CongraPlus: Towards Efficient Processing of Concurrent Graph Queries on NUMA Machines

Peitian Pan, *Student Member, IEEE,* Chao Li, *Member, IEEE,* and Minyi Guo, *Fellow, IEEE*

Abstract—Graph analytics has been routinely used to solve problems in a wide range of real-life applications. Efficiently processing concurrent graph analytics queries in a multiuser environment is highly desirable as we enter a world of edge device oriented services. Existing research, however, primarily focuses on analyzing a single, large graph dataset and leaves the efficient processing of multiple mid-sized graph analytics queries an intriguing yet challenging open problem. In this work, we investigate the scheduling of concurrent graph analytics queries on NUMA machines. We analyze the performance of several graph analytics algorithms and observe that they have diminishing performance returns as the number of processor cores increases. With concurrent graph analytics, such diminishing returns translate to no or even negative performance gains because of increasing contention on shared hardware resources. We also demonstrate the unpredictability of memory bandwidth usage for numerous graph analytics algorithms, which can lead to sub-optimal performance due to its potential to cause severe memory bandwidth contention. Motivated by the above observations, we propose CongraPlus, a NUMA-aware scheduler that intelligently manages concurrent graph analytics queries for better system throughput and memory bandwidth efficiency. CongraPlus collects the memory bandwidth consumption characteristics of graph analytics queries via offline profiling and eliminates memory bandwidth contention by computing the optimal sequence to launch queries. It also avoids computation resource contention by assigning a certain number of processor cores to the individual queries. We implement CongraPlus in C++ on top of the Ligra graph processing framework and test it with judiciously selected graph processing query combinations. Our results reveal that CongraPlus-based schemes improve query throughput by 30% compared to the conventional approach. It also exhibits a much better quality of service and scalability.

Index Terms—Concurrent graph analytics, query scheduling, memory bandwidth, efficiency, throughput.

1 Introduction

Computation over graph big data has become a hotspot due to its broad application in social network analysis, web page analysis, protein structure research, etc. Graph structures can model a wide range of entities and the links between them. Recent research literature witnessed the proposal of numerous frameworks that provide graph-specific optimizations, with an emphasis on effectively handling a single, large-scale graph dataset [1], [2].

While existing systems are efficient for single graph analytics query, they provide little support for processing concurrent graph analytics queries. 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 correlate data with other devices to provide context-aware services continuously [3]. Multi-player mobile games need to monitor and react on the status change of each player and their interactions [4]. A group of analysts may issue multiple graph analytics queries to analyze financial patterns or customer behaviors through pattern matching algorithms such as subgraph isomorphism detection [5].

The conventional way of handling concurrent queries is to create one process for each query and to run them at the same time. Unfortunately, this does not work well for concurrent graph analytics. It is generally accepted that memory bandwidth is the bottleneck for parallel graph processing [6], [7]. For concurrent graph analytics, numerous analytic queries can cause severe memory bandwidth contention and significantly degrade the response time for all queries that are running in parallel.

What is worse for conventional concurrency handling is the unpredictability of memory bandwidth consumption characteristics for a specific graph analytic application. Applications that feature a certain degree of data exploration (e.g., BFS) tend to have a decreasing number of active vertices as the number of iteration grows. In other words, these applications may consume less memory bandwidth over time. Some computation-intensive graph applications (e.g., PageRank), on the other hand, iterate over the same set of vertices until numerical values converge. These applications are observed to consume almost constant memory bandwidth.

Another obstacle to efficient concurrent graph analytics is the diminishing performance returns of single analytic application with increasing number of processor cores. Shared-memory graph processing frameworks rely heavily on more threads to achieve significant performance improvement [8]. However, our experiments indicate such improvement gets smaller as the number of threads grows larger. With concurrent graph analytics queries, the performance degradation due to computation resource contention outweighs the performance gained via increasing utilized threads. Thus an intelligent query scheduling scheme should be able to appropriately assign processor cores to each query so that the overall performance gained (i.e., the speedup achieved

- *P. Pan was with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.*
- *C. Li and M. Guo are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.*

Manuscript received ; revised .

through parallel processing minus the performance degraded due to resource contention) is maximized.

To tackle the above challenges, we use profiling to achieve accurate memory bandwidth prediction and efficient computation resource allocation. We propose CongraPlus, a novel scheme for scheduling graph analytics queries on NUMA machines, based on our prior Congra scheduler [9]. CongraPlus extends existing shared-memory graph analytics framework in two aspects. First, it enables conventional single-graph oriented designs to efficiently serve multiple users. CongraPlus uses a resource-aware query-colocation scheme to ensure the performance impact of resource sharing is minimized. Second, it allows multiple shared-memory based systems to be easily organized as a graph analytics cluster and automates the QoS-aware scheduling of incoming analytic queries.

We have built a prototype of CongraPlus based on Ligra, a lightweight graph processing framework on shared-memory machines [8]. The system profiles single analytic queries while it is offline. During the online phase, it effectively schedules concurrent analytic queries by considering their memory bandwidth requirement, performance gains through multithreading, and the interference caused by computation and bandwidth resource contention. While CongraPlus is implemented on top of Ligra, it is largely orthogonal to the underlying infrastructure and can be extended to support a variety of shared memory graph analytics frameworks.

This paper makes the following contributions:

- We characterize the resource consumption behavior of graph analytics applications on shared-memory machines and discuss the design trade-offs in such system.
- We propose CongraPlus, a NUMA-aware scheduling scheme that extends our prior Congra scheduler [9] with load balancing and thread assignment technique to support efficient concurrent graph analytics on commodity servers.
- We implement and evaluate CongraPlus system with carefully selected combinations of representative graph analytics algorithms and input graphs.

The rest of this paper is organized as follows. Section 2 introduces background and further motivates our design. 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 7 discusses related work and Section 8 concludes this paper.

2 Background and Motivation

In the past several years, most research focuses on processing a single large graph analytics workload (e.g., Web data, social network, and large scientific datasets) [1], [2]. Many of the proposed parallel frameworks in this scenario adopt a message-passing communication model rather than shared-memory [10], [11], [12].

Processing concurrent small or mid-sized graph analytics queries, however, requires a different design approach. It is preferable to analyze the graph on single shared-memory machines. State-of-the-art NUMA commodity servers that feature multiple processors and up to several terabytes of main memory are widely deployed in data centers. The abundance of computation and storage resources makes the processing of concurrent graph analytics queries possible. Moreover, the memory access in shared-memory systems has much smaller overhead than their message-passing counterparts, which is ideal for graph analytics as they are inherently data-intensive [13].

In this paper we focus on concurrent graph analytics of small to mid-sized graph structures on individual commodity servers. Large graph structures with billions of edges and vertices are generally proprietary data of companies and organizations who can afford private server clusters to analyze these data. The graph analytic workload of public cloud service providers, whose users have their own private but smaller graph structures to be analyzed, tend to be a mix of small to mid-sized graph analytic queries. However, simply treating each analytic query as either a traditional batch task or latency-critical task is inefficient: running a series of analytic queries in parallel leads to severe resource contention, which degrades performance, and dedicating the entire server to a mid-sized analytic tasks leads to low server utilization.

Our goal of efficient processing on concurrent graph analytics queries is two-fold: 1) the system should maximally utilize the computation and bandwidth resources installed to the NUMA server to increase cost efficiency; 2) the system should avoid performance degradation due to parallel execution of independent queries. To realize this goal, the scheduler must be 1) able to identify the bandwidth requirement for each query and ensure the memory bandwidth is not over contended during the entire execution process; and 2) capable of assigning an appropriate number of processor cores to each query such that they are optimally co-located. This can be challenging for two reasons: opaque bandwidth usage and limited thread scalability.

2.1 Opaque Bandwidth Usage

Graph analytics applications involve heavy communication between main memory and processor. Memory bandwidth is therefore deemed the bottleneck for most parallel graph analytics applications [6], [14]. The traditional scheduling scheme, which executes all analytic queries in parallel, can lead to severe contention in memory bandwidth and degrades the overall performance of the graph analytics engine.

Better scheduling decisions (i.e., resource-aware scheduling) can be made if the memory bandwidth usage patterns can be obtained for each analytic query. Unfortunately, such runtime patterns are typically opaque to the system and hard to predict before execution. We run three typical graph analytics algorithms and keep track of their memory bandwidth usage to demonstrate the versatility of memory bandwidth usage patterns. Fig. 1 presents the bandwidth usage traces of BellmanFord, KCore, and PageRank algorithms. Their average bandwidth usage is shown in Fig. 2.

Fig. 1a shows the three stages of memory bandwidth usage that Bellman-Ford shortest path application goes

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2899595, IEEE Transactions on Parallel and Distributed Systems

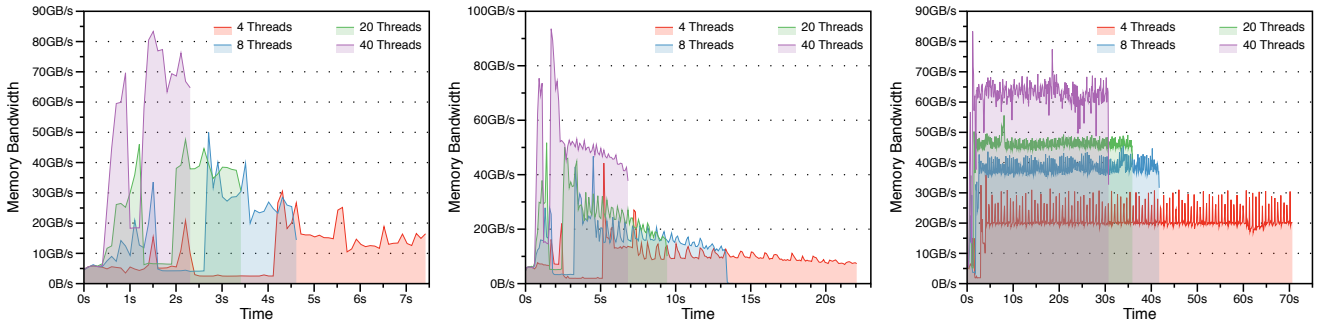


Fig. 1. Memory bandwidth usage traces of different algorithms with rMatGraph10m input graph. (a) exhibits high and low memory bandwidth utilization at different execution stages; (b) has similar bandwidth utilization distribution as (a) but exhibits a periodic change in its last execution stage; (c) has constant bandwidth usage during the entirety of its execution, with periodic fluctuation. All three presented algorithms have vastly different memory bandwidth usage patterns which are hard to predict before execution.

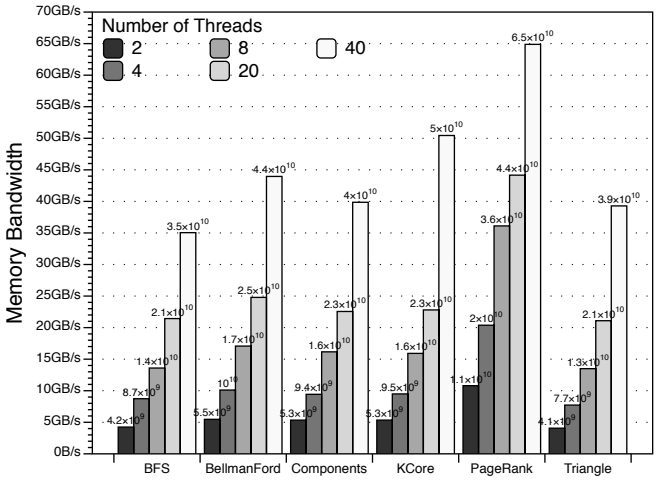


Fig. 2. Memory bandwidth usage of different algorithms with rMatGraph10m input graph. Bandwidth usage typically grows larger with more threads, but varies significantly among different algorithms.

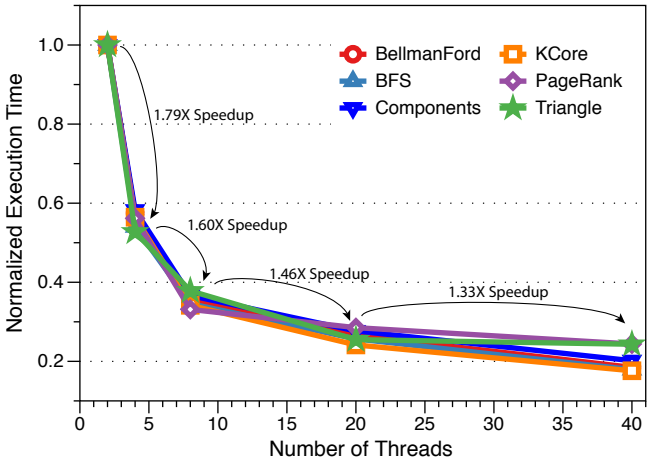


Fig. 3. Execution time of different algorithms with rMatGraph10m input graph. Performance is generally better with more threads, with speedups decreasing w.r.t. number of threads.

through: the high consumption stage, where the application is actively loading the graph structure from the hard disk; the low consumption stage, where the number of active vertices in the current vertex front is low; and the graph-dependent stage, where the bandwidth consumption depends on both the graph topology and whether the lengths of shortest paths have converged. The bandwidth usage of the K-Core algorithm in Fig. 1b resembles that of Fig. 1a in the loading stage. The following stage features decreasing bandwidth usage with periodic fluctuation, which corresponds to the iterative computation of input graph's K-decomposition. PageRank algorithm in Fig. 1c features almost constant bandwidth usage with periodic fluctuation, due to its iterative updating to the rank values of all vertices. Fig. 2 further summarizes the average bandwidth usage for different analytic applications. Even in simple average value, the bandwidth usage varies significantly across different algorithms.

To conclude, our experiments indicate that the memory bandwidth usage patterns of graph analytics applications are often vastly different across different algorithms. Such versatility makes the usage patterns hard to predict, which prevents the system from performing resource-aware scheduling.

2.2 Limited Thread Scalability

A few key techniques, such as multiprocessing, multi-core processor, and hyper-threading, are ubiquitous among shared-memory commodity servers with NUMA architecture. When combined, they benefit parallel graph analytics greatly as abundant parallel worker threads can be utilized by carefully designed and implemented graph analytics frameworks [8].

However, the performance improvement diminishes when the number of threads grows larger. We run six graph analytics algorithms with different number of threads to capture the thread scalability of different algorithms. Fig. 3 presents the execution time of the algorithms with respect to the number of threads. The speedup from single thread to double threads is significant (1.79X). Doubling the threads from 2 to 4 decreases the speedup to 1.60X. Further increasing the number of threads, although does help improve the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2899595, IEEE Transactions on Parallel and Distributed Systems

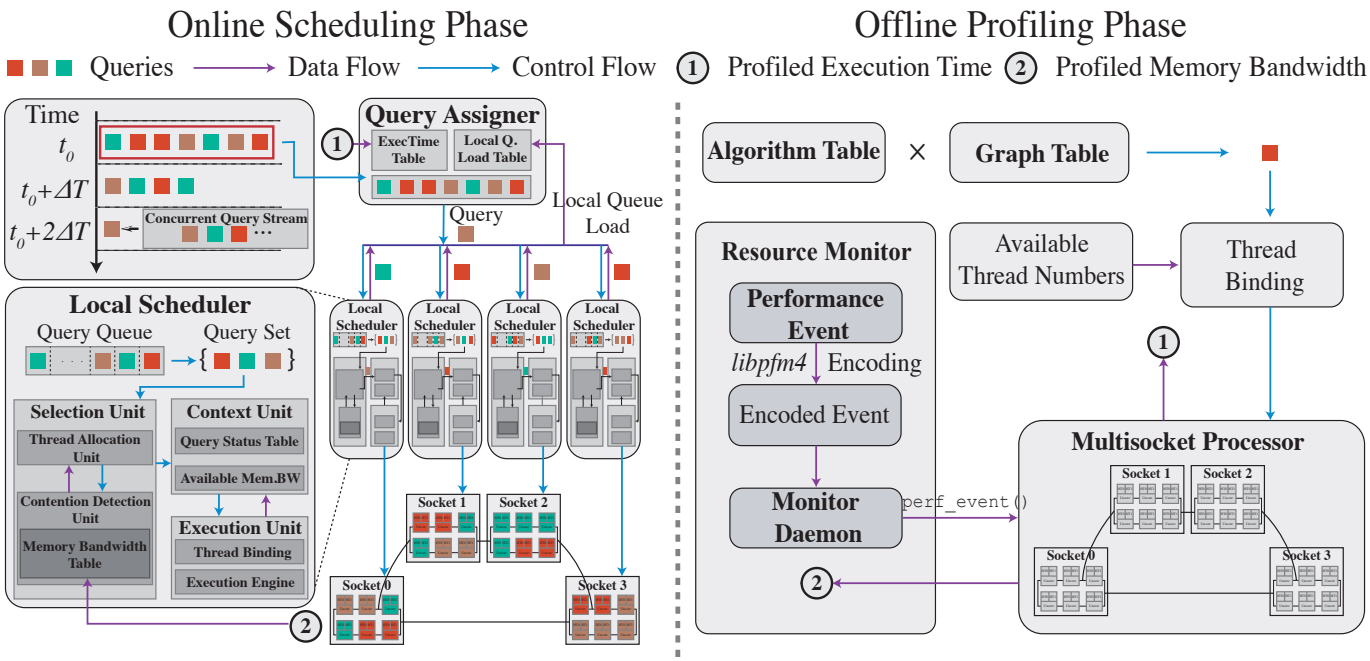


Fig. 4. Overview of CongraPlus system. During offline phase, the combination of analytic algorithms and graph structures will be enumerated and profiled with varying threads. Key run-time information such as execution time and memory bandwidth usage (①, ②) will be handed to the online phase to enable better scheduling decision. During the online phase, CongraPlus will first divide query stream into fixed time interval. The query assigner will assign queries into different nodes in the NUMA system and balances the load among them. The local scheduler determines the optimal execution sequence and thread number for each query to run with on the local node.

performance of each application, leads to less performance gained per thread. One cause of the reduced performance gain is the increased cost of maintaining cache coherence across more cores. When processors in more than one socket are involved in a single program (for threads larger than 20 in Fig. 3), the expensive remote memory accesses also has a negative performance impact.

The diminishing performance gain is problematic for concurrent graph analytics. First, allocating all processor cores to each analytic query is inefficient in a concurrent environment. The results from Fig. 3 indicates that doubling threads does not necessarily half the query's execution time. Therefore, assigning fewer processor cores to more queries can be more efficient in the sense that it reduces the overall execution time. Second, enforcing optimal performance for individual query inevitably leads to contention on shared processor resources such as reservation stations, reorder buffer, last level cache, etc. The contention will negatively impact system performance and even outweigh the speedups achieved by utilizing more threads.

Our insight from the experiments is that blindly pursuing optimal performance for the individual query will hurt the whole system's performance due to aggressive query co-location and low per-core efficiency. Optimal system performance can be achieved through intelligent processor core allocation with which shared resource contention is minimized, and each core achieves maximal speedup.

3 CongraPlus System Design

Judiciously co-locating analytic queries can improve hardware resource utilization without sacrificing the overall system performance. An intelligent scheduling scheme should be able to identify resource utilization characteristics of applications to achieve optimal system performance. In this section, we propose CongraPlus, a resource-aware analytic query scheduler for NUMA commodity servers.

Fig. 4 presents the structure of CongraPlus scheduler. We divide our scheduling scheme into two phases: the offline profiling phase and the online scheduling phase. In offline profiling phase, CongraPlus profiles all combinations of analytic algorithms and graph structures available to the system. In online scheduling phase, CongraPlus works on an analytic query stream and assigns each query to one certain processor in a way that ensures balanced loads across all processors. Each local scheduler will subsequently compute the optimal launch sequence and number of threads for each query, with which the queries will be launched.

CongraPlus system is based on our previous concurrent graph analytics framework Congra [9]. Congra targets general shared-memory machines and assumes a single-socket configuration. CongraPlus is different from Congra in that 1) it is aware of the NUMA architecture that most modern servers deploy to reduce remote memory accesses and 2) it uses an optimized local scheduler, whose unoptimized version is also included in Congra systems. Apart from these differences, they also share some similarities. Both systems are built to support efficient concurrent graph analytic queries. The profiling phase of CongraPlus is based on Congra's and both systems can share the same profiled characteristics.

3.1 Offline Profiling

The offline phase is mainly responsible for collecting runtime information that is valuable to scheduling such as

memory bandwidth usage and application execution time. The opaque bandwidth usage problem described in Section 2.1 is handled by synthesizing possible analytic applications and profiling them with varying threads. The profiling results reveal the application's resource usage patterns and resource utilization efficiency, both of which will be later (in the online scheduling phase) used to optimize the overall system performance.

This section is organized as follows: section 3.1.1 provides detailed information on how the profiled analytic applications are synthesized; section 3.1.2 describes the structure of the monitor that records hardware resource information at profile time.

3.1.1 Analytic Application Synthesis

The offline phase begins with the synthesis of analytic applications which will later be profiled. Each analytic application can be represented by two components: the analytic algorithm and the graph structure it runs on. The offline phase thus synthesizes analytic applications via enumerating the algorithms and graphs available to the system.

For concurrent graph analytics system deployed at the cloud, the analytic algorithms used in profiling can be deduced from the analytic services provided by the platform. For example, computing relative importance of each vertex in a graph structure is an important task for data mining applications. This task generally involves PageRank algorithm and its variants [15]. Therefore, the platforms that provide similar data mining services should profile PageRank algorithm and its variants. For platforms that try to provide more general-purpose graph analytic functionalities, breadth-first search (BFS) and depth-first search (DFS) should be profiled as they can serve as the primitives that help traverse a graph structure [16].

Collecting graph structures, however, is not trivial. Users of graph analytics services may have interests analyzing public graph structures which contain information related to them (exploring common neighbors in a social network, computing the shortest path from the user's starting point in a city street grid, etc.) or work on private graph structures. In the former case, the graph structure can be collected and profiled with permission from its owner as such structures tend to be updated infrequently [17]. Profiling private graph structures can be cost-inefficient since only its owner work on them. Besides, the users may be reluctant about sharing their data with cloud service providers due to privacy considerations. A workaround for this situation is to establish reasonable approximation for their run-time information, which is beyond the scope of this paper, and monitor these applications in the online scheduling phase.

3.1.2 Resource Monitor

To record the detailed resource consumption behavior of each analytic application, the resource monitor module is used to collect data from processors' performance counters and interpret them as meaningful run-time information. Specifically, the resource monitor includes a set of performance event names that are to be monitored. These names are first encoded into OS-compatible event names and then fed to monitor daemon, which will be responsible for reading data from corresponding performance counters at processors and interpreting them. The output of the resource monitor is a sequence of numerical quantities recorded with timestamps, which can be further analyzed or sent to CongraPlus online phase to assist in scheduling decision making.

3.2 Online Scheduling

The online phase aims to solve two problems that hinder efficient co-location of analytic queries: 1) how to assign queries to processors since assigning each query to all processors results in sub-optimal overall performance; 2) what is the optimal launch sequence and number of threads to execute each query. The above problems are addressed in the **QueryAssigner**[1] and **LocalScheduler** modules in Fig. 4 respectively, both of which will be described below.

This section is organized as follows: section 3.2.1 presents how CongraPlus assigns analytic queries to an individual processor and achieves well-balanced load across all processors; section 3.2.2 describes how CongraPlus computes the optimal launch sequence and thread number of each query.

3.2.1 Query Assigner

Single graph analytics application benefits from multiprocessing which provides more parallel worker threads. However, assigning all processor cores to each query, which we will refer to as a *shared* scheme, is cost-inefficient because of diminishing speedups discussed in section 2.2. The shared scheme can also incur severe contention on the last level cache, memory bandwidth, cross-socket link bandwidth, etc.

CongraPlus adopts a *private* assigning scheme, where each query can be executed only on a single processor to avoid performance degradation due to aggressive query co-location. More specifically, the query assigner maintains query queue for each processor in the system. With n processors, the workloads of queries are well-balanced among all processors if equation 1

$$OverallLoad = \max_{1 \le i \le n} \sum_{Q_j \in Queue_i} \texttt{ExecTime}(Q_j) \qquad (1)$$

is minimized. Here, $Queue_i$ represents one of the query queues that corresponds to the processors, and Q_j indicates the query that is in the queue. The execution time of each query, which has been profiled during the offline phase, will serve as an alternative indication of query's workload.

Minimizing $OverallLoad$ term in equation 1 is equivalent to the multiprocessing scheduling optimization problem, which is NP-hard [18]. Section 4.2.1 will detail the approximate algorithm CongraPlus used to solve it.

3.2.2 Local Scheduler

The local scheduler is responsible for scheduling and launching the queries assigned to its corresponding processor. The remaining section will focus on discussing how to determine the optimal query launch sequence and with

1. In this paper we assume a single machine environment. But CongraPlus can be easily scaled out to more machines by having a query allocator that dynamically allocates each incoming query to one local query stream (the concurrent query stream in Fig. 4).

how many threads a query should be run, both of which are critical to the overall system performance.

A. **Query Set**
CongraPlus computes the optimal query launch sequence with a generate-and-check method. Queries of the light workload are prioritized to be added to the query set, which will be examined by the selection unit for resource contention and the optimal number of threads. The details of the query set generation will be discussed at section 4.

B. **Selection Unit**
The selection unit takes a query set as input and checks whether launching all queries in this query set incurs resource contention. If not, the selection unit will also output the optimal thread allocation for each query.

Contention detection involves the run-time hardware resource usage information collected at the offline phase. It compares the sum of the current and provisioned resource usage with the total resource available, and determines whether resource contention occurs. With the contention detection unit, the local scheduler ensures that the resource usage is limited within the system's capability.

The thread allocation unit adopts an iterative ascent algorithm to compute the optimal number of threads for each query in the query set. During the ascent process, the query which benefits most from running with more threads is chosen and the threads allocated to it is increased. This process will iterate until resource contention occurs under the current query set and thread allocation scheme. The scheme in the second-to-last iteration is deemed optimal because it maximizes the performance gained from increasing number of threads. The iterative ascent algorithm will be detailed at section 4.2.3.

C. **Context Unit**
Either computing the optimal launch sequence or thread allocation scheme requires the knowledge of the status of the currently active queries. The context unit keeps track of the status of all launched queries so that the scheduling happening in selection unit is up-to-date.

D. **Execution Unit**
CongraPlus is compatible with all shared-memory graph analytics frameworks. Once the optimal thread allocation scheme is determined, the underlying execution engine will be called to run the analytic program.

3.3 Applicability Discussion

The scheduling techniques of CongraPlus are not exclusive to graph analytic workloads. It can be applied to other workloads if the following restrictions are satisfied: 1)the resource usage characteristics of this workload does not change significantly between each execution, 2) the workload is memory-bound and benefits from multi-threading and 3) concurrent execution of such workloads has considerable benefit in server utilization or performance and does not hurt the quality of the service provided by the service provider. The first restriction ensures the profiling overhead is acceptable. The second restriction is derived from the two challenges, opaque bandwidth usage and limited thread scalability, that the scheduler tries to tackle. The third restriction makes sure the adopting our scheduling techniques leads to benefits at an acceptable cost.

We argue that graph analytics is an ideal type of workload for the online scheduler. First, the graph datasets between each execution has little or no change so that the profiling results can be reused a large number of times before a new profiling is needed. For example, the Twitter graph, used in generating the user-to-follow recommendation, is updated less frequently than once a day [17]. For smaller graphs, the graph structure mutation has less impact on performance than with large-scale graph structures. Second, we have shown in section 2.1 and section 2.2 that opaque bandwidth usage and limited thread scalability cause inefficient concurrent execution of graph analytic applications. And finally, as we will show in section 6, mid-sized graph analytic applications benefit from our scheduling techniques in both execution time and throughput without compromising quality of service.

4 CongraPlus Implementation

This section provides more details about the CongraPlus implementation. Our prototype aims at commodity servers with Linux kernel and state-of-the-art Intel processors. Overall, the CongraPlus prototype we build includes about 600 lines of C++ code for constructing the online scheduler and about 370 lines of C++ code for offline profiling, with a few bash scripts for automated profiling and Python code for data analysis. CongraPlus uses Ligra, a light-weight parallel graph analytics framework [8], as its execution engine.

4.1 Offline Profiling Phase

This section focuses on the details of the resource monitor module. The analytic algorithms and graph structures will be described at section 5.1.

4.1.1 Resource Monitor

The resource monitor takes performance event names as input. It records and interprets the data collected from the corresponding hardware performance counters. *libpfm4* library assists in encoding human-readable names into kernel-compatible strings. Monitor daemon (a background process periodically waken up to record readings from processors' performance counters) collects and interprets the hardware resource usage information. In this paper we set the wake up period to be 0.1 second so as to plot detailed resource usage curves. In real applications, this period can be much larger (1 second to 10 second) as long as the performance counters do not overflow. The overhead of profiling with monitor daemon is low: even with the aggressive wake up period of 0.1 second we did not observe performance degradation of the analytic workload.

As an example, CongraPlus records memory bandwidth usage for each query during the profiling phase. According to Intel Software Developers' Manual, the number of Column Address Strobe (CAS) read and write commands at the integrated memory controller (iMC) side serves as a good indicator of memory bandwidth usage [19]. *libpfm4* suggests the name of this performance event is `UNC_M_CAS_COUNT`.

Require: $Query = \{Q_1, Q_2, ...\}$
Ensure: Each $Q_i \in Query$ be assigned to local scheduler
1: **procedure** QUERYASSIGNER($Query$)
2: `Sort(`$Query$`, ExecTime, Descending)`
3: **for each** $Q_i \in Query$ **do** ▷ LPT algo.
4: $Queue_j \leftarrow Queue_j \cup \{Q_i\}$ where
5: $\cdot \sum_{Q_k \in Queue_j}$ `ExecTime(`Q_k`)` is minimal
6: **end for**
7: **for** $Queue_j$ **do** ▷ Assign queries to local schedulers
8: LOCALSCHEDULER$_j$($Queue_j$, $\emptyset$)
9: **end for**
10: **end procedure**

Require: $Queue = \{Q_1, Q_2, ..., Q_n\}$; $Ctxt$: active queries
Ensure: Each $Q_i \in Queue$ be run with optimal # of threads
1: **procedure** LOCALSCHEDULER($Queue$, $Ctxt$)
2: `Sort(`$Queue$`, ExecTime, Ascending)`
3: $S \leftarrow \emptyset$
4: **repeat** when resources are released or idle
5: **for each** $Q_i \in Queue$ **do**
6: $S \leftarrow S \cup \{Q_i\}$
7: $Ready(S) \leftarrow$ CONTENTIONDETECT(S, $Ctxt$)
8: $T(S) \leftarrow$ ITERATIVEASCENT(S, $Ctxt$)
9: **end for**
10: Find $S' \subseteq S$ such that
11: $\cdot\ S' = S$ at some point in the previous for-loop
12: $\cdot\ Ready(S') =$ `True`
13: $\cdot$ `Perf(`S'`,`$T(S')$`)` is maximized
14: $Q_{LA} \leftarrow Q_i$ where
15: $\cdot\ Q_i \in Queue \wedge Q_i$ has maximal $ExecTime$
16: $Perf_{LA} \leftarrow$ `Gain(`Q_{LA}`,`$T(S')$`) - Loss(`$Ctxt$`)`
17: **if** $Perf_{LA} >$ `Perf(`S'`,`$T(S')$`)` **then**
18: **continue**
19: **end if**
20: LAUNCH(S', $T(S')$, $Ctxt$)
21: $Queue \leftarrow Queue \backslash S'$
22: **until** $Queue = \emptyset$
23: **end procedure**

Require: $S = \{Q_1, Q_2, ..., Q_n\}$; $Ctxt$: current active queries
Ensure: T: optimal number of threads for each $Q_i \in S$
1: **function** ITERATIVEASCENT(S, $Ctxt$)
2: $T' \leftarrow MinimalThread$
3: **repeat**
4: $T = T'$
5: Find $Q_i \in S$ such that
6: $\cdot$ `Gain(`Q_i`,`T`)` is maximized
7: $T'(Q_i) \leftarrow$ `Inc(`$T(Q_i)$`)`
8: **until** $\sum_{Q_i \in Ctxt}$ `BW(`Q_i`,`T`)` + $\sum_{Q_i \in S}$ `BW(`Q_i`,`T'`)` >`Limit`
9: **return** T
10: **end function**

Fig. 5. Pseudo code for algorithms of CongraPlus Online Scheduling Phase. `QueryAssigner`, `LocalScheduler` correspond to **QueryAssigner** and **LocalScheduler** modules in Fig. 4. Optional look-ahead (LA) optimization is marked in red.

Adding up the readings from performance counters at different iMC yields the total memory bandwidth usage.

It is also worth noting that `UNC_M_CAS_COUNT` event is socket-wide, which means it cannot distinguish which application generates how much of the bandwidth traffic if multiple processes are running at the same time. Therefore, the server needs to be idle before commencing profiling (one application after another) to ensure the recorded data correctly reflects the behavior of the profiled application.

4.2 Online Scheduling Phase

Fig. 5 describes the algorithms of query assigner, local scheduler, and iterative ascent in pseudo code.

4.2.1 Query Assigner Algorithm

As mentioned in section 3.2.1, the problem of balancing the query workload among processors is equivalent to the NP-hard multiprocessor scheduling optimization problem. CongraPlus uses the Least Processing Time (LPT) approximation algorithm to obtain good approximation results in the practical time limit. To be specific, the query assigner always assigns the query with the heaviest workload to the queue whose total workload is minimal. This simple yet effective algorithm has been proved to have decent approximation upper bound to the optimal solution [20].

4.2.2 Selection Unit

As part of the generate-and-check method, the selection unit mainly serves as the "checker" who decides which query set can be launched. Two aspects will be considered before the selection unit makes its launching decision: whether this query set incurs memory bandwidth contention and how much speedups can be obtained from this query set and thread allocation scheme.

The contention detection part can be done by comparing the sum of current bandwidth usage and provisioned bandwidth usage with total available bandwidth. That is, contention might be incurred if equation 2 holds

$$\sum_{Q_i \in Ctxt} \texttt{MemBW}(Q_i) + \sum_{Q_i \in S} \texttt{MemBW}(Q_i) \geq \texttt{MemBWLimit} \tag{2}$$

where S is the query set to be examined and $Ctxt$ is the set of currently running queries. In our prototype, `MemBWLimit` constant is measured while performing STREAM benchmark [21].

4.2.3 Iterative Ascent Algorithm

Iterative ascent algorithm is used to compute the optimal thread allocation scheme for a given query set and active queries. The algorithm is described in Fig. 5. Let S be the query set and $Ctxt$ be the set of currently running queries. This algorithm starts with minimum thread allocation for each query $Q_i \in S$. During each iteration, one query that benefits most from running with more thread number has its thread number increased (an "ascent"). This process is repeated until memory bandwidth contention occurs due to aggressive thread allocation. The thread allocation scheme in the last but one iteration is the output of iterative ascent algorithm.

One detail is under existing thread allocation scheme T, how CongraPlus measures the performance gained (denoted as $\texttt{Gain}(Q,\ T)$) of query Q from increasing running threads. Mathematically, equation 3 uses the difference of execution time as the measure of gained performance

$$\texttt{Gain}(Q,T) = \texttt{ExecTime}(Q,t) - \texttt{ExecTime}(Q,\texttt{Inc}(t)) \tag{3}$$

where

$$t = T(Q) \tag{4}$$

`ExecTime()` returns the execution time of a certain query running with certain number of threads, and `Inc()` returns a number of threads larger than its input.

Iterative ascent algorithm increases the performance gained from thread number at every iteration. This property ensures the output of iterative ascent maximizes overall performance gained of the input query set, as well as avoiding bandwidth resource contention.

4.2.4 LookAhead Optimization

Since the local scheduler schedules whenever any query finishes releasing resources, it is possible that the released resources are inadequate to support the following heavy workload query. The motivation behind lookahead (LA) optimization is that reserving released resources by not launching query can lead to more performance reward in the future. This optimization is critical when currently available resources only afford to execute the next query with several threads, but it requires dozens not to keep incoming queries waiting too long.

CongraPlus implements the LA optimization by computing the performance gained from LA strategy ($Perf_{LA}$) and comparing that with the reward of normal strategy. The reward of normal strategy $\texttt{Perf}(S',\ T(S'))$ is defined by equation 5, where $MinT$ stands for minimal number of threads. The LA strategy gains performance reward by reserving resources for future use but also loses immediate performance reward because the scheduler has to wait till at least one query finishes to launch the next query. Equation 6 defines the loss function as a function of currently running queries. Note that Equation 6 is just a rough estimation of the time needed till one query finishes.

$$\texttt{Perf}(S,T) = \sum_{Q\in S} \texttt{ExecTime}(Q,T(Q)) - \texttt{ExecTime}(Q,MinT) \tag{5}$$

$$\texttt{Loss}(Ctxt) = \texttt{Average}(\texttt{ExecTime}(Ctxt)) \tag{6}$$

4.2.5 Thread Allocation Implementation

According to the thread allocation scheme, each query should be bound to specific threads to reduce contention of shared resources. In Linux this can be achieved by calling the `sched_setaffinity()` system call. After the thread allocation is finished, CongraPlus calls Ligra, a lightweight graph analytics framework for shared-memory systems [8], to execute the analytic program.

It is worth noting that enforcing thread allocation scheme naturally enforces the local allocation of memory. In Linux kernel, the default memory policy is set to first-touch, which means the memory pages will be allocated from the node which triggers the page fault. By assigning threads from the same processor to the queries, the thread that triggers the page fault can only come from the query's local node. This implied consequence eliminates expensive remote memory accesses that often occur in NUMA systems and further avoids possible contention on cross-socket link bandwidth.

TABLE 1
Evaluated Server Configuration

Parameter	Setup
# of Sockets	2
Processor	Intel Xeon E5-2630 v4 @ 2.20GHz × 2
Total Logical Cores	20 × 2 [1]
Main Memory	128GB × 2 DDR4 SDRAM [2]
L1 ICache	32KB
L1 DCache	32KB
L2 Cache	256KB
L3 Cache	25MB
Linux Kernel Version	4.4.87
Compiler	g++ 5.4.0[3]

[1] Hyper threading is enabled
[2] Each socket is installed with 128GB DRAM
[3] Comes with full support for Cilk Plus

TABLE 2
Graph Datasets Used in Evaluation

Graphs				
Code	Name	‖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

5 Evaluation Methodology

In this section, we present the hardware configuration and software scheduling schemes that we use in our evaluation. Section 5.1 details what analytic algorithms and graph structures we use in the offline profiling phase. Section 5.2 discusses the scheduling schemes we evaluate, either to serve as the baseline or to demonstrate the effectiveness of our prototype.

We evaluate our prototype on a Linux commodity server with two processors. The configuration of our server is presented in table 5.

5.1 Test Set Synthesis

To evaluate our scheduler prototype, We synthesize 9 query test sets based on six graph analytics algorithms and five widely used graph instances. Table 2 and table 3 present the algorithms and graphs used in our synthesis, and table 4 details the synthesized test sets. We also use the algorithms

TABLE 3
Algorithms Used in Evaluation

Algorithms		To Compute	Property
Code	Name		
AH1	PageRank	relative importance	Bandwidth consuming
AH2	KCore	k-core decomposition	
AH3	BellmanFord	shortest path	
AL1	BFS	breadth-first search	Bandwidth conserving
AL2	Components	connected components	
AL3	Triangle	the number of triangles	

and graphs presented in table 2 and table 3 to perform application synthesis in the offline profiling phase.

Three graphs used in evaluation (i.e., LiveJournal, citePatents, and roadCA) come from a public real-world network data repository [22]. We also consider a random graph (randLocal10m) and a synthetic graph (rMatGraph10m) generated by the RMAT graph generator [23] which follows a power-law distribution. Of the all three real-world graphs, LiveJournal describes, the "friendship" relation in a social network; citePatents records the citing relation among patents; roadCA is generated from the road network of California. We categorize the graphs according to their size and mark LiveJournal, rMatGraph10m, randLocal10m as large graphs (GL) and citePatents, roadCA as small graphs (GS).

We use six representative graph analytics applications implemented with Ligra framework in our evaluation. The applications are categorized according to their memory bandwidth consumption. PageRank, KCore, and BellmanFord are deemed as algorithms of high bandwidth consumption (AH) because of their need of (iterative) graph traversal with additional computation. BFS, Components, and Triangle are deemed as algorithms of relatively low bandwidth usage (AL) because these algorithms tend to spend time mainly on exploring graph structure without doing much computation.

The synthesized test sets include nine query sets, which are shown in table 4. Each query set is specified by N input analytic algorithms and M graph instances and contains $N \times M$ ordered pairs with the form (Algorithm, Graph). For example, the Heavy query set features 15 queries generated by three bandwidth-consuming analytic algorithms and five graph instances.

5.2 Evaluated Schemes

Three scheduling schemes are evaluated in our experiments: the baseline, CongraPlus (denoted as Congra+), and CongraPlus with lookahead optimization (denoted as Congra+LA). We adopt an operating system governed approach as the baseline of our evaluation because it represents the conventional way of dealing with concurrent queries. The baseline scheme blindly parallelizes the execution of every query in the query set, which has a high possibility of incurring resource contention and thus having degraded performance. The CongraPlus scheme follows a judicious co-location principle where the loading balancing algorithm and intelligent thread allocation ensure efficient execution of co-located queries. With lookahead optimization, CongraPlus becomes better in dealing with heavy work queries by reserving shared resources for them.

6 Evaluation Results

We present the results of our experiments evaluated using the above methodology. Our evaluation includes performance, QoS implication, and the scalability of the system.

6.1 Query Throughput

We measure the query throughput of the synthesized test sets under different scheduling schemes. The throughput is defined as the number of queries finished per time unit. Fig. 6 presents the normalized throughput of test sets under all evaluated schemes. The results indicate Congra+LA scheme outperforms the baseline 30% on average, which validates the effectiveness of our workload-aware optimization. Without optimization, Congra+ still achieves 20% throughput improvement on average.

Both Congra+ and Congra+LA achieve high throughput improvement on the Heter-S1 test set. This set includes mainly medium and light workload queries and performing resource-aware scheduling on them has enormous space for improving system performance. During the experiment, both CongraPlus-based schemes dedicate a full processor socket to medium workload queries and reduce the threads allocated to light workload queries so that the contention on shared resources are alleviated. Under the Heter-S1 test set, the above method improves the throughput by 80% as compared to the baseline.

On the Heavy test set, Congra+ and Congra+LA have tremendously different throughput results with up to 2X throughput difference. The Congra+ scheme tends to be conservative in allocating threads for queries, which leads to longer execution time by allocating an insufficient number of threads to the heavy workload queries. Congra+LA, on the other hand, adopts the lookahead optimization to reserve threads for heavy workload queries. The performance of these queries under Congra+LA thus benefits more from an increased number of threads than under Congra+ scheme.

One interesting fact is that both CongraPlus-based schemes do not significantly improve the throughput on Heter-L1 and Heter-S2 test sets. Further investigation on the results shows that both test sets include an exceptionally heavy workload query (AH1 on GL2 for Heter-L1 and AH2 on GS1 for Heter-S2). CongraPlus intelligently dedicates a full processor socket to these queries but only achieves minor speedups because the threads available in a single socket is limited. Even in this worst case situation where the resource requirement of heavy workload queries cannot be fully satisfied, Congra+ and Congra+LA still manage to achieve performance improvement because the baseline scheme incurs excessive hardware resource contention, a detriment to system performance.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2899595, IEEE Transactions on Parallel and Distributed Systems

TABLE 4
Query Sets Used in Evaluation

Query Set	Algorithm	Graph	Alg. Heterogeneity	Data Property
Heavy	{AH1, AH2, AH3}	{GL1, GL2, GL3, GS1, GS2}	A mix of similar graph algorithms	A mix of different graphs
Light	{AL1, AL2, AL3}	{GL1, GL2, GL3, GS1, GS2}		
Heter-L1	{AH1, AL1, AL2}	{GL1, GL2}	More heterogeneous	Large graphs only
Heter-L2	{AH2, AH3, AL3}	{GL1, GL2}		
Heter-S1	{AH1, AL1, AL2}	{GS1, GS2}	More heterogenous	Small graphs 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}		

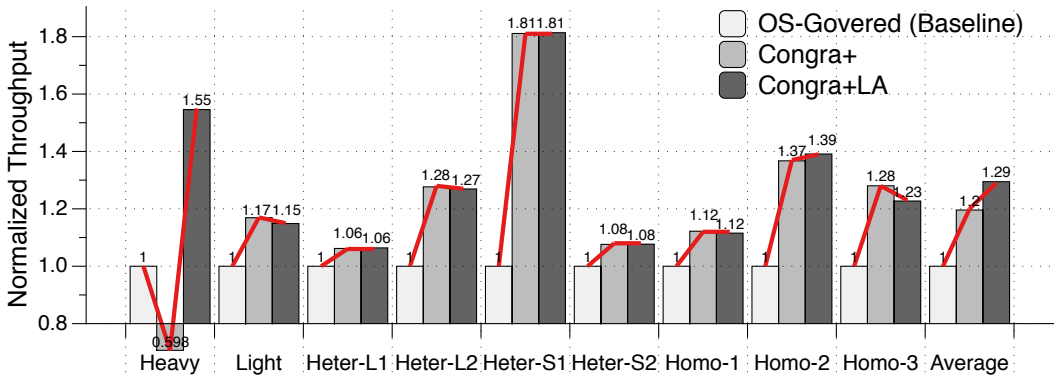


Fig. 6. Normalized query throughput under different scheduling schemes. On average, Congra+LA scheme achieves better overall throughput, followed by Congra+ scheme. Note that under Heavy test set, Congra+ scheme achieves only 60% throughput of the baseline scheme, but Congra+LA scheme improves the baseline throughput by 55%. The result indicates the effectiveness of lookahead optimization in processing heavy workload queries.

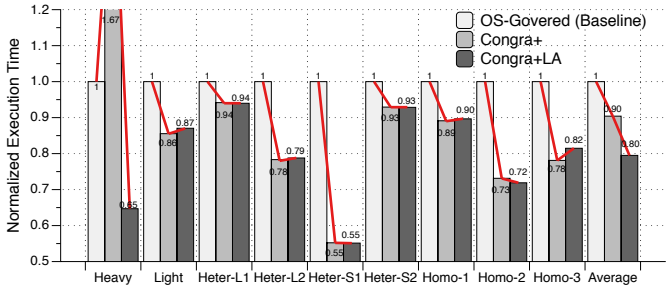


Fig. 7. Normalized test set execution time under different scheduling schemes. On average, Congra+LA scheme reduces the total execution time by 20%.

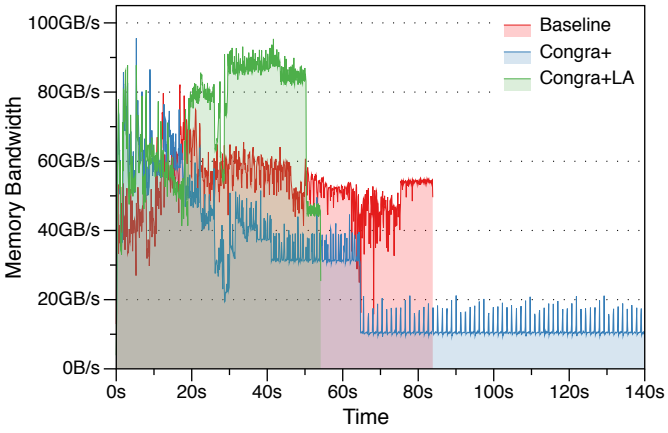


Fig. 8. Memory bandwidth usage trace of Heavy query set under different scheduling schemes. Compared to baseline, Congra+ scheme remains conservative in launching queries which results in under-utilized memory bandwidth. Congra+LA scheme reserves bandwidth resource for these heavy worload queries, a strategy that improves the memory bandwidth utilization.

6.2 Execution Time

We evaluate the impact of CongraPlus on test set execution time, which is defined as the total time needed to finish all queries in a specific test set. Fig. 7 presents the normalized execution time of each test set. The results of execution time are generally compatible with those of throughput, where high throughput test sets also have short execution time. On average, Congra+ and Congra+LA reduce the execution time of query sets by 10% and 20%, respectively.

The utilization of memory bandwidth during the experiment is crucial to better understanding the impact of our scheduling schemes. Fig. 8 presents the memory bandwidth usage trace of the Heavy test set under different schemes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2899595, IEEE Transactions on Parallel and Distributed Systems

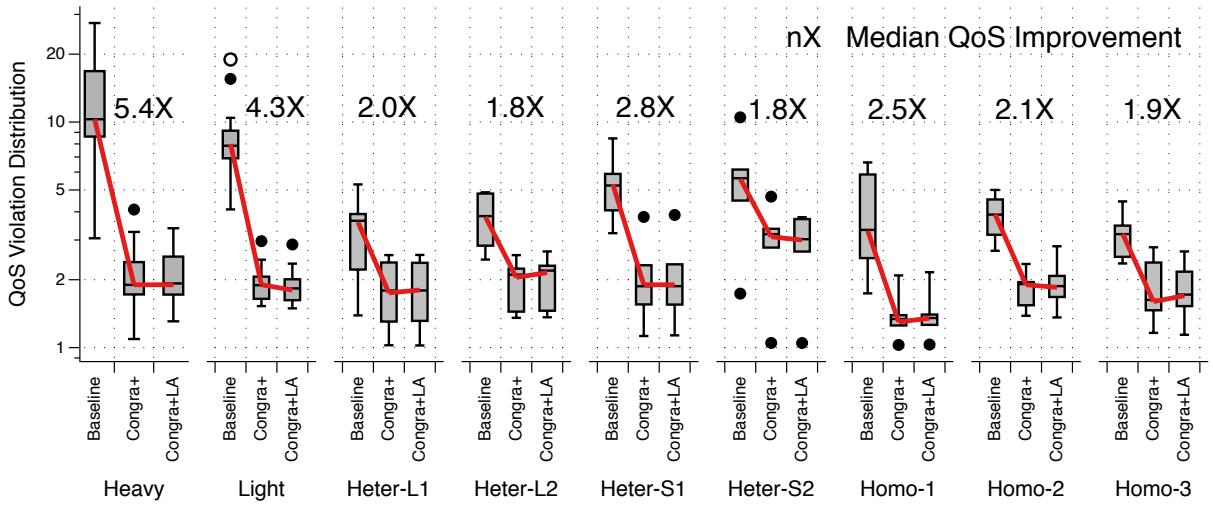
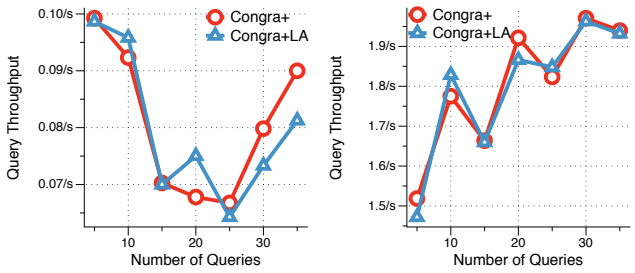


Fig. 9. The distribution of query set's QoS violation under different scheduling schemes (smaller is better). QoS violation is defined as the expected running time (as in the profiling phase) divided by the query's actual response time. The chart marks the maximum, median, minimum, upper quartiles, and lower quartiles for each set of violation numbers. Congra+ and Congra+LA have significantly smaller QoS violation compared to baseline in all test sets, with their QoS improvement ranging from 1.8X to 5.4X.



(a) Heavy workload queries (b) Light workload queries

Fig. 10. Query set finish time under different scheduling schemes. In both cases, the query throughputs of Congra+ and Congra+LA do not decrease with the increase of number of queries. The stability of CongraPlus system throughput indiecates good scalability of the system even under large number of concurreng queries.

The conservative thread allocation of Congra+ scheme leads to fewer threads allocated to each query, and therefore the much lower bandwidth utilization. The baseline scheme aggressively launches all queries with all available threads, causing severe contention on resources such as reservation stations, reorder buffer, cross-socket link, etc. Such contention prevents the processors from issuing more memory access instructions and limits the bandwidth utilization. Through judicious query co-location, Congra+LA scheme avoids the above resource contention and achieves much higher bandwidth utilization. The insights from this experiment are that the memory bandwidth utilization is critical to system performance, and that query co-location without causing resource contention is a viable approach to achieve better system performance.

6.3 Quality of Service

Another interesting metric we evaluate is the QoS (Quality of Service)[2] of our concurrent graph analytics system. We define the QoS violation of a query to be its response time (total time spent from being submitted to the server until successful retirement) dividing the expecting execution time as it is profiled. Smaller QoS violation suggests users will wait shorter time than what is needed for the results to return, a crucial property to improve user experience in the cloud side. We present the QoS violation of each query in Fig. 9 and also marks the QoS improvement of the median query. Both CongraPlus-based schemes achieve significant improvement ranging from 1.8X to 5.4X.

The QoS violation improvements should be mainly attributed to the optimal launch sequence and alleviation of resource contention. In determining the launch sequence, CongraPlus scheduler prioritizes light workload queries, a strategy to improve their QoS and reserve more resources for heavy workload queries in the future. Resource contention is also alleviated (by techniques discussed in section 3.2.2) so that the queries can run without being "dragged down" by other queries. On the contrary, under the baseline scheme, the execution time of each query becomes longer than expected because all queries contend the shared resources. This experiment result suggests that CongraPlus-based schemes have the potential to not only improve the overall system performance, but also the experience of the individual user compared to the conventional approach.

6.4 Scalability

We also evaluate the scalability of CongraPlus-based schemes, which reflects the system's performance under

2. The authors are aware that the definition of QoS in cloud services is different from what used to demonstrate our ideas here. We use this term to explain CongraPlus's capability to improve per-query response time, a metric related to the quality of graph analytics services.

a growing amount of concurrent queries. We conduct a stress test with numerous either heavy or light workload queries and record the query throughput under CongraPlus based schemes[3]. Fig. 10 presents the query throughput with respect to the number of queries.

For heavy workload queries, the query throughput first drops then increases back to normal. With a small number of heavy workload queries, the scheduler can dedicate full resources to a few queries. Increasing query number inevitably decreases throughput at this stage as more queries have to wait for resource released by finished queries. For light workload queries, the query throughput first rises then remain relatively stable. The rising phase corresponds to the stage when all lightweight queries cannot fully utilize system resources, and increasing query number increases co-located queries. When the number of lightweight queries grows so large that efficient co-location of all queries is impossible, some queries have to wait for others to release resources, which leads to stable throughput. This result shows CongraPlus-based schemes are capable of achieving stable throughput even when the number of concurrent queries is large.

7 Related Work

In this section we discuss existing research that is most relevant to our work.

7.1 Graph Analytics Framework

Many prior works have investigated the parallel processing of graph analytics workloads [1], [2]. For example, GPS [10] combines message-passing based design with dynamic graph partition to achieve efficient distributed graph processing. With graph partition and some other novel techniques, GraphChi shows that a shared-memory-based design with a disk as secondary data storage can process extremely large graphs efficiently [24]. Trinity, a graph analytics engine deployed at the cloud, jointly optimizes the memory management and network communication of graph analytics applications [25]. GraphX is a general-purpose graph processing framework built upon Spark, a widely-used distributed dataflow system [11]. It enjoys the advantage of existing dataflow system infrastructure such as fault-tolerance and broad applicability, as well as high efficiency. Most of these works focus on accelerating the processing of a single large graph.

In our work, we implement CongraPlus on top of Ligra, a light-weight shared-memory framework, as our graph processing engine [8]. Ligra uses carefully implemented parallel abstractions and multi-threading to speed up graph analytics applications.

7.2 Concurrent Graph Analytics

Processing concurrent graph analytics queries in a multi-user environment has been gaining increasing attention [4], [5], [26], [27]. Kim et al. [4] devised methods for enabling efficient processing of multiple graph queries using MapReduce. Feher et al. [5] utilized the parallelization mechanism of MapReduce to solve the graph pattern matching problem. Xue et al. [26], [27] investigated concurrent graph processing queries and proposed a graph structure sharing mechanism to avoid memory storage waste. Then et al. [28] proposed MS-BFS, which makes use the small-world network property of most real-world graphs, to efficiently support concurrent BFS traversals on the same graph. Ren et al. [29] explored the multiple-query optimization of subgraph isomorphism search with efficient common subgraph detection, storage, and query execution. Liu et al. [30] proposed iBFS algorithm that is capable of efficiently executing concurrent BFS traversals on a GPU by exploiting the shared frontiers among different traversals and optimized bitwise operations on GPUs. These works focus on adapting existing graph processing models to a multi-user environment and optimizing the performance of concurrent analytic queries by taking advantage of the common computation among different queries on a single, shared graph. In contrast, CongraPlus targets concurrent graph analytic queries that potentially run vastly different analytic algorithms on different graph datasets. To achieve high performance, CongraPlus scheduler uses a scheduling mechanism that is jointly guided by hardware resources and graph analytics query characteristics and aims to achieve efficient co-location of analytic queries.

7.3 Architectural Support for Graph Processing

Another group of research work focuses on the exploring graph processing workload from an architectural perspective or designing new hardware architecture to support such workloads. For example, Ahmad et al. [31] devised a benchmark suite to help understanding multi-threaded graph algorithms on shared-memory multicore processors. Beamer et al. [13] show that various graph analytics workloads do not fully utilize the system's off-chip memory bandwidth. Several recent studies have devised domain-specific accelerators for graph analytics workloads [6], [7], [14]. Our work explores improving the performance of concurrent graph analytics frameworks through resource-aware scheduling on a shared-memory commodity server.

8 Conclusion

There is a growing demand for processing and analyzing a variety of small graphs concurrently on a shared-memory commodity server. Nevertheless, the opaque memory bandwidth usage and the limited thread scalability of shared-memory graph analytics applications make concurrent graph analytics a challenging task. In this work we propose CongraPlus, a novel scheduling mechanism that enables efficient query co-location for NUMA architecture shared-memory systems. By utilizing the information collected through offline profiling, CongraPlus achieves balanced workloads across all processors and computes the optimal query launch sequence and thread allocation scheme. Our evaluation over a wide range of synthesized applications shows that CongraPlus significantly improves the throughput and execution time of the conventional approach. It also has good scalability under a massive amount

3. The experiment is not conducted under the baseline scheme because it has decreasing throughput when query number increases, a clear indication of its poor scalability.

of queries and enhances the quality of concurrent graph analytics services. We expect CongraPlus will enable shared-memory machines of NUMA architecture to better serve the needs of graph analytics services on the cloud side.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (#2018YFB1003500). We thank all the anonymous reviewers for their time and feedback.

REFERENCES

[1] N. Doekemeijer and A. L. Varbanescu, "A Survey of Parallel Graph Processing Frameworks ," Delft University of Technology, Tech. Rep. PDS-2014-003.

[2] R. R. McCune, T. Weninger, and G. Madey, "Thinking Like a Vertex - A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing." *ACM Comput. Surv.*, vol. 48, no. 2, pp. 1–39, 2015.

[3] B. Chandramouli, J. Claessens, S. Nath, I. Santos, and W. Zhou, "RACE - real-time applications over cloud-edge." *SIGMOD Conference*, p. 625, 2012.

[4] S.-H. Kim, K.-H. Lee, H. Choi, and Y.-J. Lee, "Parallel processing of multiple graph queries using MapReduce," in *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*. Citeseer, 2013, pp. 33–38.

[5] P. Fehér, M. Asztalos, T. Vajk, T. Mészáros, and L. Lengyel, "Detecting subgraph isomorphism with MapReduce," *The Journal of Supercomputing*, vol. 73, no. 5, pp. 1810–1851, Oct. 2016.

[6] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, 2015, pp. 105–117.

[7] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. M. Burns, and O. Ozturk, "Energy Efficient Architecture for Graph Analytics Accelerators," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 166–177.

[8] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, 2013, pp. 135–146.

[9] P. Pan and C. Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 217–224.

[10] S. Salihoglu and J. Widom, "GPS - a graph processing system." *SSDBM*, p. 1, 2013.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 599–613.

[12] Low, Yucheng, Bickson, Danny, Gonzalez, Joseph, Guestrin, Carlos, Kyrola, Aapo, and Hellerstein, Joseph M, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[13] S. Beamer, K. Asanovic, and D. A. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, 2015, pp. 56–65.

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 56:1–56:13.

[15] S. Chakrabarti, "Dynamic personalized pagerank in entity-relation graphs," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 571–580. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242650

[16] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 349–362. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151013

[17] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13. New York, NY, USA: ACM, 2013, pp. 505–514. [Online]. Available: http://doi.acm.org/10.1145/2488388.2488433

[18] M. R. Garey and D. S. Johnson, *Computers and intractability*. wh freeman New York, 2002, vol. 29.

[19] *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, Intel Corporation, Oct. 2016.

[20] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[21] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[22] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," Tech. Rep., Jun. 2014.

[23] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, Dec. 2013, pp. 442–446.

[24] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 31–46.

[25] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, 2013, pp. 505–516.

[26] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an efficient, low-cost system for concurrent graph processing," in *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, 2014, pp. 227–238.

[27] J. Xue, Z. Yang, S. Hou, and Y. Dai, "Processing Concurrent Graph Analytics with Decoupled Computation Model," *IEEE Trans. Computers*, vol. 66, no. 5, pp. 876–890, 2017.

[28] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. N. 0001, and H. T. Vo, "The More the Merrier - Efficient Multi-Source Graph Traversal." *PVLDB*, vol. 8, no. 4, pp. 449–460, 2014.

[29] X. Ren and J. Wang, "Multi-Query Optimization for Subgraph Isomorphism Search." *PVLDB*, vol. 10, no. 3, pp. 121–132, 2016.

[30] H. Liu, H. H. Huang, and Y. Hu, "iBFS," in *the 2016 International Conference*. New York, New York, USA: ACM Press, 2016, pp. 403–416.

[31] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, 2015, pp. 44–55.

Peitian Pan received his B.S. degree from the Department of Computer Science and Engineering in Shanghai Jiao Tong University. His research interests include parallel graph analytics and domain-specific accelerator design. He will continue his study as a PhD student at Cornell University in Fall 2018.

Chao Li received his Ph.D. degree from the University of Florida in 2014. He is currently a tenure-track associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research mainly focuses on computer architecture and systems for emerging applications. He has received a Best Paper Award from IEEE HPCA in 2011 and a Best Paper Award from the IEEE Computer Architecture Letter in 2015.

Minyi Guo is a Zhiyuan Chair Professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, China. He received his PhD degree in computer science from the University of Tsukuba, Japan. His research interests include parallel and distributed computing, compiler optimizations, computer architecture, cloud computing and big data. Prof. Guo has more than 250 publications in major journals and international conferences in these areas. He is a Fellow of IEEE.