRODUCTION

Computation over graph big data has become a hotspot due to its wide application in social network, web page analysis, and many other fields. Many real-world problems can be modeled by a graph, which is a set of vertices (entities/users) and edges (links between entities/users). In recent years, a number of system frameworks have been proposed to provide graph-specific optimizations, with an emphasis on effectively handle single, large-scale graph dataset [1, 2].

While existing systems can effectively handle a single graph processing query, they unfortunately provide very little support for processing multiple graph queries. Today, it is not unusual for graph processing engines to serve requests in a multi-user environment. For example, ubiquitous edge devices running smart applications may need to continuously correlate data with other devices to provide context-aware services [3]. Multiplayer mobile games need to monitor and react on the status of each player and their interactions [4]. A group of analysts may issue multiple graph queries to analyze financial patterns or customer behaviors through pattern matching algorithms such as subgraph isomorphism detection [5].

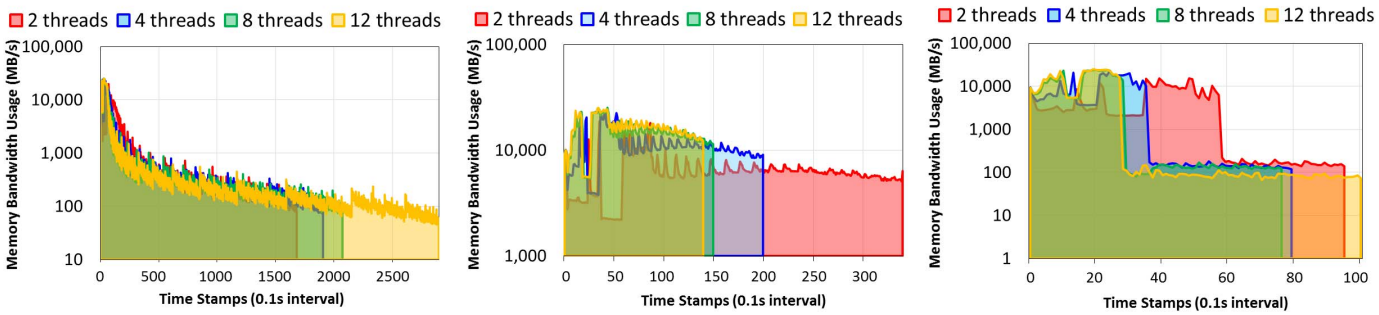
The naïve way of handling multiple queries is to create one process for each query and to run them at the same time. Unfortunately, this does not work well for large amount of graph processing queries. It is generally accepted that memory bandwidth is the bottleneck for large-scale graph processing. For multiple graph processing queries, it can cause severe memory bandwidth contention and significantly degrade the response time for all queries that are running in parallel.

Processing of many concurrent graph queries can be a daunting task. It is hard to predict the memory bandwidth consumption characteristics of a specific graph processing application without prior knowledge. Applications that features certain degree of data exploration (e.g., BFS) tend to have decreasing number of vertices as the number of iterations grow. In other words, these applications may consume less memory bandwidth over time. Some computation-intensive graph applications, on the other hand, iterate over the same set of vertices until numerical values converge. These applications are observed to consume constant memory bandwidth.

More importantly, the scalability of graph applications, which indicates whether the performance improves as the number of computing threads assigned to it grows, can vary across algorithms and even input graphs. For example, *BellmanFord*, a single source shortest path algorithm, scales poorly in graphs that are randomly weighted. We also noted that Breadth First Search (BFS) does not scale well on some graphs, due to the large number of small connected components. Again, it is hard to predict how well some queries may scale because of the complicated interactions between algorithm, systems, and hardware.

In this paper we propose *Congra*, a novel scheme for scheduling concurrent graph processing queries on shared-memory based systems. In general, *Congra* extends existing shared-memory graph processing framework in two aspects. First, it enables conventional single-graph oriented designs to serve multiple users through resources-aware colocation of graph queries. Second, it allows shared-memory based systems to be efficiently organized as a cluster and automates QoS-aware load balancing of large-amount of graph queries.

We have built a prototype of *Congra* based on Ligra, a lightweight graph processing framework on shared-memory machines. The system profiles each query while it is offline. During online phase, it effectively schedules all graph queries by considering their estimated execution time, memory bandwidth requirement, and scalability, etc. We implement all the components of *Congra* with about 700LOC in C++, and



(a) BellmanFord (b) K-Core (c) BFS

Fig. 1. Memory bandwidth usage traces of different algorithms (graph data input: rMatGraph10m)

another 200LOC for extended support for cluster-wide scheduling. While *Congra* is implemented on top of Ligra, it is largely orthogonal to the underlying infrastructure. It can be easily extended to support a variety of shared memory graph processing system framworks.

This paper makes the following contributions:

- We characterize the resource consumption behavior of graph processing query on shared-memory machines and describe the design trade-offs in such system.
- We propose *Congra*, a novel scheduling scheme for improving resource utilization and query throughput on shared-memory systems in a multi-user environment.
- We implement all the components of our design on Ligra. We also extend our design to support cluster-level query scheduling in a scale-out environment.
- We evaluate our technique and system with carefully selected combinations of representative graph processing algorithms and input graphs.

The rest of this paper is organized as follows. Section 2 introduces background. Section 3 proposes the framework and algorithm for scheduling multiple graph processing queries. Section 4 presents our implementation details. Section 5 describes evaluation methodology followed by Section 6 showing our experiment results. Finally, Section 6 discusses related work and Section 7 concludes this paper.

II. BACKGROUND AND MOTIVATION

In the past several years, most researches are focused on processing a single, large graph analytic workload (e.g., Web data, social network, and large scientific datasets) [1, 2]. Many of the proposed schemes in this scenario adopt a message-passing communication model rather than shared-memory.

Processing small or mid-sized graph queries, however, requires a different design approach. It is preferable to analyze the graph on a single node since memory access incurs much smaller overhead than contemporary networked systems. By leveraging query-level parallelism on a cluster of independent shared-memory machines, one can efficiently handle large amount of concurrent graph queries. With growing installed node memory capacity, shared-memory systems have great potential in improving the quality of service (QoS) of graph processing in a multi-user environment.

Graph applications involve heavy communication between memory and the processor. Memory bandwidth is therefore the bottleneck for efficient graph processing. Traditional scheduling method, which runs all queries at the same time with all threads, leads to severe contention in memory bandwidth. Although memory bandwidth is fully utilized under traditional scheduling, resource contention degrades the response time of all queries.

Our goal of efficient processing on many concurrent graph queries is two-fold: 1) the system should maximally utilize its installed memory bandwidth to increase the cost efficiency; 2) it is important to avoid performance degradation due to parallel execution of independent queries. To realize this goal, one must be able to identify any available memory bandwidth and optimally co-locate selected graph queries to make the best use of the resource. This can be challenging for two reasons:

A. *Opaque Capacity*

The possible memory bandwidth savings during runtime is often opaque to the system. Since each graph query can be served by different numbers of logical cores (threads), it is possible that a graph query receives over-provisioned capacity. To demonstrate this, we show the memory bandwidth usage trace of three graph applications in Figure 1 and present their average memory bandwidth usage in Figure 2. Our results reveals that keeping using the minimum (e.g., 2 threads) or maximum (12 threads) configurations does not always yield a desirable execution time. Unfortunately, existing graph frameworks have little knowledge of the potential memory bandwidth it can exploit through optimal resource allocation

B. *Variable Property*

The resource management problem become more acute for multiple graph queries that have diverse behaviors. First, the job execution behavior is affected by the type of graph algorithm. Figure 3 shows the impact of thread assignment on average job execution time. It shows that execution time of certain graph applications (e.g. *Components* and *K-Core*) are monotonically decreasing with increasing number of assigned threads. This is because more threads lead to higher memory bandwidth utilization and thus shorter execution time. The most interesting result is *BFS*, which shows that neither 2 threads nor 12 threads can yield the best execution time. In addition, the type of data input (graph instance) can also greatly changes the behavior of workload as well. For example, *BFS* shows totally different performance scaling trend in Figures 3(a) and 3(b) .

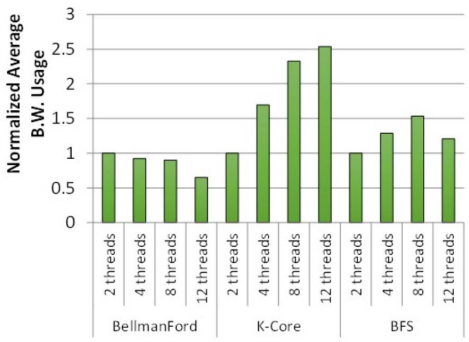
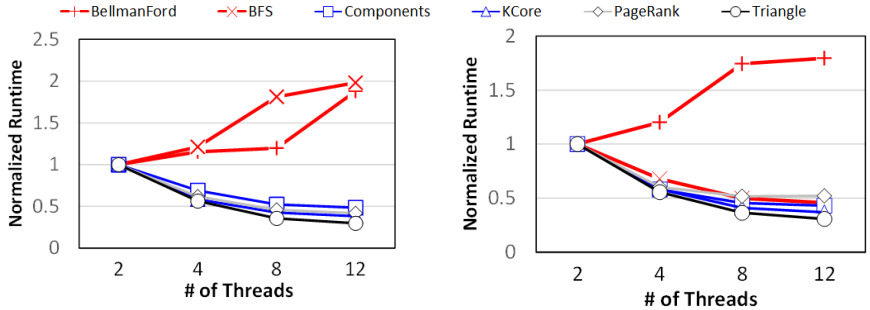


Fig. 2. Average bandwidth usage



(a) citePatents (b) randLocal10m

Fig. 3. Total execution time

III. CONGRA SYSTEM DESIGN

Efficiently assigning threads and collocating tasks can improve utilization without affecting performance. A better scheduling method should try to avoid severe memory bandwidth contention and still preserve high bandwidth utilization. In this section we introduce *Congra*, our scheduling module for multiple concurrent graph queries. Figure 4 describes the scheduling process in pseudo code.

We divide our scheduling algorithm into two phases: the offline profiling phase and the online scheduling phase. In offline profiling phase, we profile all combinations of algorithms and input graphs we have in our database. In online scheduling phase, we schedule all queries according to their expected running time, memory bandwidth requirement, and scalability. We also use hardware performance counters at run time to provide feedback to the scheduler.

A. *Offline Profiling*

The difficulty of accurately predicting the memory bandwidth characteristics and scalability can be easily handled by profiling. We use hardware performance counters at the integrated memory controller (IMC) to get the detailed memory bandwidth usage characteristics of each query. We run each query with varying number of threads to determine its scalability. Queries with shorter running time on fewer threads are deemed scalable, otherwise they are not scalable.

B. *Online Scheduling*

Our scheduler first sorts all graph processing queries by their expected running time in ascending order. Afterwards, it puts them in a queue and schedules each queue in turn. If there is enough bandwidth for current query, the system will run it either with *MAX_THREAD* number of threads (if it's scalable) or *MIN_THREAD* number of threads(if it's not scalable). Otherwise the scheduler waits until some query/queries is/are finished and there is enough memory bandwidth.

C. *Cluster-Level Scaling*

Congra allows dynamic node scaling to support increased graph queries. Our current design assumes a homogeneous cluster, namely, all the nodes are identical. Every node in the cluster is a shared-memory machine that supports *Congra*'s offline profiling and online scheduling discussed above.

Input: Ligra query stream *qs;* periodically and asynchronously updated memory bandwidth usage *cur_bw;* memory bandwidth threshold *threshold*

```
procedure schedule(qs)
  batch = 0
  forever loop
    queryQueue = {}
    for next query q_i in qs do // add queries to query queue
      if batch*L <= arriving time of q_i < (batch+1)*L then
        queryQueue = queryQueue ∪ {q_i}
      else
        break
      endif
    end for
    for each unprofiled query q_i in queryQueue do
      send(q_i, profiler) // make sure every query is profiled
    end for
    if recv(profiler) == "All queries profiled" then
      read_profiling_table(q_i, time_i, avg-bw_i, threads_i)
    else
      raise exception "queries unprofiled"
    endif
    sort(queryQueue, ascending, time)
    // schedule the queries in query queue
    for query q_i at head of queryQueue do
      pop_head(queryQueue)
      while(avg-bw_i + cur_bw > threshold) do
        wait till finished queries free enough bandwidth
      end while
      fork a new process and run q_i with threads_i threads
    end for
    batch = batch + 1
  end forever loop
```

Fig. 4. Single machine scheduling algorithm

For the cluster-level management, a global coordinator and a global job queue are required. During runtime, the global coordinator keeps monitoring the resource utilization status on each node (mainly memory bandwidth). Once the query response time on current active nodes has available capacity, the global coordinator will add queries to the local job queue of a selected node from the global queue. Similarly, if all the current active nodes are running out of resources, the global coordinator will bring up a new node for offloading. Note that we do not perform task migration. It is possible to have very large metadata on the Ligra framework and live migration can significantly degrade the overall performance.

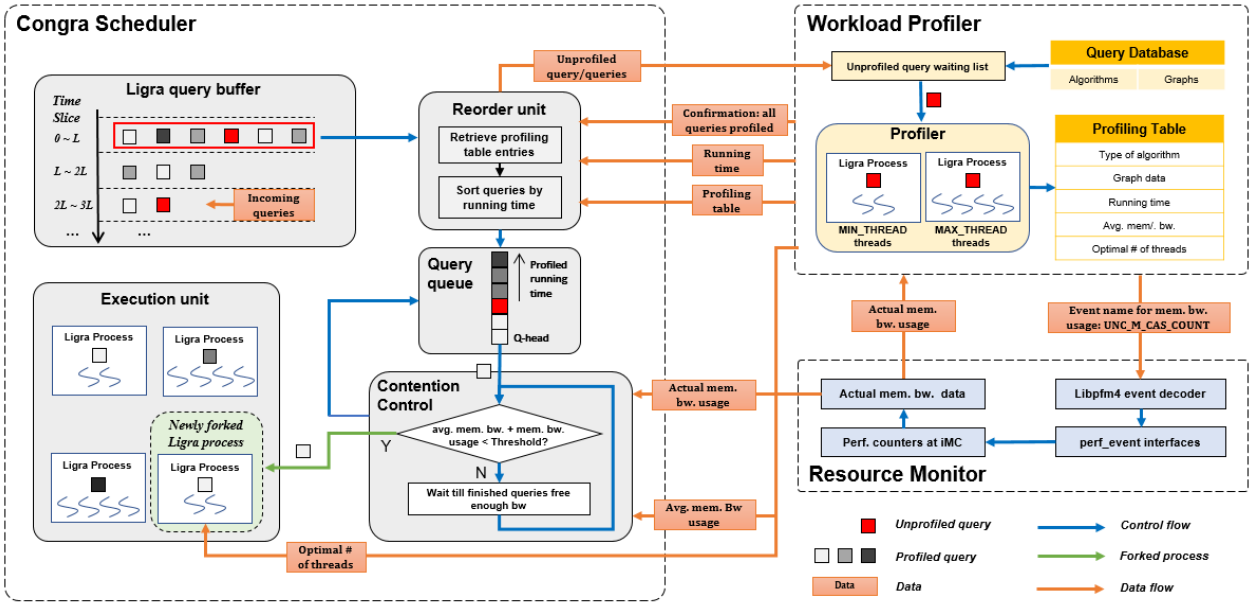


Fig. 5. System architecture of *Congra* (online phase)

IV. Implementation

This section provides more details of our prototype. Figure 5 shows a schematic diagram of *Congra*'s system architecture. Our prototype mainly consists of three components: the *Congra* scheduler, a workload profiler and a resource monitor.

The *Congra* scheduler is the crux of our system. Its basic control flow has been discussed in Section 3. In the following we mainly introduce the resource monitor and workload profiler, and the interaction between different components.

The resource monitor distills crucial information for the system to make informed decisions. It uses *libpfm4* library to translate human readable performance event names to machine readable event code. The monitor then uses the *perf_event* interfaces of Linux to access hardware performance counters. The performance event names used in the memory bandwidth monitor, as is suggested by Intel [7], is *hswep_unc_imcX::UNC_M_CAS_COUNT*, which counts the number of DRAM CAS commands recorded at integrated memory controller (iMC) at CPU socket X.

The resource monitor periodically fetches the readings of performance counters (once per 0.1s) and records it in *BW_USAGE* file. *BW_USAGE* will be used later by the workload profiler to calculate average memory bandwidth usage. When used together with the *Congra* scheduler, the monitor converts readings of performance counters to an integer *CUR_BW* and raises user-defined signal *SIGUSR1* to deliver this integer to the scheduler.

When running or profiling a Ligra query, the number of threads of its process is changed by calling *sched_setaffinity()*. This system call restricts the process to run on a certain subset of logical cores, and therefore with certain number of threads. The subsets of logical cores are selected in a manner where all logical cores are treated as a cyclic queue, in hope that the load is evenly shared among all logical cores.

The selection of memory bandwidth threshold is obtained by taking average of *BW_USAGE* file of STREAM memory benchmark. Note that memory bandwidth monitor is an independent module and can provide memory bandwidth usage information for applications other than Ligra queries. The factor choice means to fully utilize memory bandwidth especially when memory bandwidth usage of queries drops significantly over time.

We implemented all components of the single machine scheduler with about 700LOC C++ code, and about 900LOC C++ code for the cluster version. We run all test cases with Ligra compiled from a gcc 4.8 compiler with Cilk plus support. Since the *BellmanFord* algorithm requires the graph to be weighted, the input graphs to such queries are randomly weighted with *adjGraphAddWeights* utility from Ligra.

V. Evaluation Methodology

We setup a four-node Linux cluster connected via Gigabit Ethernet. Each node is a single-socket machine that uses an Intel E5-2620v3 2.4GHz processor (6 cores, 12 threads) with 32 GB of shared RAM and 1TB disk. We experiments with a variety of graph query sets and different scheduling schemes.

A. Graph Query Set

We use five widely used graph instances in our experiments. Our experiment includes real world graph datasets from a public data repository [7]. LiveJournal is a very irregular structure and citePatents is relatively regular. We also used a RMAT graph generator [8] to generate synthetic graphs that follow a power-law distribution like many realistic graphs. Table 1 summarizes the properties of all the graph dataset evaluated in this study.

TABLE I. DATA SETS USED IN EVALUATION

Graphs		|V|	|E|	Description
GL1	LiveJournal	4.0M	34.7M	social network
GL2	rMatGraph10m	16.8M	100M	power graph
GL3	randLocal10m	10.0M	100M	random graph
GS1	citePatents	3.8M	16.5M	temporal, labeled
GS2	roadCA	2.0M	2.8M	road network/grid

TABLE II. GRAPH ALGORITHMS USED IN EVALUATION

Algorithms		Description	Property
AH1	PageRank	measuring the relative importance	Bandwidth-consuming
AH2	KCore	computes the k-core decomposition	
AH3	BellmanFord	computes the shortest path	
AL1	BFS	traverse a graph, breadth-first	Bandwidth-conserving
AL2	Components	strongly connected components	
AL3	Triangle	counts the number of triangles	

TABLE III. QUERY SETS USED IN EVALUATION

Query Set	Algorithms	Input Graphs	Alg. Heterogeneity	Data Property
Heavy	{AH1, AH2, AH3}	{GL1, GL2, GL3, GS1, GS2}	A mix of similar graph alg.	A mix of different graphs
Light	{AL1, AL2, AL3}	{GL1, GL2, GL3, GS1, GS2}		
Heter-L1	{AH1, AL1, AL2}	{GL1, GL2}	More heterogeneous	Large graph only
Heter-L2	{AH2, AH3, AL3}	{GL1, GL2}		
Heter-S1	{AH1, AL1, AL2}	{GS1, GS2}	More heterogeneous	Small graph only
Heter-S2	{AH2, AH3, AL3}	{GS1, GS2}		
Homo-1	{AH1}	{GL1, GL2, GL3, GS1, GS2}	More homogeneous	A mix of different graphs
Homo-2	{AH2}	{GL1, GL2, GL3, GS1, GS2}		
Homo-3	{AH3}	{GL1, GL2, GL3, GS1, GS2}		

TABLE. IV. GRAPH ALGORITHMS USED IN EVALUATION

Schemes	Descriptions	Parallelism
Private	Give dedicated node to each graph query	Low
Shared	Use OS to run multiple graphs in parallel	High
Congra	BW-aware multiple graph scheduling	Medium

We select six representative graph processing applications in Ligra and roughly divide them into two categories based on multiple trials of experiment (Table 2). Three algorithms including *PageRank*, *K-Core* and *BellmanFord* are observed to be highly memory bandwidth-consuming. The other three are relatively light-weight in terms of resource consumption.

To evaluate concurrent graph processing environment, we have devised 9 query sets, as shown in Table 3. Each query set are specified by N input graph algorithms and M input graph instances. It contains all the ordered pairs (e.g., $N \times M$) of {Algorithm, Graph}. For example, the *Heavy* query sets features 15 queries generated by 3 resource-consuming graph algorithms and all the 5 graph instances. We assume all the queries arrival simultaneously at each time stamp.

B. Evaluated Schemes

We compare Congra with two strategies: *Private* and *Shared*. The former scheme is close to existing single-graph based processing (e.g., original Ligra framework). It aims to avoid resource contention and runs each query separately in a serial manner. Differently, the latter scheme creates a new process for each query. It relies on the OS to manage parallel applications. Table 4 lists our evaluated schemes.

Two different implementation approaches of Congra are considered. In its default configuration, Congra uses hardware performance counters for precise bandwidth-aware scheduling. It allows the system to better identify free bandwidth resource with the cost of slightly increased response time. We also consider another light-weight design alternative that does not require monitoring performance counters. By checking the profiled data of active tasks, one can estimate the average memory bandwidth usage in real-time.

VI. RESULTS

In this section, we experimentally analyze the performance of Congra in a concurrent graph processing environment.

A. Job Throughput

We measure the query throughputs of graph systems managed by *Congra*. The throughput is defined as the number of queries processed per time unit. Figure 6 compares the throughput of four different schemes under different graph sets. The results are normalized to *Private*.

It is evident that *Congra* outperforms existing systems no matter what kind of bandwidth monitoring mechanisms are used. On average, Congra (w/o hardware monitoring in default) improves query by 60% compared to existing scheduling strategy that aggressively fits all the queries on to shared machines (e.g., *Shared*). With hardware monitor, we can see a slightly increased throughput (5%).

In Figure 6 we notice that our scheduler works particularly well for *BellmanFord*. This is mainly because our system is aware of the poor scalability of *Homo-3* (*BellmanFord*) and decides to run it with less computation threads. Differently, *Homo-1* (*K-Core*), and *Homo-2* (*PageRank*) both have poor throughput improvement over baseline since they are memory bandwidth-hungry workloads. The scheduler can do little to further improve bandwidth utilization on our hardware system for these cases because the bandwidth is already saturated.

If we look at each individual result, *Shared* does not outperform *Private* across all the evaluated graph query sets. For some workloads such as *Heter-L2* and *Heter-S2*, *Private* can yield 15% higher throughput compared to *Shared*. Similarly, we observe that *Shared* shows better throughput compared to *Private* on our homogeneous workloads and the differences can be up to 21%. Overall, the average throughput of running multiple graph queries in serial (*Private*) is almost the same as *Shared*. The difference is less than 1%. Our results demonstrate that graph analytic queries are special

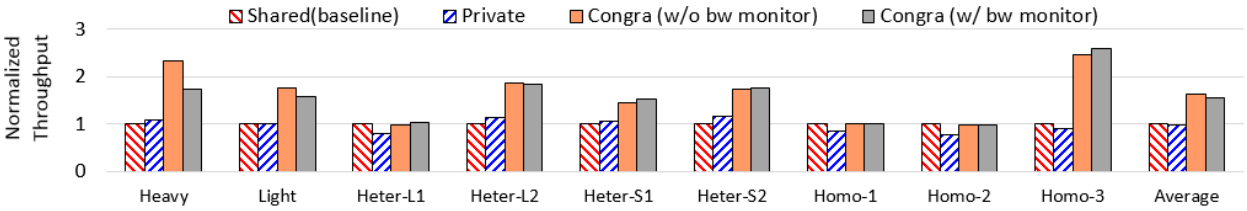


Fig. 6. Normalized throughput of different methods

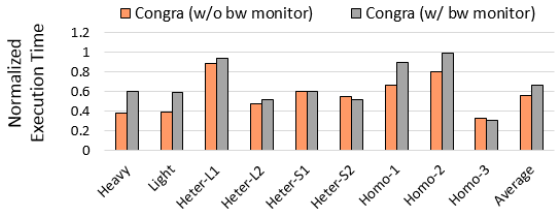


Fig. 7. Normalized execution of different methods

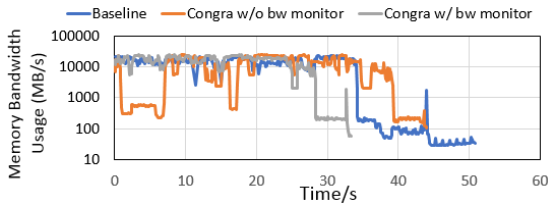


Fig. 8. Memory bandwidth usage of graph exploration

workloads that must be carefully scheduled to unleash the full potential of the graph processing framework.

B. *Execution Time*

We next evaluate the impact of our scheduling on query execution time. Figure 7 presents the total execution time of each query set and the results are normalized to that of *Shared*. As long as *Congra* is used, we observe improved execution time compared to existing graph processing framework. If hardware monitor is enabled, our scheduler can reduce the execution time by 33%.

In many cases, *Congra* with hardware monitoring shows worse results compared to *Congra* without monitoring. For example, the difference between the two evaluated *Congra* implementations can be up to 21 percentage points. This can be explained by the operation mechanism of *Congra*. During runtime, *Congra* with hardware monitoring is able to accurately identify the available free memory bandwidth resources. As a result, it has better opportunity to increase the parallelism by exploiting the memory bandwidth slack. Although this may slightly increase the overall query throughput (Section 6.1), it can lead to non-trivial cost in execution time due to increased resource contention.

To further demonstrate the resource utilization behavior of different schemes, we draw a memory bandwidth curve obtained from hardware performance counters in Figure 8. The graph query application used is *BFS*, *Components*, and *Triangle*. Our result shows clearly that *Congra* without hardware monitor exhibits much more memory bandwidth slacks. In contrast, Congra with hardware monitor achieves very high memory bandwidth utilization just like *Shared*, while reduces total execution time by 65%. Graph applications typically involve heavy communication between memory and the processor. Generally, applications with large memory bandwidth usage run faster. Although using hardware monitoring mechanism may negatively affect the query response time, it maintains better system utilization. In other words, it provides an alternative trade-off between cost efficiency and workload performance.

C. *Quality of Service*

Another interesting observation is the QoS of our graph processing system. We define the QoS of a query to be its running time (execution time plus waiting time) dividing the executing time when it is profiled (e.g., without any resource contention). We create a box and whisker chart, as shown in Figure 9. The chart shows distribution of data into quartiles, highlighting the mean, highest and lowest numbers.

We find the performance variation of existing single-query oriented frameworks is much higher than *Congra*. This is because conventional designs have no knowledge of the characteristics of different queries and the underlying memory usage patterns. Such a large performance variance poses serious problems to future graph-based edge services.

From the figure we can clearly see that our scheduling algorithm causes much smaller QoS violation as compared to the baseline. Our calculation of QoS takes in the amount of time where the query is waiting. *Congra*'s scheduling algorithm can minimize the total waiting time by serving short queries first. In this way, the overall response time is decreased and each query enjoys better QoS.

D. *Sensitivity and Scalability*

Another advantage of *Congra* is that it shows fairly good scalability. To understand how the number of queries influences the performance of *Congra*, we conduct a stress test on our system. In the experiment, we vary the number of concurrent graph queries. Figure 10 shows the normalized results obtained from two different types of queries.

Congra is believed to scale well if it yields almost constant throughput with growing number of queries. As shown in Figure 10(a), under lightweight queries, the system has relatively stable throughput between 0.58~0.64, which means each query takes roughly the same amount of time. For heavy-weight queries, increasing the number of concurrent queries above 15 causes the system (with hardware monitor) to stop responding. This is because the scheduler issues too many queries and the system instantly runs out of server memory.

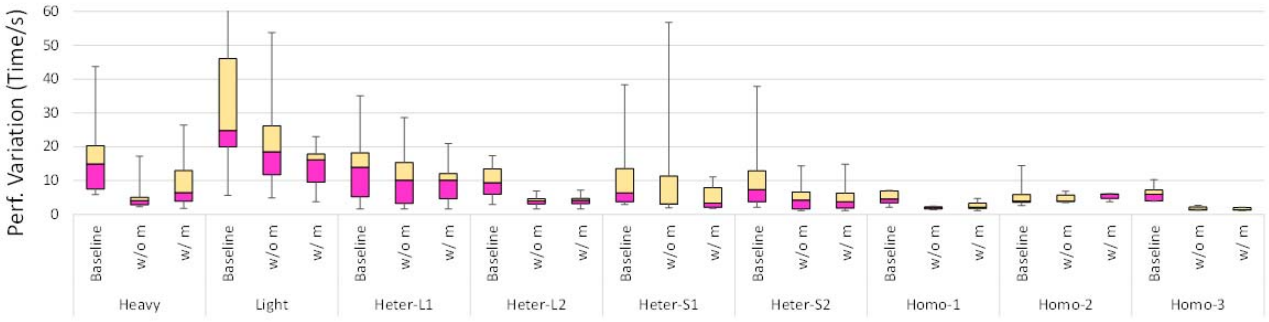
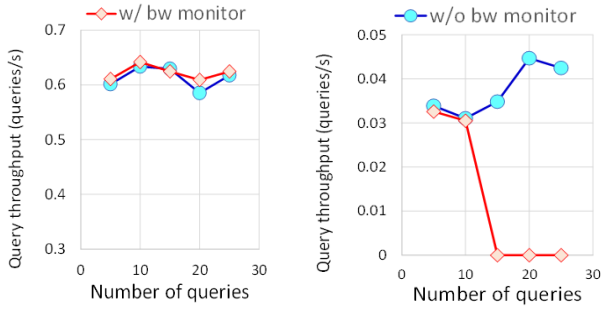


Fig. 9. Impact of scheduling algorithm on performance variation



(a) Lightweight queries (b) Heavyweight queries

Fig.10. Scalability under large number of queries

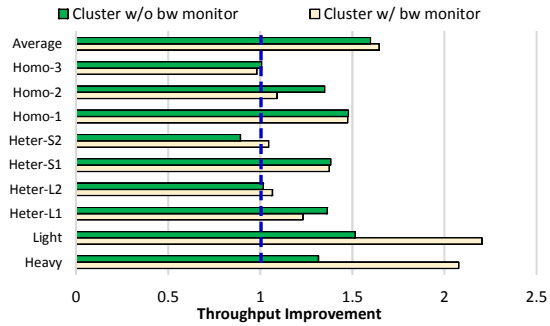


Fig. 11. Cluster-wide scheduling results

E. Cluster-Level Scheduling

We finally evaluate the impact of our cluster-wide scheduling strategy. The results are shown in Figure 11.

As we can see, our cluster scheduling algorithm achieves more than 2X throughput (with 2 active nodes) for *Heavy* and *Light*. In the figure, *Heter-S2* and *Homo-3* (*BellmanFord*) are where the algorithm shows poor performance. This is because *BellmanFord* consumes too small memory bandwidth such that only one server is enough for all *BellmanFord* queries to run. *Heter-S2* has considerable portion of *BellmanFord* query, and therefore it also shows lower performance. From the results we also notice that a cluster of shared-memory machines requires hardware monitor to achieve better performance. It is better to use memory bandwidth monitor to periodically update the current value of bandwidth usage.

VII. RELATED WORK

In this section we discuss representative prior studies that are most relevant to our work.

A. Single-Graph Processing Systems

Many prior works have investigated the parallel processing of graph analytic workloads [1, 2], including both messaging-passing based designs [9] or shared-memory based designs [10]. For example, GPS [9] combines message-passing based design with dynamic graph partition with optimized graph structure partitioning. If the data can fit on a single node, shared-memory based design is shown to be more attractive. By breaking large graphs into smaller parts, GraphChi shows that a shared memory based design with disk as additional data store can handle large graph efficiently [10]. Sometimes both communication schemes can be seen in a framework, such as Trinity [11]. It jointly optimizes the memory and network resources by considering graph access patterns. Differently, GraphX [12] represents another way of implementing the graph processing system. It uses a data flow based approach. Most of these prior arts focus on accelerating the processing of a single large graph, rather than a variety of small graphs. They can incur unnecessary performance overhead when facing multiple concurrent graph queries.

In this paper we implement *Congra* on top of the Ligra framework. Ligra is introduced by Shun and Blelloch [13] as a light-weight shared memory framework for large graph processing. Ligra uses multi-threading to speedup graph applications are that scalable.

B. Concurrent Graph Query Processing

Processing concurrent graph queries in a multi-user environment has been gaining increasing attention [4, 5, 14, 15]. For example, Kim et al. [4] devised methods for enabling efficient processing of multiple graph queries using MapReduce. Similarly, Feher et al. [5] utilized the parallelization mechanism of MapReduce to solve the graph pattern matching problem. Recently, Xue et al. [14, 15] investigated concurrent graph queries and proposed a graph structure sharing mechanism for avoiding memory waste. This work considers a single graph, rather than multiple queries on different graphs. In general, all these works focus on adapting existing graph processing models to a multi-user environment. Differently, we explore a scheduling mechanism that is jointly guided by hardware resources and graph query characteristics. Further, we consider the interaction of heterogeneous graph queries and investigate optimal thread allocation.

C. Architectural Support for Graph Processing

Another group of relate work focusing on the implication of architecture design on graph analytic workloads. For example, Ahmad et al. [16] devised a benchmark suite for understanding multi-threaded graph algorithms for shared multicore processors. Beamer et al. [17] show that many workloads may not fully utilize the systems off-chip memory bandwidth. Several recent studies have devised graph accelerators for single-graph processing [18-20]. Different from these prior works, we explore improving the performance of concurrent graph processing through resource-aware scheduling in a multi-user environment.

D. Resoruce Aware System Design and Managemnt

Resource aware management has been focused on workload chateristics [21, 22], system architecture [23, 24], as well as power behavior [25-27]. In this work we combine hardware resource statistics with graph analytic workload characteristics to optimize the resource usage of a shared-memory node.

VIII. CONCLUSIONS

There is a growing demand to process and analyze a variety of small graphs concurrently. Nevertheless, the opaque memory consumption behaviors and the variability of graph query properties make concurrent graph processing a challenging task. In this work we propose *Congra*, a novel scheduling mechanism at enables highly efficient query-level concurrency. Our evaluation on a wide range of applications shows that *Congra* significantly outperforms conventional designs in both throughput and response time. It also maintains attractive scalability and quality of services. *Congra* allows shared-memory machines to better serve the needs of graph analytic services in a multi-user environment.

IX. ACKNOWLEDGEMENTS

This work is supported in part by the National Basic Research Program of China (973 Program, #2015CB352403), the National Natural Science Foundation of China (No. 61502302), and a CCF-Tencent Open Fund.

REFERENCES

[1] R. McCune, T. Weninger, and G. Madey. "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing, *ACM Computing Surveys*, Vol 48, Issue 2, 2015

[2] N. Doekemeijer and A. Varbanescu. "A survey of parallel graph processing frameworks". Technical Report #PDS-2014-003, Delft University of Technology, 2014

[3] B. Chandramouli, J. Claessens, S. Nath, I. Santos, and W. Zhou. "RACE: real-time applications over cloud-edge". *SIGMOD Int. Conference on Management of Data* (SIGMOD), 2012

[4] S.H. Kim, Y.H. Lee, H. Choi, and Y.J. Lee. "Parallel processing of multiple graph queries using MapReduce", *Int. Conf. on Advances in Databases, Knowledge, and Data Applications* (DBKDA), 2013

[5] P. Feher, M. Asztalos, T. Vajk, T. Meszaros, and L. Lengyel. "Detecting subgraph isomorphism with MapReduce". *Journal of Supercomputing*. Vol. 73, Issue 5, 2017.

[6] Intel® 64 and IA-32 Architectures Software Developer Manuals: https://software.intel.com/en-us/articles/intel-sdm

[7] Stanford large network dataset collection https://snap.stanford.edu/data/soc-LiveJournal1.html

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos. "R-MAT: A recursive modelfor graph mining", *The 2004 SIAM International Conference on Data Mining* , 2004

[9] S. Salihoglu and J. Widom. "GPS: a graph processing system". *The 25th Int. Conference on Scientifc and Statistical Database Management* (SSDBM), 2013.

[10] A. Kyrola, G. Blelloch, and C. Guestrin. "GraphChi: Large-scale graph computation on just a PC", *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 2012

[11] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. *ACM SIGMOD Int. Conference on Management of Data* (SIGMOD), 2013

[12] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica. "GraphX: Graph processing in a distributed dataflow framework" *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 2012

[13] J. Shun, and G. Blelloch. "Ligra: a lightweight graph processing framework for shared memory", *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), 2013

[14] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. "Seraph: An efficient, low-cost system for concurrent graph processing", *Int. Symp. on High-Performance Parallel and Distributed Computing* (HPDC), 2014

[15] J. Xue, Z. Yang, S. Hou, and Y. Dai. "Processing concurrent graph analytics with decoupled computation model". *IEEE Transactions on Computers*, Vol 66, No. 5, 2017

[16] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. "CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores", *IEEE Int. Symposium on Workload Characterization* (IISWC), 2015

[17] S. Beamer, K. Asanovic, and D. Patterson. "Locality exists in graph processing: Workload characterization on an Ivy Bridge server", *IEEE Int. Symposium on Workload Characterization* (IISWC), 2015

[18] J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi. "A scalable processing-in-memory accelerator for parallel graph processing", *Int. Symposium on Computer Architecture* (ISCA), 2015

[19] M. Ozdal, S. Yesil, T. Kim, A. Ayupov. J. Greth. S. Burns, O. Ozturk. "Energy efficient architecture for graph analytics accelerators", *Int. Symposium on Computer Architecture* (ISCA), 2016

[20] T. Ham, L. Wu, N. Sundaram, N. Satish, M. Martonosi. "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics", *Int. Symposium on Microarchitecture* (MICRO), 2016

[21] Q. Chen, M. Guo, and Z. Huang. CATS: cache aware task-stealing based on online profiling in multi-socket multi-core architectures, *International conference on supercomputing* (ICS), 2012

[22] Q. Chen and M. Guo, "locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures", *ACM Trans. On Architecture and Code Optimization* (TACO), 2015

[23] M. Liu, C. Li, and T. Li, "Understanding the Impact of vCPU Scheduling on DVFS-based Power Management in Virtualized Cloud Environment", *IEEE Int. Symp. On Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS), 2014

[24] M. Song, Y. Hu, Y. Xu, C. Li, H. Chen, J. Yuan, and T. Li, Bridging the semantic gaps of GPU acceleration for scale-out CNN-based big data processing: Think big, see small, *Int. Conference on Parallel Architectures and Compilation Techniques* (PACT), 2016

[25] Y. Zu, C. Lefurgy, J. Leng, M. Halpern, M. Floyd, and V.J. Reddi, "Adaptive guardband schduling to improve system-level efficiency of the POWER7+", *Int. Symposium on Microarchitecture* (MICRO), 2015

[26] W. Zheng, K. Ma, and X. Wang, "TECfan: Coordinating Thermoelectric Cooler, Fan and DVFS for CMP Energy Optimization", *IEEE Int. Parallel and Distributed Processing Symposium* (IPDPS), 2016

[27] Y. Hu, C. Li, L. Liu, and T. Li, "HOPE: enabling efficient service orchestration in software-define data centers ", *Int. Conference on Supercomputing* (ICS), 2016