GRAM: Scaling Graph Computation to the Trillions

Ming Wu[†], Fan Yang[†], Jilong Xue[†§], Wencong Xiao[†‡], Youshan Miao[†*], Lan Wei[†◇], Haoxiang Lin[†], Yafei Dai[§], Lidong Zhou[†]

[†]Microsoft Research, [§]Peking University, [‡]Beihang University, [*]University of Science and Technology of China, [◇]Institute of Software - Chinese Academy of Sciences

Abstract

GRAM is an efficient and scalable graph engine for a large class of widely used graph algorithms. It is designed to scale up to multicores on a single server, as well as scale out to multiple servers in a cluster, offering significant, often over an order-of-magnitude, improvement over existing distributed graph engines on evaluated graph algorithms. GRAM is also capable of processing graphs that are significantly larger than previously reported. In particular, using 64 servers (1,024 physical cores), it performs a PageRank iteration in 140 seconds on a synthetic graph with over one trillion edges, setting a new milestone for graph engines.

GRAM's efficiency and scalability comes from a judicious architectural design that exploits the benefits of multicore and RDMA. GRAM uses a simple message-passing based scaling architecture for both scaling up and scaling out to expose inherent parallelism. It further benefits from a specially designed multi-core aware RDMA-based communication stack that preserves parallelism in a balanced way and allows overlapping of communication and computation. A high degree of parallelism often comes at the cost of lower efficiency due to resource fragmentation. GRAM is equipped with an adaptive mechanism that evaluates the cost and benefit of parallelism to decide the appropriate configuration. Combined, these mechanisms allow GRAM to scale up and out with high efficiency.

Categories and Subject Descriptors C.2.4 [*Computer Systems Organization*]: Computer-Communication Networks—Distributed Systems; C.4 [*Computer Systems Organization*]: Performance of Systems

General Terms Design, Experimentation, Performance

Keywords Graph computation engine, RDMA, scalability

SOCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA.
© 2015 ACM. ISBN 978-1-4503-3651-2/15/08...$15.00.
DOI: http://dx.doi.org/10.1145/2806777.2806849

1. Introduction

Graph structures are prevalent and offer valuable information about relations and connections. For examples, social graphs help reveal the influence of each individual and the formation of communities. Distributed graph computation engines (e.g., Pregel [24], PowerGraph [13], GraphX [14], and PowerLyra [8]) have been proposed to mine insights from graphs through multiple iterations of vertex-centric computation and communication along the edges, with iterations often separated by barriers. Graph computation is particularly challenging to scale because graphs are notoriously hard to partition and small random accesses are dominant in such a workload. Yet, we are seeing the real needs to process increasingly large graphs on the order of hundreds of billions and even trillions of edges [10, 19]. Even with detailed performance studies [15] on the existing graph engines, there remains a lack of understanding on the scaling of such graph engines and its cost.

We present GRAM, a new graph engine that takes advantage of the modern hardware available in data centers, with multi-core servers, abundant memory in a cluster, and high-speed network with RDMA (Remote Direct Memory Access) support [17]. GRAM's design and implementation are further guided by a careful study on performance, scalability, and cost, with the following key design choices.

First, GRAM adopts a deceivingly simple model for both scaling up and scaling out, where we affinitize a thread with each core, with threads (even within a server) communicating through message passing. We choose message passing over shared-memory primitives even for the intra-server case because our evaluation results (Section 5) reveal that, with sufficient batching, message passing exhibits better performance than shared-memory primitives due to better locality and fewer inter-core communications. This model fully exposes inherent hardware-level parallelism. The simplicity of the model makes it easy to understand any potential scalability bottleneck introduced by the software stack, while allowing an efficient implementation that scales.

Second, to scale out, GRAM incorporates a multicore-aware RDMA-based communication stack. The communication stack maximizes parallelism through fine-granularity message buffer pools to reduce interference among different communicating thread-pairs. Because graph computation often involves barriers across iterations, load balancing across threads becomes key to reducing overall computation latency. The design of GRAM's communication stack carefully takes balance into account: it adopts a symmetric and balanced design of a light-weight coordination mechanism and supports NUMA-aware allocation to cope with the underlying heterogeneity due to servers' NUMA architecture.

The benefit of using RDMA goes beyond achieving high bandwidth and low latency in network communication. GRAM seizes the opportunity to overlap communication with computation because of reduced CPU involvement due to RDMA. We observe the surprising result, where GRAM is shown capable of achieving the same scaling out across servers as scaling up on a local server: the communication cost, although significantly higher across servers than within a server, can be masked by the overlapped computation.

There is a tension between masking communication cost and scaling. As scaling increases the level of parallelism and (consequently) reduces the computation cost on each thread, there might no longer be sufficient computation to mask the communication cost. An increased level of parallelism could also lead to fragmentation, where data and buffers are partitioned at a finer granularity, thereby reducing the efficiency of memory uses and also the communication channels. In those cases, GRAM adapts to multiplex and batch at a coarse granularity to improve communication efficiency.

GRAM has been fully implemented and extensively studied on a cluster connected by an RDMA-capable high-speed network. We evaluate GRAM with common graph computation algorithms, such as page rank, weakly connected components (WCC), and single-source shortest path (SSSP), on a variety of representative graphs with sizes ranging from 1 billion edges to 1 trillion edges. The evaluations not only show that GRAM scales up and out, but also reveal the important limiting factors to scalability and validate our design choices. Overall, GRAM is competitive ($\sim 33\%$ slower) with a highly optimized single-threaded implementation on a single core, achieves lower execution time at 4 cores, and scales up and out to 1,024 cores to handle huge graphs with more than 1-trillion edges.

Although GRAM is designed for and evaluated in a specific (RDMA-capable) environment, we focus on presenting the insights from our detailed analyses that are valuable to building efficient and scalable computation engine in general, even in a different configuration, where different design trade-offs are made.

The rest of the paper is organized as follows. Section 2 describes the class of graph algorithms that GRAM targets and their characteristics that affect GRAM's design. We present the overall architectural design of GRAM in Section 3 and describe GRAM's communication stack in Section 4. The evaluation results are shown in Section 5, followed by a discussion on the general lessons we have learned from GRAM in Section 6. We survey related work in Section 7 and conclude in Section 8.

2. Graph Computation: An Overview

The recent work on graph computation [13, 14, 23, 24] advocates a vertex-based graph-parallel computation model. Given a graph $G = \{V, E\}$, a *vertex-program* P is executed on each vertex $v \in V$ and interacts with neighboring instances $P(u)$, where $(u, v) \in E$. A vertex-program often maintains an application-specific state associated with vertices and with edges, exchanges the state values among neighboring vertices, and computes new values during graph computation. For example, each iteration of a PageRank graph computation involves a vertex propagating the associated rank value divided by its degree to the neighbors and computing its new rank value based on the current value and the values from neighbors.

Many graph computation algorithms, such as PageRank, are executed iteratively in a Bulk Synchronous Parallel model [38], where a barrier is inserted between two consecutive iterations. Such a model is simple to understand and tends to offer well-defined convergence properties. While some graph engines [14, 24] adopt a synchronous model, others [13, 23] employ asynchronous scheduling of vertex-programs to avoid barriers. Recent research [15] however shows that the asynchronous scheduling in GraphLab [23] does not perform as well as its synchronous scheduling for PageRank due to lock contention, lack of message batching, and the expensive termination detection. Our experience so far has also favored the synchronous model. GRAM therefore focuses on optimizing the synchronous case, but also introduces a relaxed model for graph algorithms such as connected components, where allowing multi-hop propagation in each iteration helps reduce the number of iterations and expedite convergence.

3. GRAM Architecture

GRAM's design is driven by the characteristics of graph computation described in Section 2: (i) graph parallelism at the vertex level, (ii) dominant small random accesses in each iteration, and (iii) barriers across iterations. GRAM therefore adopts a simple scaling architecture to exploit the inherent parallelism of graph computation, as shown in Figure 1. In this architecture, each worker thread is assigned to a dedicated CPU core and owns a graph partition. All graph partitions are loaded into main memory at the beginning of the computation. GRAM lays out an in-memory graph partition using data structures similar to the ones used in Grace [32] with a vertex array containing vertex data (e.g., rank value) and an edge array containing the edges grouped by the corresponding source vertices.

During graph computation, each pair of threads communicate through a message-passing mechanism, no matter whether they are running on the same server or not. We opt for message-passing, rather than shared-memory primitives even for threads on the same server, mainly to allow batching of small random inter-core data accesses. The simple architecture also makes it easy to identify the limiting factors to

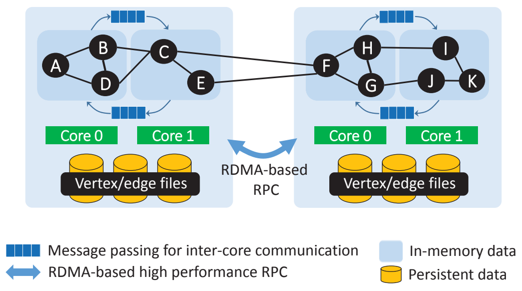


Figure 1. Overview of GRAM architecture.

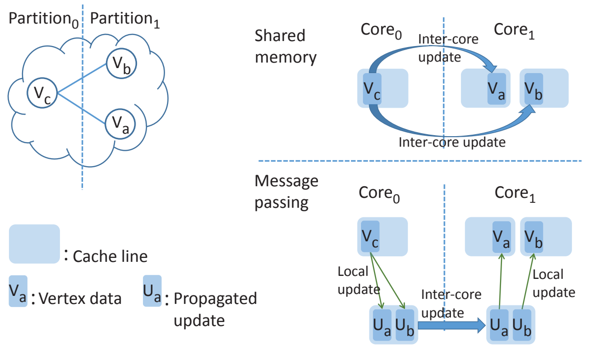


Figure 2. Thread message passing vs. shared-memory

scaling graph computation as the architecture fully exposes core-level parallelism. Optimizing communication overhead is clearly the key to scaling. In addition, due to common use of barriers in graph computation, load balancing across cores is another important goal because in each iteration the computation time is dominated by the slowest thread.

Modern multi-core architecture usually provides a shared-memory abstraction for all cores to access any memory address. It seems to be a natural choice to adopt this abstraction for graph workloads on single multi-core server. Several existing graph engines take this choice, e.g., GraphLab/PowerGraph [13, 23], Grace [32], and Chronos [16]. Surprisingly, our analysis shows that the shared-memory model actually offers suboptimal performance for graph computation, compared to using message passing, largely due to the dominant small random access pattern in the graph workload. Graphs tend to be complex and highly dimensional: they are hard to partition or to lay out in a way that exhibits good locality: neighboring vertices cannot always be placed together. Typical graph computation workloads such as PageRank, Spectral Partitioning [37], and sparse matrix-vector multiplication, incur a large number of random data accesses at the vertex granularity that often have to go across cores and servers to propagate data between neighbors. One source of overhead for shared-memory is locking and lock contention because vertices might have common neighbors and different threads might want to update the value associated with the same vertex. The other source of overhead can be the cost of inter-core communication due to false sharing or bad data locality across cores. With message passing, those random inter-core (inter-server) data accesses can often be batched naturally across vertices.

Figure 2 illustrates one of the potential benefits of inter-thread message passing on mitigating inter-core communication overhead. The example graph contains three vertices V_a, V_b, and V_c, with V_c in a partition assigned to $Core_0$, and V_a and V_b (which are neighbors of V_c) in a partition assigned to $Core_1$. Consider the vertex data propagation from V_c to V_a and V_b, and assume that the vertex data of V_a and V_b are in different cache lines. If the thread on $Core_0$ directly updates V_a and V_b through shared-memory primitives, the operations may involve two inter-core communications at the cache-line level. In contrast, through message passing, the thread on $Core_0$ first packs the two updates into a single cache line of message buffer, and then the cache line of data will be fetched by the thread on $Core_1$ through only one inter-core communication at the cache-line level. Fundamentally, with message passing, a random inter-core data access pattern is transformed into a sequential one through batching to improve locality and maximize communication throughput. This benefit turns out to be significant for the graph computation workload even though message passing introduces the additional overhead of intra-core memory copy to a message buffer.

Message passing is a natural choice for inter-server communication and our analysis also shows that it performs well compared to the shared-memory primitives offered by RDMA, for similar reasons. This leads to a simple architecture that allows GRAM to scale up and out. For scaling out, GRAM uses a specially designed RDMA-based networking stack that (i) enables largely concurrent data transfers between thread pairs on different servers, (ii) balances the coordination load on multiple cores and copes with heterogeneity due to NUMA, and (iii) enables overlapping of communication with computation and adaptively adjusts the multiplexing granularity based on the ratio of communication and computation. We present the detailed design next.

4. Multicore-Aware Communication Stack with RDMA

To scale out, GRAM has a specially designed communication stack that (i) takes into account the server's multi-core architecture and aims to preserve core-level parallelism in a balanced way, and (ii) leverages RDMA, which is becoming available on data centers thanks to technologies such as RDMA over Converged Ethernet (RoCE) [17]. RDMA *queue pairs* provide reliable connections, where requests on registered remote memory regions are sent directly to the NIC without involving the kernel and are handled by the remote NIC without involving the remote CPU. GRAM is built on FaRM's message-passing mechanism [12] that is implemented using RDMA writes: an RPC sender transmits data to the receiver through a unidirectional message-passing channel with a dedicated circular buffer on the receiver side or the *receiver buffer*. The RPC sender uses RDMA writes to deliver data to the (remote) receiver buffer directly. GRAM does not use the shared memory primitives in FaRM based on the same principle as in the case for the intra-server case: for the graph computation workload, it is more effective to transform small random data shuffling into batching with the message-passing primitives. In contrast, FaRM's driving scenarios are storage abstractions that support queries and (transactional) updates, where latency of individual request is the main concern, rather than the parallel computation workloads that GRAM targets. This design choice also leads to a uniform scaling model for GRAM.

4.1 Overlapping communication and computation

For graph computation, the main benefit of RDMA comes from the opportunity to overlap communication and computation to hide communication latency, because CPU is no longer involved in RDMA-based communication. The design goal of GRAM is therefore to allow all cores to execute concurrently on graph computation tasks and to schedule communication to overlap with the computation. This naturally leads to a symmetric threading model that pins threads to hardware cores as in FaRM. Each thread is programmed in an event-driven style to execute graph computation work items and to poll for RDMA-based communication.

The computation includes enumerating vertex and edge arrays to process vertex data, preparing and filling sender buffers for propagating data to neighbors on remote servers, and processing data in receiver buffers from remote servers. In practice, GRAM's local computation tasks for enumerating graph data and sending messages are given $100\mu s$ maximum slices, while the tasks for processing received messages have an elastic time slice that could be much larger. The intention of this setup is to give priority to message consuming tasks when necessary (to prevent excessive message queuing in sender buffers) while offering a balance between computation and communication. Our experience shows that GRAM's performance is not particularly sensitive to the scheduling setting and we have not found a need to implement a more sophisticated scheduling mechanism.

Overlapping communication and computation allows GRAM to scale out as effectively as scaling up, even though cross-server communication is significantly more expensive and with much lower throughput compared to intra-server communication and memory bandwidth. As shown in Section 5, we observe the same performance with multiple GRAM workers running on multiple servers, each on a different server, as with the same number of GRAM workers running on multiple cores of the same server.

4.2 NUMA-awareness and balanced multiplexing

Because many graph computation algorithms use barriers across iterations, having all threads complete each iteration at approximately the same time is important to end-to-end performance. This is also why GRAM starts with a symmetric threading model. To cope with the existing heterogeneity due to NUMA, all GRAM's data and buffer allocations are NUMA-aware to minimize the chance of a thread accessing data or a buffer on a different NUMA node. For the graph data, GRAM guarantees that the graph partition a thread owns is allocated from the same NUMA node as the core that the thread is dedicated to.

The communication stack must also preserve symmetry across threads. GRAM's RPC abstraction is at the granularity of threads, where a thread has a separate RPC connection with each of the remote threads. A naive implementation would dedicate a receiver buffer to each pair of threads, thereby requiring $O(M^2 \times N)$ buffers on each server, where N is the number of servers and M the number of cores per server. The total number of receiver buffers quickly becomes a limiting factor for scaling as the number of servers increases in the system. (With each receiver buffer of 64MB and 16 cores per server, having a receiver buffer per pair of threads/cores would need 1TB per server in a 64-server setting.) Receiver buffers are therefore multiplexed in GRAM. A *channel* with a dedicated receiver buffer connects a group of sending threads with a group of receiving threads. We always ensure that a receiver buffer is on the same NUMA node as the receiving threads, as those threads will access the buffer. To enable multiplexing and reduce contention, a *coordinator* is introduced both on the sending side and on the receiving side. The coordinator on the sender side keeps the tail of the receiver buffer. When a thread wants to send a message, it asks the coordinator to reserve the receiver buffer for the message to be sent. And the coordinator will update the tail address accordingly to prevent other threads from writing to overlapped address space. The coordinator on the receiver side polls the receiver buffer. When there is a new message, it peeks the header to find out which thread this message is for. It then notifies the corresponding thread along with the offset and size of the message in the buffer, so that the corresponding thread can read the message from the receiver buffer directly. After it is done with the message, the thread informs the coordinator that the buffer region in the receiver buffer can be reused.

The responsibility of a coordinator is assigned to a core. Our experience shows that an uneven distribution of coordinators to cores would adversely impact GRAM's performance due to imbalanced coordination load on the cores. In GRAM, we always configure the system so that the coordinator's responsibility is distributed approximately evenly across threads: there are multiple channels and therefore multiple coordinators in each group of threads on a server as the server communicates with multiple other groups on peer servers. In a configuration where the number of servers is much smaller than the number of cores on each server, GRAM creates an appropriate number of groups and a sufficient number of channels per group to allow an approximately even assignment.

Figure 3 illustrates a case where two servers, servers 1 and 2, communicate with server 0. Each server has two groups (e.g., threads within a NUMA node can form a group). Each group establishes a channel to the corresponding group of a remote server, threads within the group shares

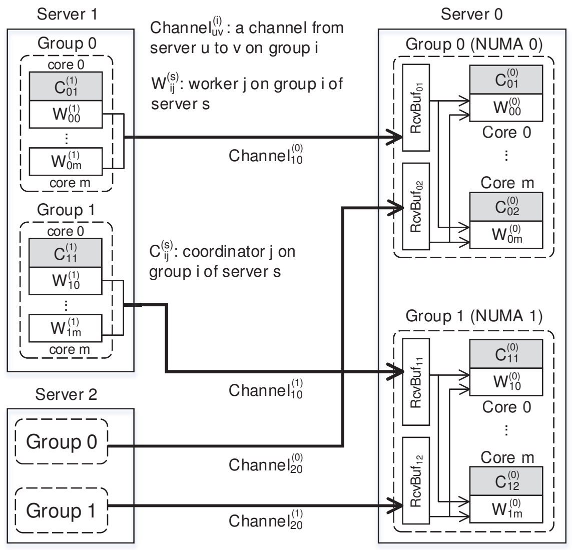


Figure 3. NUMA-aware and balanced multiplexing

the same channel. Coordinators $C_{01}^{(1)}$ and $C_{11}^{(1)}$ are introduced to mediate on the sender side of $Channel_{10}^{(0)}$ and $Channel_{10}^{(1)}$, respectively. Coordinators $C_{01}^{(0)}$ and $C_{11}^{(0)}$ are mediating on the receiver side for the two channels. (For simplicity, we omit details in server 2 and other channels/-coordinators in the configuration, as the setup is symmetric. On group 0 of server 0, another coordinator $C_{02}^{(0)}$ is introduced to mediate $Channel_{20}^{(0)}$ from group 0 of server 2. For load balance in group 0 of server 0, we should assign $C_{01}^{(0)}$ and $C_{02}^{(0)}$ to different threads running on different cores in the group. In Figure 3, if a thread in group 0 of server 1 wants to send messages to a thread in group 1 of server 0, it would ask coordinator $C_{11}^{(1)}$ in group 1 of server 1 to mediate the communication.

4.3 Trade-off between parallelism and efficiency

GRAM's design has paid special attention to exposing high parallelism in the communication stack. For example, a thread maintains a sender buffer for each remote thread because the RPC provides a thread-granularity abstraction: in this configuration, no extra coordination or de-multiplexing is needed on an RPC receiver, i.e., each receiver thread processes messages delivered to it in the receiver buffers independently, thus maximizing parallelism. Such a one-per-peer-thread sender buffer design works best when there is sufficient parallel computation that can keep all cores busy, while hiding the communication cost. This is not always the case; for example, when some phases of certain graph computation do not involve all vertices in a graph or when the cluster scale is large and a graph is partitioned to a point where the computation on a partition is no longer sufficient to mask the communication cost. Our evaluation has shown that there is non-negligible overhead associated with each RPC call to prepare the buffer, invoke the communication stack, and transmit on the network [28]. Sending many small messages is therefore costly and inefficient. A batch size of 512KB is needed for obtaining high communication efficiency. GRAM's original design essentially favors parallelism over batching and is no longer appropriate in this setting. We therefore introduce a different setup to trade off parallelism for communication efficiency by merging all the sender buffers of a thread to all threads on the same remote server (i.e., a one-per-peer-server buffer arrangement). In this scheme, RPC is then done at a coarse granularity to allow better batching. The merged buffer is sent to a delegate thread in the remote server, which de-multiplexes the message and re-dispatches to the appropriate threads through thread message passing. This introduces extra overhead, but our evaluation shows that this trade-off works well when cluster scale is large.

To achieve better performance for all the cases, GRAM adopts a simple strategy to adaptively switch between the two schemes in a graph computation. When GRAM observes that the time spent on non-overlapped communication is larger than a threshold, it will switch to one-per-peer-server buffer scheme in the next iteration, and vice versa.

5. Evaluation

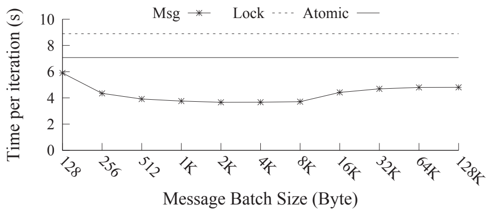
GRAM is implemented in about 6,000 lines of C++ code. In this section, we present the detailed evaluation results to support our design choices and demonstrate the efficiency and scalability of GRAM, as well as comparing with other state-of-the-art engines such as Naiad [27] and GraphX [14].

Experimental setup. We evaluate GRAM on a cluster of commodity multi-core servers. Each server runs Windows Server 2012 R2 on dual 2.6 GHz Intel Xeon E5-2650 processors (16 physical cores and 32 logical cores), 256GB of memory, and a Mellanox ConnectX-3 InfiniBand NIC with 54Gbps bandwidth.

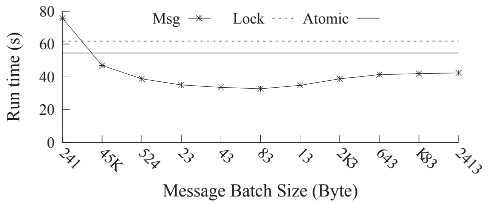
Our evaluation was performed using 3 graph applications, PageRank, weakly connected component (WCC), and single-source shortest path (SSSP), on 6 graph datasets. We run 5 iterations in all our PageRank experiments. Table 1 lists the graph datasets in the ascending order of graph sizes. The "edge size" column shows the memory footprint of the edge lists in the corresponding graphs. The data structure to store edge information usually dominates the memory footprint in a graph engine. The first 5 graphs are publicly available real-world social or web graphs ranging from 1.5 billion edges to 128 billion edges. Note that the original uk-union graph has 5.5 billion edges. To have a graph sufficiently larger than uk-2007-05, we make all edges bidirectional. The last and the largest graph is a synthesized graph with 1.2 trillion edges, where the degree of each vertex is set by a random number from 1 to 281 and each edge from the vertex is randomly connected to any of the remaining vertices. To our best knowledge, this is so far the largest graph processed in a distributed in-memory graph engine [10]. Unless stated otherwise, all the graphs are randomly partitioned.

Graph name	# of vertices	# of edge	edge size
Twitter [20]	41.7 M	1.5 B	11.9 GBytes
uk-2007-05 [4, 5]	105.9 M	3.7 B	28.2 GBytes
uk-union [6]	133.6 M	9.3 B	69.9 GBytes
ClueWeb12 [1]	6.3 B	71.7 B	557 GBytes
HyperLink12 [22]	3.6 B	128.7 B	972 GBytes
Synthesized	8.6 B	1206.9 B	9024 GBytes

Table 1. Graph dataset statistics (M: million, B: billion).



(a) PageRank



(b) WCC

Figure 4. Thread message-passing vs. shared-memory on Twitter graph on single server with 32 threads.

The rest of the section answers the following questions: 1) How effective is inter-thread message-passing on improving graph computation performance on multi-core server compared to shared-memory primitives? 2) How well does the RDMA-based network communication stack of GRAM perform, especially with respect to computation and communication overlapping, balanced multiplexing, and the trade-off between parallelism and communication efficiency? 3) What is the COST [25] of scaling in GRAM and how does it compare with an efficient single-threaded graph algorithm implementation? 4) How does the performance of GRAM compare to the existing state-of-the-art alternative systems? 5) How well does GRAM perform on a trillion-edge graph?

5.1 Inter-thread message passing vs. shared memory

In this set of experiments, we validate GRAM's choice of using message passing over shared-memory primitives even for inter-core communication within a server. For message passing, batching is an important factor and we study how message batch size affects performance in our experiment. For shared-memory primitives, we examine two primitives: one with locking to protect graph data and the other uses atomic instructions to update graph data. Figures 4(a) and 4(b) compare the performance (running time in seconds) of PageRank (one iteration) and WCC on the Twitter graph on a single server (with a 32-thread configuration) under three configurations: message passing with different batch sizes (Msg), shared memory with locking (Lock), and shared memory with atomic operations (Atomic).

The results show that the optimal message batch sizes to maximize the benefit of inter-thread message-passing are around 2K∼8K bytes, which are much larger than the cache line size. This is to amortize the constant overhead attributed to each message-passing; e.g., allocations of message buffer, operations on message queue, and so on. A large batch size will start to affect message-delivery latency negatively. For PageRank, with the optimal batch size (2KB), GRAM achieves about a 140% improvement by using thread message-passing (3.7 seconds) compared to shared-memory with locking (8.9 seconds) and a 90% performance gain compared to shared-memory with atomic instructions (7.1 seconds).

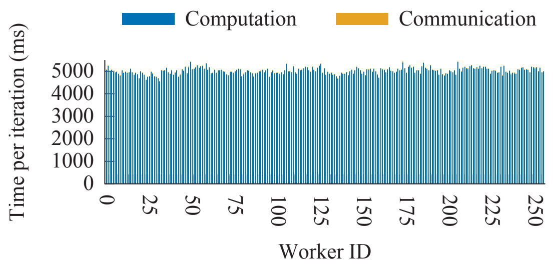
For WCC, the performance improvement relative to *Lock* is about 100% at 2KB batch size, which is less than the performance gain for PageRank. This is because WCC has fewer updates per vertex than PageRank (in WCC, if the propagated component ID is larger than that of target vertex, the propagated value will be discarded).

We observed similar results for other graph datasets that we used. For PageRank on uk-union graph, with the best batch size, message-passing improves by 100% compared to *Lock* and by 46% to *Atomic*. For WCC, the corresponding improvements are 50% and 34%, respectively.

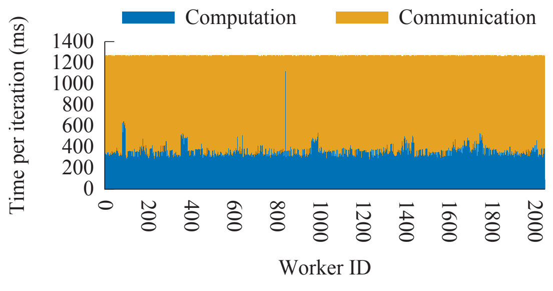
5.2 Efficient RDMA Communication for Scaling Out

An efficient RDMA-based communication stack is key to GRAM's performance and scalability. We design a series of experiments to evaluate different aspects of the communication stack. First, we study the effect of communication-computation overlapping and how well GRAM does this. Thanks to such overlapping, we show that GRAM can scale out as effectively as scaling up despite significantly higher communication cost for scaling out. Then, we investigate the importance of balance by looking into coordinator assignment and also NUMA-awareness. We further examine the different configurations in GRAM to trade off parallelism and communication efficiency by adjusting the granularity of multiplexing in the communication stack. Finally, we compare an RDMA-based network stack with TCP. We show the evaluation results on uk-union graph only. We observed similar results on other graphs.

Overlapping communication and computation. An important source of GRAM's efficiency comes from overlapping communication with computation with RDMA. We examine how well GRAM manages to overlap communication with computation in various configurations. Figure 5 shows a breakdown of computation and observable non-overlapped portion of communication when running PageRank on the uk-union graph with different numbers of servers. The x-axis is the worker ID globally assigned to all threads on all servers, with workers 0∼31 on server 0, workers 32∼63 on server 1, and so on. The y-axis is the execution time in milliseconds of one PageRank iteration. We break the execution time into 2 phases: a computation phase and a non-



(a) 8 servers



(b) 64 servers

Figure 5. Overlapping of computation and communication for one iteration of PageRank on the uk-union graph.

overlapped communication phase. In the computation phase, shown as the dark blue bar, a worker enumerates vertices and edges, prepares data to be propagated to neighbors, and processes messages received from other workers. The communication portion of this phase is overlapped with computation. The light yellow bar denotes the non-overlapped communication phase of a worker, which follows the computation phase. The operations in this phase include draining the remaining messages in the buffer and receiving messages sent from other workers (if any remaining).

Figure 5(a) shows communication is almost completely masked by the graph computation on 8 servers. The light yellow bars are barely visible, which means the communication is completed right after computation. However, such ideal overlapping cannot always be achieved, especially when the size of the graph partition owned by a worker is too small: data communication becomes inefficient due to fragmentation and there is not sufficient computation to hide the communication cost. Figure 5(b) shows such a case. With 64 servers, the communication phase is a significantly larger portion of the overall time spent than that in the 8-server case. The suboptimal overlap between computation and communication explains why the performance speedup is sub-linear from 8 to 64 servers. Computing one iteration of PageRank on uk-union costs slightly more than 5 seconds on 8 servers, as shown in Figure 5(a). It would have been reduced to around 0.625 seconds on 64 severs, assuming linear scaling. While the actual value is only 1.2 seconds, as shown in Figure 5(b). When the computation time falls into sub-second, the communication could become too fragmented and hence hard to take advantage of batching, which leads to much worse overlapping and overall performance.

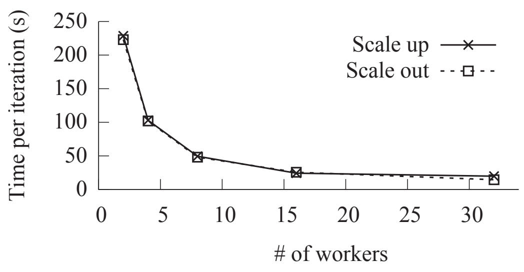
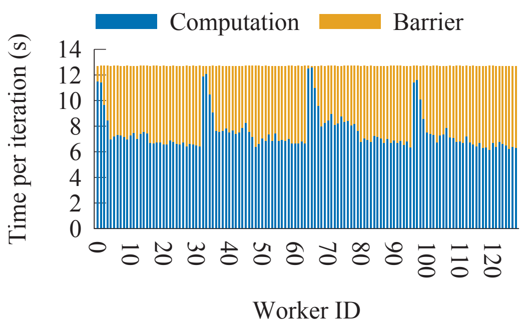


Figure 6. Scale-up vs. Scale-out with PageRank on the uk-union graph.

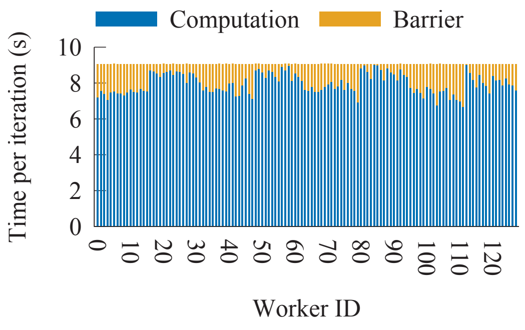
Scale-out vs. scale-up. Overlapping communication and computation allows GRAM to scale out as effectively as scaling up, as shown in the surprising result of the following experiment. In this experiment, we examine two configurations. In the *scale-up* configuration, we use a single server with each worker thread dedicated to one logical core. All inter-thread communication is therefore within a server. In the *scale-out* configuration, we place each worker thread on a dedicated server and also affinitize the worker to a single core. All inter-thread communication is therefore across servers through the RDMA network.

Figure 6 shows the comparison between these two configurations. As shown in the figure, GRAM's performance in the scale-out configuration is almost indistinguishable from the performance in the scale-up configuration, thanks to reduced CPU involvement in RDMA and the overlap of communication and computation. Note that in the 32-worker case, the performance in the scale-out configuration even outperforms that in the scale-up configuration. This is because there are only 16 cores on a single server and hyper-threading is used to support 32 threads.

Balanced coordinator assignment. Balanced parallelism is important for graph computation that uses barriers between iterations. GRAM's design carefully assigns coordinators to cores to achieve such balance, as discussed in Section 4.2. Figure 7(a) shows in detail how an unbalanced coordinator assignment affects overall performance. In this experiment, we run PageRank on the uk-union graph with 4 servers. As in Figure 5, we show the behavior on each thread (a total of 32×4) on all 4 servers and break down the execution time into 2 phases: *computation* and *barrier*, where the barrier phase refers to the time a worker spends waiting on a barrier. In this configuration, there are 3 sender/receiver channels and the first three threads on each server acts as a coordinator: the worker ids of the coordinators are 0∼2, 32∼34, 64∼66, and 96∼98, respectively. As shown in the figure, the workers serving as coordinators spend much longer time in the computation phase than others, while other workers, which enter the barrier phase much earlier, have to wait for those workers. To alleviate the issue, we introduce 8 groups in each server with 3 sender/receiver channels in each group. Each server then has 24 sender/receiver coordinators to be spread among the 32 cores. This roughly



(a) Imbalance in coordinator distribution.



(b) Imbalance in data allocation.

Figure 7. PageRank on the uk-union graph with 4 servers, and each server runs 32 worker threads. For allocation imbalance case, all data are allocated from NUMA node 0.

balances the load. Our experiment shows that alleviating coordinator imbalance this way improves the overall performance by 33%.

NUMA awareness. Another source of imbalance can be introduced by NUMA-agnostic data allocation. Figure 7(b) shows the effect of this issue. In this experiment, all data (including graph data and RPC buffers) are allocated from memory of NUMA node 0. The result clearly shows that all workers in NUMA node 0 execute faster than those in NUMA node 1 in the computation phase because the data in memory are closer to CPU cores in NUMA node 0. Figure 8 shows a breakdown of performance gains of applying NUMA-awareness to different parts of data. It shows that, when applying NUMA-awareness to both graph data and RPC buffer, GRAM realizes a performance gain of 40.9%.

Trade-off between parallelism and efficiency. As described in Section 4, there is an inherent trade-off between parallelism and communication efficiency. GRAM explores different points of the trade-off in different configurations by adaptively switching between the two sender buffer arrangement schemes: 1) one sender buffer for each peer thread and 2) one sender buffer for each peer server. The latter requires an additional dispatching phase to deliver messages to their destination threads, but gains efficiency because of the batching effect across receiving threads on the same server.

Figure 9 shows the performance comparison of the two schemes with different numbers of servers and different graph applications. First, the one-per-peer-thread buffer

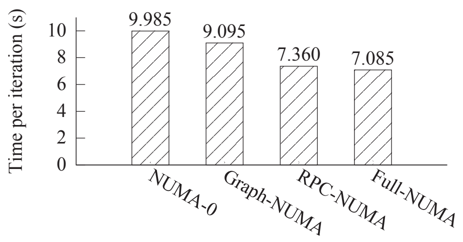
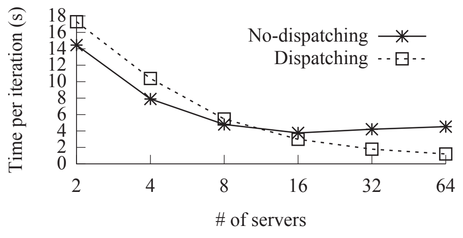
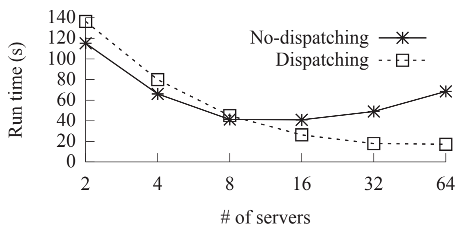


Figure 8. Performance comparison before and after NUMA-aware data allocation for PageRank on the uk-union graph with 4 servers. The experiment ran 32 threads on each server. NUMA-0 means all data allocated from NUMA node 0, Graph-NUMA means graph data like vertex and edge data are NUMA-aware but not for RPC buffers, and RPC-NUMA means RPC buffer is NUMA-aware while not for graph data. Full-NUMA applies NUMA-awareness for both data.



(a) PageRank



(b) WCC

Figure 9. Performance comparison on the uk-union graph with different buffer arrangement schemes.

arrangement (denoted as *no-dispatching*) performs better when the number of servers used is small because favoring parallelism is the right choice in those configurations where the computation cost dominates and effective batching is achievable at the level of parallelism. When the number of servers increases to 8 servers and more, the computation on each thread is reduced and the messages exchanged between threads become too small, thereby leading to reduced network efficiency. In those cases, the one-per-peer-server buffer scheme (denoted as *dispatching*), which favors communication efficiency, starts to outperform because it batches messages to all threads of the same remote server. The benefits of batching outweighs the extra overhead of redispatching messages to targeted threads. With a small number of servers, this dispatching overhead is evident (up to 21% performance degradation compared to *no-dispatching*).

The *dispatching* scheme achieves 3.7× and 3.9× performance improvements for PageRank and WCC in the case

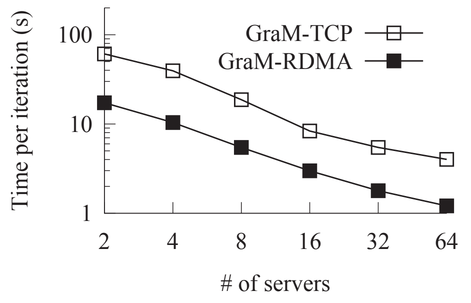


Figure 10. TCP/IP vs. RDMA for PageRank on the uk-union graph.

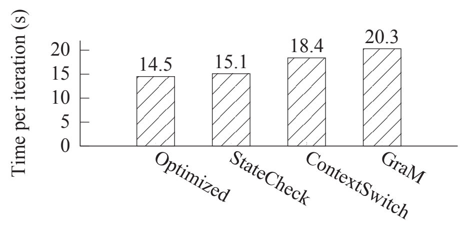


Figure 11. Overhead compared to an optimized single-threaded implementation of PageRank on Twitter graph.

of 64 servers, respectively. Note that WCC generally suffers more from the fragmentation issue because in WCC only a small portion of vertices are active in the computation in most iterations, thereby benefiting more from the merged buffer arrangement (i.e., *dispatching*).

RDMA vs. TCP/IP. We further show the efficiency gain due to RDMA by comparing with a version of GRAM configured to use TCP/IP. To maximize TCP's throughput in a high-bandwidth network, we tune the TCP window size to make it optimal for the experiment. Figure 10 compares the performance of PageRank on the uk-union graph with the RDMA and TCP communication stacks. GRAM with RDMA consistently performs $3\times$ better than GRAM with TCP. The performance degradation with a TCP stack comes from the significant overhead of the heavy TCP/IP kernel stack including switching between user and kernel modes, a complex multi-layer protocol, lock contention on kernel data structures, inevitable message data copy, and lack of flexibility to apply NUMA-aware optimizations.

5.3 COST analysis

A system might trivially scale to multiple servers if its single-threaded implementation has a high cost. A recent study shows that some distributed systems indeed scale at a high cost [25] and proposes a new metric *COST* (or the Configuration that Outperforms a Single Thread) to evaluate a system's efficiency. More precisely, a graph engine has a cost C if it takes C cores to outperform a single-threaded implementation of the same graph algorithm. We now report the COST of GRAM, analyze the additional overhead compared to a single-threaded specialized implementation, and demonstrate that the overhead is mainly the result of being a general-purpose distributed platform.

For our COST analysis, we compare GRAM with a public version of an optimized single-threaded PageRank implementation [25]. Because the original implementation is in C# while GRAM is implemented in C++, we port the C# version to a C++ implementation to remove the potential overhead from the language runtime. When running the single-threaded implementation, we affinitize the thread to a dedicated core. On the twitter graph, the C# implementation takes 15.5 seconds and our C++ version takes 14.5 seconds[1], which are reasonably close. In comparison, GRAM's single thread performance is 20.3 seconds. We further analyze the sources that contribute to the performance differences. It turns out that the major sources are related to our intention to make GRAM a general purpose platform. Here, we highlight the major ones. The first type of overhead comes from the cost of checking vertex states. For some graph algorithms, the engine needs to check whether the state of a certain vertex has changed in the previous iteration. This check helps decide when the execution of the algorithm converges and can terminate. Algorithms such as PageRank, WCC, and SSSP all require such state check for a complete computation process. However, the single-threaded PageRank implementation just performs a fixed number of iterations, without any logic to check vertex state. The "StateCheck" bar in Figure 11 shows a 4.1% slowdown compared to the optimized single-threaded implementation (the "Optimized" bar) due to the addition of those checks.

The second platform overhead comes from the scheduling overhead. In order to overlap computation and communication, every worker in GRAM has to yield the computation periodically and execute some communication tasks to poll and receive messages. This introduces some necessary context-switch cost. The "ContextSwitch" bar in Figure 11 shows such performance overhead. With yielding, the performance slows down to 18.4 seconds, which amounts to 21.9% additional overhead.

Finally, as a platform, GRAM introduces some necessary branching cost, e.g., to find out which partition the data should be propagated to. The "GRAM" bar includes such overhead. Note that GRAM provides the Gather-Apply-Scatter graph interface [13] in template functions, and hence enables effective inlining by the compiler, to avoid the overhead of interface invocations when applied to each edge.

Our investigation also reveals other factors that affect the performance of a graph engine. We initially do fine-grained graph partitioning for ease of load balancing. We observe that an excessive number of partitions could adversely impact performance. In our experiment, we partition the Twitter graph into 32 partitions and run a single-threaded GRAM on the 32 partitions. The execution time increases from 20.3 seconds to 42 seconds. This is because random partition-

[1] We used the default graph layout instead of the data arrangement based on Hilbert curve. The impact of graph layout is out of the scope of this paper.

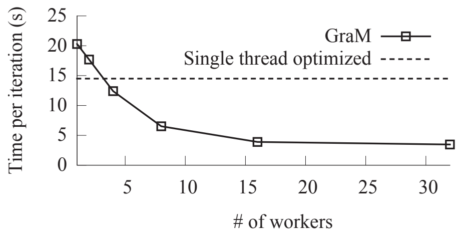


Figure 12. GRAM vs. an optimized single-threaded implementation of PageRank on Twitter graph.

ing worsens the locality of the graph layout, as observed in [25, 32]. Also the system introduces additional branching to allow a single worker to handle multiple partitions.

After understanding the single-threaded performance, we further show the scalability of GRAM in order to compute its COST. Figure 12 shows GRAM's performance speedup on a multi-core server. GRAM outperforms the optimized single thread implementation with 4 cores, thereby having a COST of 4. As shown in the figure, the speedup from 1 core to 2 cores is relatively small, primarily due to the additional platform overhead to support communication between concurrent worker threads with inter-core message passing. We see nearly perfect scaling from 2 cores to 16 cores. The performance continues to improve to 32 cores, but only marginally, due to hyper-threading.

5.4 GRAM vs. Naiad and GraphX

We compare GRAM with two state-of-the-art graph systems: Naiad [27] and GraphX [14]. Due to its efficient and compact data structures, GRAM can process much larger graphs on the same server resources than Naiad and GraphX. GRAM can run computations on the first three graphs (in Table 1) in the memory of one multi-core server in our cluster. The largest real-world graph, HyperLink12, can be accommodated using 8 servers. We are not able to run GraphX on graphs larger than uk-union even with 64 servers. The system reports out of memory exceptions for all the graphs larger than uk-union. Naiad can run ClueWeb12 and HyperLink12 starting from 16 and 32 servers, respectively.

We compare the performance of the three systems using PageRank and WCC and show the results in Figures 13 and 14, respectively. Note that we exclude GraphX's WCC results in Figure 14 because GraphX's WCC implementation produced incorrect results. Also, Naiad's WCC implementation crashed on larger graphs. We re-implement WCC using Naiad's GraphLINQ interface based on the same WCC algorithm we use in GRAM. In Figures 13 and 14, the x-axis denotes the number of servers, whereas the y-axis is the execution time in seconds shown in a log scale. As shown in the figures, GRAM outperforms Naiad and GraphX in all these experiments due to its highly efficient inter-core and network communication stack. It often achieves more than an order of magnitude higher performance than Naiad and GraphX. In WCC, GRAM achieves up to 2 orders of magnitude improvement as shown in Figure 14(b).

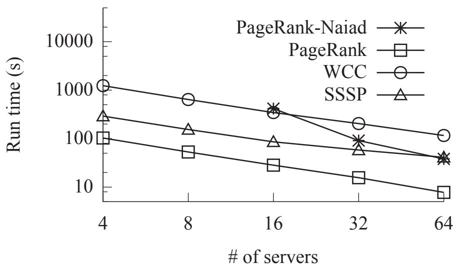


Figure 15. GRAM performance on ClueWeb12 graph.

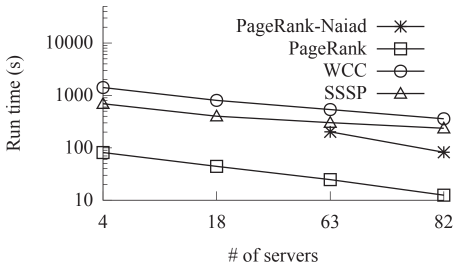


Figure 16. GRAM performance on hyperlink12 graph.

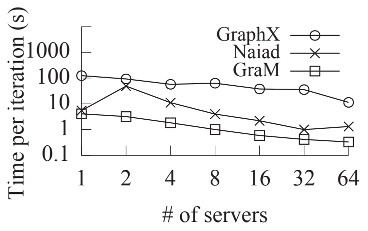
In the single server case, GRAM is faster than Naiad due to its compact data structure, less usage of function pointers, and more efficient C++ runtime. The performance gap in this case is smaller than that in distributed cases. This shows the additional benefit of GRAM's RDMA-based communication stack compared to the heavy TCP/IP protocol stack used in Naiad, even over the same high bandwidth network (54Gbps). We observe that, when performing the computation, Naiad only achieves 700MB/s network throughput per server on average, much less than the 3GB/s bandwidth that GRAM can achieve.

On the Twitter graph, GRAM does not scale well from 32 servers to 64 servers, especially for WCC. This is because Twitter is a small graph. When divided into 2048 partitions (32×64), each worker is assigned only about 6MB of data, on which computation can hardly overlap with communication. Even with GRAM's adaptive buffer management, communication remains inefficient due to lack of sufficient batching. This is worse in the case of WCC because in most iterations only a small subset of vertices are involved in the computation, leading to even less effective batching.

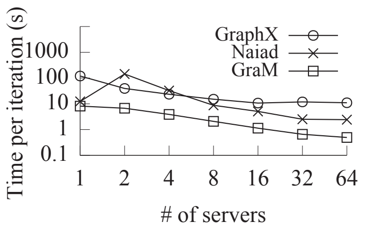
Note that in many cases GraphX can be significantly slower than Naiad and GRAM because the latest GraphX build employs a disk-based shuffle stage to perform the vertex data propagation. The relative performance we observed for the two comparing systems is consistent with what was reported earlier [25].

5.5 GRAM on large graphs

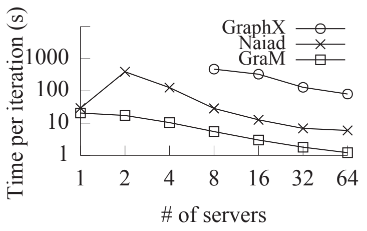
GRAM is capable of processing graphs that are significantly larger than those reported for other existing distributed in-memory graph engines. We now analyze GRAM's performance on those graphs. The first set of experiments are performed on two public graph datasets ClueWeb12 and Hy-



(a) Twitter

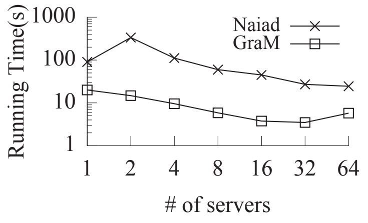


(b) uk-2007-05

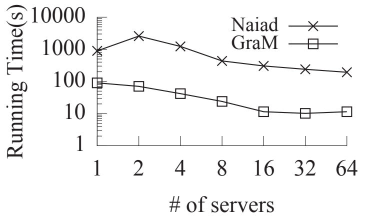


(c) uk-union

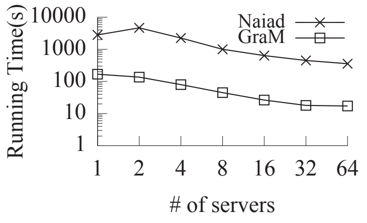
Figure 13. GRAM vs. Naiad vs. GraphX: PageRank.



(a) Twitter



(b) uk-2007-05



(c) uk-union

Figure 14. GRAM vs. Naiad vs. GraphX: WCC.

perLink12, with 71.7 billion and 128.8 billion of edges, respectively. HyperLink12 is to our knowledge the largest real world graph that is publicly available. Figure 15 and Figure 16 show the performance of GRAM when executing PageRank (one iteration), WCC, and SSSP. As a point of comparison, we also include Naiad's performance on PageRank. Again, GRAM's memory management is efficient and can process ClueWeb12 (557GB) on 4 servers. In contrast, Naiad needs at least 16 servers to process ClueWeb12 and 32 for HyperLink12.

For PageRank, we observe linear scaling of GRAM on both graphs. The shortest time is 7.8 seconds per iteration on ClueWeb12 and 12.5 seconds on Hyperlink12, outperforming Naiad by 5× to 14× under the same configurations. For WCC and SSSP, GRAM also scales well initially, but starts to see reduced benefit when scaling to 32 servers and above. Unlike the case for PageRank, where all vertices participate in each iteration, for WCC and SSSP, a small portion of vertices might be active in later iterations of the computation, reducing the effectiveness of batching.

Finally, we run GRAM on the synthesized graph with 1.2 trillion edges. Our evaluation shows that one iteration of PageRank takes 140 seconds on 64 servers. We observe that the average network throughput per server for GRAM is about 2GB/s, which is still lower than the upper bound of 6.75GB/s. This shows that CPU, rather than network, remains the bottleneck of GRAM in this case and indicates that we are likely to see further improvement when using more servers for the computation. As a point of comparison, the only available public number for in-memory graph processing at the scale of one trillion edges is reported by Facebook [10], where the per-iteration PageRank time using Giraph is reported to be 4 minutes on 200 commodity servers. The detailed information about Facebook's graph dataset and server configuration is unknown to the public.

6. Discussion

While the design of GRAM focuses on optimizations that leverage RDMA, our experiences have revealed principles and lessons that could be valuable to designers of other general scalable distributed computation engines.

First, the design of GRAM shows the importance of striking a balance between computation and communication, which is key to the efficiency and scalability of distributed computation engines in general. While a system can recruit more cores (servers) to speed up computation through partitioning and parallelization, the scalability of the system is limited by several factors: (i) the communication cost and overhead goes up when the number of end points increases; (ii) the efficiency of communication drops due to reduced batching opportunities, and (iii) insufficient computation to overlap and mask communication. Overlapping communication and computation is particularly effective and important with RDMA and other kernel-bypassing messaging substrate, just as database systems are designed to hide disk latency. How to manage communication cost, maintain communication efficiency, and balance communication and computation therefore becomes the common theme. As we have observed in GRAM with graph computation, the balance between communication and computation is often workload-dependent and dynamic in that the balance shifts as the system executes, thereby necessitating an adaptive strategy.

Second, the conventional wisdom is that the network cost dominates in distributed systems. Our experiences with GRAM indicate that local computation efficiency and multicore parallelism matter greatly even to a distributed computation engine. A careful design that takes into account issues such as locality, NUMA-awareness, and balanced parallelism makes a huge difference in the end. And a good design can and should achieve both efficiency (against a highly optimized single-threaded implementation) and scalability at the same time.

Third, when comparing against other distributed computation engines, we also realize that the choice of programming language has subtle, but sometimes significant, implications, especially related to RDMA. For example, a managed runtime for a high-level language such as Java and C# could impose constraints on the threading model and memory management. GRAM adopts FaRM's polling-based threading model with threads dedicated to cores polling RDMA messages, while a managed runtime tends to have background threads for activities such as garbage collection, which reduces the level of parallelism. Also, the memory regions accessed by RDMA network cards need to be non-pageable, and therefore not allowed to be swapped out during the network data transfer. This might not be in harmony with the memory model of managed runtime with automatic memory management.

7. Related Work

The research and development of parallel or distributed graph processing systems have been driven by the growing scale and importance of graph data recently. Some graph systems [21, 30, 32, 33, 36] focus on single-server optimizations to scale up graph computation on multi-core server while others [8, 9, 13, 14, 23, 24, 27, 31, 34] emphasize scaling out on a cluster of servers to exploit further the parallelism at a larger scale. Chaos [34] is shown capable of processing a 1-trillion edge graph over two days on 32 servers as it provides a scale-out solution from secondary storage, an interestingly different design point from GRAM.

GRAM embraces message-passing even in single-server case for higher performance. Barrelfish [3] advocates a similar design in the operating system scenario, while we demonstrate the benefit of message passing and the importance of batching for graph workloads. Grappa [28] also employs this idea by translating shared-memory abstractions to an underlying message-passing execution.

A recent study [25] shows that many distributed graph processing systems scale, but at a high COST compared to a single-threaded implementation of the same graph algorithm. Some of the published systems either need a large number of cores to outperform that single-threaded implementation or never manage to match its performance. GRAM therefore reports not only scalability, but also its efficiency in terms of COST. The combined efficiency and scalability makes it possible for GRAM to process a large graph with over 1 trillion edges and outperform the existing graph engines by a large margin in our experiments.

Given the low latency and high bandwidth brought by RDMA, some work [18, 26] explored the opportunities to benefit key-value stores, while others [12, 28] revisited and advocated distributed shared-memory abstractions. In contrast, the characteristics of graph computation leads to a different choice in GRAM, where balanced parallelism, overlap of computation and communication, and network efficiency/throughput are more important than absolute latency.

Some optimization techniques used in single-server graph engines are complementary to GRAM. Grace [32] proposed a technique to exploit better data locality of graph workload through a graph-aware data placement, which can be easily integrated to enhance GRAM's efficiency. Galois [30] extends the vertex-program interface to allow applications to specify priority and employs a priority-based scheduling with a scalable priority queue for workloads (e.g., WCC) that may benefit from it. Their techniques may not be easily extended to distributed setting directly, but GRAM adopts the priority interface and uses a priority queue for each thread to get a similar benefit for those workloads. Polymer [39] proposed NUMA-aware graph data layout and scheduling strategy for a single-server setting, while GRAM's network stack is also designed to be NUMA-aware. GRAM's system design does not impede integration of some complementary techniques for a distributed setting, e.g., the graph partitioning schemes used in Surfer [7], PowerGraph [13], and PowerLyra [8]. Advanced partitioning schemes for huge graphs is a daunting task, while GRAM demonstrates its efficiency with simple hash partitioning due to the effective RDMA communication stack. There is another line of graph system research that targets online graph query scenarios [2, 11, 29, 35]. It would be interesting to explore how RDMA could benefit such systems.

8. Conclusion

There is a growing need to process increasingly large graphs efficiently to extract valuable insights from graph structures of relations and connections. With GRAM, we demonstrate that an efficient and also scalable graph engine capable of processing graphs with over one trillion edges can be built with a principled design that leverages new hardware capabilities such as multi-core and RDMA. Our detailed evaluation further validates the key design choices and sheds light on the key to scaling distributed computation engines.

Acknowledgements

We thank our shepherd Wyatt Lloyd and the anonymous reviewers for their valuable comments and suggestions. The paper also benefited greatly from the insightful feedback from Miguel Castro on an early version of the paper. We are grateful to Aleksandar Dragojevic, Dushyanth Narayanan, and the entire FaRM team in Microsoft Research for their help and discussions related to FaRM. Jilong Xue and Yafei Dai are partially supported by National Science Foundation of China under Grant No. 61472009.

References

[1] The lemur project: Clueweb12 web graph. http://www.lemurproject.org/clueweb12/webgraph.php/.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, 2007.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, 2004.

[5] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8), 2004.

[6] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2), 2008.

[7] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[8] R. Chen, J. Shi, Y. Chen, H. Guan, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, 2015.

[9] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, 2012.

[10] A. Ching. Scaling Apache Giraph to a trillion edges. https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920.

[11] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11), 2013.

[12] A. Dragojevi, D. Narayanan, O. Hodson, , and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, 2014.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[14] 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, 2014.

[15] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 2014.

[16] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, 2014.

[17] InfiniBand Trade Association. Supplement to infiniband architecture specification volume 1 release 1.2.2 annex a16: RDMA over converged ethernet (RoCE), 2010.

[18] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, 2014.

[19] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in MapReduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '14, 2014.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, 2010.

[21] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, volume 8 of *OSDI'12*, 2012.

[22] O. Lehmberg, R. Meusel, and C. Bizer. The graph structure of the web aggregated by pay-level domain. In *Web Science Conference*, WebSci '14, 2014.

[23] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8), 2012.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, 2010.

[25] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[26] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference*, USENIX ATC '13, 2013.

[27] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[28] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference*, USENIX ATC '15, 2015.

[29] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), 2010.

[30] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th*

ACM Symposium on Operating Systems Principles, SOSP '13, 2013.

[31] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[32] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[34] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '15, 2015.

[35] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[36] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[37] D. A. Spielman and S.-H. Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS '96, 1996.

[38] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[39] K. Zhang, R. Chen, and H. Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, 2015.