ARGO: Architecture-Aware Graph Partitioning

Angen Zheng, Alexandros Labrinidis, Panos K. Chrysanthis, Jack Lange
Department of Computer Science, University of Pittsburgh
{anz28, labrinid, panos, jacklange}@cs.pitt.edu

Abstract—**The increasing popularity and ubiquity of various large graph datasets has caused renewed interest for *graph partitioning*. Existing graph partitioners either scale poorly against large graphs or disregard the impact of the underlying hardware topology. A few solutions have shown that the nonuniform network communication costs may affect the performance greatly. However, none of them considers the impact of resource contention on the memory subsystems (e.g., LLC and Memory Controller) of modern multicore clusters. They all neglect the fact that the bandwidth of modern *high-speed* networks (e.g., Infiniband) has become comparable to that of the memory subsystems. In this paper, we provide an in-depth analysis, both theoretically and experimentally, on the contention issue for distributed workloads. We found that the slowdown caused by the contention can be as high as 11x. We then design an *architecture-aware* graph partitioner, ARGO, to allow the full use of all cores of multicore machines without suffering from either the contention or the communication heterogeneity issue. Our experimental study showed (1) the effectiveness of ARGO, achieving up to 12x speedups on three classic workloads: Breadth First Search, Single Source Shortest Path, and PageRank; and (2) the scalability of ARGO in terms of both graph size and the number of partitions on two billion-edge real-world graphs.**

Index Terms—**Heterogeneity; Contention; Multicore; Graph Partitioning; Distributed Graph Processing;**

I. INTRODUCTION

Large graph datasets are becoming increasingly popular. For example, graphs, like Web Graphs, Biological Networks, and Social Networks, are often at the scale of hundreds of billions or even a trillion (10ˆ12) edges, and they are continuously growing. As a consequence, many distributed graph computing frameworks (e.g., Pregel [23], GraphLab [20] and PowerGraph [11]) have been developed.

In such systems, distributing vertices evenly across partitions often corresponds to an even load distribution, while minimizing the *edge-cut* (the number of edges connecting different partitions) helps minimize the amount of data communication incurred by the computation. Balanced graph partitioning has been proved to be NP-hard [3]. Most of the solutions are heuristic-based [12], [34]. The most well-known approaches are *multi-level* ones [17]. **However, these solutions often scale poorly against large graphs** [36], [27].

To address the scalability issue, a *(re)streaming graph partitioning* model [36], [39], [27], [7] was recently proposed for large graph partitioning. In this model, the graph is treated as a stream of vertices. Upon arrival of a vertex, the partitioner places the vertex to one of the partitions based on the distribution of the vertices that arrived previously. **Nevertheless, none of the above partitioners considers the nonuniform network communication costs** of modern parallel computing infrastructures [6], [41], [42].

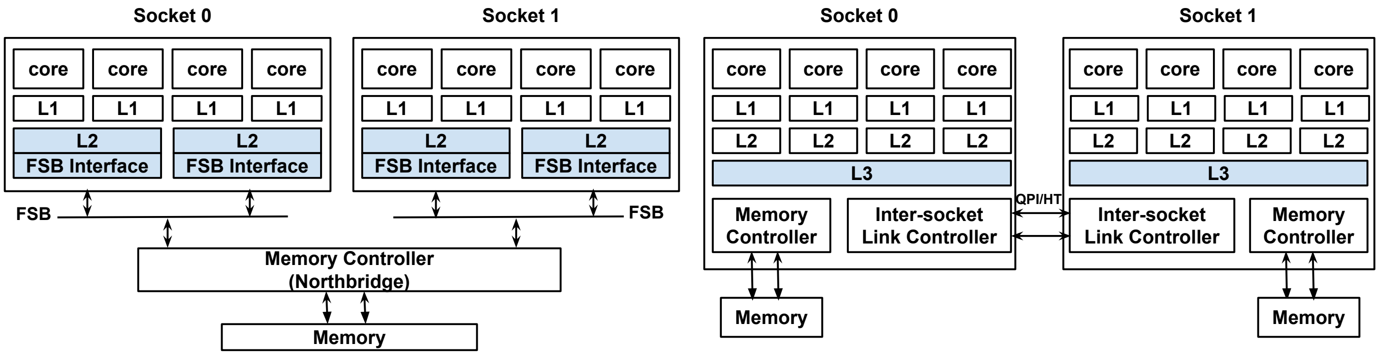
Three recent works [6], [41], [42] attempted to tackle this heterogeneity issue by trying to avoid any edges being cut among partitions having higher network communication costs (minimizing *hop-cut*). However, these graph partitioners are all **built on the assumption that the network is the bottleneck**, since they all aim to minimize either the edge-cut or the hop-cut. The assumption is typically true for geo-distributed clusters and the cloud computing environment. However, for clusters connected via high-speed networks like InfiniBand, this assumption no longer holds: the data transfer on these networks has been reported to be almost as fast as moving data from memory to CPU [9]. Actually, several recent works [45], [44], [13] have found that the contention for the shared hardware resources on the memory subsystems (e.g., last level cache, memory controller, and front-side bus) of modern multicore machines can greatly impact the performance of distributed workloads. Specifically, work [13] investigates the contention issue for MPI [1] workloads, whereas our works [45] and [44] are *architecture-aware (heterogeneity- and contention-aware) graph repartitioners* designed to avoid the heterogeneity and contention issue for distributed graph workloads.

Contributions This paper advances the state-of-the-art with regards to contention for distributed (graph) workloads with the following three contributions:

1. We provide a holistic view on: (a) why we have to care about the contention for distributed workloads (Section II); and (b) to what extent it may impact the performance of distributed workloads (Section V).
2. We present an architecture-aware graph partitioner, ARGO, which avoids both the heterogeneity and the contention issue without doing so at the cost of resource underutilization, for static graph partitioning (Section III).
3. We evaluate our approach extensively using three large real-world graphs, showing up to 12x speedups on three classic graph workloads (Section VI & VII).

II. THE CURSE OF CONTENTION

Multicore machines usually consist of multiple sockets and each socket has multiple cores. Each core is a logical processing unit, but they are not physically isolated. Cores of the same socket have to contend with each other for the shared hardware resources. For example, in the architecture depicted in Figure 1a, cores sharing the L2 caches have to compete



(a) Uniform Memory Access (UMA) Node

(b) Nonuniform Memory Access (NUMA) Node

Fig. 1: Example Architectures of Modern Compute Nodes

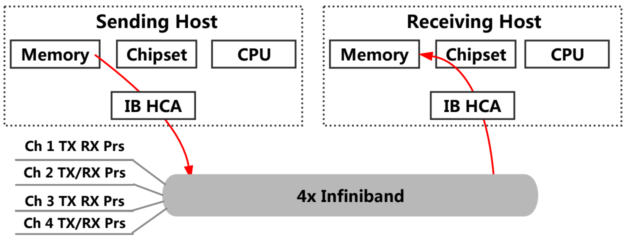


Fig. 2: Memory transactions of inter-node data communication via RDMA [14]

TABLE I: Intra-Node Shared Resource Contention

Cores/Resources		Sharing		Contention		
Core Groups		Socket	LLC	LLC	FSB/QPI(HT)	Memory Controller
UMA Fig. 1a	G1	✓	✓	✓	✓	✓
	G2	✓			✓	✓
	G3					✓
NUMA Fig. 1b	G1	✓	✓	✓		✓
	G2				✓	

with each other for the shared L2, *Front-Side Bus* (FSB), and the Memory Controller. Although cores on different sockets do not share the L2, they may still contend for the shared FSB and Memory Controller. In fact, even if they are residing on different sockets, they may have to contend for the shared Memory Controller. Table I provides a concise summary for the resources that different cores may have to contend for, in the *Uniform Memory Access* (UMA) architecture of Figure 1a and the *Non-Uniform Memory Access* (NUMA) architecture of Figure 1b. The summary is based on whether the cores are on the same socket and whether they share the *last level cache* (LLC).

The impact of contention is becoming more and more noticeable because network nowadays may no longer be the bottleneck due to the presence of *remote direct memory access* (RDMA) technology [9]. RDMA-enabled networks allow a compute node to read data from the memory of another compute node without involving the processor, cache, or operating system of either node, enabling true zero-copy data communication [14] (Figure 2). At the same time, the bandwidth of modern RDMA-enabled networks has been reported to be in the same ballpark as memory bandwidth [9]. As shown in Figure 3, DDR3 memory bandwidth is currently between 6.25GB/s (DDR3-800) and 16.6GB/s (DDR3-2133) per memory channel, whereas InfiniBand bandwidth ranges from 1.7GB/s (FDR 1x) to 37.5GB/s (EDR 12x) per NIC port. Thus, the memory bandwidth of a machine with 4-channel DDR3-1600 memory can be roughly provided by four dual-port FDR 4x NICS.

The fact that intra-node data communication is often achieved via shared memory further amplifies the contention, because communication requires additional data copies [16], [5], leading to cache pollution and thus saturating the memory

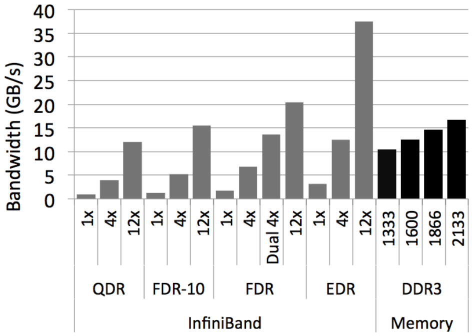


Fig. 3: Theoretic bandwidth for different InfiniBand and memory technologies (Binnig et. al. [9].)

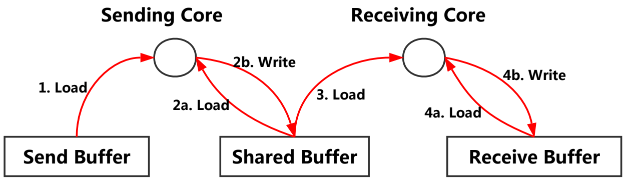


Fig. 4: Memory transactions of intra-node data communication via shared memory

controller. Figure 4 shows the corresponding memory/cache transactions for sending a message from one core to another. The sending core first needs to load the message from the application send buffer into its cache (Step 1 in Figure 4) and then write the data to the shared buffer (Step 2b). However, the write may require loading the shared buffer block into the sender's cache first (Step 2a). Then, the receiving core reads the data from the shared memory (Step 3). Finally, the receiver writes the data to the receiving buffer (Step 4b), which may again require loading the receiving memory block into the receiver's cache first (Step 4a).

Thus, if the sending core shares the same last level cache with the receiving core, there will be multiple copies of the same message in LLC. This is because in addition to the cached message for the send and receiving buffer, the message in the shared memory has also to be cached in the LLC. Even if the sender and receiver do not share LLC, the LLC of both sender and receiver may still have to maintain multiple copies of the message as long as they reside on the same machine (one for the shared memory buffer and the other one for the send or receive buffer). Clearly, intra-node data communication may lead to serious cache pollution and therefore saturate the memory controller.

What is even worse is that graph workloads are known to be communication-intensive, and that cores on the same machine are often communicating with each other at the same time for parallel computation, further increasing the contention for the shared resources. The fact that graph workloads often have poor locality [22] (because of the irregular and unstructured nature of real-world graphs) and high memory access to computation ratio [22] (since graph algorithms are often based on the exploration of the graph structure with little computation work per vertex) further aggravates the contention issue. We have experimentally confirmed and quantified the performance impact of the contention on the distributed graph workloads in Section V.

Take-Away *Focusing solely on minimizing the edge-cut or the hop-cut may not be sufficient for scalable performance. This is because edge-cut based solutions have no guarantee on how the edge-cut is distributed across partitions. They may end up with lots of data communication among partitions that are assigned to the same machine, leading to contention on the memory subsystems. On the other hand, hop-cut based solutions advocate to group neighbouring vertices as close as possible, further aggravating the contention the memory subsystems.*

III. Architecture-Aware Graph Partitioning

In this section, we first introduce the partitioning model adopted by our proposed Architecture-Aware Graph PartitiOning technique, ARGO (Section III-A). Then, we show how ARGO takes the communication heterogeneity into account while partitioning (Section III-B). Finally, we describe how ARGO considers the contentiousness of the underlying hardware architectures (Section III-C).

A. ARGO*: Graph Partitioning Model*

ARGO follows the same streaming model first proposed by [36]. In such a model, vertices arrive at the partitioner in a certain order along with their adjacency lists. Upon the arrival of each vertex, the partitioner decides the placement of the vertex to one of the partitions based on the placements of vertices previously arrived. The placement of the vertex never changes once it is assigned to a partition.

A variety of heuristics have been proposed by [36] for the vertex placement, among which the *linear deterministic greedy* (*LDG*) performs the best. *LDG* tries to assign a vertex, v, to a partition, P_i, which maximizes:

$$(1 - \frac{w(P_i)}{C(P_i)}) * \sum_{e=(u,v)\in E \text{ and } u\in P_i} w(e) \tag{1}$$

where $w(P_i)$ is the aggregated weights of vertices that have been assigned to P_i (indicating the computational requirement of the vertices of the partition), $C(P_i)$ denotes the maximal amount of work P_i can have, and $w(e)$ is the edge weight (reflecting the amount of data communication between neighboring vertices). Essentially, *LDG* places each vertex to a partition with the maximum number of its neighbors while penalizing the placement based on the load of the partition.

B. ARGO*: Incorporating Heterogeneity-Awareness*

ARGO takes the nonuniform network communication costs into account by replacing the vertex placement heuristics to maximize the following objective:

$$\frac{1}{comm(v, P_i) + 1} * (1 - \frac{w(P_i)}{C(P_i)}) \tag{2}$$

where $comm(v, P_i)$ is defined as

$$comm(v, P_i) = \sum_{e=(u,v)\in E \text{ and } u\in P_j \text{ and } i\neq j} w(e) * c(P_i, P_j) \tag{3}$$

$comm(v, P_i)$ reflects the communication cost that v would incur during the computation if it is assigned to P_i. Here, $w(e)$ denotes the edge weight, whereas $c(P_i, P_j)$ is the relative network communication cost between the computing elements that P_i and P_j are assigned to. The computing element can either be a core (one partition per core), a socket (one partition per socket), or a server (one partition per node). By default, we assume that the computing elements are cores since we target for clusters of multicore machines. Guided by a cost matrix, ARGO will put neighboring vertices to partitions as close as possible. We denote this version of ARGO as ARGO-H, since it only considers the communication heterogeneity while ignoring the contentiousness of the underlying hardware architecture.

C. ARGO*: Incorporating Contention-Awareness*

As analysed in Section II and will be demonstrated in Section V, edge-cut (e.g., *LDG*) and hop-cut (e.g., ARGO-H) based solutions may lead to serious resource contention on the memory subsystems of modern multicore clusters. One common way to avoid this contention issue is to disallow the use of all the cores of the machine, which leads to resource underutilization.

Fortunately, we found that the contention is caused by the communication among cores of the same node and can be avoided by offloading a certain amount of intra-node communication across compute nodes. This is because inter-node data communication is often implemented using RDMA and rendezvous protocols [37], which allows a compute node to read data from the memory of another compute node without involving the processor, cache, or operating system of either node (Figure 2), thus alleviating the traffic on memory subsystems and cache pollution. In fact, with Intel Data Direct I/O technology [15], it is even possible to transfer data from one machine into the cache of another. Another reason why offloading intra-node data communication across compute nodes (via contention-aware graph partitioning) works is that graph workloads are often data-driven. The computations performed by a graph algorithm are dictated by the vertex and edge structure of the graph on which it is operating rather than being directly expressed in code [22].

Thus, to make ARGO contention-aware, we penalize intra-node network communication costs via a penalty score in the same way as in our previous work [45], [44]. The score is computed based on the degree of contentiousness between the communication peers. By doing this, the amount of intra-node data communication and the contention on the memory subsystems will decrease accordingly. Recall that guided by a cost matrix, ARGO-H can gather neighboring vertices close to each other (Eq. 2), which causes contention on the memory subsystems due to the excess intra-node data communication. To solve this, we simply refine the intra-node communication costs as follows:

$$c(P_i, P_j) = c(P_i, P_j) + \lambda * (s_1 + s_2) \tag{4}$$

where P_i and P_j are two partitions collocated in a single compute node; λ is a value between 0 and 1, denoting the degree of contention; and s_1 denotes the maximal inter-node network communication cost, while s_2 equals 0 if P_i and P_j reside on different sockets and equals the maximal inter-socket network communication cost otherwise. s_1 is used to avoid excess intra-node data communication, whereas s_2 is used to prevent load imbalance on the memory controllers and to further avoid the contention on the shared LLC.

Clearly, if $\lambda = 0$, ARGO degrades to ARGO-H, and $\lambda = 1$ means that contention on the memory subsystems is the biggest bottleneck and should be prioritized over the communication heterogeneity. Note that ARGO with any $\lambda \in (0, 1]$ considers both the contention and the communication heterogeneity. Considering the impact of resource contention and communication heterogeneity is highly application- and hardware-dependent; users will need to do simple profiling of the target applications on the actual computing environment to determine the ideal λ for them. Typically, for multicore clusters with high-speed network, a larger λ is recommended, and vise verse.

TABLE II: Datasets used in our experiments

Dataset	$\|V\|$	$\|E\|$	Description
com-orkut [2]	3,072,627	234,370,166	Social Network
Friendster [2]	124,836,180	3,612,134,270	Social Network
Twitter [19]	52,579,682	3,926,527,016	Social Network

TABLE III: Cluster Compute Node Configuration

Socket (2 Intel Haswell Sockets)			Memory	
Cores/Socket	Clock speed	L3 Cache	Capacity	Bandwidth
10	2.6GHz	25MB	128 GB	65 GB/s

IV. Evaluation Setup

In our experimental study, we first quantified the performance impact of the contention issue using three representative graph workloads: Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), and PageRank in Section V. Then, we evaluated the effectiveness of ARGO in avoiding contention using the same workloads in Section VI. Finally, we examined the scalability of ARGO in terms of both graph size and the number of partitions in Section VII.

A. Workload Implementation

All the workloads were implemented using MPI [1] based on the idea presented in [4], [21]. The specific MPI implementation we used in the experiment was OpenMPI 1.8.6 [28]. Note that the workloads were implemented using MPI_Isend/MPI_Irecv functions.

B. Datasets

Table II describes the datasets used. com-orkut and Friendster datasets were undirected, whereas the original Twitter dataset was directed but was treated as a undirected graph in the experiment. Note that these datasets were all *scale-free* and *small-world* graphs. The vertex degree-distribution of the scale-free graphs asymptotically follows a power law distribution [8], [30], whereas small-world graphs are known to have low diameters.

Throughout the paper, the graphs were partitioned with the vertex weights (i.e., computational requirement) set to their vertex degree and edge weights (i.e., amount of data communicated) set to 1. Vertex degree is a good approximation of the computational requirement of each vertex for the execution of BFS, SSSP, and PageRank, while an edge weight of 1 is a close estimation of their communication patterns. By default, the graphs were partitioned across cores of a given set of machines with one partition per core. During the partitioning, we allowed up to 2% load imbalance among partitions.

C. Algorithms

We compared ARGO to three graph partitioners: (a) *METIS*, the most well-known multi-level graph partitioner [26], (b) *LDG*, a state-of-the-art streaming graph partitioner [36], and (c) ARGO-H (Section III).

TABLE IV: BFS, SSSP, and PageRank Execution Time in Seconds on com-orkut Dataset Under Different Configurations

Configuration	BFS (10 Source Vertices)			SSSP (10 Source Vertices)			PageRank (30 Iterations)		
	METIS	*LDG*	Argo-H	*METIS*	*LDG*	Argo-H	*METIS*	*LDG*	Argo-H
1:2:8	53.05	95.82	68.61	633	2,632	1,549	174	690	859
2:2:4	55.01	105.71	88.17	654	2,565	1,505	222	619	618
4:2:2	36.85	55.82	64.02	521	631	861	202	269	247
8:2:1	19.16	45.81	14.84	222	280	132	95.84	133	108

TABLE V: BFS, SSSP, and PageRank LLC Misses in Millions on com-orkut Dataset Under Different Configurations

Configuration	BFS (10 Source Vertices)			SSSP (10 Source Vertices)			PageRank (30 Iterations)		
	METIS	*LDG*	Argo-H	*METIS*	*LDG*	Argo-H	*METIS*	*LDG*	Argo-H
1:2:8	609	424	283	10,292	44,117	23,632	1,945	6,216	10,209
2:2:4	662	601	766	10,626	44,689	23,770	2,719	6,836	9,087
4:2:2	59	73	70	2,541	1,061	2,787	48	100	82
8:2:1	52	67	66	96	187	141	44	98	87

D. Evaluation Platform

All the experiments were performed on a 32-node university cluster [33]. The cluster had a flat network topology with all the compute nodes connected to a single switch via 56Gbps FDR Infiniband. Table III depicts the node configuration of the cluster.

E. Network Communication Cost Modeling

The relative network communication costs among partitions (cores) were approximated using a variant of osu_latency benchmark [29]. To ensure the accuracy of the cost matrix, we bound each MPI rank (process) to a core using options provided by OpenMPI 1.8.6 [28].

V. Performance Impact of Resource Contention

In this section, we experimentally demonstrated and quantified the performance impact of the contention on distributed graph computing using *METIS*, *LDG*, and Argo-H. This is achieved by comparing runs of an MPI implementation of PageRank, BFS, and SSSP with different process (rank) affinity patterns.

For presentation clarity, we labelled an execution of a workload under a specific partition (rank) to core mapping as *m:s:c*, where *m, s*, and *c*, respectively, denote the number of machines used, the number of sockets used per machine, and the number of cores used per socket. For example, label *1:2:8* indicates that the experiment was performed on one dual-socket machine with eight MPI ranks per socket (one rank per core). To quantify the performance impact of the contention, we ran each workload with a fixed number of MPI ranks (16) under four different configurations: {*1:2:8, 2:2:4, 4:2:2, 8:2:1*}. Note that the degree of contention gradually decreased from configuration *1:2:8* to configuration *8:2:1*. This is because the number of active cores per socket of the configurations gradually decreased from 8 to 4, to 2, and finally to 1. This also explains why we only used 16 cores per node at most (8 cores per socket) in this experiment, although each compute node of the cluster had 20 cores.

To mitigate the impact of other factors, executions of BFS/SSSP under different configurations all started from the same set of randomly selected source vertices (10 by default). Also, given the long execution time of the jobs, we grouped multiple (256) messages sent by the same MPI rank to the same destination into a single one. In the experiment, the dataset was partitioned into 16 partitions across corresponding cores (one partition per core) using *METIS*, *LDG*, and Argo-H. Note that we observed similar results on the other datasets of Tabel II.

A. Results in terms of execution time (Table IV)

Table IV shows the resulting execution time of the workloads under different configurations on the com-orkut dataset (Table II). As expected, the higher the contention was, the longer the execution time would be. When compared with configuration *8:2:1*, the slowdown caused by the contention can be as high as 5.94, 11.69, and 7.94 times for the execution of BFS, SSSP, and PageRank, respectively. We also noted that even if we reduced the number of active cores per socket by half (configuration *2:2:4*), the application may still suffer from serious contention. The reason why the execution of BFS under configuration *2:2:4* sometimes took longer than that of configuration *1:2:8* was probably because configuration *2:2:4* and configuration *1:2:8* has similar degree of contentiousness, but configuration *2:2:4* required data communication across machines (which was typically slower than intra-node data communication).

Another interesting observation was that *METIS* performed better than *LDG* and Argo-H in most configurations except configuration *8:2:1*. This was probably because the partitionings computed by *METIS* had the lowest edge-cut and thus lowest amount of contention on the memory subsystems. The reason why Argo-H was worse than *METIS* and sometimes even worse than *LDG* in dense configurations (i.e., *1:2:8, 2:2:4*, and *4:2:2*) was because Argo-H was a hop-cut based solution. It aims to avoid inter-machine data communication by gathering neighbouring vertices as close as possible, which may lead to significant intra-node data communication and thus increase the contention on the memory subsystems.

However, Argo-H outperformed *METIS* and *LDG* on two out of the three workloads under configuration *8:2:1*. This

TABLE VI: BFS, SSSP, and PageRank Execution Time (Second) on com-orkut Dataset with Varying Message Grouping Size

Configuration	BFS (10 Source Vertices)			SSSP (10 Source Vertices)			PageRank (30 Iterations)		
	64	128	256	64	128	256	64	128	256
METIS	196	27.27	8.59	3,730	787	125	1,435	121	32.74
LDG	136	33.32	9.52	3,003	523	71.84	1,110	161	48.93
Argo-H	306	40.84	9.28	4,750	1,033	147	2,088	179	31.81
Argo	73.11	19.12	5.20	1,528	196	49.84	406	71.74	16.68

TABLE VII: BFS, SSSP, and PageRank LLC Misses in Millions on com-orkut Dataset with Varying Message Grouping Size

Configuration	BFS (10 Source Vertices)			SSSP (10 Source Vertices)			PageRank (30 Iterations)		
	64	128	256	64	128	256	64	128	256
METIS	843	50	17	38,942	6,313	471	10,605	529	22
LDG	194	27	22	30,096	1456	59	4,605	69	43
Argo-H	1,702	36	22	51,774	8,173	589	17,360	748	35
Argo	35	26	21	8,702	163	49	142	49	37

was expected because under configuration *8:2:1* reducing inter-machine data communication became more critical than mitigating the contention. This also confirmed the fact that the network may not always be the bottleneck. The reason why Argo-H did not outperform *METIS* on PageRank execution was probably because PageRank was more communication-intensive than BFS and SSSP, and thus the contention on the memory subsystems was still the dominant factor even under the sparsest configuration.

B. Results in terms of LLC misses (Table V)

To confirm that the slowdown was indeed caused by the contention on the memory subsystems, we also reported the LLC misses for each execution of the workloads in Table V. The LLC misses were collected via the PAPI_L3_TCM event provided by the hardware performance counter programming tool, PAPI [31], and the values reported were the average LLC misses across partitions (MPI processes). By comparing Tables V and IV, we observed that the timing results were highly consistent with the LLC miss results. The denser the configuration was, the larger the LLC misses and thus the longer execution time of the workload would be. We also observed that under configuration *8:2:1* Argo-H had much higher LLC cache misses than that of *METIS* for BFS and SSSP, but it still outperformed *METIS* in terms of the execution time. This further confirmed our assumption that under configuration *8:2:1* reducing inter-machine data communication was more critical to the performance than mitigating contention on memory subsystems (e.g., cache pollution caused by inter-socket data communication), for BFS and SSSP.

C. Discussions

The above experimental results can be summarized as follows:

Take-Away 1 *The contention on the memory subsystem can also have significant performance impact on distributed workloads, especially for multicore machines connected via high-speed networks.*

Take-Away 2 *Heterogeneity-aware graph (re)partitioners are designed for cases where the network is the bottleneck, especially for geo-distributed clusters or cloud computing environments.*

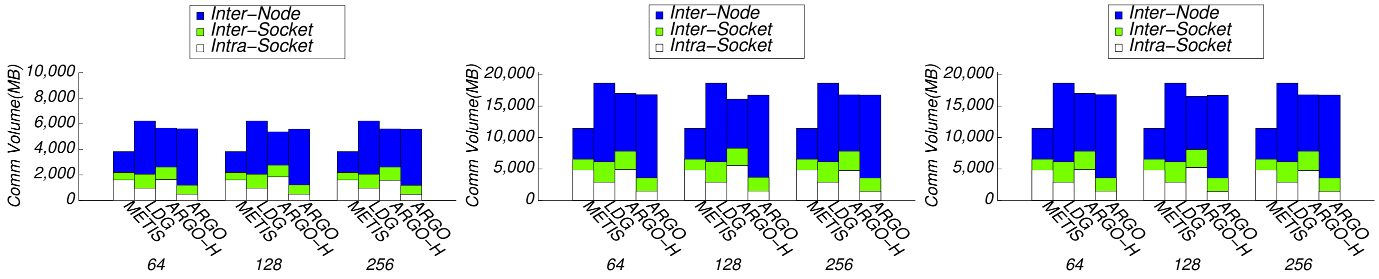
VI. Effectiveness in Avoiding Contention

This experiment evaluated the effectiveness of Argo in avoiding contentiousness using BFS, SSSP, and PageRank on the com-orkut dataset. In the experiment, the dataset was partitioned across three 20-core compute nodes with one partition per core. As demonstrated in Section V, the contention on the memory subsystems on the cluster was the primary bottleneck. Hence, we set λ to 1 for all the experiments presented below.

A. Results in terms of Execution Time (Table VI)

Table VI shows the workload execution time on decompositions computed by *METIS*, *LDG*, Argo-H, and Argo with three different message grouping sizes: 64, 128, and 256. As expected, Argo had the lowest workload execution time in all cases. In comparison to *METIS*, *LDG*, and Argo-H, Argo, respectively, speeded up the execution of BFS by up to 2.67, 1.85, and 4.18 times; the execution of SSSP by up to 4, 2.66, and 5.26 times; and the execution of PageRank by up to 3.53, 2.93, and 5.14 times.

Interestingly, we found that Argo-H performed the worst in almost all cases. This was also expected because Argo-H aimed to grouping neighbouring vertices as close as possible, which may cause an increase in the intra-node data communication and thus aggravate the contention on the memory subsystems. However, as the message grouping size increased, the gap between Argo-H and *METIS*/*LDG* was gradually closed up. This was because, the larger the message grouping size was, the fewer the messages were exchanged and thus the less contention on the memory subsystems. As a result, the importance of reducing inter-machine data communication gradually increased, calling for heterogeneity-aware graph partitioners. This also explained the reason why the improvement achieved by Argo decreased sometimes as the message grouping size increased.



(a) BFS Communication Volume (b) SSSP Communication Volume (c) PageRank Communication Volume

Fig. 5: Breakdown communication volume for the execution of BFS, SSSP, and PageRank on com-orkut partitionings, and ARGO. Here, intra-socket, inter-socket, and inter-node, respectively, represent the communication volume among partitions that were assigned to the same sockets, the communication volume among partitions that were residing on different sockets but on the same machines, and the communication volume among partitions of different machines.

Take-Away ARGO *performs better for workloads with a large number of small message exchanges, whereas* ARGO-H *seems to be more suitable for workloads with lots of large message exchanges.*

B. Results in terms of LLC Misses (Table VII)

To further show that the improvement was indeed caused by the reduced contention on the memory subsystems. We also recorded the LLC misses for the execution of the workloads in Table VII. As shown, the LLC miss results were highly consistent with the timing results: (1) ARGO had the lowest LLC misses in almost all cases whereas ARGO-H had the highest LLC misses in most cases; and (2) the larger the message grouping size was, the fewer the misses were.

Interestingly, we found that with message grouping size of 256, *METIS* actually had lower LLC misses than that of ARGO for the execution of BFS and PageRank. However, ARGO still beat *METIS* in terms of execution time (Table VI). We attributed this to two facts (1) that intra-node data communication required the involvement of the CPU (CPU spending time in communicating the data), while inter-machine data communication relieved the CPU from the communication (allowing it to focus on computation: processing the messages received); and (2) that the larger message grouping size allowed a larger degree of the overlap between the computation and communication, further amplifying the benefits of RDMA-enabled networks.

Take-Away *It is important to take both the contention on the memory subsystems and the communication heterogeneity into account while partitioning.*

C. Results in terms of Communication Volume (Figure 5)

To further confirm that the reduction in the contention was indeed caused by the reduced intra-node data communication, we also present the breakdown communication volume for each execution of the workloads in Figure 5. As shown, ARGO had the lowest intra-node data communication in all cases, while ARGO-H had the highest intra-node data communication. When compared with *METIS* and *LDG*, ARGO, respectively, reduced the intra-socket data communication by up to 70% and 40% for the execution of BFS, by up to 70% and 50% for the execution of SSSP, and by up to 70% and 50% for the execution of PageRank. All these matched the timing and LLC misses results. Another interesting observation was that even though *METIS* had lower overall communication volume than that of ARGO, ARGO still outperformed *METIS* in terms of execution time due to the reduced communication volume in critical components (intra-node data communication).

Take-Away *Putting too much data communication into cores of the same machine may lead to significant contention on the memory subsystems and thus hurt the performance. Counter-intuitively, offloading a certain amount of intra-node data communication across machines may sometimes achieve better performance due to the presence of RDMA-enabled networks.*

VII. SCALABILITY STUDY

A. Scalability in terms of Graph Size (Table VIII)

Configuration This experiment evaluated the scalability of ARGO as the size of the graph increased. Towards this, we generated six additional datasets by sampling the edge set of the Friendster and Twitter datasets. Then, we examined the execution time of the workloads on the datasets when they were partitioned across four 20-core machines (with one partition per core and message grouping size of 512). Note that *METIS* failed to partition the datasets.

Results Table VIII shows the corresponding workload execution time as the size of the graphs increased. As can be seen, ARGO outperformed both *LDG* and ARGO-H in all cases, whereas ARGO-H was always the worst. Compared to *LDG*, ARGO achieved by to 2.71x, 2.72x, and 3.58x speedups for the execution of BFS, SSSP, and PageRank, respectively. As expected, the speedups against ARGO-H were much higher, since what ARGO-H did during the partitioning aggravated the contention issue. The speedups were quite consistent in spite of the increasing graph size, showing the stability and scalability of ARGO.

TABLE VIII: BFS, SSSP, and PageRank Execution Time in Seconds as the Graph Size Increased

# of Edges (in Billion)	BFS (5 Source Vertices)			SSSP (5 Source Vertices)			PageRank (15 Iterations)		
	LDG	Argo-H	Argo	*LDG*	Argo-H	Argo	*LDG*	Argo-H	Argo
Friendster									
0.9	10.74	16.46	7.93	111	266	54.46	36.79	65.92	18.80
1.8	37.46	74.76	24.24	599	1,700	243	156	479	108
2.7	78.78	147	49.87	2,273	3,429	1,007	476	4,972	1346
3.6	156	470	80.26	3,243	4,531	1,687	757	2,259	361
Twitter									
0.98	13.10	15.68	7.58	126	414	66.09	51.46	79.88	33.65
1.96	44.94	157	28.44	1,190	1,932	437	262	1,019	169
2.94	146	399	72.08	3,788	4,690	2,071	1,071	2,071	430
3.92	285	607	105	6,875	8,610	4,688	2,208	2,951	617

TABLE IX: BFS, SSSP, and PageRank Execution Time in Seconds as the # of Partitions Increased

Number of Partitions	BFS (5 Source Vertices)			SSSP (5 Source Vertices)			PageRank (15 Iterations)		
	LDG	Argo-H	Argo	*LDG*	Argo-H	Argo	*LDG*	Argo-H	Argo
Friendster									
80	156	470	80.26	3,243	4,531	1,687	757	2,259	361
100	68.66	212	37.72	1,747	3,304	541	350	1,248	182
120	42.71	210	21.52	878	2,210	262	252	975	141
140	42.63	121	22.07	384	2,059	162	152	626	83.43
160	29.20	81.81	20.45	228	1,732	151	134	441	65.40
180	24.26	61.88	18.81	201	1,350	72.42	82.94	282	52.49
200	20.17	48.47	18.83	146	1,079	120	58.28	244	51.79
Twitter									
80	285	607	105	6,875	8,610	4,688	2,208	2,951	617
100	124	457	69.83	3,647	4,859	2,062	651	2,012	359
120	85.93	160	39.10	2,297	3,903	848	488	1,427	241
140	75.20	149	24.81	948	2,737	351	264	880	128
160	35.32	145	23.84	475	1,765	174	173	305	108
180	25.37	80.12	22.88	283	1,754	158	118	260	64.37
200	28.24	57.74	21.36	261	1,177	135	116	214	63.81

B. Scalability in terms of # of Partitions (Tables IX & X)

Configuration This experiment inspected the effectiveness of Argo as the number of partitions increased. Towards this, we partitioned the original Friendster and Twitter dataset across four up to ten 20-core machines (one partition per core) and then examined the BFS, SSSP, and PageRank execution time on the partitionings (with message grouping size of 512) computed by *LDG*, Argo-H, and Argo.

Results in terms of Execution Time (Table IX) Table IX presents the corresponding results. As expected, Argo performed the best in all cases whereas Argo-H performed the worst. In comparison to *LDG*, Argo, respectively, speeded up the execution of BFS by up to 3.03x, the execution of SSSP by up to 3.36x, and the execution of PageRank by up to 3.58x. The corresponding speedups against Argo-H were as high as 9.78x, 12.70x, and 6.9x, respectively.

We also noted that the workload execution time decreased, as the number of partitions increased. One of the reasons for this was that as the number of partitions increased, the degree of parallelism also increased. Another possible reason was that the degree of contention on the memory subsystems decreased due to the reduced intra-node data communication volume. The drop in the intra-node data communication was caused by the increasing number of inter-machine communication peers. For example, with four machines (80 partitions), each partition only had 60 inter-machine communication peers, whereas with five machines (100 partitions), the number of inter-machine communication peers of each partition increased to 80. This also explains the reason why the improvement achieved by Argo became smaller as the number of partitions increased. Nevertheless, one thing should be aware of here was that the size of the graph remained unchanged and that Argo reduced the execution time of each core used by this much.

TABLE X: Partitioning Time in Seconds

# of Partitions	Friendster		Twitter	
	LDG	Argo	*LDG*	Argo
80	68.70	313	99.02	110.09
100	71.57	387	59.74	157.57
120	72.37	477	68.99	176.86

Results in terms of Partitioning Overhead (Table X) We also reported the partitioning overhead (vertex placement decision time) of Argo in Table X. The longer partitioning time of Argo was caused by an optimization we made: Argo loaded vertices of the graph from the file system in blocks and streamed each in-memory vertex block twice to further improve the partitioning quality. Thus, ARGO was a partial restreaming graph partitioner. Fortunately, the partitioning only has to be performed once and can be used multiple times. Also, graph processing often require the processing of the entire graph (e.g., SSSP for a large set of source vertices or PageRank with more iterations) which will have significantly longer execution time when compared with partitioning time.

VIII. Related Work

Distributed Graph Computing Many distributed graph computing frameworks, such as Pregel [23], Giraph [10], GraphLab [20], PowerGraph [11], Mizan [18], Giraph++ [38], GoFFish [35], and Blogel [43], have been proposed for big graph processing. These systems hide the complexity of data partitioning, computation parallelization, and fault tolerance from users, providing a simple and elegant way for users to design and implement scalable distributed graph algorithms.

Pregel, as one of the most popular graph computing engines, adopts a *vertex-centric* model. In such a model, users only need to specify the logic for one vertex, whereas the system will hide the complexity of executing the logic on all vertices in a distributed fashion. The execution is carried out in a sequence of *supersteps* separated by a global synchronization barrier. In each superstep, the vertex can change its state and the state of its outgoing edges, send messages to its neighbours to be processed in the next superstep, or even modify the structure of the graph. Vertices can vote to halt at the end of each superstep and be reactivated by messages from its neighbors. The execution ends when all vertices are inactive.

Graph Partitioning Most of the systems partition the vertices of the graph across workers by cutting edges (edge-cut), except PowerGraph which partitions the edges of the graph across workers by cutting vertices (vertex-cut). Edge-cut based graph partitioning has been studied for decades [12], [34]. The well-studied multilevel graph partitioners, like *METIS* [26], are known for their capability of producing high-quality decompositions. However, they scale poorly against large graphs even if performed in parallel since they require full knowledge of the graph for partitioning.

Streaming [36], [39], [27] and restreaming [27] partitioners are the recent solutions to large graph partitioning. In the streaming setup, the graph is treated as a stream of vertices. Upon arrival of a vertex, the partitioner places the vertex to one of the partitions permanently based on the distribution of the vertices that previously arrived. As can be seen, the information that the partitioner can use for vertex placement decision is limited. To address this, restreaming partitioning consists of several passes of streaming partitioning and allows subsequent passes can have access to the results of previous passes. By doing this, the partitioner is capable of leveraging more information about the graph for partitioning. They were reported to be able to output decompositions comparable to *METIS* but within a relatively short time.

Recently, a new distributed graph partitioner, Sheep [24], has been proposed for large graph partitioning. It is similar in spirit to *METIS*. They both first reduce the original graph to a smaller tree or a sequence of smaller graphs, then do a partition of the tree or the smallest graph, and finally map the partitioning back to the original graph. In terms of partitioning time, Sheep performs better than both *METIS* and streaming partitioners. For partitioning quality, Sheep is competitive with *METIS* for a small number of partitions and is competitive with streaming graph partitioners (such as *LDG* [36]) for larger numbers of partitions. Since Sheep has similar characteristics as *METIS* and streaming partitioners in terms of partitioning quality, we omitted its comparison, especially considering *METIS* and *LDG* are more prevail graph partitioners.

Several heterogeneity-aware graph partitioners [6], [41], [42] have been proposed. However, none of them considers the contention issue on the memory subsystems of modern multicore machines. The only two heterogeneity- and contention-aware work are our prior work [45], [44]. Nevertheless, they are graph *repartitioners*, whereas ARGO is a graph *partitioner*. This is also the reason why we did not compare ARGO to [45], [44]. Additionally, this paper extends our prior work [45], [44] by providing a holistic view on (1) why contention on the memory subsystems may become a problem for distributed (graph) workloads (Section II); and (2) to what extent the contention may harm the performance (Section V).

In fact, several vertex-cut graph partitioners [40], [32], [11], [25] were also proposed to improve the performance of distributed graph computation. Although they belong to a different type graph partitioners, they all have to face the heterogeneity and contention issue as edge-cut solutions. In fact, work [25] is a first attempt to address the heterogeneity issue for vertex-cut solutions.

IX. Conclusion

In this paper, we first demonstrated that the contention in the memory subsystems of modern multicore clusters with high-speedup networks can have significant performance impact on distributed workloads. Then, we presented an architecture-aware graph partitioner, ARGO, which considers the impact of both the contention on the memory subsystems and the heterogeneity in the network communication costs while partitioning. Our experimental results show that ARGO achieved up to 12x speedups for the execution of BFS, SSSP, and PageRank on real-world graphs and scaled quite well in terms of both graph size and the number of partitions.

Acknowledgment

We would like to thank Peyman Givi, Patrick Pisciuneri, and the anonymous reviewers for their help on the paper. This work was funded in part by NSF awards CBET-1250171 and CBET-1609120.

References

[1] https://en.wikipedia.org/wiki/Message_Passing_Interface.

[2] http://snap.stanford.edu/data.

[3] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 2006.

[4] A. Buluç and K. Madduri. Parallel Breadth-First Search on Distributed Memory Systems. *CoRR*, 2011.

[5] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In *ICPP*, 2009.

[6] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, 2012.

[7] G. Echbarthi and H. Kheddouci. Fractional greedy and partial restreaming partitioning: New methods for massive graph partitioning. In *BigGraphs*, 2014.

[8] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, 1999.

[9] C. B. A. C. A. Galakatos and T. K. E. Zamanian. The End of Slow Networks: It's Time for a Redesign. *VLDB*, 2016.

[10] Giraph. http://giraph.apache.org/.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[12] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 2000.

[13] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *IPDPS*, 2010.

[14] RDMA protocol: improving network performance. http://h21007.www2.hpe.com/portal/download/files/unprot/c00589475.pdf.

[15] http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[16] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *ICPP*, 2005.

[17] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, 1995.

[18] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, 2013.

[19] http://konect.uni-koblenz.de/networks/.

[20] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv:1408.2041*, 2014.

[21] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *VLDB*, 2014.

[22] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 2007.

[23] 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 *SIGMOD*, 2010.

[24] D. Margo and M. Seltzer. A Scalable Distributed Graph Partitioner. *VLDB*, 2015.

[25] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning. In *ICDCS*, 2016.

[26] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[27] J. Nishimura and J. Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *KDD*, 2013.

[28] http://www.open-mpi.org/.

[29] http://mvapich.cse.ohio-state.edu/benchmarks/.

[30] E. Papalexakis, B. Hooi, K. Pelechrinis, and C. Faloutsos. Power-Hop: A Pervasive Observation for Real Complex Networks. *PloS one*, 2016.

[31] http://icl.cs.utk.edu/papi/.

[32] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *CIKM*, 2015.

[33] http://core.sam.pitt.edu/MPIcluster.

[34] K. Schloegel, G. Karypis, and V. Kumar. *Graph partitioning for high performance scientific simulations*. AHPCRC, 2000.

[35] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *EuroPar*. 2014.

[36] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, 2012.

[37] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *PPoPP*, 2006.

[38] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *VLDB*, 2013.

[39] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, 2014.

[40] C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis. In *NIPS*. 2014.

[41] N. Xu, B. Cui, L.-n. Chen, Z. Huang, and Y. Shao. Heterogeneous Environment Aware Streaming Graph Partitioning. *TKDE*, 2015.

[42] J. Xue, Z. Yang, S. Hou, and Y. Dai. When computing meets heterogeneous cluster: Workload assignment in graph computation. In *BigData*, 2015.

[43] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *VLDB*, 2014.

[44] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Repartitioning. In *ICDE*, 2016.

[45] A. Zheng, A. Labrinidis, P. Pisciuneri, P. K. Chrysanthis, and P. Givi. Paragon: Parallel Architecture-Aware Graph Partitioning Refinement Algorithm. In *EDBT*, 2016.