ARCHITECTURE- AND WORKLOAD-AWARE GRAPH (RE)PARTITIONING

by

Angen Zheng

BS, Beijing Information Science and Technology University, 2009

MS, Beijing University of Posts and Telecommunications, 2012

Submitted to the Graduate Faculty of

the DIETRICH SCHOOL OF ARTS AND SCIENCES in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Angen Zheng

Dr. Alexandros Labrinidis, Department of Computer Science, University of Pittsburgh

Dr. Panos K. Chrysanthis, Department of Computer Science, University of Pittsburgh

Dr. Jack Lange, Department of Computer Science, University of Pittsburgh

Dr. Peyman Givi, Department of Mechanical Engineering and Materials Science, University of Pittsburgh

Dissertation Director: Dr. Alexandros Labrinidis, Department of Computer Science, University of Pittsburgh

ARCHITECTURE- AND WORKLOAD-AWARE GRAPH (RE)PARTITIONING

Angen Zheng, PhD

University of Pittsburgh, 2017

Graph partitioning and repartitioning have been studied for several decades. Yet, they are receiving more attention due to the increasing popularity of large graphs from various domains, such as social networks, web networks, telecommunication networks, and scientific simulations. Traditional well-studied graph (re)partitioners often scale poorly against these continuously growing graphs. Recent works on streaming graph partitioning and lightweight graph repartitioning usually assume a homogeneous computing environment. However, modern parallel architectures may exhibit highly non-uniform network communication costs. Several solutions have been proposed to address this, but they all consider the network as the primary bottleneck of the system, even though transferring data across modern high-speed networks is now as fast as the local memory access. As such, minimization of the network data communication may not be a good choice. We found that putting too much data communication into partitions assigned to cores of the same machines may result in serious contention for the shared hardware resources (e.g., last level cache, memory controller, and front-side bus) on the memory subsystems in modern multicore clusters. The performance impact of the contention can even become the dominant factor in limiting the scalability of the workload, especially for multicore machines connected via high-speed networks. Another issue of existing graph (re)partitioners is that they are usually not aware of the runtime characteristics of the target workload. *To enable efficient distributed graph computation, this thesis aims to (1) understand the performance impact of non-uniform network communication costs, the impact of contention on the memory subsystems, as well as the*

impact of workload runtime characteristics on distributed graph computation; and (2) design and implement new scalable graph (re)partitioners that take these factors into account.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF EQUATIONS

PREFACE

First and foremost, I want to thank my advisor Alexandros Labrinidis and co-advisor Panos K. Chrysanthis. It has been an honor to be their Ph.D. student. They have taught me, both consciously and unconsciously, how good scientific research is done. I appreciate all their contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating.

The members of the ADMT group have contributed immensely to my personal and professional time at Pitt. The group has been a source of friendships as well as good advice and collaboration.

For this dissertation, I would like to thank my thesis committee members: Alexandros Labrinidis, Panos K. Chrysanthis, Jack Lange, Peyman Givi, and Patrick Pisciuneri for their time, interest, insightful questions, and helpful comments. I would also like to thank Christos Faloutsos at the Carnegie Mellon University for his collaboration on the last piece of my thesis work. I also want to thank Juli Stresing, Wenchen Wang, Zhaohong Wu, and Xiang Xiao for proofreading my thesis.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. My work was supported by the NSF awards CBET-1250171, CBET-1609120, and OIA-1028162.

My time at Pitt was made enjoyable in large part due to the many friends and groups that became a part of my life. I am grateful for the time spent with roommates and friends, and for many other people and memories.

Lastly, I would like to thank my family for all their love and encouragement.

1.0 INTRODUCTION

Large graph datasets are becoming increasingly popular nowadays. For example, graphs like Web Graphs, Biological Networks, and Social Networks, are often at the scale of hundreds of billions or even a trillion (10^{12}) edges, and they are continuously growing. As a consequence, many distributed graph computing frameworks, such as Pregel [1], GraphLab [2] and PowerGraph [3], have been developed. In addition to the data that is inherently represented as graphs, many problems in scientific simulations [4] as well as machine learning and data mining [2] can be modeled as graph problems.

In such systems, distributing vertices of the graph evenly across partitions often corresponds to an even load distribution, while minimizing the *edgecut* (the number of edges connecting different partitions) helps minimize the amount of data communication. This is known as the *balanced graph partitioning* problem. Balanced graph partitioning has been extensively studied and proved to be NP-hard [5, 6, 4, 7, 8, 9, 10]. The most well-known heuristic-based approaches are *multi-level* ones [5, 6, 4] and *streaming* ones [8, 9, 10].

One of the pitfalls of a big portion of the proposed solutions is that minimal data communication does not always mean minimal network communication cost, since the computing infrastructures may exhibit highly non-uniform network communication costs. Several *hopcut*-based solutions [11, 12, 13, 14] have been proposed to make the graph partitioning procedure aware of the issue of non-uniform network communication costs.

Nevertheless, *both edgecut- and hopcut-based solutions are designed with the assumption that the network is the primary bottleneck and should be avoided at all costs.* This assumption has been broken by new advancements in HPC (high performance computing) infrastructures. It is common to see clusters that are, nowadays, connected via high-performance networks with RDMA (remote direct memory access) capabilities, like Infiniband. In fact,

Infiniband was reported to connect 65% of the HPC platforms, and 39% of the overall TOP500 systems by November 2016 [15]. It was also reported that transferring data across the network (like Infiniband) has been almost as fast as local memory access [16]. The next generation of Infiniband (EDR Infiniband) is even able to deliver up to 100Gbps network bandwidth. As a result, focusing on minimizing network data communication may not always lead to performance improvement.

At the same time, the microprocessor industry has shifted from boosting the clock speed of uniprocessors to multicore processors by continuously integrating many small processing units onto a single chip. Multicore machines nowadays have become the backbone of modern HPC infrastructures. Due to the increasing core count per node, the contention for the shared resources (e.g., last level cache, memory controller, and front-side bus) on the memory subsystems is becoming more and more notorious. It has been shown that the contention can significantly impact the performance of the collocated workloads [17, 18] even for collocated processes from the same distributed workload [19]. What is even worse is that the core count per node is continuously increasing. It is expected that there will be nodes with hundreds of cores in the near future. The fast development of high-speed networks further aggravates the issue. Nevertheless, none of the existing graph partitioning algorithms is aware of the contention issue. In fact, they may even increase the degree of the contention on the memory subsystems, especially the hopcut-based solutions. This is because existing solutions always try to avoid network data communication even at the cost of increasing intra-node data communication (data communication among partitions assigned to cores of the same machine), despite the fact that excess intra-node data communication may saturate the memory subsystems, amplifying the contention.

As mentioned earlier, existing graph (re)partitioners often assume that distributing vertices of the graph evenly across partitions corresponds to an even load distribution. In other words, they assume that vertices of the graph are always active during the computation. Nevertheless, for some workloads, like Breadth-First Search (BFS) and Single-Source Shortest Path (SSSP), only a subset of the vertices participate in the computation in a specific time period. As a result, the execution of such workloads on the partitionings computed by existing graph (re)partitioners may suffer from significant *time-varying skewness*, leading to

resource underutilization. Another issue of existing graph (re)partitioners is that many of them assume that vertices of the graph have uniform computation requirements and uniform communication requirements with their neighbors. Nevertheless, the computation and communication requirements of the vertices are highly application-dependent.

In this dissertation, we argue that *computations performed on the partitionings computed by existing graph partitioning algorithms could not always fully utilize the underlying computing infrastructures*, which impedes the efficiency of the computing infrastructures as well as the scalability of the target workload. To enable efficient distributed graph computation on modern HPC infrastructures, we advocate for *architecture- and workload-aware graph partitioning. Architecture-aware means that we should consider the performance impact of these new hardware trends while partitioning, whereas workload-aware indicates that the (re)partitioner should be aware of the runtime characteristics of the target workload.*

1.1 RESEARCH OVERVIEW

1.1.1 Thesis Statement

Architecture- and workload-aware graph partitioning enables efficient distributed graph computation on modern HPC infrastructures.

1.1.2 Target Computing Infrastructure

In this thesis, we study the graph partitioning problem for distributed graph computing on *dedicated HPC clusters*. In such clusters, the compute nodes are equipped with multiple CPU sockets and each CPU socket has multiple cores. Compute nodes of the cluster are often connected via high-speed networks, like Infiniband. To perform a computation on such systems, users need to first submit their jobs to the cluster. The cluster manager will allocate the resources requested for the user. Once allocated, the user will have dedicated access to the allocated resources.

1.1.3 Assumptions

In our study of the problem, we assume (1) that graphs are partitioned across cores of the allocated compute nodes with one partition per core for parallel processing; (2) that the mapping of a partition to a core, as well as the number of partitions remains unchanged throughout the computation; and (3) that the set of the allocated cores used for graph processing is also the set of cores used for the (re)partitioning.

1.1.4 Main Contributions

We first investigated the performance impact of the new hardware trends (i.e., multicore machines connected via high-speed networks) on distributed graph workloads. As a result of this study, we identified two important factors that one should be aware of while partitioning the graphs: (a) the non-uniform network communication costs of the underlying computing infrastructures; and (b) the contention for the shared hardware resources on the memory subsystems of modern HPC clusters. We also provided a holistic view on: (a) why we have to be aware of such factors for distributed graph workloads; and (b) to what extent these factors may impact the performance of distributed graph workloads.

To avoid such negative performance impact, we proposed an **architecture-aware graph partitioning algorithm**, Argo [20], for efficient distributed graph computation on static graphs. Argo follows the same streaming partitioning model proposed by others. In this model, vertices arrive at the partitioner in a certain order along with their adjacency lists. The partitioner decides the placement of each newly arrived vertex to one of the partitions permanently based on the placements of the vertices previously arrived. The key novelty of Argo lies in making the vertex placement aware of (a) the non-uniform network communication costs of the underlying computing infrastructures; and (b) the contentiousness of the memory subsystems of modern HPC clusters. We also make Argo aware of the non-uniform computation and communication requirements of the vertices by encoding such information into the vertex and edge weights of the graph. Since making the (re)partitioner aware of such workload characteristics is fairly straightforward, we will primarily focus on the discussion of architecture-awareness throughout the thesis.

We also proposed four new **architecture-aware graph repartitioning algorithms**: Aragon [21], Paragon [22], Planar [23], and Planar+ [24] for efficient distributed graph computation on dynamic graphs. They all attempt to adapt the current partitioning to the changes in the graph by migrating vertices among the partitions. The migration is only allowed if the gain of moving the vertex from its current partition to an alternative partition is positive. The gain of migrating a vertex is defined as the reduction in the communication cost incurred by the vertex during the computation. One of the key contributions of this thesis is that we make the vertex gain computation process aware of both the communication heterogeneity and contentiousness of the underlying computing infrastructures. Nevertheless, Aragon is a centralized solution with the assumption that the graphs are small enough to be held in the memory of a single machine, whereas Paragon is a parallel version of Aragon designed for median-sized graphs. Planar and Planar+ overcome the drawbacks of Paragon by scaling it to even larger graphs and by increasing the degree of parallelism the repartitioning algorithm can have. In addition to being highly scalable, the partitionings computed by them also have much lower hopcut/edgecut than that of Aragon and Paragon. Planar+ further reduces the overhead of Planar, by introducing an efficient way of modeling the communication heterogeneity and contentiousness. This, in turn, enables an optimized vertex gain computation. Making the partitioning algorithms scale efficiently against large graphs is another key contribution of the thesis. Similar to Argo, Aragon, Paragon, Planar, and Planar+ are all aware of the non-uniform computation and communication requirements of the vertices.

Finally, we examine a special type of workload-aware graph partitioning: **skew-resistant graph partitioning**. In particular, we would like to distribute the vertices that are active in the same time period evenly across the partitions and thus avoid the issue of time-varying skewness. Towards this, we studied the runtime characteristics of two representative traversal-style graph workloads: BFS and SSSP. Based on the study, we introduced the idea of multi-label graph partitioning (MLGP) and an application of this idea is to do skew-resistant graph partitioning. We also identified a set of target workloads that MLGP can be applied to.

1.1.5 Main Impact

The main impact of this thesis is that we identified an important aspect that has been ignored by the current graph processing community, that is, the performance impact of the underlying HPC infrastructures, especially the contentiousness of the memory subsystems, on distributed graph computation. In fact, this is also a blind spot for general distributed computation, where people often assume that the network is the bottleneck.

In particular, we made an in-depth analysis about the factors that one should consider while (re)partitioning the graph for distributed graph computing, namely, the non-uniform network communication costs (Section 2.2.1) and the contention on the memory subsystems (Section 2.2.2). We also experimentally demonstrated that (1) the network may not always be the bottleneck in modern HPC clusters (Section 2.2.3); and (2) the contention on the memory subsystems can impact the performance of distributed graph computation significantly (Section 2.2.3). Based our analysis and our observations, we showed that even with simple managed graph (re)partitioning we can achieve significantly better performance (Chapters 3 & 4). All these observations will enable the graph processing community to rethink the design of graph (re)partitioning algorithms and even the design of distributed graph computing frameworks.

1.2 OUTLINE

The rest of the dissertation is organized as follows:

In Chapter 2, we will first review the literature on the topics of distributed graph computation and graph partitioning and then motivate our work by demonstrating the importance of architecture-awareness.

In Chapter 3, we will introduce Argo, an architecture-aware graph partitioning algorithm we proposed for static graph partitioning.

In Chapter 4, we will study the problem of architecture-aware graph partitioning for dynamic graphs. In particular, we will present four solutions we proposed: Aragon,

Paragon, Planar, and Planar+.

In Chapter 5, we will investigate the problem of skew-resistant graph partitioning.

Finally, we will conclude in Chapter 6

2.0 BACKGROUND AND MOTIVATION

2.1 LITERATURE REVIEW

In this section, we review the topic of distributed graph computation as well as the state-of-the-art graph partitioning and repartitioning algorithms.

2.1.1 Distributed Graph Computation

Recently, many distributed graph computing frameworks, such as Pregel [1], Giraph [25], GraphLab [2], PowerGraph [3], Mizan [26], Giraph++ [27], GoFFish [28], and Blogel [29], 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 the *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 the 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 neighbors 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 the vertices are inactive.

However, the vertex-centric model has its own drawbacks (e.g., high communication cost for graphs with high average vertex degree and long convergence time for graphs with

large diameters). To address these limitations, Giraph [25] allows the use of customized graph partitioners to mitigate the communication cost, and Mizan [26] exploits dynamic load balancing to avoid runtime skewness. GraphLab [2] introduces the asynchronous execution mode to eliminate the need of per superstep global synchronization, whereas PowerGraph [3] proposes the idea of vertex-cut graph partitioning to speed up the processing of power-law graphs. In addition to Giraph++ [27], GoFFish [28], and Blogel [29], which adopt the *block- or subgraph-centric* model, there are also *query-centric* systems, like Horton [30] and Quegel [31], which are designed for online querying of big graphs.

2.1.2 Graph Partitioning and Repartitioning

A common characteristic of these graph computing engines is that the distribution of the graph (the partitioning of the graph) across the computing elements can impact the performance greatly. Most of the systems adopt the edgecut-based graph partitioning solution, where vertices of the graph are distributed across the partitions while edges connecting different partitions are cut. This is also the type of graph partitioners this thesis focuses on.

Heavyweight Graph Partitioning Edgecut-based graph partitioning have been extensively studied [32, 33, 34, 35, 36]. Among these, the multi-level graph partitioner, METIS [32], is the most well-known one. Nevertheless, these graph partitioners often scale poorly against large graphs, even if performed in parallel like PARMETIS [33] and ZOLTAN [36]. Besides, none of them is architecture-aware. Although work [11] and [12] are heterogeneity-aware (i.e., they consider the issue of non-uniform network communication costs while partitioning), neither of them is contention-aware (i.e., none considers the contention issue on the memory subsystems of modern multicore clusters). Last but not the least, they both rely on the use of heavyweight graph partitioners, making them infeasible for large graph partitioning.

Streaming Graph Partitioning To address the scalability issue of the heavyweight graph partitioners, a new family of graph partitioning heuristics, namely streaming graph partitioning [8, 10, 37], has been proposed recently for online graph partitioning. They can produce partitionings that are comparable to METIS in terms of partitioning quality (edgecut) but within a relatively short time. However, they are not architecture-aware. Although [13, 14]

are two heterogeneity-aware streaming graph partitioners, they are not contention-aware. To address this issue, we proposed ARGO, an architecture-aware (heterogeneity- and contention-aware) streaming graph partitioner.

In addition to these streaming graph partitioners, a new distributed architecture-agnostic graph partitioner, Sheep [38], 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 [8], for larger numbers of partitions.

Lightweight Graph Repartitioning The majority of the above referenced graph partitioners (except PARMETIS and ZOLTAN) are designed for static graph partitioning. However, many graphs are dynamic, continuously evolving over time. The changes can be either in the graph structure (like vertex/edge addition/deletion) or in the graph properties (like changes in vertex weights and edge weights). As a result, the quality of the partitioning computed on the stale graph may degrade over time, requiring the graph to be repartitioned periodically to maintain good performance. Indeed, the model adopted by the streaming graph partitioners can handle a certain type of graph dynamism (e.g., vertex/edge addition). However, they tend to lead to suboptimal performance for the computation in the presence of dynamism [39] and they are incapable of handling other types of dynamism. Although we could potentially use existing graph partitioners to compute a new partitioning for the changed graph from scratch, they usually lead to high migration costs. Besides, existing graph partitioners cannot be used as online graph repartitioners, since they scale poorly against large graphs.

Consequently, many lightweight graph repartitioners [40, 39, 41, 26, 42] have been proposed for dynamic graph partitioning. They tackle the scalability issue of heavyweight graph (re)partitioners and the issue of streaming graph partitioners with dynamic graphs by incrementally migrating vertices among the partitions on-the-fly based on some heuristics. Nevertheless, they are architecture-agnostic. Also, many of them assume uniform vertex weights (uniform computation requirements of the vertices) and vertex sizes (uniform

amount of application state associated with the vertices), and some [39, 41] even assume uniform edge weights (uniform amount of data communication along the edges). However, real-world graphs often have non-uniform vertex weights, vertex sizes, and edge weights. In fact, the weights and sizes are highly algorithm-dependent.

Work [26] is a Pregel-like graph computing engine, which allows it to migrate vertices based on the runtime characteristics of the target workload (i.e., the number of messages sent/received by each vertex and response time). Paper [42] also presents a repartitioning system which migrates vertices on-the-fly based on some runtime statistics (i.e., the average compute and communication time of each superstep, and the probability of a vertex becoming active in the next superstep). Because of this, they are tightly coupled with their systems. In contrast, we encode such workload characteristics into the vertex and edge weights, avoiding coupling our proposed solutions with a specific graph computing engine.

Vertex-Cut Graph Partitioning Several vertex-cut graph partitioners [43, 44, 3, 45] were also proposed to improve the performance of distributed graph computation on power-law graphs. In this case, edges are mapped to partitions and vertices are cut if their edges happen to be assigned to different partitions. Although they belong to a different type of graph partitioners, they all have to face the heterogeneity and contention issue as edgecut-based solutions. Work [45] is a first vertex-cut attempt to address the heterogeneity issue.

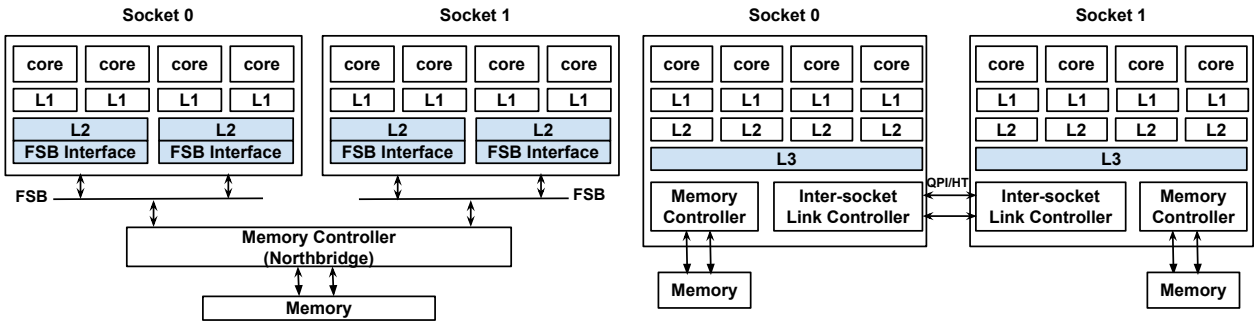
State-of-The-Art Graph (Re)Partitioners We summarize the state-of-the-art edgecut graph (re)partitioners in Table 2.1, according to three dimensions: the supported graph properties, architecture-awareness, and algorithmic properties. In terms of *graph properties (GP)*, we characterize each approach as to whether it can handle graphs with (a) dynamism, (b) weighted vertices (i.e., non-uniform computation), (c) weighted edges (i.e., non-uniform data communication), and (d) vertex sizes (i.e., non-uniform data sizes on each vertex). In terms of *architecture-awareness (AA)*, we distinguish three aspects: (a) CPU heterogeneity, (b) network heterogeneity, and (c) resource contention. Lastly, in terms of *algorithmic properties (AP)*, we characterize each approach as to whether it (a) runs in parallel, and (b) is lightweight. The reason why we omit the comparison of vertex-cut solutions is because the focus of this thesis is edgecut-based solutions.

Table 2.1: State-of-the-art Graph (Re)Partitioners

Graph RePartitioners	Algorithm Properties					Graph Properties			
	Parallel	Lightweight	Architecture-Aware			Dynamism	Weighted		Vertex Size
			CPU	Network	Contention		Vertex	Edge	
Graph Partitioners									
METIS [32]							✓	✓	
Chaco [35]							✓	✓	
ICA3PP'08 [11]			✓	✓			✓	✓	
SoCC'12 [12]				✓			✓	✓	
LDG [8]		✓				Limited			
Fennel [10]		✓				Limited			
arXiv'13 [37]		✓				✓			
TKDE'15 [13]		✓	✓	✓		Limited	✓	✓	
Sheep [38]	✓	✓							
ARGO [20]		✓		✓	✓	Limited	✓	✓	
Graph Repartitioners									
PARMETIS [33]	✓					✓	✓	✓	✓
ZOLTAN [36]	✓					✓	✓	✓	✓
Scotch [34]						✓	✓	✓	✓
CatchW [40]	✓	✓				✓	✓	✓	
xdgp [39]	✓	✓				✓			
Hermes [41]	✓	✓				✓	✓		
Mizan [26]	✓	✓				✓	✓	✓	
LogGP [42]	✓	✓				✓	✓	✓	
ARAGON [21]				✓		✓	✓	✓	✓
PARAGON [22]	✓			✓	✓	✓	✓	✓	✓
PLANAR [23]	✓	✓		✓	✓	✓	✓	✓	✓
PLANAR+ [24]									

2.2 IMPORTANCE OF ARCHITECTURE-AWARENESS

In this section, we first describe two important factors that one should consider while partitioning the graphs for efficient distributed graph computation on modern HPC clusters (Section 2.2.1 & 2.2.2), followed by an experimental demonstration of their performance impact on three representative distributed graph workloads (Section 2.2.3).



(a) Uniform Memory Access (UMA) Node (b) Non-uniform Memory Access (NUMA) Node

Figure 2.1: Example architectures of modern compute nodes

2.2.1 Network Characteristics of Modern HPC Infrastructures

For distributed graph computations on multicore systems, communication can be either *inter-node* (i.e., among cores of different compute nodes) or *intra-node* (i.e., among cores of the same compute node). In general, intra-node communication is an order of magnitude faster than inter-node communication. This is because in many modern parallel programming models like MPI [46, 47], a predominant messaging standard for HPC applications, intra-node communication is implemented via shared memory/cache [48, 49], while inter-node communication needs to go through the network interface. Additionally, both inter-node and intra-node communication are themselves non-uniform.

Non-uniform Inter-Node Network Communication Modern parallel architectures, like supercomputers, usually consist of a large number of compute nodes linked via a network. Consequently, the communication costs among compute nodes vary a lot because of their varying locations. For example, in the Gordon supercomputer [50], the network topology is a 4x4x4 3D torus of switches with 16 compute nodes attached to each switch. As a result, the distance to different compute nodes starting from a single node varies from 0 to 6 hops. Also, supercomputers often allow multiple jobs to concurrently run on different compute nodes and contend for the shared network links, limiting the effective network bandwidth available for each job and thus amplifying the heterogeneity.

Non-uniform Intra-Node Network Communication Communication among cores of

the same compute node is also non-uniform because of the complex memory hierarchy. Communication among cores sharing more cache-levels can achieve lower latency and higher effective bandwidth than cores sharing fewer cache-levels. For example, in the architecture described by Figure 2.1a, communication among cores sharing L2 caches (e.g., between the first and second core of Socket 0) offers the highest performance, while communication among cores of the same socket but not sharing any L2 cache (e.g., between the first and third core of Socket 0) delivers the next highest performance. Communication among cores of different sockets performs the worst. Similarly, in Figure 2.1b, cores of the same socket (intra-socket communication) usually communicate faster than cores residing on different sockets (inter-socket communication). This is because intra-socket communication can be achieved via the shared caches, while inter-socket communication has to go through the front-side bus and the off-chip memory controller (Figure 2.1a) or the inter-socket link controller (Figure 2.1b).

Take-away *To improve the performance of graph-based big-data applications, we should not only minimize the number of edges across different partitions (edgecut), but also the number of edges connecting partitions having higher network communication costs (hopcut).* This is the major difference between architecture-agnostic solutions (that only minimize edgecut) and architecture-aware ones (that try to minimize both edgecut and hopcut).

2.2.2 Resource Contention on HPC Memory Subsystems

Clearly, it is critical to make the graph partitioning or repartitioning procedure aware of the non-uniform network communication costs in cases where the network is the bottleneck. Nevertheless, network nowadays may no longer be the bottleneck due to the presence of *remote direct memory access* (RDMA) technology [16]. RDMA-enabled networks allow a compute node to read/write data from/to the memory of another compute node without involving the processor, cache, or operating system of either node, enabling true zero-copy data communication [51] (Figure 2.3a). Besides, the bandwidth of modern RDMA-enabled networks has been reported to be in the same ballpark as memory bandwidth [16]. As shown in Figure 2.2, 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

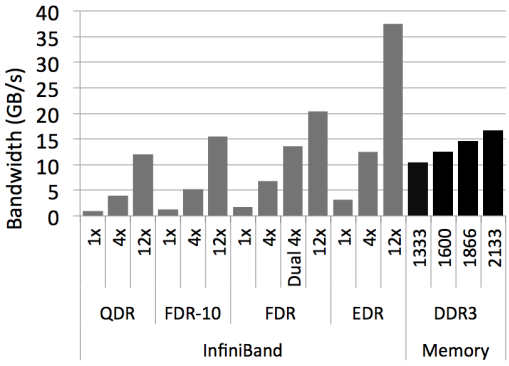
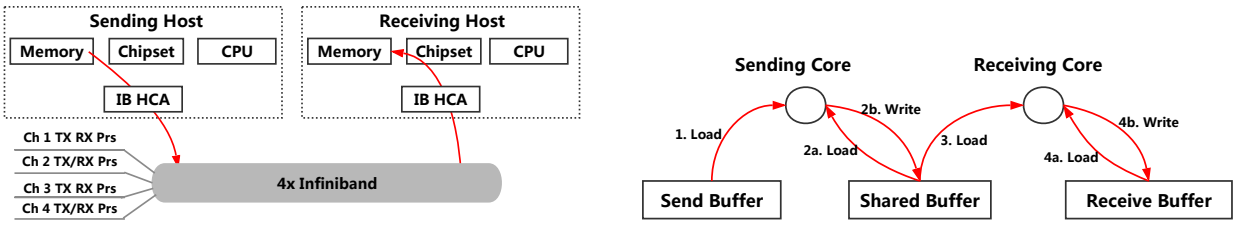


Figure 2.2: Theoretic bandwidth for different generations of InfiniBand and memory technologies [16].

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. As a result, the contention for the shared resources on the memory subsystem of modern multicore machines is becoming more and more noticeable.

Inherent Contention in Multicore Machines 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 2.1a, cores sharing the L2 caches have to compete 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 2.2 provides a concise summary for the resources that different cores may have to contend for, in the *Uniform Memory Access* (UMA) architecture of Figure 2.1a and the *Non-Uniform Memory Access* (NUMA) architecture of Figure 2.1b. The summary is based on whether the cores are on the same socket and whether they share the *last level cache* (LLC).

Contention and Intra-Node Data Communication The fact that intra-node data communication is often achieved via shared memory further amplifies the contention, because



(a) Inter-node data communication via RDMA [51]

(b) Intra-node data communication via shared memory

Figure 2.3: Memory transactions of inter- and intra-node data communication

Table 2.2: Intra-node shared resource contention

Cores/Resources		Sharing		Contention		
Core Groups		Socket	LLC	LLC	FSB/QPI(HT)	Memory Controller
UMA Fig. 2.1a	G1	✓	✓	✓	✓	✓
	G2	✓			✓	✓
	G3					✓
NUMA Fig. 2.1b	G1	✓	✓	✓		✓
	G2				✓	

intra-node data communication requires additional data copies [48, 49], which, in turn, may lead to significant cache pollution and thus saturate the memory controller. Figure 2.3b 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 sending buffer into its cache (Step 1 in Figure 2.3b) 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 sending 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 sending or receiving buffer). Clearly, intra-node data communication may lead to serious cache pollution and therefore saturate the memory controller.

What is even worse is that cores of the same machine are often communicating with each other at the same time for parallel computation and the number of cores per node is continuously increasing. The fact that graph workloads often have poor locality [52] (because of the irregular and unstructured nature of real-world graphs) and high memory access to computation ratio [52] (since graph algorithms are often based on the exploration of the graph structure with little computation work per vertex) further aggravates the contention.

Take-Away *Focusing solely on minimizing the edgecut or the hopcut may not be sufficient for scalable performance. This is because edgecut-based solutions have no guarantee on how the edgecut is distributed across the 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, hopcut-based solutions advocate for grouping neighboring vertices as close as possible, further aggravating the contention on the memory subsystems.*

2.2.3 Understanding the Performance Impact of Heterogeneity and Contentiousness

In this section, we experimentally demonstrate and quantify the performance impact of architecture-awareness (especially contentiousness) on distributed graph computing using four graph partitioners: (a) METIS, the most well-known graph partitioner [32], (b) LDG, the most well-known streaming graph partitioner [8], (c) ARGO, an architecture-aware graph partitioner presented in Chapter 3, and (d) ARGO-H, a variant of ARGO that only considers the communication heterogeneity. The demonstration was achieved by comparing runs of an MPI implementation of three classic graph workloads: PageRank, Breadth-First Search (BFS), and Single-Source Shortest Path (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

Table 2.3: Workload execution time in seconds on com-orkut dataset

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

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. More details about the evaluation platform are presented in Section 3.2.2.

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 com-orkut dataset was partitioned into 16 partitions across corresponding cores (one partition per core) using METIS, LDG, ARGO, and ARGO-H. Orkut is a social network ran by Google for people across the world to discuss their common interests [53]. The dataset used is a subset of the Orkut user population (around 11.3% at the time crawled by A. Mislove et. al. [54]). The dataset has around 3M vertices and 234M edges. The degree distribution of the dataset follows the power-law distribution with average and maximal vertex degree equal 76.281 and 33,313, respectively. The maximal diameter of the dataset is 10 with the effective diameter of 5.4489.

Results in terms of execution time (Table 2.3) Table 2.3 shows the resulting execution time of the workloads under different configurations on the com-orkut dataset. As expected, the higher the contention, 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. We believe the reason why the execution of BFS under configuration *2:2:4* sometimes took longer than that of configuration *1:2:8* was 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*. We believe this was because the partitionings computed by METIS had the lowest edgecut and thus the 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 hopcut-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 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 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.

Results in terms of LLC misses (Table 2.4) 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 2.4. The LLC misses were collected via the PAPI_L3_TCM event provided by the hardware performance counter programming tool,

Table 2.4: Workload LLC misses in millions on com-orkut dataset

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

PAPI [55], and the values reported were the average LLC misses across partitions (MPI processes). By comparing Tables 2.4 and 2.3, 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 the execution time of the workload. 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 observation that under configuration *8:2:1* reducing inter-machine data communication was more critical to the performance than mitigating contention on the memory subsystems (e.g., cache pollution caused by inter-socket data communication) for BFS and SSSP.

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.*

3.0 ARCHITECTURE-AWARE STATIC GRAPH PARTITIONING

In this chapter, we first formally define the problem of architecture-aware graph partitioning for static graphs (Section 3.1). Then, we introduce a streaming-based implementation of such a graph partitioner, ARGO (Section 3.2).

3.1 PROBLEM STATEMENT

Let $G = (V, E)$ be a graph, where V is the vertex set and E is the edge set.

Workload-Awareness To make the partitioning aware of the runtime characteristics of the target workload, we assign each vertex and edge a weight for partitioning. *Vertex weight*, $w(v)$, indicates the computation requirement of vertex v, while *edge weight* $w(e)$ reflects the amount of the data communicated along edge e during the computation. Since the encoding of such information to the vertex and edge weights is fairly straightforward, we will primarily focus on the discussion of how to make the partitioning algorithm architecture-aware here.

Architecture-aware graph partitioning aims to partition the graph into n *balanced* partitions:

$$P = \{P_i : \cup_{i=1}^{n} P_i = V \text{ and } P_i \cap P_j = \phi \text{ for any } i \neq j\} \tag{3.1}$$

such that *the communication cost of the partitioning is minimized*.

We define the *communication cost* of a partitioning P as:

$$comm(G, P) = \sum_{\substack{e=(u,v)\in E \\ \text{and } u\in P_i \text{ and } v\in P_j \text{ and } i\neq j}} w(e) * c(P_i, P_j) \tag{3.2}$$

where $c(P_i, P_j)$ can be either the relative network communication cost, the degree of shared resource contentiousness between P_i and P_j or a hybrid of both. Existing architecture-agnostic graph partitioners usually assume $c(P_i, P_j) = 1$, which fails to reflect the characteristics of modern HPC infrastructures. Thus, to minimize $comm(G, P)$, we should avoid data communication among partition pairs having high $c(P_i, P_j)$ as much as possible.

We say a partitioning is *balanced* if the skewness of the partitioning is within a user-defined threshold. The *skewness* of a partitioning P is defined as:

$$skewness(G, P) = \frac{\max\{w(P_1), w(P_2), \cdots, w(P_n)\}}{\frac{\sum_{i=1}^{n} w(P_i)}{n}} \tag{3.3}$$

where $w(P_i) = \sum_{v \in P_i} w(v)$.

Assumptions Throughout this chapter, we assume that (a) P_i is assigned to server M_i for parallel processing; (b) P_i and M_i are used interchangeably; and (c) the server can be either a hardware thread, a core, a socket, or a machine. By default, the servers are cores since we target for clusters of multicore machines.

3.2 ARGO: ARCHITECTURE-AWARE GRAPH PARTITIONING

In this section, we introduce ARGO, an architecture-aware graph partitioner we implemented for static graph partitioning. ARGO is short for Architecture-Aware Graph PartitiOning.

3.2.1 Algorithm Design and Implementation

3.2.1.1 Graph Partitioning Model ARGO follows the same streaming partitioning model first proposed by [8]. 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 [8] 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:

$$(1 - \frac{w(P_i)}{C(P_i)}) * \sum_{e=(u,v)\in E \text{ and } u\in P_i} w(e) \tag{3.4}$$

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 along the edge). 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.

3.2.1.2 Incorporating Heterogeneity Awareness

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

$$(1 - \frac{w(P_i)}{C(P_i)}) * \frac{1}{comm(v, P_i) + 1} \tag{3.5}$$

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

$$comm(v, P_i) = \sum_{e=(u,v)\in E \text{ and } u\in P_j \text{ and } i\neq j} w(e) * c(P_i, P_j) \tag{3.6}$$

Thus, if $c(P_i, P_j)$ represents the relative network communication cost between P_i and P_j, $comm(v, P_i)$ defines the communication cost that v would incur during the computation if it is assigned to P_i. As a result, the above heuristic will put neighboring vertices to partitions as close as possible according to the relative network communication cost matrix. We denote this version of ARGO as ARGO-H, since it only considers the heterogeneity of the network communication costs while ignoring the contentiousness of the underlying computing infrastructures.

3.2.1.3 Incorporating Contention Awareness

As analyzed and demonstrated in Section 2.2, edgecut (e.g., LDG) and hopcut (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 excess data communication among cores of the same node and can be avoided by offloading a certain amount of intra-node data communication across compute nodes. This is because inter-node data communication is often implemented using RDMA and rendezvous protocols [56], which allows a compute node to read/write data from/to the memory of another compute node without involving the processor, cache, or operating system of either node (Figure 2.3a), thus alleviating the traffic on memory subsystems and cache pollution. In fact, with Intel Data Direct I/O technology [57], it is even possible to transfer data from one machine directly 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 [52].

Recall that guided by a relative network communication cost matrix, ARGO-H can gather neighboring vertices close to each other (Eq. 3.5), which causes contention on the memory subsystems due to the excess intra-node data communication. Thus, to make ARGO contention-aware, we penalize intra-node network communication costs via a penalty score. 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. Specifically, we refine the intra-node network communication costs as follows:

$$c(P_i, P_j) = c(P_i, P_j) + \lambda * (s_1 + s_2) \tag{3.7}$$

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. 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 infrastructures to determine the ideal λ for them. Typically, for multicore clusters with high-speed network, a larger λ is recommended, and vice-versa.

3.2.2 Evaluation

3.2.2.1 Setup In our experimental study, we first evaluated the effectiveness of Argo in avoiding contention using three representative graph workloads: Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), and PageRank (Section 3.2.2.2). Then, we examined the scalability of Argo in terms of both graph size and the number of partitions (Section 3.2.2.3 & 3.2.2.4).

Workload Implementation All the workloads were implemented using MPI [58] based on the idea presented in [59, 60]. The specific MPI implementation we used in the experiment was OpenMPI 1.8.6 [46]. Note that the workloads were implemented using MPI_Isend and MPI_Irecv functions.

Algorithms We compared Argo to three graph partitioners: (a) Metis, the most well-known multi-level graph partitioner [32], (b) LDG, a state-of-the-art streaming graph partitioner [8], and (c) Argo-H, a variant of Argo that only considers the communication heterogeneity.

Datasets Table 3.1 describes the datasets used. com-orkut and Friendster datasets were undirected, whereas the original Twitter dataset was directed but was treated as an undi-

Table 3.1: Datasets used in our experiments

Dataset	$\|V\|$	$\|E\|$	Description
com-orkut [61]	3,072,627	234,370,166	Social Network
Friendster [61]	124,836,180	3,612,134,270	Social Network
Twitter [62]	52,579,682	3,926,527,016	Social Network

Table 3.2: 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

rected 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 follow a power law distribution [63, 64], whereas small-world graphs are known to have low diameters.

Throughout our experimental study, the graphs were partitioned with the vertex weights (i.e., computational requirement) set to their vertex degree and the edge weights (i.e., amount of data communicated) set to 1. The 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 the partitions. Note that for the streaming graph partitioners: LDG, Argo, and Argo-H, the vertices of the graphs were presented to the partitioner in the BFS order [8].

Evaluation Platform All the experiments were performed on a 32-node university cluster [65]. The cluster had a flat network topology with all the compute nodes connected to a single switch via 56Gbps FDR Infiniband. Table 3.2 depicts the compute node configuration of the cluster.

Network Communication Cost Modeling The relative network communication costs among the partitions were approximated using a variant of the osu_latency benchmark [66].

To ensure the accuracy of the cost matrix, we bound each MPI rank (process) to a core using the options provided by OpenMPI 1.8.6 [46]. OpenMPI 1.8.6 is the specific MPI implementation that was available on the cluster.

3.2.2.2 Effectiveness of Being Architecture-Aware

Configuration 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 2.2.3, the contention on the memory subsystems on the cluster was the primary bottleneck. Hence, we set λ (from Eq. 3.7) to 1 for all the experiments presented below.

Table 3.3: Workload execution time in seconds 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

Results in terms of Execution Time (Table 3.3) Table 3.3 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. During the execution of BFS, SSSP, and PageRank, we grouped multiple messages sent by each MPI rank to the same destination into a single one. As expected, Argo had the lowest workload execution time in all the 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 the cases. This was also expected because Argo-H aimed to group 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 gradually closed up. This was because, the larger the message grouping size was, the fewer 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.

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.*

Table 3.4: Workload LLC misses in millions on com-orkut dataset with varying message grouping size

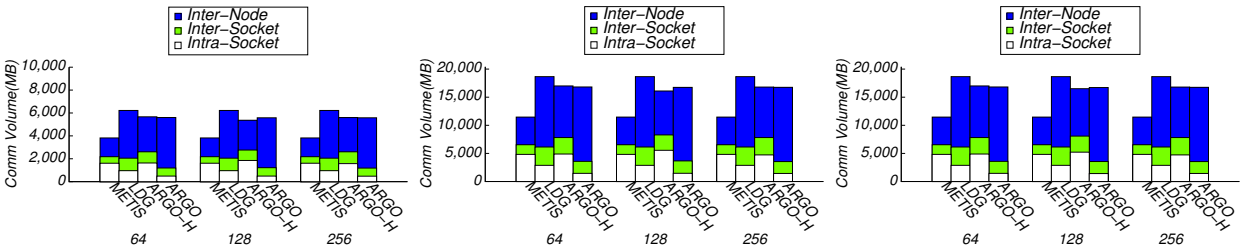
Configuration	BFS (10 Source Vertices)			SSSP (10 Source Vertices)			PageRank (30 Iterations)		
	64	128	256	64	128	256	64	128	256
METIS	843	50	17	38,942	6,313	471	10,605	529	22
LDG	194	27	22	30,096	1456	59	4,605	69	43
ARGO-H	1,702	36	22	51,774	8,173	589	17,360	748	35
ARGO	35	26	21	8,702	163	49	142	49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

Results in terms of LLC Misses (Table 3.4) 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 3.4. As shown, the LLC miss results were highly consistent with the timing results: (1) ARGO had the lowest LLC misses in almost all the 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 3.3). 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 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.*



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

Figure 3.1: Breakdown communication volume for the execution of BFS, SSSP, and PageRank on com-orkut partitionings.

Results in terms of Communication Volume (Figure 3.1) 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 3.1. 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.

As shown, Argo had the lowest intra-node data communication in all the 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 miss 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.*

Table 3.5: Workload 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

3.2.2.3 Scalability in terms of Graph Size

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 (even the smallest size graphs of this experiment, i.e., 09 billion and 0.98 billion edges).

Results (Table 3.5) Table 3.5 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 the cases, whereas Argo-H was always the worst. Compared to LDG, Argo achieved by up 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 3.6: Workload execution time in seconds as the # of partitions increased

Number of Partitions (cores)	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

3.2.2.4 Scalability in terms of Number of Partitions

Configuration This experiment inspected the effectiveness of Argo as the number of partitions increased. Towards this, we partitioned the original Friendster and Twitter datasets 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 3.6) Table 3.6 presents the corresponding results. As expected, Argo performed the best in all the 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, the improvement was still non-negligible, since ARGO reduced the execution time of each core used by this much.

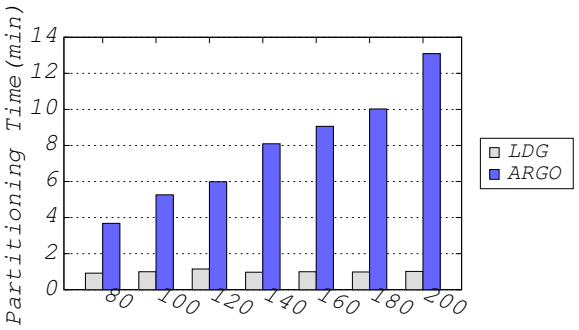


Figure 3.2: Partitioning time on Twitter dataset

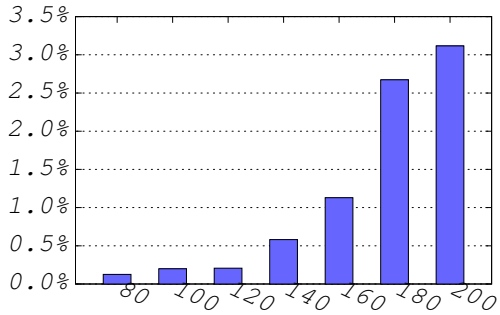


Figure 3.3: ARGO partitioning time as a percentage of CPU time saving

Results in terms of Partitioning Overhead (Figures 3.2 & 3.3) We also reported the partitioning overhead (vertex placement decision time) of ARGO in Figure 3.2. The main reason for the extra overhead was because 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. Also, the time complexity of ARGO was $O(|E| + |V| * k^2)$ whereas that of LDG was $O(|E| + |V| * k)$. Here, $|V|, |E|$, and k, respectively, denote the number of number of vertices of the graph, the number of edges of the graph, and the number

of partitions. The reason why the overhead of LDG remained quite stable was because the overhead was dominated by the iterating of each vertex's neighbors. However, if we compare the partitioning time to the CPU time saved by Argo against LDG, the overhead was negligible. The CPU time saving was the reduction in the workload execution time multiplied by the number of CPU cores used (e.g., Figure 3.3 shows the partitioning time of Argo as a percentage of the CPU time saved by Argo for the execution of SSSP with 5 randomly selected vertices on Twitter dataset with different number of partitions). Besides, the partitioning only has to be performed once and can be used multiple times. Also, graph analytics 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.

3.3 CHAPTER SUMMARY

In this chapter, we first defined the problem of architecture-aware graph partitioning and then presented a streaming-based implementation of such partitioner, Argo. In addition to being aware of the the contentiousness of the memory subsystems and the heterogeneity in the network communication costs, it also considers the runtime characteristics of the target workload 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 (up to 3.9 billion edges) and the number of partitions (up to 200 partitions).

4.0 ARCHITECTURE-AWARE DYNAMIC GRAPH PARTITIONING

In our previous chapters, we have demonstrated the importance of architecture-awareness (Chapter 2) and presented an architecture-aware graph partitioner, Argo (Chapter 3), for static graph partitioning. Although the partitioning model adopted by Argo can handle a certain type of dynamism (vertex/edge addition/deletion), it may lead to suboptimal performance in the presence of graph dynamism and it is incapable of dealing with other types of dynamism. In other words, to maintain the performance, the graph has to be repartitioned periodically. Towards this, we will first define the problem of architecture-aware dynamic graph partitioning in Section 4.1. Dynamic graph partitioning is also known as the graph repartitioning problem. Then, we will introduce four architecture-aware graph repartitioners we proposed: Aragon (Section 4.2), Paragon (Section 4.3), and Planar/Planar+ (Section 4.4). For the presentation of Aragon, Paragon, Planar, and Planar+, we will primarily focus on explaining how we make them heterogeneity-aware, since we can easily make them contention-aware by penalizing the intra-node data communication costs in the same way as Argo does.

4.1 PROBLEM STATEMENT

Let $G = (V, E)$ be a graph, where V is the vertex set and E is the edge set, and P be a partitioning of G with n partitions, where

$$P = \{P_i : \cup_{i=1}^{n} P_i = V \text{ and } P_i \cap P_j = \phi \text{ for any } i \neq j\} \tag{4.1}$$

and M be the current assignment of the partitions to the servers, where P_i is assigned to server M_i. Throughout this chapter, P_i and M_i are used interchangeably. The server that each partition is assigned to can be either a hardware thread, a core, a socket, or a machine. By default, the servers are cores since we target for clusters of multicore machines. Here, we assume that this is also the set of servers used for repartitioning.

Workload-awareness To make the repartitioning workload-aware, we allow each vertex of the graph to be assigned a weight and size. *Vertex weight*, $w(v)$, denotes the computation requirement of v, whereas *vertex size*, $vs(v)$, indicates the amount of application data represented by v. Each edge of the graph can also be assigned a weight for repartitioning. *Edge weight*, $w(e)$, reflects the amount of data communicated along the edge in each computation superstep. Again, given the easiness of encoding such information into the graph, we will focus on the discussion of how to make the repartitioning architecture-aware here.

Architecture-aware graph repartitioning aims to improve the mapping of the application communication pattern to the underlying hardware topology by modifying the current partitioning of the graph, such that the communication cost of the target application, given the specific hardware topology, is minimized. The modification usually involves migrating vertices from one partition to another partition. Hence, in addition to the communication cost, the repartitioning should also minimize the data migration cost among the partitions. Also, to ensure balanced load distribution in terms of the computation requirement, the refinement/repartitioning should keep the skewness of the partitioning as small as possible.

We define the *communication cost* of a partitioning P as:

$$comm(G, P) = \alpha * \sum_{\substack{e=(u,v)\in E \\ \text{and } u\in P_i \text{ and } v\in P_j \text{ and } i\neq j}} w(e) * c(P_i, P_j) \tag{4.2}$$

where α specifies the relative importance between communication and migration cost, which is usually set to be the number of supersteps carried out between two consecutive refinement/repartitioning steps, and $c(P_i, P_j)$ can be either the relative network communication cost, the degree of shared resource contentiousness between P_i and P_j or a hybrid of both. Existing architecture-agnostic graph (re)partitioners usually assume $c(P_i, P_j) = 1$, which fails to reflect the reality of modern computing infrastructures. Thus, to minimize $comm(G, P)$,

we should avoid data communication among partition pairs having high $c(P_i, P_j)$ as much as possible.

The *migration cost* of the refinement is defined as:

$$mig(G, P, P') = \sum_{\substack{v \in V \\ \text{and } v \in P_i \text{ and } v \in P'_j \text{ and } i \neq j}} vs(v) * c(P_i, P'_j) \tag{4.3}$$

where P' denotes the partitioning after being refined/repartitioned. Note that $c(P_i, P'_j)$ refers only to the relative network communication costs between P_i and P_j. Similarly, to keep $mig(G, P, P')$ minimized, we should avoid migrating both (a) vertices having large neighborhoods or application state and (b) the migration among partitions having high network communication costs. Generally speaking, communication cost is more important than migration cost, since data communication occurs in every superstep, whereas migration is performed only once at the end of each repartitioning phase.

The *skewness* of a partitioning, P, is defined as:

$$skewness(G, P) = \frac{\max\{w(P_1), w(P_2), \cdots, w(P_n)\}}{\frac{\sum_{i=1}^{n} w(P_i)}{n}} \tag{4.4}$$

where $w(P_i) = \sum_{v \in P_i} w(v)$.

4.2 ARAGON: ARCHITECTURE-AWARE GRAPH REPARTITIONING

In this section, we introduce our centralized architecture-aware graph repartitioner, ARAGON, for small dynamic graph partitioning.

4.2.1 Algorithm Design and Implementation

ARAGON is a two-level hierarchical repartitioner, that performs *inter-node* repartitioning (Section 4.2.1.1) and *intra-node* repartitioning (Section 4.2.1.2). The goal of *inter-node repartitioning* is to rebalance the load across compute nodes while minimizing the inter-node communication and migration costs. The latter is achieved by minimizing the number of hops each data item needs to traverse, by grouping together vertices that communicate a lot. In contrast, *intra-node repartitioning* aims to equalize the load assigned to each compute node across its different cores while minimizing the intra-node communication and migration cost, by co-locating vertices communicating a lot to cores sharing more cache levels.

4.2.1.1 Inter-Node Graph Repartitioning Inter-node repartitioning consists of three phases: (1) A *regrouping phase* in which ARAGON regroups partitions currently assigned to the same compute node into a single partition; (2) A *repartitioning phase* where ARAGON repartitions this regrouped graph into balanced parts using existing topology-agnostic graph repartitioners, such as ZOLTAN [36] and PARMETIS [33]; and (3) A *refinement phase* where the decomposition produced by the previous phase is modified according to the current mapping of partitions to compute nodes and the relative inter-node communication costs via a topology-aware refinement algorithm, to further reduce the communication and migration cost. We named this refinement algorithm TopoFM and present it next.

The regrouping and repartitioning phases are straightforward. They take the mapping of vertices to compute nodes (rather than the mapping of vertices to the old partition number) as input. The mapping of partitions to compute nodes (i.e., the mapping of vertices to compute nodes) is readily available in the initial partitioning, which is part of the input to our algorithm.

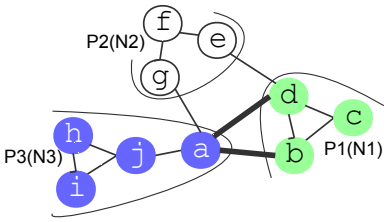


Figure 4.1: Old Decomposition

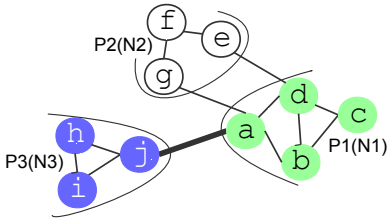


Figure 4.2: Better Decomposition

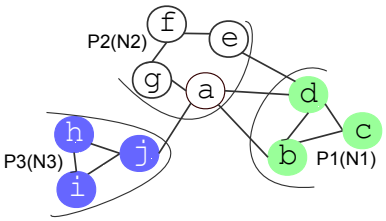


Figure 4.3: Best Decomposition

Table 4.1: Relative network communication costs

	N_1	N_2	N_3
N_1		1	6
N_2	1		1
N_3	6	1	

TopoFM Overview TopoFM is an iterative algorithm and is a variant of the Fiduccia-Mattheyses (FM) algorithm [67]. Its input includes two partitions of the k-way decomposition, the current mapping of partitions to compute nodes, and the relative network communication costs among the compute nodes. During each iteration, TopoFM tries to find a single vertex, v, such that moving it from its current partition to the alternative partition would lead to a maximal gain, $g(v)$. The gain is defined as the reduction in the communication and migration cost. This process is repeated until all the vertices are moved once or the decomposition cannot be further improved after a certain number of vertex movements. Since TopoFM can only refine one partition pair at a time, it is repeatedly applied to all the partition pairs sequentially.

Motivating Example Before we dive into the details of TopoFM, we first go through a simple motivating example. Let us assume that the graph in Figure 4.1 captures the computation and communication pattern of an application. For simplicity, we assume that all weights and sizes of the graph are 1. Originally, the graph is partitioned into 3 partitions, and partition P_i is assigned to compute node N_i for the parallel execution of the applica-

tion. Vertices of the same color belong to the same partition, whereas the relative network communication costs among N_1, N_2, and N_3 are shown in Table 4.1.

A topology-agnostic repartitioner (i.e., assuming uniform network communication costs) could repartition the decomposition of Figure 4.1 into the one of Figure 4.2, reducing the number of edges among the partitions from 4 to 3. However, if we consider the case where all network costs are not equal, i.e., we want to make our repartitioner *architecture- and topology-aware* (e.g., using the communication costs from Table 4.1), then the decomposition in Figure 4.2 can be further improved by moving vertex a to P_2 (Figure 4.3). Even though the movement increases the communication cost between P_1 and P_2 by 1, it actually reduces the communication cost between a and its neighbors in P_3 by 5, since the network cost between N_1 and N_3 is 6, while that of N_2 and N_3 is 1. For the same reason, moving a to P_2 also decreases the migration cost of a by 5, since vertex a originally belonged to N_1.

Architecture-Aware Vertex Gain Computation Motivated by the example above, for the vertex gain computation, we first focus on how the movement of vertex v will impact the communication between v's current partition and the refinement partner. For notation simplicity, let P_i and P_j be the two partitions of the k-way decomposition of graph $G = (V, E)$ we want to refine, and N_i and N_j be the compute nodes that hold P_i and P_j, respectively, and P_i be the partition that v currently belongs to. We define the gain of moving v in terms of its impact on the communication between P_i and P_j as:

$$g_{std}(v) = \alpha * (d^j_{ext}(v) - d^i_{int}(v)) * d(N_i, N_j) \tag{4.5}$$

Here, $d(N_i, N_j)$ is the relative network cost between N_i and N_j, whereas $d^j_{ext}(v)$ is the relative external communication volume of v with respect to P_j, formally defined as

$$d^j_{ext}(v) = \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_j \text{ and } i\neq j} w(e) \tag{4.6}$$

Here, $w(e)$ denotes the edge weight. In contrast, $d^i_{int}(v)$ is the relative internal communication volume of v with respect to P_i, formally defined as

$$d^i_{int}(v) = \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_i} w(e) \tag{4.7}$$

Algorithm 1: TopoFM

```
Data: Two balanced partitions (P_i, P_j), partition assignment A, inter-node communication cost
      matrix c
1  orderedList ← {}
2  unmarkVertices(P_i, P_j)
3  computeInitialGain(P_i, H_1, A, c)
4  computeInitialGain(P_j, H_2, A, c)
5  while exists unmarked vertex and # of useless moves ≤ LIMIT do
6      heap = FMHeapSelection(P_i, P_j, H_1, H_2)
7      v = heapGetMaxGainVertex(heap)
8      mark(v)
9      append(v, orderedList)
10     updateNborGain(P_i, P_j, H_1, H_2, v, A, c)
11 applyMove(P_i, P_j, orderedList)
```

We then consider the impact of moving v from P_i to P_j on the communication between v and its neighbors which do not belong to either P_i or P_j, defined as:

$$g_{topo}(v) = \alpha * \sum_{\substack{e=(v,u)\in E \\ \text{and } v\in P_i \text{ and } u\in P_k \\ \text{and } k\neq i \text{ and } k\neq j}} w(e) * (d(N_i, N_k) - d(N_j, N_k)) \tag{4.8}$$

Here, N_k is the compute node where P_k belongs.

Next, we consider the impact of moving v from P_i to P_j on the migration cost. Let $vs(v)$ be the amount of data vertex v represents and P_k be the partition that contained v in the old decomposition. We formally define the gain of moving v to P_j in terms of its impact on the migration cost as:

$$g_{mig}(v) = vs(v) * (d(N_i, N_k) - d(N_j, N_k)) \tag{4.9}$$

Thus, the total gain of moving vertex v from its current partition to the refinement partner is:

$$g(v) = g_{std}(v) + g_{topo}(v) + g_{mig}(v) \tag{4.10}$$

TopoFM Implementation Algorithm 1 presents the basic idea of TopoFM. The input to TopoFM includes two balanced partitions (P_i, P_j) of the k-way decomposition, the current assignment of the partitions to the compute nodes, A, and the relative inter-node communication costs, c. First, TopoFM unmarks all the vertices of P_i and P_j, indicating that no

vertex has been moved. Second, it computes the initial gain of these vertices and inserts vertices having edges connecting to the other partitions, referred as boundary vertices, into corresponding heaps (one heap per partition), which are sorted by the gain.

Then, the following procedure is repeated until all the vertices are moved once or the communication and migration cost could not be further reduced after a certain number of vertex movements. As a first step, TopoFM attempts to find an umarked vertex v with maximum gain from P_i or P_j. As long as the imbalance between P_i and P_j is within the user-defined threshold (2% by default), TopoFM always selects the max gain vertex from the partition whose max gain vertex has the largest value. Otherwise, it will return the max gain vertex from the overloaded partition. Then, TopoFM marks v as moved, and appends v to the end of an ordered list. Subsequently, TopoFM updates the gain of v's neighbors that are in P_i or P_j as if v was moved. During the update, TopoFM checks whether any boundary vertices are no longer boundary ones. If so, these vertices are removed from the corresponding heaps. TopoFM also checks if any non-boundary vertices become boundary vertices. If so, TopoFM interts them into the corresponding heaps.

Once the procedure terminates, TopoFM finds the best number of moves θ in the ordered list such that $\sum_{i=0}^{\theta} g(v)$ is maximized. Only if the sum is positive will these θ vertices be moved. Otherwise, TopoFM simply terminates.

Moreover, although TopoFM is topology-aware, it is also topology-independent since it only requires that the relative inter-node communication costs are available, and its implementation is similar to that of standard FM algorithms, like [68]. The differences include:

1. In standard FM algorithms, $g(v) = \alpha * (d^j_{ext}(v) - d^i_{int}(v))$, which is unaware of the communication heterogeneity.
2. The refinement between P_i and P_j of standard FM only needs to consider moving boundary vertices of P_j/P_i with respect to P_i/P_j. However, TopoFM needs to consider boundary vertices of P_i/P_j with respect to all the partitions.
3. In standard FM algorithms, refinement only happens between partition pairs that communicate. In contrast, TopoFM needs to refine all the partition pairs even though no

communication occurs between the partition pair.

4. Unlike TopoFM, standard FM algorithms usually select max gain vertices alternatively from P_i and P_j. Thus, our maximal gain vertex selection policy has a greater potential to improve the decomposition while satisfying the balance requirement, and speed up the convergence of a good decomposition.

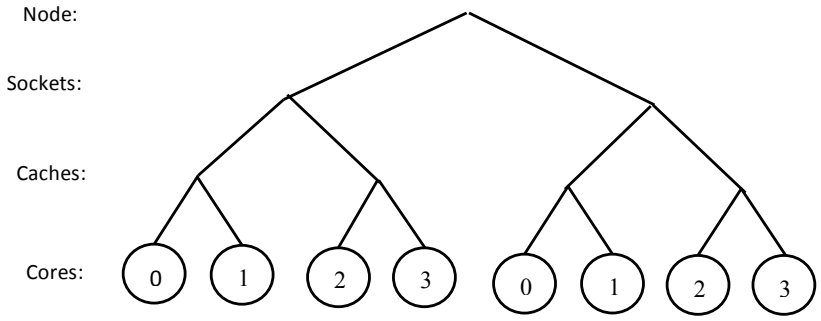


Figure 4.4: Topology Tree

4.2.1.2 Intra-Node Graph Repartitioning Partitions computed by the inter-node repartitioning are treated as individual subgraphs, each of which requires one round of parallel intra-node repartitioning. For each such round, we offer two repartitioners: *FlatCacheLB* and *HierCacheLB*, both of which try to equalize the load assigned to each compute node across its cores, while minimizing the communication and migration cost (by co-locating frequently communicating vertices to cores sharing more cache levels).

HierCacheLB HierCacheLB first models the topology of each compute node as a tree like [69] does. For example, the tree in Figure 4.4 denotes a compute node with two quad-core sockets, where two cores in the same socket share a cache, like the one in Figure 2.1a. Then, HierCacheLB partitions the subgraph assigned to the compute node hierarchically according to the tree. This automatically minimizes the communication volume across tree nodes at each level. At the end of each level's partitioning, HierCacheLB remaps the new decomposition to the old one to maximize the amount of data in place.

For instance, assume that we want to repartition a subgraph assigned to a compute node modeled by Figure 4.4. HierCacheLB will first partition the subgraph into two balanced partitions while minimizing the edgecut, which approximates equalizing the load across the

sockets while minimizing the inter-socket communication cost. Then, HierCacheLB remaps these two partitions to the old decomposition to minimize the migration cost. This step is recursively applied to the next level until it reaches the leaf level.

The remapping phase can be done in an efficient way using the Hungarian algorithm [70], because the topology tree is small. The algorithm takes as input a cost matrix, M, where $M[i][j]$ denotes the migration cost of assigning partition i of the subgraph to the jth socket/cache/core of the compute node. Along with this input, the Hungarian algorithm will output an assignment of the partitions to the sockets/caches/cores of the node with minimal migration cost.

Since each process may monopolize a vertex portion of each subgraph due to the parallel inter-node repartitioning, to compute $M[i][j]$ all processes need to iterate over its vertex portion of partition i to see if the socket/cache/core originally owning the vertex is the jth one. If not, the assignment will lead to a migration cost of moving the vertex from its original socket/cache/core to the jth one. The migrating cost of a vertex is the vertex size. Then, an MPI reduce operation is performed to aggregate the result.

FlatCacheLB Unlike HierCacheLB, FlatCacheLB first partitions the subgraph assigned to each node directly into the corresponding number of partitions. Then, it explores all possible assignments of the partitions to the cores of each compute node to find the one with minimal cost. The exploration phase takes two cost matrices M and C as its input. As with HierCacheLB, $M[i][j]$ denotes the migration volume of assigning partition i of the subgraph to core j of the node. $C[i][j]$ reflects the communication volume between partition i and j, defined as the aggregated weights of edges crossing partition i and j. The computation of $C[i][j]$ is similar to that of $M[i][j]$ except that we are visiting edges of the subgraph now. The cost of an assignment, A, where partition i is assigned to core $A[i]$ of the compute node, is defined as:

$$\sum_{i=1}^{n} M[i][A[i]] * 2 * D_{L_n} + \alpha * \sum_{i=1}^{n} \sum_{j=i+1}^{n} C[i][j] * c(A[i], A[j]) \tag{4.11}$$

where n is the number of subgraph partitions, while $c(A[i], A[j])$ is the communication cost of $64B$ data within a compute node ($64B$ is the typical cache line size), which is approximated by the access latency to the first cache level shared by core $A[i]$ and $A[j]$. The inter-socket

communication cost and the intra-node migration cost of $64B$ data within a compute node are both approximated by 2 times of the access latency to the highest cache level. Although FlatCacheLB needs to explore all possible combinations to figure out the optimal assignment, this is feasible since in practice each compute node only has dozens of cores at most.

4.2.2 Evaluation

In section, we first describe our experimental setup, followed by an evaluation of ARAGON in improving the quality of the partitionings using a dataset generated from a scientific simulation.

Table 4.2: Original combustion simulation dataset

$\|V\|$	$\|E\|$	Vertex Degree		
		Min	Max	Avg.
115, 351	1, 432, 950	7	26	24

Table 4.3: Synthetic datasets

Graph	Num. of Partitions	Degree of Imbalance (Eq. 4.4)
G8	8	2.51
G64	64	2.81
G128	128	2.82
G256	256	2.85
G512	512	2.98

4.2.2.1 Setup

Datasets For our experimental study, we used data provided by the authors of [71]. The dataset (Table 4.2) is a 26-degree mesh, which models the computation and communication pattern of the large eddy simulation (LES) of Sandia Flame D [72]. Out of this dataset, we constructed 5 synthetic graphs (Table 4.3) in order to evaluate a range of workloads. For each new graph, we first randomized its edge weights to between 20% and 50% of the sum of the pairwise vertex sizes, and then partitioned the graph into balanced parts using the

partitioner from [36]. Later, 20% of the partitions are selected and the weights and sizes of vertices in these partitions are randomly increased to between 1.5 and 7.5 times of their original values, to simulate load fluctuations of the simulation.

Platform The evaluation was performed with a simulated supercomputer, whose compute nodes are interconnected by a 3D-torus interconnect (5*5*5 by default). Each compute node has 2 quad-core sockets with shared L3 caches and private L1/L2 caches. Partitions of each graph are mapped to cores of the supercomputer at the beginning of each experiment as follows. We first sort the randomly allocated compute nodes by their x coordinates, then by their y coordinates and finally by their z coordinates. Then, we either map partitions of each graph to cores sequentially starting from cores of the first compute node as in the SMP policy [73] or place sequential partitions to the compute node next in the list following the RR policy [73]. In the end, the partitioned graph along with the mapping of partitions to cores and the relative inter-node communication costs serves as the input to repartitioners. The relative inter-node communication costs were estimated by the number of hops (the Manhattan Distance) among compute nodes. We need to clarify here that we do not aim to simulate a full-featured supercomputer. Instead, we just want to evaluate the impact of inter- and intra-node topology on graph repartitioners.

Table 4.4: Four flavors of Aragon

Aragon	InterNode Repartitioner	IntraNode Repartitioner
PTF	ParmetisRepart + TopoFM	FlatCacheLB
PTH	ParmetisRepart + TopoFM	HierCacheLB
ZTF	ZoltanRepart + TopoFM	FlatCacheLB
ZTH	ZoltanRepart + TopoFM	HierCacheLB

Algorithms We compared Aragon with two widely used architecture-agnostic repartitioners: ZoltanRepart [36] and ParmetisRepart [33]. For Aragon, we evaluated 4 different combinations of our inter- and intra-node repartitioners (Table 4.4). For intra-node repartitioning, we only considered the partitioner from [36] because Parmetis [33] kept failing due to an error originating from its code with our dataset. Supposedly, both inter- and intra-node repartitioning can be any (re)partitioners. All the results presented are the means of 5 trials

Table 4.5: Cache access latencies

Cache	L1	L2	L3
Latency (ns)	1	7	15

with 8 MPI processes on an 8-core machine with our simulated architecture. Initially, the graph was evenly distributed across processes for parallel repartitioning.

Metrics The quality of a decomposition in terms of the expected communication and migration cost is defined by Equation 4.2 and 4.3. Throughout the evaluation, the cost of communicating or migrating $64B$ data among compute nodes is approximated by 2 times of the access latency to the highest cache level weighted by the number of hops. In contrast, the communication cost of $64B$ data between two cores of the same compute node was estimated by the access latency to their first shared cache level. In cases where cores of the same node share no caches, we used 2 times of the access latency to the highest cache level as an approximation. The same process was used for the intra-node migration cost. Table 4.5 shows the cache access latencies used.

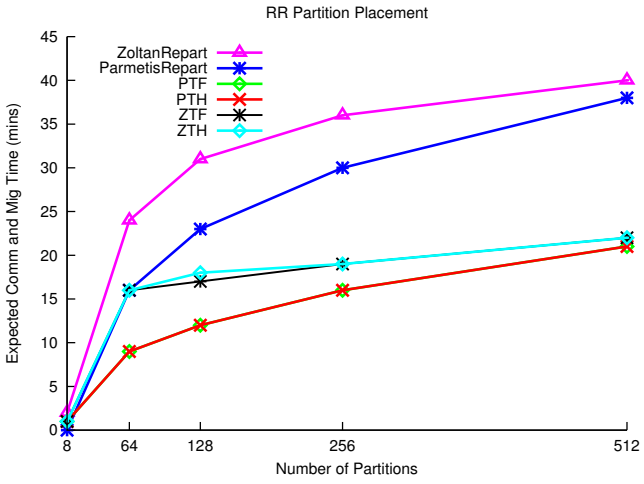


Figure 4.5: Varying num. of partitions (RR)

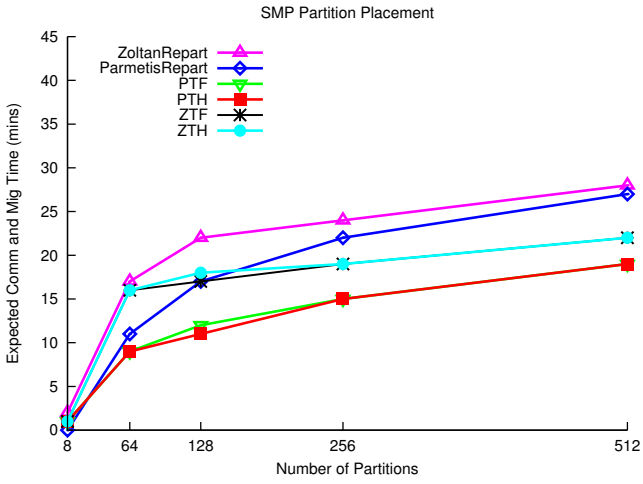


Figure 4.6: Varying num. of partitions (SMP)

4.2.2.2 Varying Number of Partitions

Our first experiment investigated ARAGON's robustness to graphs (Table 4.3) of varying partitions with $\alpha = 500$ (number of compu-

tation steps). The results in Figures 4.5 & 4.6 indicate that if only a few compute nodes are needed, Parmetis may obtain decompositions of similar quality as ARAGON (i.e., PTF/PTH/ZTF/ZTH) and sometimes even better, especially when ARAGON uses Zoltan for its inter-node repartitioning (i.e., ZTF/ZTH). We believe this was caused by the limited heterogeneity among a small number of compute nodes allocated on a relative small sized 3D-torus (providing limited refinement space for TopoFM). In contrast, with more compute nodes, ARAGON can outperform Zoltan and Parmetis by up to 60% and 46%, respectively, and the improvement became bigger as the number of partitions increased (due to the increasing heterogeneity). Since scientific computation usually requires hundreds of compute nodes, we believe that our approach will be beneficial for most applications.

The difference between PTF/H and ZTF/H was probably caused by the fact that Zoltan embraces the hypergraph model rather than the graph model like Parmetis. Thus, before Zoltan starts to (re)partition a graph, it first needs to convert the graph to a hypergraph, which may result in information loss from the original graph, leading to decompositions of lower quality.

Finally, we found that the improvement under the RR placement policy was bigger than that of SMP, which further confirms the general belief that SMP usually produces better partition mappings than RR, thus offering a smaller refinement space for TopoFM. Except for this difference, the results under both polices were similar. As such, given the space limitation, for the rest of our experimental study we will only present results under SMP policy, although the RR policy consistently showed bigger gains for ARAGON over its competitors in our experiments.

4.2.2.3 Varying Number of Computation Steps

This experiment evaluated the influence of α values (number of computation steps) using G512. The results in Figure 4.7 show that PTF/H and ZTF/H improved the decomposition quality by around 30% and 17%, respectively. We also observed that the improvement became more evident as α increased. As was the case in our previous experiment, the results of PTF/PTH were similar and always outperformed those of ZTF/ZTH. As such, for the rest of our experimental study, we will only present the results of PTH against Zoltan and Parmetis. The reason we prefer

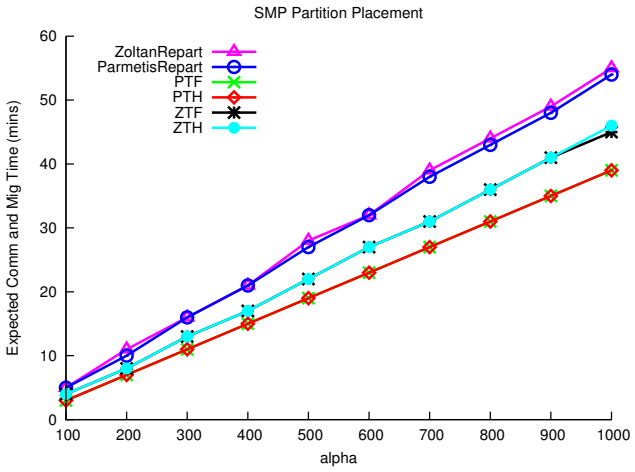


Figure 4.7: Num. of computation steps

PTH over PTF is that PTH does not require any quantitative information about the cache architecture (i.e., cache access latency).

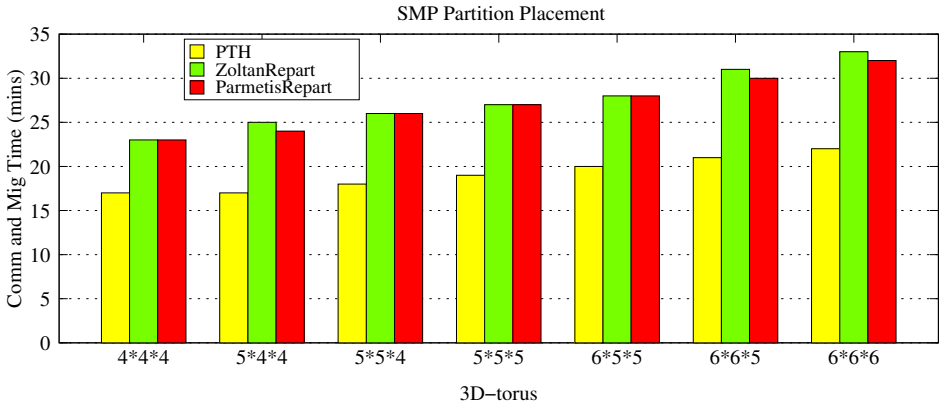


Figure 4.8: Different sized 3D-torus

4.2.2.4 Varying Sized 3D-Torus This experiment evaluated the impact of different-sized interconnects using G512 with $\alpha = 500$. Figure 4.8 shows that PTH outperformed Zoltan and ParMetis by 26%-32%, and that the expected communication and migration time of PTH increased slower than that of Zoltan and ParMetis, as the size of interconnect increased, indicating PTH's robustness to increasing heterogeneity.

Also, we expect a bigger difference between PTH and its competitors if the evaluation was carried out with a real application on a real supercomputer. This is because in our simulation-based evaluation, we did not consider the contention for the memory bandwidth

and interconnect links, which usually plays a critical role in communication cost. This contention would further favor PTH due to its ability to reduce the communication cost (and the resulting contention). We expect the network link contention (and therefore the difference of PTH with the state-of-the-art) to increase further, as the size of the datasets becomes bigger.

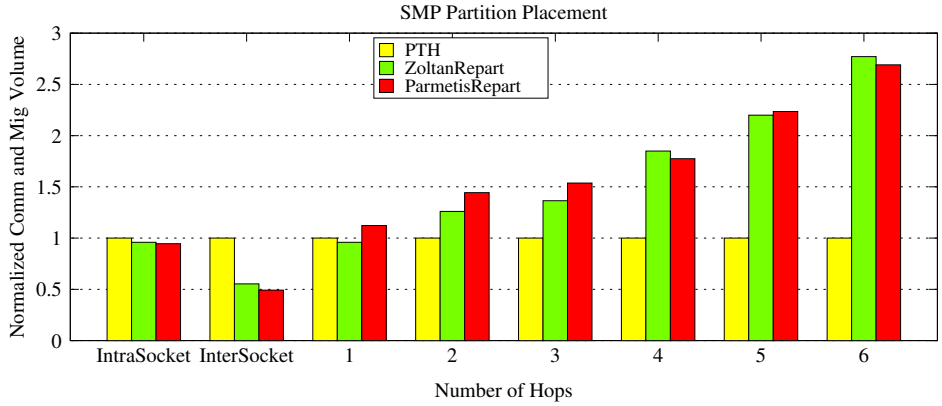


Figure 4.9: Normalized communication and migration volume distribution in terms of the number of hops each byte travels.

4.2.2.5 Communication and Migration Volume Breakdown To pinpoint the source of the improvement, we computed the breakdown of the communication and migration volume of G512 decompositions output by different algorithms over the different network distances (i.e., number of hops that the data needs to travel) with $\alpha = 500$. Figure 4.9 presents the overall communication and migration volume distribution in terms of the number of hops each byte traversed, normalized to that of PTH[1] As shown, PTH produced lower inter-node volume than ZoltanRepart and ParmetisRepart in all the cases, and the reduction in inter-node volume became more significant as the number of hops increased. All these reductions added together contributed to around 30% and 35% reduction in the overall inter-node volume of PTH against ZoltanReapart and ParmetisRepart, respectively. The reduction is mainly due to the ability of TopoFM in grouping the most communication-heavy vertices as close as possible and the design of the two-tier architecture offering more freedom

[1]The first and second group of columns represent the aggregated communication and migration volume among partitions within and across each socket in each compute node, whereas groups 3-8 denote the aggregated communication and migration volume among partitions that require 1-6 hops, respectively.

for inter-node repartitioning to group the frequently-communicating vertices into a single partition.

The reduction in inter-node volume also explained the increase in intra- and inter-socket volume. The improvement in intra-socket volume also demonstrated the effectiveness of FlatCacheLB and HierCacheLB in clustering the vertices communicating a lot to cores sharing more cache levels. Also, we noticed that PTH maintained the same total communication and migration volume as ZoltanReapart and ParmetisRepart, implying that PTH can improve the communication mapping without deteriorating the decomposition.

Table 4.6: Degree of imbalance

Algorithms	Average	Std. Deviation
PTH	1.0340	0.0053
ZoltanRepart	1.0428	0.0156
ParmetisRepart	1.0525	0.0163

4.2.2.6 Degree of Imbalance In terms of the imbalance degree, PTH produced decompositions slightly better than ZoltanRepart and ParmetisRepart. The average imbalance degree of different algorithms over all the experiments we ran and their standard deviations are presented in Table 4.6.

4.2.2.7 Repartition Time Currently, PTH is 4 to 10 times slower than ZoltanRepart due to the sequential refinement phase. However, this time is negligible compared to the actual simulation time, because scientific simulations often run for a very long time, and repartitioning is not a frequent operation as load changes during each computation step are often negligible but the accumulated changes across computation steps are usually significant. Besides, we plan to further parallelize and evaluate the refinement phase with a real workload in the future.

4.2.3 Section Summary

In this work, we proposed an architecture-aware graph repartitioner, ARAGON, that is particularly suited for data- and compute-intensive applications on modern parallel computing infrastructures, i.e., typical Big Data scientific applications. ARAGON considers the characteristics of both the underlying computing infrastructures and target workload while repartitioning. For compute-intensive applications that are also data-intensive, such considerations are extremely crucial. In fact, we showed that ARAGON outperforms the state-of-the-art (Parmetis and Zoltan) by up to 60% using data derived from a real dataset.

4.3 PARAGON: PARALLEL ARCHITECTURE-AWARE GRAPH REPARTITIONING

The main problem of Aragon is that it requires the use of the heavyweight graph repartitioners for its inter-node repartitioning and has a centralized component, TopoFM. To address the issue, Paragon first separates TopoFM out from Aragon and then makes a parallel implementation of the algorithm. In other words, Paragon is a parallel implementation of TopoFM, which can be used to improve the mapping of the application communication and computation pattern to the underlying computing infrastructures. Because of this, we will refer Aragon as the centralized implementation of TopoFM for the rest of the thesis.

4.3.1 Algorithm Design and Implementation

In our previous implementation, Aragon requires all the servers to send their local partitions to a single server for TopoFM to be carried out. The centralized server is responsible for the refinement of all the partition pairs. Although such a solution, only requires sending the entire graph over the network once, the server has to be able to store the entire graph in memory. As a result, the server can easily become a performance and scalability bottleneck.

Another naive implementation of Aragon could be as follows: server M_i is responsible for the refinement of P_i with all its partners $P_{i+1}, P_{i+2}, \cdots, P_n$, and server M_{i+1} can not start its refinement for P_{i+1} until server M_i finishes its refinement. One major issue of this approach is that it requires the entire graph to be sent across the network $\frac{n-1}{2}$ times. An advantage of this approach is that each server only needs to hold two partitions in memory at a time (one for its local partition and the other one for the refinement partner).

To address the issues identified above, Paragon takes a middle point of the two extremes for TopoFM parallelization, where it allows multiple servers to do the refinement in parallel, each of which is responsible for the refinement of a group of partitions. In this way, we can enjoy the benefits of both extremes without worrying about their drawbacks.

Algorithm 2 describes the main idea of Paragon. During the refinement, each server

Algorithm 2: PARAGON

```
Data: P_l, c
Result: new locations of vertices of P_l
1  masterNodeSelection(c)
2  partitionStat(P_l, ps)
3  if server M[l] is master node then
4      pg = partitionGrouping()
5      gs = optGroupServerSelection(pg, ps, c)
6      partitionGroupServerBcast(gs);
7  sendPartitionToGroupServers(P_l, gs)
8  if server M[l] is a group server then
9      pg = recvPartitionsFromMyGroupMembers(gs)
10     foreach P_i ∈ pg do
11         foreach P_j ∈ pg do
12             if i ≠ j then
13                 AragonRefinement(P_i, P_j, c)
14     shuffleRefinement(pg)
15     vertexLocationUpdate(pg)
16 physicalDataMigraton(P_l)
```

runs an instance of the algorithm with its local partition P_l and the relative network communication cost matrix c as its input. The algorithm first selects a server as master node (Line 1), and then computes everything needed by the master node to make the parallization decision (Line 2). The master node decides how to split the partitions into groups such that each group can be refined independently on different servers and the selection of the group servers (Line 4–6). The group servers take responsibility for the refinement of each group. Once the decision has been made, each server will send their vertices to the corresponding group servers (Line 7). Upon receiving all the vertices from their group members, the group servers will start to do the refinement of each group independently (Line 8–13). After finishing the refinement of its group, the group servers will notify their group members about the new locations of their vertices (Line 15). Then, each server will physically migrate vertices to their new owners accordingly (Line 16).

4.3.1.1 Partition Grouping To assign a partition to a group, we consider three factors: (1) to minimize the refinement time, each group should have roughly equal number of

partitions; (2) members of each group should be carefully selected, since the gain of refining each partition pair may vary a lot. Thus, to maximize the effectiveness of the refinement, we should group together partitions leading to high refinement gain; and (3) we should minimize the cross-group refinement interference, because the gain of refining one partition pair heavily relies on the amount of data they communicate with other partitions. This is different from the standard FM algorithms, which solely compute the gain of migrating each vertex based on the data it communicates with vertices of the partition pair. For example, in the decomposition of Figure 4.2, the communication between vertex a and j contributes most to the gain of moving a from P_1 to P_2 for PARAGON. However, for standard FM algorithms, the gain of migrating a to P_2 will be -1, since a has two neighbors in P_1 and 1 in P_2. Unfortunately, there is no clear way to do the grouping, since we could not use the state-of-the-art graph partitioners (i.e., METIS) to compute a high-quality initial decomposition, due to their poor scalability. As a result, the input decomposition to PARAGON will probably have edge-cuts across all the partitions. Fortunately, we found that random grouping along with the *shuffle refinement* (the remedy technique presented below) works quite well.

Theoretically, the number of groups we can have can be any integer between 1 and $\frac{n}{2}$, where n is the number of partitions of the graph. Clearly, if the number of groups equals 1, PARAGON degrades to ARAGON, in which all servers will send their local partitions to a single group server for sequential refinement. The reason why there is an upper bound is because each group needs to have at least 2 partitions for the refinement to proceed. Typically, the higher the number is, the faster the refinement will finish. However, there is a tradeoff between the degree of parallelism and the quality of the resulting decomposition we can have. This is because the higher the number is, the fewer partitions each group will have and thus the fewer partition pairs will be refined. Given a graph with n partitions and m groups, PARAGON only refines $\frac{n(n-m)}{2m}$ partition pairs, while ARAGON refines all $\frac{n(n-1)}{2}$ partition pairs. In other words, ARAGON will eventually select an optimal migration destination among all the partitions for each vertex, whereas PARAGON only considers a subset of the partitions for each vertex. This also explains the reason why the resulting decompositions computed by PARAGON are usually poorer than those of ARAGON. Fortunately, the shuffle refinement technique we proposed helps to address the issue.

4.3.1.2 Shuffle Refinement

To mitigate the impact of cross-group refinement interference and increase the gain of the refinement, we perform an additional round of refinement once all the group servers finish the refinement of their own groups. We call this *shuffle refinement*. In this round, each group server first exchanges the changes it made to the decompositions such that each group server has the up-to-date load information of each partition and the up-to-date locations of the neighbors of each vertex. Then, each group server swaps some of its partitions randomly with other group servers. Subsequently, each group server starts another round of refinement with the new grouping.

The reason why shuffle refinement is a remedy to the above issue is because it increases the number of partition pairs refined by PARAGON and thus the solution space that PARAGON explores. For example, for a graph with 4 partitions and 2 groups, PARAGON originally only refines 2 out of the 6 partition pairs. However, if the group servers swap one of their partitions, PARAGON will refine 4 partition pairs instead of 2. In fact, we can repeat this shuffle refinement multiple times to further expand the solution space PARAGON explores, thus further alleviating the impact of cross-group refinement interference and increasing the gain we can obtain.

The idea of shuffle refinement is very straightforward, but it is not easy to efficiently implement, especially the propagation of the changes that each group server made. One easy way to achieve this is to use a distributed data directory, like the one provided by Zoltan [36]. In this scheme, each group server only needs to make an update to the data directory first, and then all the group servers can pull the up-to-date locations for the neighbors of their vertices. We found that this approach is very inefficient for really big graphs in terms of both memory footprint and execution time. It requires around $O(|V| + |E|)$ data communication.

Another way to achieve this is to maintain an array at each group server, forming a mapping from vertex global identifiers[2] to their locations. In this way, the exchange can easily be achieved via a single (MPI) reduce operation, requiring only $O(|V|)$ data communication. This approach is much more efficient than the distributed data directory approach in terms of execution time, but it is not memory scalable for large graphs.

[2]In distributed graph computation, each vertex has a unique global identifier across all the partitions and a unique local identifier within each partition.

In our implementation, we adopt a variant of the second approach. That is, we first chunk the entire global vertex identifier space into multiple smaller equal sized regions. Each region contains vertices within a contiguous range. By default, the region size equals $k = \min\{2^{26}, |V|\}$, where V is the vertex set of the entire graph. Correspondingly, the exchange is split into multiple rounds. Each round only exchanges the locations of vertices of one region. With this scheme, we only need to maintain a smaller array at each group sever and thus the amount of data communication remains unchanged. Although this scheme requires scanning the edge lists of each partition multiple times, it is much more efficient than the distributed data directory approach.

4.3.1.3 Group Server Selection Once the master node finishes the grouping process, it will select an optimal server for each group, such that the cost of sending partitions of the group to the group server is minimized. For example, in case of Figure 4.2, where we assume that P_1, P_2, and P_3 are of one group, we should select server M_2 as the group server intuitively since $c(P_1, P_2) = c(P_2, P_3) = 1$ while $c(P1, P_3) = 6$. To achieve this, we define the cost of selecting server M_s as the group server for group g as:

$$\sum_{P_i \in g} ps[i] * c(P_i, P_s) * (1 + \frac{\sigma(s)}{drp}) \tag{4.12}$$

Here, $ps[i]$ denotes the number of edges associated with vertices of P_i, which is a good approximation for the amount of data each server needs to send to their group servers. $\sigma(s)$ is the number of group servers that have been designated on the compute node that server M_s belongs to. It should be noticed that server M_s can be a hardware thread, a core, a socket, or a machine. drp is the degree of refinement parallelism (number of group servers). The last term $(1 + \frac{\sigma(s)}{drp})$ is the penalty that is added to avoid the concentration of multiple group servers into a single compute node, reducing the chance of memory exhaustion. Once all the group servers are selected, the master node will broadcast the group servers of the groups to the slave nodes. Then, each server will send its vertices (as well as their edge lists) to their corresponding group servers, after which the group servers will start to refine partitions of their own groups independently.

4.3.1.4 Reducing Communication Volume

Clearly, PARAGON with the shuffle refinement disabled requires the entire graph to be sent over the network once, and PARAGON with the shuffle refinement enabled demands more data communication. For really big graphs, the communication volume may get very high. Thus, we follow the same approach proposed in [68] to reduce the communication volume. Specifically, instead of sending the entire partition to their group servers, each server only needs to send vertices that can be reached by a breadth-first search from boundary vertices of each partition within k-hop traversal. Boundary vertices are vertices that have neighbors in other partitions. The rationale behind this is that if a vertex is very far from the boundary vertex, the chance that it get moved by PARAGON to another partition to improve the decomposition is very small. Surprisingly, we found that PARAGON is not sensitive to k in terms of the partitioning quality, and that a larger k does not always lead to partitionings of higher quality. However, it may increase the refinement time greatly. Thus, in our implementation, we set $k = 0$ by default. In other words, we only send boundary vertices of each partition to the group servers.

In fact, [68] has presented a solution to parallellize the standard FM algorithms [67]. However, it may require a graph with n partitions to be sent over the network $n - 1$ times in case the initial decomposition has edge-cuts across all partition pairs. Furthermore, the presence of communication heterogeneity complicates things greatly. First, ARAGON has to be applied to all the partition pairs, whereas standard FM algorithms, which assume uniform network communication costs, only need to refine partition pairs that have edge-cuts between them. Second, during each refinement iteration of a single partition pair, standard FM algorithms only need to consider migrating vertices of both partitions that have neighbors in the alternative partition. On the other hand, PARAGON has to consider migrating all boundary vertices.

4.3.1.5 Master Node Selection

As presented so far, each server (slave node) needs to send some auxiliary data (i.e., the number of vertices/neighbors) of their local partitions to the master node for the parallelization decision, and the master needs to broadcast the decision it made to all the slave nodes. To reduce the communication cost between the

master node and the slave nodes, we also select the master node in an intelligent way using the following heuristic:

$$\min_{m \in [1,n]} \sum_{i=1 \text{ and } i \neq m}^{n} c(P_i, P_m) \tag{4.13}$$

The heuristic tries to find a server $M[m]$ that will result in minimal network communication cost as the master node. For example, in case of Figure 4.2, we should select server M_2 as the master node. Clearly, the selection of master node can be made locally by each server without synchronizing with each other.

4.3.1.6 Incorporating Contention-Awareness

To make PARAGON also aware of the issue of shared resource contention on the memory subsystems, we adopt the same solution as ARGO, where we penalize intra-node network communication costs by a score. The score is computed based on the degree of the contentiousness between the communication peers. By doing this, the amount of intra-node communication will decrease accordingly. The parameter λ can be used to specified the degree of contention. Similar to ARGO, if $\lambda = 0$, PARAGON will only consider the communication heterogeneity, whereas $\lambda = 1$ means that intra-node shared resource contention is the biggest performance bottleneck, which should be prioritized over the communication heterogeneity. It should be noticed that PARAGON with any $\lambda \in (0, 1]$ considers both the contention and the communication heterogeneity. Again, considering the impact of the 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.

4.3.2 Evaluation

4.3.2.1 Setup

In this section, we first evaluate the sensitivity of PARAGON to varying input decompositions computed by different initial partitioners and the impact of its two important parameters: the degree of parallelism and the number of shuffle refinement times (Section 4.3.2.2). We then validate the effectiveness of PARAGON using two real-world graph workloads: Breadth-First Search (BFS) [59] and Single-Source Shortest Path (SSSP) [60],

which we implemented using MPI (Section 4.3.2.3). Finally, we demonstrate the scalability of PARAGON via a billion-edge graph (Section 4.3.2.4).

Table 4.7: Datasets used in our experiments

Dataset	$\|V\|$	$\|E\|$	Description
wave [74]	156,317	2,118,662	2D/3D FEM
auto [74]	448,695	6,629,222	3D FEM
333SP [75]	3,712,815	22,217,266	2D FE Triangular Meshes
CA-CondMat [61]	108,300	373,756	Collaboration Network
DBLP [62]	317,080	1,049,866	Collaboration Network
Email-Eron [61]	36,692	183,831	Communication Network
as-skitter [61]	1,696,415	22,190,596	Internet Topology
Amazon [61]	334,863	925,872	Product Network
USA-roadNet [76]	23,947,347	58,333,344	Road Network
PA-roadNet [61]	1,090,919	6,167,592	Road Network
YouTube [62]	3,223,589	24,447,548	Social Network
com-LiveJournal [61]	4,036,537	69,362,378	Social Network
Orkut [61]	3,072,627	234,370,166	Social Network
Friendster [61]	124,836,180	3,612,134,270	Social Network
Twitter [62]	52,579,682	3,926,527,016	Social Network

Datasets Table 4.7 describes the datasets used. By default, the graphs were (re)partitioned with vertex weights (i.e., computational requirement) set their vertex degree, with vertex sizes (i.e., amount of the data of the vertex) set their vertex degree, and with edge weights (i.e., amount of data communicated) set to 1. The degree of each vertex is often a good approximation of the computational requirement and the migration cost of each vertex, while a uniform edge weight of 1 is a close estimation of the communication pattern of many graph algorithms, like BFS and SSSP. Given the fact that communication cost is usually more important than migration cost, all the experiments were performed with $\alpha = 10$ (Eq. 4.2). Unless explicitly specified, all the graphs were initially partitioned by DG (deterministic greedy heuristic), a state-of-the-art streaming graph partitioner [8], across cores of the com-

Table 4.8: Cluster compute node configuration

Node Configuration	MPICluster (Intel Haswell Processor)	Gordon (Intel Sandy Bridge Processor)
Sockets	2	2
Cores	20	16
Clock Speed	2.6 GHz	2.6 GHz
L3 Cache	25 MB	20 MB
Memory Capacity	128 GB	64 GB
Memory Bandwidth	65 GB/s	85 GB/s

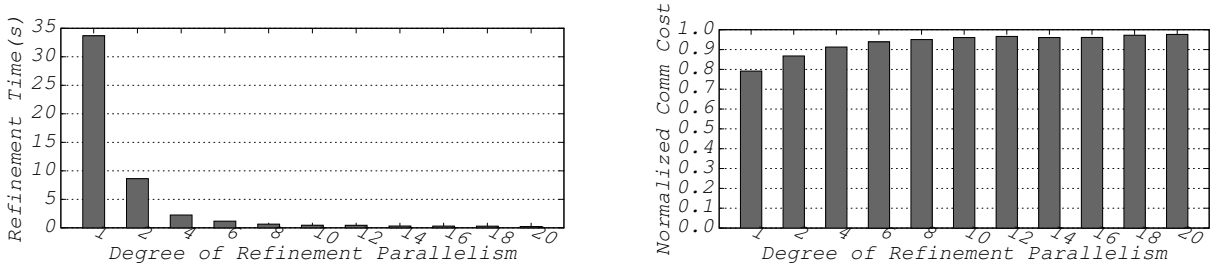
pute node used (one partition per core). The partitionings were then improved by PARAGON. During the (re)partitioning, we allowed up to 2% load imbalance among the partitions. For fairness, DG/LDG were extended to support vertex- and edge-weighted graphs. Vertices of the graphs were presented to DG/LDG in some unknown random order.

Platforms We evaluated PARAGON on two clusters: MPICluster [65] and Gordon supercomputer [50]. MPICluster had a flat network topology, with all 32 compute nodes connected to a single switch via 56Gbps FDR Infiniband. On the other hand, the Gordon network topology was a 4x4x4 3D torus of switches connected via QDR Infiniband with 16 compute nodes attached to each switch (with 8Gbps link bandwidth). Table 4.8 depicts the compute node configuration of the clusters. The results presented were the means of 5 runs, except the execution of SSSP on Gordon (Section 4.3.2.3) and the scalability test (Section 4.3.2.4).

Network Communication Cost Modelling The relative network communication costs among the partitions (cores) were approximated using a variant of the osu_latency benchmark [66]. To ensure the correctness of the cost matrix, each MPI rank (process) was bound to a core using the mechanism provided by MVAPICH2 1.9 [47] on Gordon and OpenMPI 1.8.6 [46] on MPICluster. MVAPICH2 and OpenMPI were two different MPI implementations available on the clusters.

4.3.2.2 MicroBenchmarks

Varying Degree of Parallelism In this experiment, we examined the impact of the degree



(a) Refinement time

(b) Comm cost of the resulting partitionings

Figure 4.10: Refinement time and normalized communication costs of the com-lj decompositions after being refined with varying degree of refinement parallelism on two 20-core compute nodes.

of parallelism in terms of the refinement time (i.e., the time that the refinement took) and the refinement quality (i.e., the communication cost of the resulting decomposition). Towards this, we first partitioned the com-lj dataset into 40 partitions using DG across two compute nodes of MPICluster, and then applied PARAGON to the decompositions with varying degree of refinement parallelism but with shuffle refinement disabled.

Results (Figures 4.10a & 4.10b) Figure 4.10a plots the runtime of PARAGON on the com-lj dataset for various degrees of parallelism. As expected, the higher the degree of parallelism, the faster the refinement would finish, and PARAGON significantly reduced the refinement time of ARAGON (PARAGON with degree of parallelism of 1). However, the speedup was achieved at the cost of higher communication cost of the resulting decompositions (Figure 4.10b). The communication costs presented were normalized to that of the initial decomposition computed by DG. However, in the end, PARAGON still resulted in lower communication cost in all the cases when compared to the initial decompositions.

Impact of Shuffle Refinement In our second experiment, we were interested to see whether the shuffle refinement technique could address the issue we identified in the previous experiment. Towards this, we repeated the same experiment but with a fixed degree of refinement parallelism (8) and varying number of shuffle refinement times (from 0 to 15).

Results (Figure 4.11) Figure 4.11 shows the corresponding refinement time and the normalized communication costs of the resulting decompositions with the decompositions computed by ARAGON as the baseline. As shown, PARAGON (with shuffle refinement en-

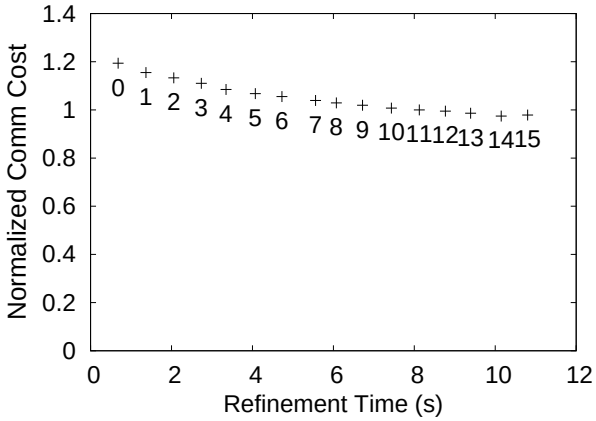


Figure 4.11: Y-axis corresponds to the communication costs of the com-lj decompositions after being refined with varying number of shuffle refinement times on two 20-core compute nodes when they were normalized to that of the decompositions refined by ARAGON; X-axis denotes the corresponding refinement time; the labels on each data point were the number of refinement times.

abled) not only produced decompositions of lower communication costs than ARAGON (when the number of shuffle refinement times was greater than 11), but also completed the refinement faster (ARAGON took around 33s to finish the refinement vs 8.12s by PARAGON with 11 shuffle refinement times).

Impact of Initial Partitioners This experiment examined the refinement overhead and the quality of the resulting decompositions, when PARAGON was provided with decompositions computed by four different partitioners: (a) HP, the default graph partitioner of many parallel graph computing engines; (b) DG and LDG, two state-of-the-art streaming graph partitioning heuristics [8]; and (c) METIS, a state-of-the-art multi-level graph partitioner [32]. The graphs were initially partitioned across cores of the same two machines used in our prior experiments but with both the degree of refinement parallelism and the number of shuffle refinement times set to 8.

Quality of the Initial Decompositions (Figure 4.12) Figure 4.12 denotes the communication cost of the initial decompositions computed by HP, DG, LDG, and METIS for a

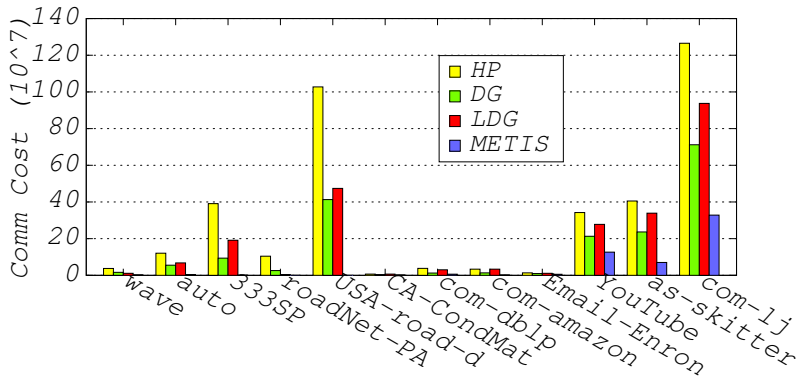
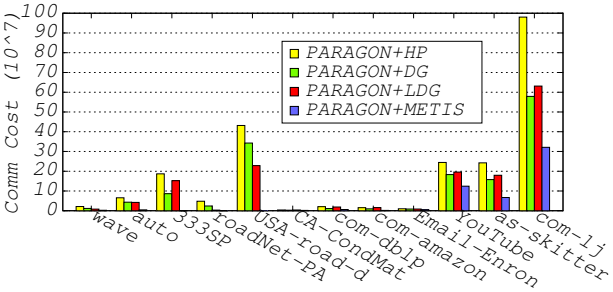
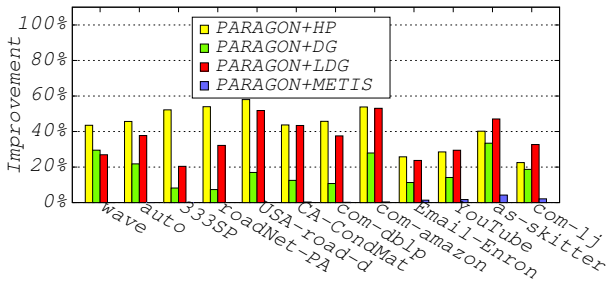


Figure 4.12: Communication cost of the initial decompositions computed by HP, DG, LDG, and METIS across cores of two 20-core compute nodes for a variety of graphs.

variety of graphs. As anticipated, METIS performed the best and HP was the worst. However, METIS is a heavyweight serial graph partitioner, making it infeasible for large-scale distributed graph computation either as an initial partitioner or as an online repartitioner (repartitioning from scratch). It was reported in prior work [10] that METIS took up to 8.5 hours to partition a graph with 1.46 billion edges. Unexpectedly, DG outperformed LDG, the best streaming partitioning heuristic among the ones presented in [8]. This was probably because the order in which the vertices were presented to the partitioner favored DG over LDG (the results of DG and LDG rely on the order in which vertices are presented). This was also the reason why we picked DG as the default initial partitioner for PARAGON.



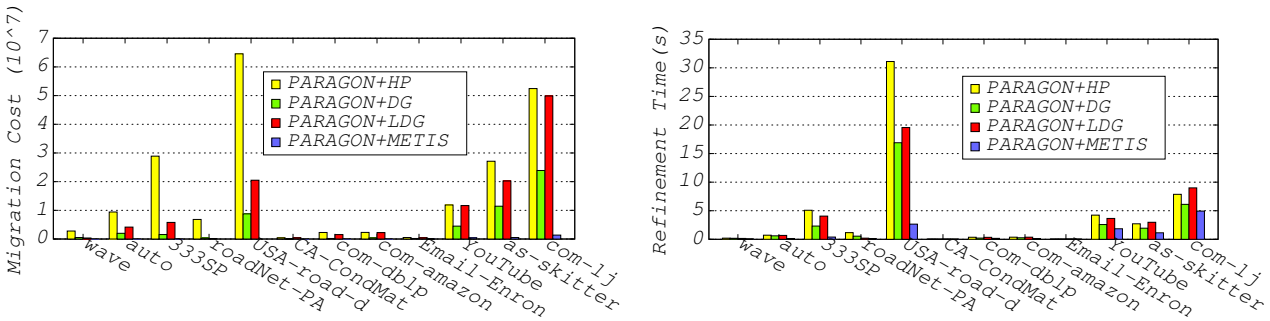
(a) Communication cost of the decompositions after being refined.



(b) Improvement achieved by PARAGON against the initial decomposition.

Figure 4.13: PARAGON's sensitivity to varying initial decompositions in terms of the communication cost for a variety of graphs, which were initially partitioned by HP, DG, LDG, and METIS across cores of two 20-core compute nodes.

Quality of the Resulting Decompositions (Figures 4.13a & 4.13b) Figures 4.13a and 4.13b show the corresponding communication cost of the resulting decompositions and the improvement achieved by PARAGON in terms of the communication cost when compared to the initial decompositions. As shown, the better the initial decomposition was, the better the resulting decomposition would be. In comparison with the initial decompositions computed by HP, DG, and LDG, PARAGON reduced the communication cost of the decompositions by up to 58% (43% on average), 29% (17% on average), and 53% (36% on average), respectively. Although PARAGON did not improve significantly the decompositions computed by METIS for easily partitioned FEM and road networks (left 7 datasets), it achieved an improvement of up to 4.5% for complex networks (right 5 datasets). Given the size of the dataset, the improvement was still non-negligible. Fortunately, we found that PARAGON with DG as its initial partitioner can achieve even better performance than METIS on real-world workloads (Section 4.3.2.3).



(a) Migration Cost (b) Refinement Time

Figure 4.14: Overhead of the refinement on varying decompositions that were initially partitioned by HP, DG, LDG, and METIS across cores of two 20-core compute nodes.

Refinement Overhead (Figures 4.14a & 4.14b) We also noticed that the quality of the initial decomposition impacted the refinement overhead greatly. Figures 4.14a and 4.14b plot the migration cost (Eq. 4.3) and the refinement time. Clearly, the poorer the initial decomposition was, the higher the migration cost and the longer the refinement time would be. Finally, for decompositions, which PARAGON failed to make much improvement, PARAGON only led to a very small amount of overhead.

4.3.2.3 Real-World Applications (BFS & SSSP)

Configuration This experiment evaluated PARAGON using BFS and SSSP on the YouTube, as-skitter, and com-lj datasets. Initially, the graphs were partitioned across cores of three compute nodes of the two clusters using DG. Then, the decomposition was improved by PARAGON with the degree of refinement parallelism and the number of shuffle refinement times both set to 8. During the execution of BFS/SSSP, we grouped multiple messages sent by each MPI rank to the same destination into a single one (8 for YouTube and as-skitter dataset and 16 for com-lj dataset). The reason why we picked 8 and 16 was because any larger values would make the execution time too short for consideration, especially for the execution of BFS.

Resource Contention Modeling To capture the impact of resource contention, we carried out a profiling experiment for BFS and SSSP with the 3 datasets on both clusters by increasing λ gradually from 0 to 1. Interestingly, we found that intra-node shared resource contention was more critical to the performance on MPICluster, while inter-node communication was the bottleneck on Gordon. This was probably caused by the differences in network topologies (flat vs hierarchical), core count per node (20 vs 16), memory bandwidth (65GB vs 85GB), and network bandwidth (56Gbps vs 8Gbps) between the two clusters, and that BFS/SSSP had to compete with other jobs running on Gordon for the network resource, while there was no contention on the network communication links on MPICluster. Hence, we fixed λ to be 1 on MPICluster and 0 on Gordon for the experiment.

Table 4.9: BFS job execution time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
MPICluster			
DG	30	59	218
METIS	8.50	67	27
PARMETIS	29 (21.00)	59 (9.65)	185 (4.71)
UNIPARAGON	25 (2.70)	27 (2.26)	159 (7.54)
PARAGON	8 (4.00)	10 (3.31)	40 (10.00)
Gordon			
DG	322	577	4319
UNIPARAGON	264 (2.70)	350 (2.07)	3310 (6.98)
PARAGON	220 (3.83)	228 (2.96)	2586 (9.08)

Table 4.10: SSSP job execution time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
MPICluster			
DG	2136	1823	5196
METIS	545	822	955
PARMETIS	1842 (19.00)	582 (9.28)	3268 (4.50)
UNIPARAGON	1805 (2.45)	1031 (2.07)	3136 (6.98)
PARAGON	468 (3.88)	472 (3.14)	1549 (9.71)
Gordon			
DG	3436	7092	10732
UNIPARAGON	3402 (2.76)	3355 (2.13)	7831 (9.75)
PARAGON	2838 (3.89)	2731 (2.97)	6841 (29.00)

Job Execution Time (Tables 4.9 & 4.10) Tables 4.9 and 4.10 show the overall execution time of BFS and SSSP with 15 randomly selected source vertices on the three datasets and

the overhead of PARAGON. The execution time of a distributed graph computation is defined as: $JET = \sum_{i=1}^{n} SET(i)$, where n is the number of supersteps the job has, while $SET(i)$ denotes the execution time of the ith superstep and is defined as the ith superstep execution time of the slowest MPI rank. In the table, DG and METIS mean that BFS/SSSP was performed on the datasets without any repartitioning/refinement, PARMETIS is a state-of-the-art multi-level graph repartitioner [33], UNIPARAGON was a variant of PARAGON that assumes homogeneous and contention-free computing environment, and the numbers within the parentheses were the overhead of repartitioning/refining the decomposition computed by DG.

As expected, PARAGON beat DG, PARMETIS, and UNIPARAGON in all the cases. Compared to DG, PARAGON reduced the execution time of BFS and SSSP on Gordon by up to 60% and 62%, respectively, and up to 83% and 78% on MPICluster, respectively. If we time the improvements by the number of MPI ranks (48 for Gordon and 64 for MPICluster), the improvements were more remarkable. Yet, the overhead PARAGON exerted (sum of the refinement time and physical data migration time) was very small in comparison to the improvement it achieved and the job execution time. By comparing the results of UNIPARAGON with DG, we can conclude that PARAGON not only improved the mapping of the application communication pattern to the underlying hardware, but also the quality of the initial decomposition (edgecut). What we did not expect was that PARAGON with DG as its initial partitioner outperformed the gold standard, METIS, in 4 out the 6 cases and was comparable to METIS in other cases.

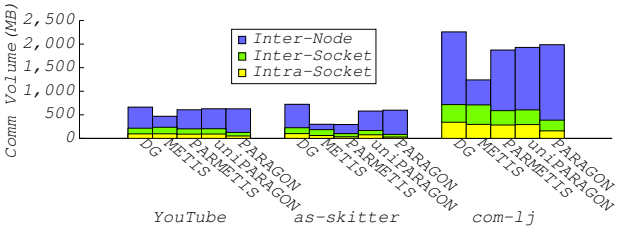


Figure 4.15: The breakdown of the accumulated communication volume across all supersteps for BFS on PittMPICluster.

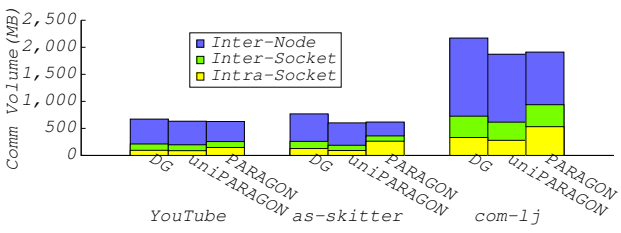


Figure 4.16: The breakdown of the accumulated communication volume across all supersteps for BFS on Gordon.

Communication Volume Breakdown (Figures 4.15 & 4.16) To further confirm our observations, we also collected the total amount of data remotely exchanged per superstep by BFS and SSSP among cores of the same socket (intra-socket communication volume), among cores of the same compute node but belonging to different sockets (inter-socket communication volume), and among cores of different compute nodes (inter-node communication volume). Since we observed similar patterns for BFS and SSSP in all the cases, we only present the breakdown of the accumulated communication volume across all the supersteps for the execution of BFS here.

As shown in Figures 4.15 (for MPICluster) and 4.16 (for Gordon), PARAGON and uniPARAGON have much lower remote communication volume than DG in all the cases, and PARAGON has the lowest inter-node communication volume and highest intra-node (inter-socket & intra-socket) communication volume on Gordon (vice versa on MPICluster), which was expected given our choices of λ. It is worth mentioning that on MPICluster, intra-node data communication was the bottleneck. Another interesting thing was that in spite of its higher total communication volume when compared to METIS, PARMETIS, and uniPARAGON, PARAGON still outperformed them in most cases due to the reduced communication on critical components.

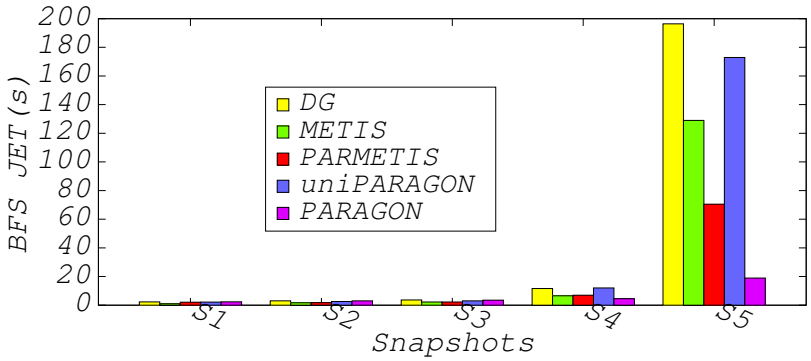


Figure 4.17: BFS JET with Graph Dynamism

Graph Dynamism (Figure 4.17) To further validate the effectiveness of PARAGON in the presence of graph dynamism, we split the YouTube dataset (a collection of YouTube users and their friendship connections over a period of 225 days) into 5 snapshots with an interval of 45 days. Thus, snapshot S_i denotes the collection of YouTube users and their friendship connections appearing during the first $45 * i$ days. We then ran BFS on snapshot S_1 across

three 20-core machines and injected vertices newly appeared in each snapshot to the system using DG whenever BFS finished its computation for every 15 randomly selected vertices. The injection also triggered the execution of PARAGON, uniPARAGON, and PARMETIS on the decomposition.

Figure 4.17 plots the BFS execution time for 15 randomly selected source vertices on each snapshot. As shown, both architecture-awareness and the capability to cope with graph dynamism were critical to achieve superior performance. This is especially true as the graph changes a lot from its original version: at snapshot S_5, PARAGON performed 90% better than DG, 85% better than METIS, 73% better than PARMETIS, and 89% better than uniPARAGON.

4.3.2.4 Billion-Edge Graph Scaling

Configuration In this experiment, we investigated the scalability of PARAGON as the graph scale increased. Towards this, we generated three additional datasets by sampling the edge list of the friendster dataset (3.6 billion edges). We denote the datasets generated as friendster-p, where p was the probability that each edge was kept while sampling. Hence, friendster-p would have around $3.6 * p$ billion edges. Interestingly, the number of vertices remained almost unchanged in spite of the sampling. We ran the experiment on three compute nodes of MPICluster with the degree of refinement parallelism, the number of shuffle refinement times, and the message grouping size set to 10, 10, and 256, respectively.

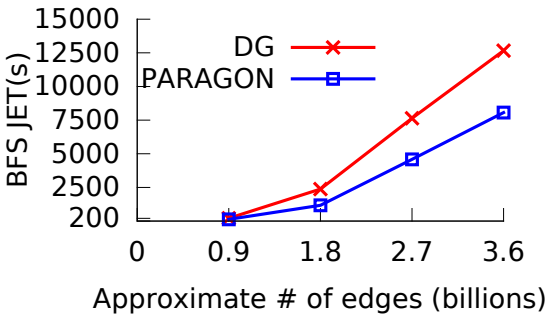


Figure 4.18: BFS JET vs Graph Size

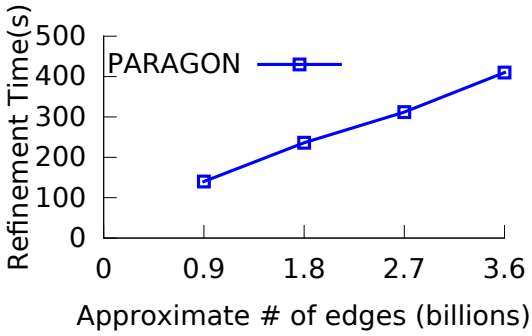


Figure 4.19: Refinement Time vs Graph Size

Results (Figures 4.18 & 4.19) Figures 4.18 and 4.19 present the execution time of BFS with 15 randomly selected source vertices and the overhead of PARAGON at different graph

scales. As shown, PARAGON not only led to lower job execution times, but also to lower speed in which the job execution time increased as the graph size increased. It should be noticed that PARAGON reduced the execution time of all machines (3*20 cores) not just one. Also, the refinement time increased at a much slower rate (from 140s, to 236s, to 312s, and to 410s) than that of the graph size. The reason why we did not present the results of METIS or PARMETIS here was because they failed to (re)partition the graphs (even for the first dataset, of 0.9 billion edges).

4.3.3 Section Summary

In this work, we presented PARAGON, a *parallel architecture-aware* graph partition refinement algorithm that bridges the mismatch between the application communication pattern and the underlying hardware topology. PARAGON achieves this by modifying a given decomposition according to the non-uniform network communication costs and consideration of the contentiousness of the underlying computing infrastructures. To further reduce its overhead, we made PARAGON itself architecture-aware. Compared to the state-of-the-art, PARAGON improved the quality of graph decompositions by up to 53%, achieved up to 5.9x speedups on real workloads, and successfully scaled up to a 3.6 billion-edge graph.

4.4 PLANAR AND PLANAR+: PARALLEL LIGHTWEIGHT ARCHITECTURE-AWARE GRAPH REPARTITIONING

In this section, we first introduce our *Parallel Lightweight Architecture-Aware graph Repartitioner*, PLANAR (Section 4.4.1) that we designed for large dynamic graph partitioning. We then introduce an optimized implementation of PLANAR, PLANAR+ (Section 4.4.3). They both overcome the scalability limitation of PARAGON by increasing the degree of parallelism that the repartitioning algorithm can have; PLANAR+ introduces implementation optimizations that reduces the cost of repartitioning by up to 9x compared to PLANAR+. We describe and evaluate PLANAR first, and then describe PLANAR+ along with some additional experimental results.

4.4.1 PLANAR: Algorithm Design and Implementation

Algorithm 3: Planar Overview

```
Data: P_l, c, σ, τ
if the partitioning has not converged then
    // Phase-1 (Section 4.4.1.1 & 4.4.1.2)
    LogicalVtxMigration(P_l, c, &pv)
    // Phase-2 (Section 4.4.1.3)
    PhysicalVtxMigration(P_l, pv)
    // Convergence Check (Section 4.4.1.4)
    CheckPartitionConvergence(σ, τ);
```

Rather than costly repartitioning the entire graph at once, PLANAR adapts the current partitioning in the presence of changes by incrementally migrating vertices among the partitions, while considering the non-uniformity of the network communication costs. Algorithm 3 presents PLANAR at a high level. It is triggered whenever there are enough changes in the graph or imbalance among the partitions. Once triggered, it is performed at the beginning of each superstep until the partitioning is *convergent*. We say a partitioning is convergent if the improvement achieved in the expected communication cost (Eq. 4.2) between two consecutive adaptations is within a user-defined threshold σ, after τ consecutive adaptation (Section 4.4.1.4).

Each of such adaptations has two phases: *logical vertex migration phase* (Phase-1) and

Algorithm 4: Phase-1a: Vertex Migration

```
Data: P_l, c
identify boundary vertices of P_l
foreach boundary vertex v ∈ P_l do
    optimal migration destination selection
foreach boundary vertex v ∈ P_l do
    marked v as moved with a probability proportional to the gain
```

physical vertex migration phase (Phase-2). Phase-1 attempts to improve the decomposition by logically migrating vertices among the partitions while considering the communication heterogeneity. Logically means that we only locally *mark* vertices chosen by PLANAR for migration as if they were moved. Phase-2 (Section 4.4.1.3) is responsible for the actual vertex and application data migration. Phase-1 is further split into two sub phases: Phase-1a and Phase-1b. Phase-1a (Section 4.4.1.1) tries to improve the decomposition in terms of communication cost as much as possible. Phase-1b (Section 4.4.1.2) aims to improve the decomposition in terms of load distribution without significantly increasing the communication cost of the decomposition determined in Phase-1a.

4.4.1.1 Phase-1a: Minimizing Communication Cost

In this phase, each server runs an instance of Algorithm 4 *in parallel* to decide which vertices should be moved out from its local partition and which partition should each vertex migrate to, such that both the communication and migration cost are minimized. The input to the algorithm includes the local partition P_l owned by each server and the relative network communication cost matrix c. The algorithm first tries to identify vertices of P_l having neighbors in other partitions (boundary vertices). Then, each boundary vertex independently selects the partition leading to a maximal gain as its optimal migration destination. Afterwards, boundary vertices are locally marked with a migration probability that is proportional to their gain.

Architecture-Aware Vertex Gain Computation The gain of moving a vertex, v, from its current partition to an alternative partition is defined as the reduction in the communication cost. The communication cost consists of two parts: the communication that v would incur during the computation and the cost of migrating v. The communication cost that v

would incur during the computation when it is placed in P_i is defined as:

$$comm(v, P_i) = \alpha * \sum_{k=1 \text{ and } k \neq i}^{n} d_{ext}(v, P_k) * c(P_i, P_k) \tag{4.14}$$

where $c(P_i, P_k)$ indicates the relative network communication costs between P_i and P_k, whereas $d_{ext}(v, P_k)$ represents the amount of data that v communicates with vertices of P_k and is further defined as:

$$d_{ext}(v, P_k) = \sum_{e=(u,v) \in E \text{ and } u \in P_k} w(e) \tag{4.15}$$

Here, $w(e)$ is the edge weight, indicating the amount of data communicated along the edge in a single computation superstep. The cost of migrating v from its current partition P_i to another partition P_j is defined as:

$$mig(v, P_i, P_j) = vs(v) * c(P_i, P_j) \tag{4.16}$$

Here, $vs(v)$ denotes the amount of application state associated with the vertex, indicating the cost of migrating the vertex. Hence, the gain of migrating v from P_i to P_j is:

$$g^{i,j}(v) = comm(v, P_i) - comm(v, P_j) - mig(v, P_i, P_j) \tag{4.17}$$

In case of $P_i = P_j$, $g^{i,j}(v)$ becomes 0. If $g^{i,i}(v)$ happens to be maximal, v will choose to stay. Clearly, migrating non-boundary vertices of P_l to other partitions would not lead to any gain since they only communicate with vertices of P_l.

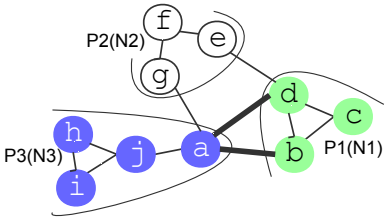


Figure 4.20: Old Decomposition

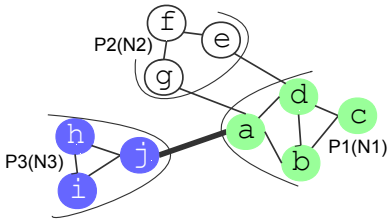


Figure 4.21: Better Decomposition

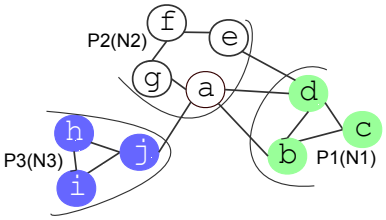


Figure 4.22: Best Decomposition

Migration Destination Selection Example (Figures 4.20–4.22) Consider a decomposition given by Figure 4.20 with three partitions and unit weights and sizes, and the relative

Table 4.11: Relative network communication costs

	N_1	N_2	N_3
N_1		1	6
N_2	1		1
N_3	6	1	

network communication costs among the partitions as shown in Table 4.11. Now, let us examine how vertices in P_3 make their migration decisions with $\alpha = 1$ (equal importance of communication and migration costs). Take for example vertex a, the only boundary vertex of P_3. Clearly, the gain of moving a from P_3 to P_1 (Figure 4.21) and to P_2 (Figure 4.22) is 0 and 9, respectively, since $comm(a, P_3) = 13$, $comm(a, P_1) = 7$, $comm(a, P_2) = 3$, $mig(a, P_3, P_1) = 6$, and $mig(a, P_3, P_2) = 1$. Thus, vertex a would select P_2 as its migration destination. On the other hand, architecture-agnostic repartitioners would choose the decomposition of Figure 4.21 over Figure 4.22 due to its lower edgecut (3 vs 4).

Cross-Partition Migration Interference As is evident, the gain of migrating a vertex from its current partition to another partition heavily relies on the amount of data that the vertex communicates with its neighbors in other partitions. For example, in Figure 4.20, the amount of data communicated between vertex a and P_1 contributes most to the gain of moving a to P_2. However, due to the independent nature of the migration decisions, neighbors of vertex a that are in P_1 may decide to migrate to other partitions. Consequently, the gain of moving vertex a to P_2 may no longer exist.

To mitigate this cross-partition migration interference, each vertex u is migrated with a probability proportional to the gain they may introduce. Considering the gain of migrating the vertices may vary significantly. A vertex, v, is migrated from its current partition P_i to its optimal destination P_j with a probability of

$$0.5 + \begin{cases} -1 * \frac{avgGain}{g(v,P_i,P_j)} * 0.05 & g(v, P_i, P_j) < avgGain \\ \frac{g(v,P_i,P_j)}{avgGain} * 0.05 & Otherwise \end{cases} \tag{4.18}$$

where *avgGain* denotes the average of the gain of migrating vertices of my P_i to their optimal migration destinations. In this way, vertices having a higher possible gain are more likely to be migrated (maximizing the chance of performance improvement), and vice versa. This also reduces the chance of migrating a high-degree vertex, since the gain of migrating a high-degree vertex is often small according to our gain heuristic given its large neighborhood.

Analysis As presented, each vertex only needs to know the locations of its neighbors and the amount of data it communicates with each partition for the migration decisions. The former is readily available to each partition in real-world systems for neighboring vertices to communicate with each other, while the latter can be locally computed. Each vertex only has to examine the accumulated weights of its edges that have one endpoint in another partition. Clearly, Phase-1a is lightweight, since it does not require any global coordination.

Also, Algorithm 4 only requires two arrays of size $O(n)$ and $O(|V_l|)$ to store the information about the amount of data a vertex communicates with each partition and the information about boundary vertices. Here, n denotes the number of partitions and $|V_l|$ is the number of (boundary) vertices of each partition. The time complexity of Algorithm 4 is $O(|E_l| + n^2 * |V_l|)$ with E_l denoting the edge set of each partition, because the identification of boundary vertices takes $O(E_l)$ and the selection of optimal migration destination for boundary vertices takes $O(|E_l| + n^2 * |V_l|)$.

4.4.1.2 Phase-1b: Ensuring Balanced Partitions

Since each partition makes its migration decisions independently in Phase-1a, vertices in different partitions may decide to migrate to the same partition, leading to load imbalance. To ensure a balanced load distribution, we carry out another quota-based vertex migration phase (if necessary), where we only allow a limited number of vertices to be migrated from each overloaded partition to each underloaded one. To achieve this, PLANAR needs to decide: (1) How much work should P_i migrate to P_j? and (2) What vertices should P_i move to P_j?

To resolve our first question, we first compute the amount of work that needs to be moved out from each overloaded partition:

$$Q(P_i) = w(P_i) - C(P_i) \tag{4.19}$$

Algorithm 5: Phase-1b: Quota Allocation

Data: P_l, Q, c
Result: $quota^l$
1 load information exchange
2 potentialGainCompute(P_l, Q, c, pg)
3 insert P_i, P_j and $pg(P_i, P_j)$ into a heap sorted by the gain
4 **foreach** *popped partition pair* P_i *and* P_j **do**
5 quota[i][j] = max {0, min {$Q(P_i)$, -$Q(P_j)$}}
6 update $Q(P_i)$ and $Q(P_j)$
7 quota^l[i][j] = quota[i][j] * λ

where $w(P_i)$ is the aggregated weight of vertices in P_i and $C(P_i)$ denotes the maximal load that P_i can have. $C(P_i) = (1 + \varepsilon) * \frac{\sum_{i=1}^{n} w(P_i)}{n}$ with ε denoting the user-defined imbalance tolerance. Clearly, $-Q(P_i)$ corresponds to the remaining capacity of P_i.

Architecture-Aware Quota Allocation Algorithm 5 describes how PLANAR distributes the remaining capacity of each underloaded partition across the overloaded ones. It is an iterative, architecture-aware quota allocation algorithm. During each iteration, the algorithm attempts to find a single partition pair, (P_i, P_j), such that allocating as much quota as possible from the underloaded partition, P_j, for the overloaded partition, P_i, would lead to a maximal gain. To do this, PLANAR first computes the potential gain of migrating vertices of each overloaded partition to each underloaded partition. The partition number of each partition pair is then inserted into a heap sorted by the potential gain. Then, PLANAR computes the quota allocation iteratively starting from the heap top. For each popped partition pair (P_i, P_j), P_j will allocate $quota[i][j] = max\{0, min \{Q(P_i), -Q(P_j)\}\}$ quota share for P_i. $quota[i][j] = 0$ indicates that either P_i is already balanced or the remaining capacity of P_j is 0. Upon each allocation, $Q(P_i)$ is also updated to reflect the allocation. This process is repeated until all the partitions are balanced.

Thanks to Phase-1's vertex migration, each server may hold a vertex portion of P_i, requiring $quota[i][j]$ to be properly distributed across the servers. Here, we take a simple yet effective approach (line 7), where $quota[i][j]$ is distributed across the servers proportionally to the amount of work of P_i held by each server. To this end, each server first exchanges the amount of work (vertices) it migrated to every other server with each other. By doing this, each server knows exactly how much work it imports from other partitions. Let $IW(P_i)$

denote the amount of work server M_i/P_i imported from others. If $IW(P_i) \geq Q(P_i)$, each server can simply scale $quota[i][j]$ by $\frac{w^l(P_i)}{IW(P_i)}$, where $w^l(P_i)$ denotes the amount of work of P_i held by each server. In case of $IW(P_i) < Q(P_i)$, $quota[i][j]$ is scaled by $1 - \frac{IW(P_i)}{Q(P_i)}$ for P_i and by $\frac{w^l(P_i)}{Q(P_i)}$ for others.

Potential Gain Computation The potential gain of migrating vertices from an overloaded partition P_i to an underloaded partition P_j is defined as:

$$pg(P_i, P_j) = \sum_{v \in P_i} g^{i,j}(v) \tag{4.20}$$

Each server only needs to consider migrating boundary vertices of the overloaded partitions to each underloaded ones, and only needs to count vertices that lead to positive gain for $pg(P_i, P_j)$. To facilitate our next step's vertex migration, we maintain a sorted heap to keep track of the gain of migrating each vertex to each possible migration destination here.

Analysis As presented, Phase-1b only requires a small amount of global coordination to compute the load distribution for quota allocation decisions. In addition to this, Algorithm 5 can be run in parallel on each server without coordination with other nodes. The time complexity of Algorithm 5 is $O(n * |V_l| + n^2)$, since the complexity of the partition pair potential gain computation phase (Line 2) and the final quota allocation phase (Line 3–7) are $O(n * |V_l|)$ and $O(n^2)$, respectively.

Also, Algorithm 5 only requires a small amount of additional memory, including two arrays of size n (for $Q(P_i)$ and $d_{ext}(v, P_j)$), one $n * n$ matrix (for $pg(P_i, P_j)$), a heap of n^2 elements (to record the potential gain of each partition pair), another heap of size $n * |V_l|$ (to keep track of the gain of migrating boundary vertices of the overloaded partitions to all the possible migration destinations), and another $n * n$ matrix (for the quota allocation result).

Given the quota allocation, each overloaded server knows how much work it should migrate to each underloaded partition. Along with the sorted heap we maintained while computing the potential gain, we can easily figure out the vertices to migrate and their optimal migration destinations, which is described by Algorithm 6. Clearly, Algorithm 6 does not require any global coordination, and its time complexity is $O(n * |V_l|)$. This indicates that our Phase-1b vertex migration is also lightweight.

Algorithm 6: Phase-1b: Vertex Migration

```
Data: P_l, quota, sortedHeap
for i = 0 → size(sortedHeap) do
    HeapGet(sortedHeap, i, &v, &dest, &gain)
    if v's current owner o(v) is overloaded then
        if quota[o(v)][dest] > 0 then
            mark v as moved to the dest partition
            update Q(o(v)) and quota[o(v)][dest]
```

4.4.1.3 Phase-2: Physical Vertex Migration

Based on the result of Phase-1 vertex migration, PLANAR will physically migrate vertices that were chosen to move out to their destinations (including the associated application data). For example, in SSSP, each vertex often maintains two fields: $\{prev(v), dist(v)\}$, where $prev(v)$ is the vertex preceding v on the current shortest path and $dist(v)$ is the length of the current shortest path [60]. To ensure correctness, we also need to migrate these two fields along with the vertex. Clearly, physical vertex migration is highly application-dependent and developing a general-purpose solution is out of the scope of this work. Hence, the output of PLANAR will simply be an array indicating the new location of each vertex, based on which the physical migration can be performed either using a customized migration service or a general migration service (like the one provided by Zoltan [36]).

4.4.1.4 Phase-3: Convergence

To avoid unnecessary execution of PLANAR at the beginning of each superstep, we check if the partitioning converges and discontinue PLANAR if does. However, PLANAR can be re-enabled in the presence of sufficient load imbalance and graph dynamism. We define as *convergent* the state where the improvement achieved by each adaptation in terms of the communication cost is within a user-defined threshold σ after τ consecutive adaptations. Normally, the partitioning converges quickly, since each adaptation usually produces a better partitioning and after a certain point the partitioning could not be further improved (Section 4.4.2.2).

However, there may exist cases where the improvement achieved never meets the threshold, or it oscillates around the threshold. To eliminate this issue, we double σ every τ supersteps or once we detect two consecutive oscillations. We define as *oscillation* the sit-

uation where a newly computed partitioning fails to meet the threshold, but its immediate prior has met the threshold. In this way, the algorithm will always converge timely, thus reducing the overhead of PLANAR.

Also, there is a chance that PLANAR outputs a decomposition worse than its immediate prior during some adaptation supersteps, since vertex migration is performed using only local information available to each partition. One way to avoid this is to rollback the movements we made. However, to do this we have to put the convergence check before the physical data migration phase. As a result, each server would first need to exchange the up-to-date vertex locations with each other, because each vertex needs to know the up-to-date vertex locations of their neighbors for convergence check, leading to additional coordination overhead. In contrast, if we put the convergence check after the physical data migration phase, we can combine the vertex location updates along with the updates of other application data (i.e., the mapping of global vertex identifiers to local vertex identifiers[3]), thus reducing the communication overhead. Furthermore, the rollback may be an overreaction, because these movements may lead to a big performance improvement in the following adaptation supersteps. Besides, we only observed this negative performance impact in few adaptation supersteps on the datasets we tested and the deterioration was very small (less than 1%). This has convinced us that it is not beneficial to tackle this issue.

It should be noted that we assume that the changes in the graph during each of PLANAR's adaptation supersteps is not drastic. This is a reasonable assumption, since repartitioning is performed in a periodic manner in real-world scenarios.

4.4.1.5 Incorporating Contention-Awareness

To make PLANAR also aware of the issue of shared resource contention on the memory subsystems, we adopt the same solution as ARGO, where we penalize intra-node network communication costs by a score. The score is computed based on the degree of contentiousness between the communication peers. By doing this, the amount of intra-node communication will decrease accordingly. The parameter λ can be used to specified the degree of contention. If $\lambda = 0$, PLANAR will only

[3]In distributed graph computation, each vertex has one global identifier unique across partitions and one local identifier unique within each partition.

consider the communication heterogeneity, whereas $\lambda = 1$ means that intra-node shared resource contention is the biggest bottleneck and should be prioritized over the communication heterogeneity. It should be noticed that PLANAR 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.

4.4.2 PLANAR: Evaluation

4.4.2.1 Setup In this section, we first evaluate the sensitivity of PLANAR to (a) its two important parameters (Section 4.4.2.2) and (b) varying input decompositions computed by different initial partitioners (Section 4.4.2.3). We then validate the effectiveness of PLANAR using two graph workloads: Breadth-First Search (BFS) [59] and Single-Source Shortest Path (SSSP) [60] (Section 4.4.2.4). Finally, we demonstrate the scalability of PLANAR using a billion-edge graph (Section 4.4.2.5). Towards this, we implemented the two workloads and a prototype of PLANAR using MPI [46, 47].

Datasets Table 4.7 describes the datasets used. By default, the graphs were (re)partitioned with both the vertex weights (i.e., computational requirement) and vertex sizes (i.e., amount of the data of the vertex) set to their vertex degree. Their edge weights (i.e., amount of data communicated) were set to 1. Vertex degree is a good approximation of the computational requirement and the migration cost of each vertex, while an edge weight of 1 is a close estimation of the communication pattern of BFS and SSSP. Considering the communication cost is more important than migration cost, all the experiments were performed with $\alpha = 10$ (Eq. 4.2). Unless explicitly specified, the graphs were initially partitioned by the deterministic greedy heuristic, DG [8], across cores of the machines used (one partition per core). The partitionings were then improved by PLANAR until it converges. During the (re)partitioning, we allowed up to 2% load imbalance among partitions. It should be noted (a) that DG/LDG were extended to support vertex- and edge-weighted graphs for fair comparison; and (b) that vertices of the graphs were presented to DG/LDG in some unknown order.

Platforms We evaluated PLANAR on two clusters: MPICluster [65] and Gordon supercomputer [50]. MPICluster had a flat network topology, where all the 32 compute nodes were connected to a single switch via 56Gbps FDR Infiniband. On the other hand, the Gordon network topology was a 4x4x4 3D torus of switches connected via QDR Infiniband with 16 compute nodes attached to each switch (with 8Gbps link bandwidth). Table 4.8 depicts the compute node configuration of both clusters. All results presented were the means of 5 runs, except the execution of SSSP on Gordon.

Network Communication Cost Modelling The relative network communication costs among the partitions were approximated using a variant of osu_latency benchmark [66]. To ensure the accuracy of the cost matrix, we bound each MPI rank (process) to a core using the options provided by OpenMPI 1.8.6 [46] on MPICluster and MVAPICH2 1.9 [47] on Gordon. OpenMPI and MVAPICH2 were two different MPI implementations available on the clusters.

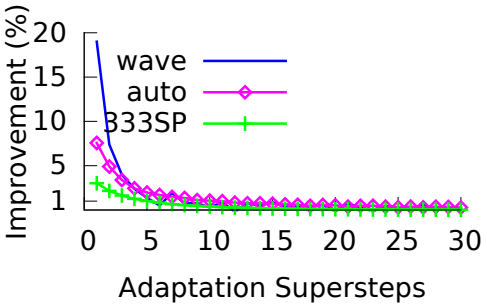


Figure 4.23: PLANAR parameter selection

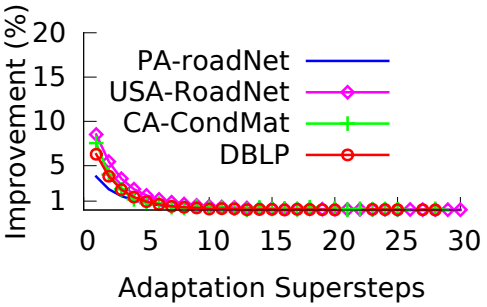


Figure 4.24: PLANAR parameter selection

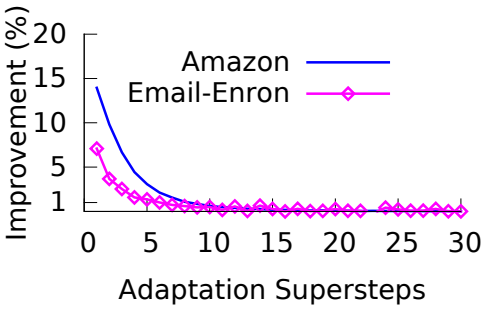


Figure 4.25: PLANAR parameter selection

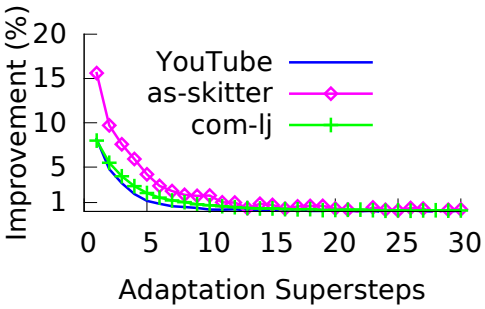


Figure 4.26: PLANAR parameter selection

4.4.2.2 Parameter Selection

Configuration This experiment studied the sensitivity of PLANAR to its two critical parameters: σ and τ (Section 4.4.1.4). Theoretically, σ should be a value *large enough*, so that PLANAR can converge quickly, especially for decompositions that it cannot improve much. Also, it should be *small enough*, offering PLANAR sufficient time to refine graph decompositions with large improvement space. Towards this, we applied PLANAR to various graph decompositions computed by the deterministic greedy (DG) partitioner across cores of two 20-core compute nodes for 30 consecutive adaptation supersteps, and examined the improvement achieved by PLANAR in terms of communication cost in each adaptation superstep (against the input decomposition to each adaptation superstep).

Results Figures 4.23 to 4.26 present the corresponding results. Interestingly, we found that most of the improvements were achieved in the first 5 adaptation supersteps. After that, the improvement achieved in each adaptation superstep dropped quickly below 1%, and asskitter and Email-Enron were the only two datasets exhibiting some small oscillations. Thus, in our implementation, we set σ and τ to 1% and 10, respectively, and do not perform any convergence check for the first 5 adaptation supersteps.

4.4.2.3 Microbenchmarks

Configuration This experiment examined the effectiveness of PLANAR in terms of partitioning quality (Eq. 4.2 and 4.3), when it was provided by various decompositions computed by HP, DG, LDG, and METIS. HP is the default graph partitioner used by many parallel graph computing engines; DG and LDG are two state-of-the-art streaming graph partitioning heuristics [8]; and METIS is a state-of-the-art multi-level graph partitioner [32]. The graphs were initially partitioned across two 20-core compute nodes on MPICluster.

Quality of the Initial Decompositions (Figure 4.27) Figure 4.27 presents the initial communication costs of the decompositions computed by HP, DG, LDG, and METIS for a variety of graphs in log-scale. As expected, METIS performed the best and HP was the worst. However, METIS is a heavyweight serial graph partitioner, making it infeasible for large-scale distributed graph computation either as an initial partitioner or as an online repartitioner

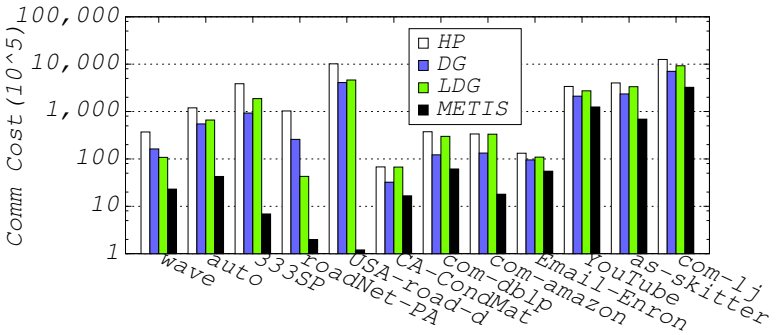
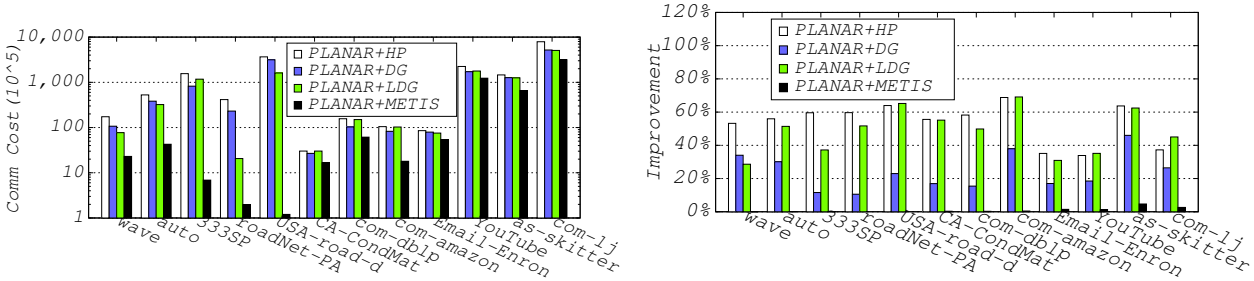


Figure 4.27: Communication costs of the initial decompositions partitioned by HP, DG, LDG, and METIS into 40 partitions.

(repartitioning from scratch). It was reported in [10] that METIS took 8.5 hours to partition a graph with 1.46 billion edges. Surprisingly, DG performed better than LDG, the best streaming partitioning heuristic among the ones presented in [8]. This was probably because the order (some unknown random order) in which vertices were presented to the partitioner favored DG over LDG, since the results of streaming partitioning heuristics rely on the order in which vertices are presented to them.



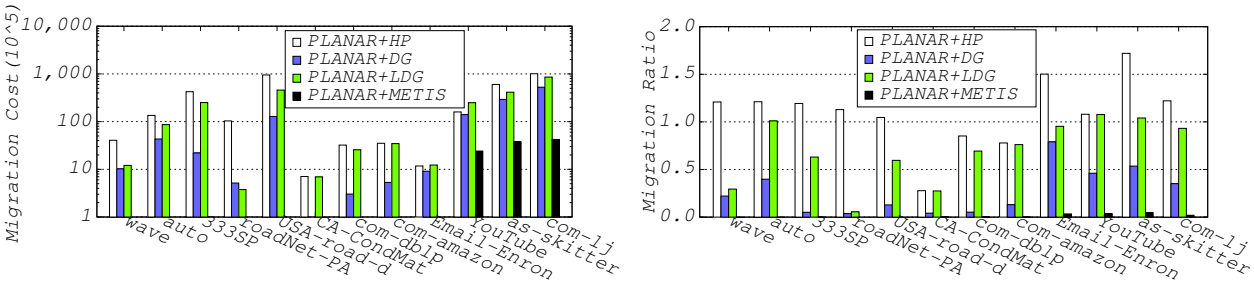
(a) Communication Cost

(b) Improvement Achieved

Figure 4.28: Communication cost of the resulting decompositions and improvement achieved after running PLANAR over varying initial decompositions generated by HP, DG, LDG, and METIS across two 20-core machines.

Quality of the Resulting Decompositions (Figures 4.28a & 4.28b) Figures 4.28a and 4.28b, respectively, plot the log-scale communication cost of resulting decompositions and the improvements achieved by PLANAR in terms of communication cost against the initial decompositions. As shown, the better the initial decomposition was the better the resulting decomposition would be, and PLANAR reduced the communication cost of decom-

positions computed by HP, DG, and LDG by up to 68%, 46%, and 69%, respectively, whereas it only slightly improved the decompositions computed by METIS. One reason for this is that METIS usually produces decompositions much better than others, providing PLANAR limited improvement space. Yet, PLANAR still achieved an improvement by up to 4.6% for complex networks (right 5 datasets) against METIS. On the other hand, this also showed the stability of PLANAR, since it did not deteriorate any decompositions computed by METIS. Also, we found that PLANAR with DG as its initial partitioner can achieve even better performance than METIS in real-world workloads (Section 4.4.2.4).



(a) Migration Cost (b) Accumulated Vertex Migration Ratio

Figure 4.29: Overhead of the adaptation on varying initial decompositions computed by HP, DG, LDG, and METIS into 40 partitions.

Migration Cost (Figures 4.29a & 4.29b) In the experiment, we also examined the migration cost introduced by PLANAR in terms of Eq. 4.3 and the accumulated vertex migration ratio (# of vertices migrated as a percentage of the entire graph) across all the adaptation supersteps. Figures 4.29a and 4.29b present the corresponding results. As shown, the better the initial decomposition was, the lower the migration cost was. The reason why the migration ratio exceeded 1 in some cases was because each vertex may be migrated multiple times during the repartitioning. We also observed that PLANAR improved the decompositions computed by DG only with a very small amount of data migration for most of the datasets. Also, PLANAR only led to a very small amount of data migration for decompositions with limited improvement space, further demonstrating the stability of PLANAR.

Convergence Time (Figure 4.30) Another item of interest in this experiment is the average number of supersteps PLANAR took to converge (Figure 4.30). As presented, for graph decompositions that have limited improvement space, PLANAR only took around 8

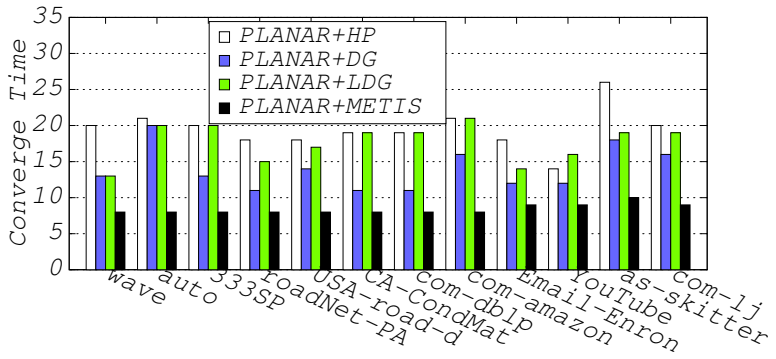


Figure 4.30: PLANAR converge time in terms of supersteps

supersteps to converge. In contrast, graph decompositions with large improvement space were provided with sufficient time. This further validated the robustness of σ and τ's default values. The reason why the converge time dropped below 15 in some cases was because we made some additional optimizations to the convergence check phase.

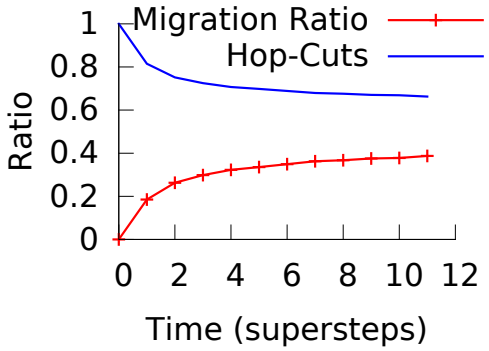


Figure 4.31: PLANAR convergence study on the wave dataset

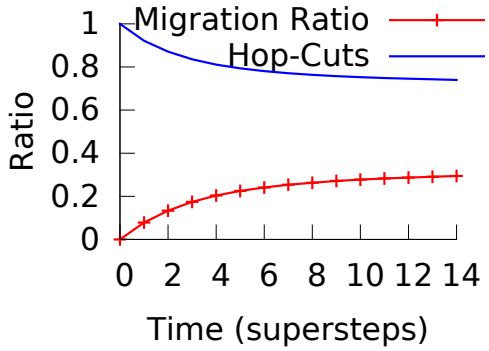


Figure 4.32: PLANAR convergence study on the com-lj dataset

Convergence Process (Figures 4.31 & 4.32) Another thing of interest is the exact converge process: the number of vertices migrated by PLANAR (with DG as its initial partitioner) during each adaptation superstep and the evolution of the corresponding hopcut across the adaptation supersteps. Figures 4.31 and 4.32 show the accumulated vertex migration ratio and the normalized hopcut (with the initial decomposition as the baseline) for the wave and the com-lj dataset, respectively. In both figures, superstep 0 corresponds to the initial decomposition. All the datasets followed the same pattern where PLANAR greatly reduced the hopcut in the first 5 adaptation supersteps, which were also the places where

most vertices got migrated.

4.4.2.4 Real-World Applications (BFS & SSSP)

Configuration This experiment evaluated PLANAR using BFS and SSSP on the YouTube, as-skitter, and com-lj datasets. Initially, the graphs were partitioned across cores of three machines of two clusters using DG. Then, the decomposition was improved by PLANAR until convergence. During the execution, we grouped multiple messages sent by each MPI rank to the same destination into a single one (8 for the YouTube and as-skitter dataset and 16 for the com-lj dataset). The reason why we picked 8 and 16 was because larger values would make the execution time too short, especially for the execution of BFS.

Resource Contention Modelling To capture the impact of resource contention, we ran a profiling experiment for BFS and SSSP with the three datasets on both clusters by increasing λ gradually from 0 to 1. Interestingly, we found that intra-node shared resource contention was more critical to the performance on MPICluster, while inter-node communication was the bottleneck on Gordon. This was probably caused by the differences in network topologies (flat vs hierarchical), core count per node (20 vs 16), memory bandwidth (65GB vs 85GB), and network bandwidth (56Gbps vs 8Gbps) of the two clusters, and that BFS/SSSP had to compete with other jobs running on Gordon for the network resource, while there was no contention on the network communication links on MPICluster. Hence, we fixed λ to be 1 on MPICluster and 0 on Gordon for our experiments.

Results in terms of Job Execution Time (Tables 4.12 & 4.13) Tables 4.12 and 4.13 show the execution time of BFS and SSSP with 15 randomly selected source vertices on the three datasets. The job execution time is defined as: $JET = \sum_{i=1}^{n} SET(i)$, where n corresponds to the number of supersteps the job has, while $SET(i)$ is the ith superstep execution time of the slowest MPI rank. In the table, DG and METIS mean that BFS/SSSP was performed on the datasets without any repartitioning/refinement, UNIPLANAR is a variant of PLANAR assuming homogeneous and contention-free computing environment (serving as a representative of the state-of-the-art adaptive solutions). We also show the overhead of each algorithm (in parentheses). Note that METIS is performed offline, and typically takes

Table 4.12: BFS job execution time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
MPICluster			
DG	21	79	221
METIS	5.28 (off)	66 (off)	23 (off)
PARMETIS	21 (21.92)	51 (9.75)	175 (4.89)
UNIPLANAR	10 (1.78)	36 (1.90)	109 (4.13)
ARAGON	8.99 (21.18)	13 (17.41)	55 (61.97)
PARAGON	9.03 (4.12)	12 (3.44)	67 (10.43)
PLANAR	7.95 (6.74)	8.76 (6.91)	21 (17.20)
Gordon			
DG	353	660	956
UNIPLANAR	222 (3.14)	217 (2.97)	587 (6.59)
ARAGON	240 (21.18)	238 (17.10)	501 (59.94)
PARAGON	217 (3.76)	248 (2.98)	558 (9.03)
PLANAR	166 (7.43)	205 (6.63)	477 (16.07)

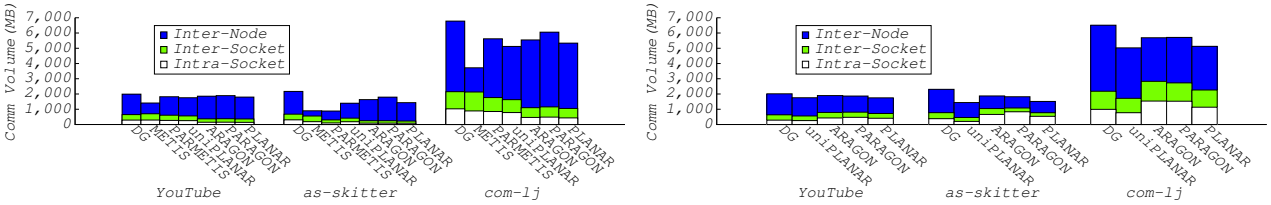
Table 4.13: SSSP job execution time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
MPICluster			
DG	2166	1754	4693
METIS	520 (off)	694 (off)	907 (off)
PARMETIS	1908 (21.91)	492 (9.70)	3055 (4.76)
UNIPLANAR	1128 (2.61)	615 (2.61)	2043 (5.47)
ARAGON	303 (21.26)	291 (16.95)	1283 (61.86)
PARAGON	405 (4.08)	312 (3.36)	1439 (10.38)
PLANAR	257 (7.68)	288 (7.08)	890 (18.76)
Gordon			
DG	3581	6517	11011
UNIPLANAR	2691 (4.62)	2184 (4.15)	7080 (9.04)
ARAGON	2874 (20.66)	3474 (15.41)	7395 (68.75)
PARAGON	2613 (3.85)	2741 (2.94)	7363 (9.03)
PLANAR	2322 (9.16)	2801 (8.11)	6381 (17.57)

a long time to complete (even hours for large graphs).

As expected, PLANAR beat DG, PARMETIS, and UNIPLANAR in almost all the cases. Compared to DG, PLANAR reduced the execution time of BFS and SSSP on Gordon by up to 69% and 57%, respectively, and by up to 90% and 88% on MPICluster, respectively. So, in the best case, PLANAR was 10 times better than DG. Yet, the overhead PLANAR exerted (sum of the repartitioning time and physical data migration time) was very small compared to the improvement it achieved and the job execution time. By comparing the results of UNIPLANAR with DG, we can conclude that PLANAR not only improved the mapping of the application communication pattern to the underlying hardware, but also the quality of the initial decomposition (edgecut). What we did not expect was that PLANAR, with DG as its initial partitioner, outperformed the gold standard, METIS, in 3 out the 6 cases and was comparable to METIS in other cases, and that PLANAR performed even better than both ARAGON and PARAGON. We attributed this to the greedy nature of our Phase-1 vertex migration.

Results in terms of Communication Volume Breakdown (Figures 4.33a & 4.33b) To further confirm our observations, we also measured the total amount of data remotely



(a) MPICluster (b) Gordon

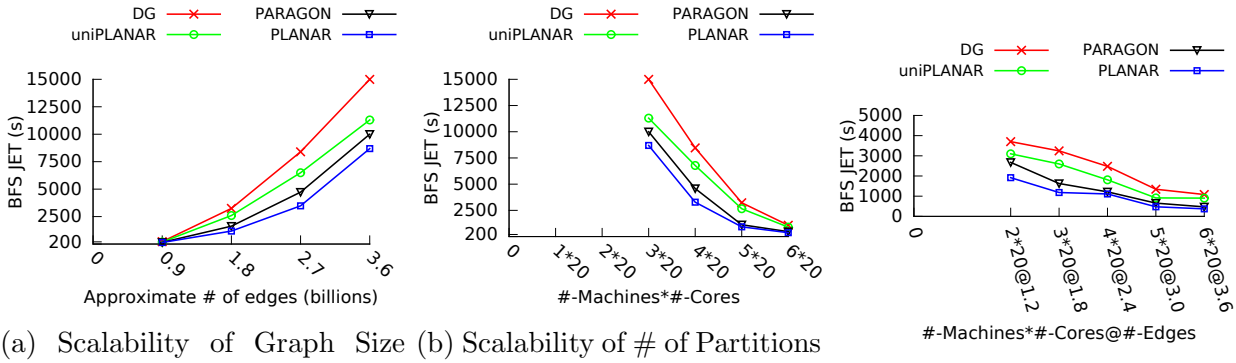
Figure 4.33: The communication volume breakdown of SSSP on both clusters.

exchanged per superstep by BFS and SSSP among cores of the same socket (intra-socket communication volume), among cores of the same compute node but belonging to different sockets (inter-socket communication volume), and among cores of different compute nodes (inter-node communication volume). Since we observed similar patterns for BFS and SSSP in all the cases, we only present the breakdown of the accumulated communication volume across all the supersteps for the execution of SSSP on both clusters here.

As shown in Figures 4.33a and 4.33b, comparing to the architecture-agnostic solutions (i.e., DG, Metis, ParMetis, and uniPlanar), Planar had the lowest intra-node (inter-socket & intra-socket) communication volume on MPICluster and lowest inter-node communication volume on Gordon. It should be noticed that on MPICluster intra-node communication was the bottleneck, and vice verse on Gordon. In comparison to Aragon and Paragon, Planar not only led to lower communication volume on critical components, but also had lower total remote communication volume. Another interesting thing was that, in spite of the higher total communication volume of the architecture-aware solutions (i.e., Aragon, Paragon, and Planar) when compared to Metis, ParMetis, and uniPlanar, architecture-aware solutions still outperformed them in most cases due to the reduced communication on critical components.

4.4.2.5 Billion-Edge Graph Scaling

Configuration This experiment investigated the scalability of Planar using the friendster dataset (3.6 billion edges) in three different setups: (1) Scalability of Graph Size; (2) Scalability of Number Partitions; and (3) Hybrid. In Setup 1, we demonstrated the scalability



(a) Scalability of Graph Size (3*20 Partitions)

(b) Scalability of # of Partitions (3.6B Edges)

(c) Hybrid Scalability

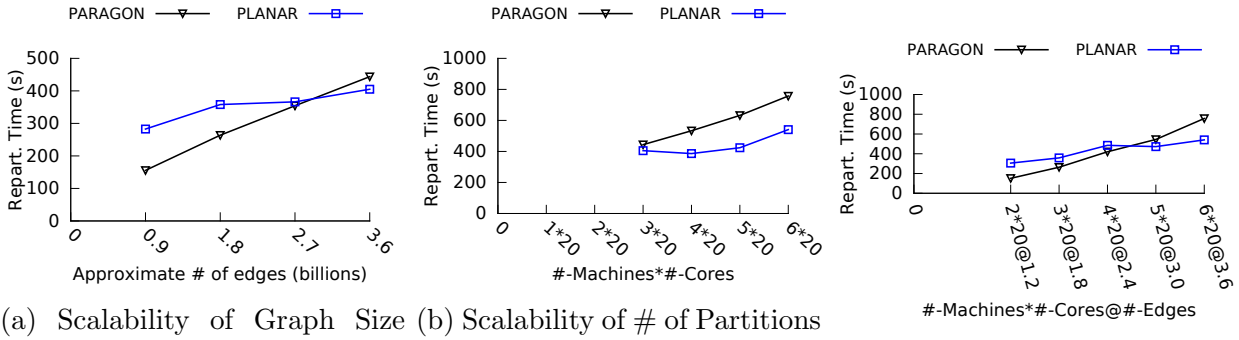
Figure 4.34: BFS Job Execution Time (JET)

of PLANAR as the graph scaled (from 0.9 up to 3.6 billion edges) but with a fixed number of partitions (60). In Setup 2, we showed the scalability of PLANAR using the original com-friendster dataset when it was partitioned into varying number of partitions (from 60 up to 120). In Setup 3, we exhibited the scalability of PLANAR as the number of partitions increased (from 40 up to 120) but with an approximately fixed number of edges per partition. That is, we varied the graph size accordingly (from 1.2 up to 3.6 billion edges) as the number of partitions increased.

Towards this, we generated some additional datasets by sampling the edge lists of the friendster dataset. We denoted the datasets as friendster-p, where p ($0 < p \leq 1$) was the probability that each edge was kept while sampling. Hence, friendster-p would have around $3.6 * p$ billion edges. Interestingly, the number of vertices remained almost unchanged in spite of the sampling. The experiment was performed on MPICluster with BFS message grouping size set to 256. We would only present the results of DG, PARAGON, UNIPLANAR, and PLANAR, since METIS, PARMETIS, and ARAGON failed to (re)partition the graphs even for the smallest graph of this experiment, due to their heavyweight nature.

Results in terms of BFS Execution Time (Figures 4.34) Figures 4.34 plots the BFS execution time with 15 randomly selected source vertices in different setups. As shown, PLANAR had the lowest BFS execution time in all the cases. We also noticed that in Setup 1 (Figure 4.34a), PLANAR had the lowest speed in which the BFS execution time increased as the graph scaled, and that in Setup 2 & 3, the more the machines used, the faster BFS

completed. Interestingly, we found that the improvement achieved by PLANAR gradually decreased as the number of partitions increased. This was probably because the fraction of intra-node communication dropped greatly as the number of partitions increased due to the increasing inter-node communication peers, weakening the impact of architecture-awareness on MPICluster. Even though the improvement decreased, PLANAR still achieved up to 2.9x speedups with 6 machines (Setup 2). It should be noted that PLANAR reduced the execution time of all the computing elements (6*20 cores) by this much not just one.



(a) Scalability of Graph Size (3*20 Partitions)

(b) Scalability of # of Partitions (3.6B Edges)

(c) Hybrid Scalability

Figure 4.35: Repartitioning Time

Results in terms of Repartitioning Time (Figures 4.35) Figure 4.35 shows the corresponding repartitioning time of PLANAR and PARAGON. As shown, PLANAR's repartitioning time increased at a much slower rate than that of PARAGON in all the setups. The reason why PLANAR had higher repartitioning time for smaller graphs was because PLANAR requires a migration phase at the end of each adaptation superstep (the major source of the overhead). Fortunately, as the graph and the deployment scale increased, PLANAR was the clear winner. This was because PARAGON requires more knowledge about the graph for repartitioning and has lower degree of repartitioning parallelism. In fact, if we average the repartitioning time across the adaptation supersteps, the overhead introduced by PLANAR in each adaptation superstep would be very small.

4.4.3 PLANAR+: Optimized PLANAR

In this section, we introduce three major optimizations made by PLANAR+ to further reduce the repartitioning overhead of PLANAR. The main optimizations include: (a) the elimination of per adaptation superstep physical vertex migration; (b) an optimized relative network communication costs measuring method; and (c) an optimized vertex gain computation algorithm.

Algorithm 7: PLANAR+ Full Repartitioning

```
   Data: P_l, c, σ, τ
1  while the partitioning has not converged do
2      // Phase-1 (Section 4.4.1.1 & 4.4.1.2)
3      LogicalVtxMigration(P_l, c, &pv)
4      // Vertex Location Update (Section 4.4.3.1)
5      VertexLocationUpdate(P_l, pv)
6      // Convergence Check (Section 4.4.1.4)
7      CheckPartitionConvergence(σ, τ)
8  // Phase-2 (Section 4.4.1.3)
9  PhysicalVtxMigration(P_l, pv)
```

4.4.3.1 Eliminating Per Adaptation Superstep Physical Vertex Migration In our previously published work, PLANAR repartitions the graph in an adaptive manner, where it (Algorithm 3) is performed at the beginning of each computation superstep until the partitioning converges. As a result, it requires a physical migration phase at the end of each adaptation superstep, which could potentially increase the repartitioning overhead. To address the issue, PLANAR+ provides an alternative *full repartitioning* mode. Algorithm 7 describes the whole process of full repartitioning. In the full repartitioning mode, PLANAR+ (Algorithm 7) only needs to be executed once and will automatically adapt the graph continuously until the partitioning converges. As can be seen, we eliminate the need of the physical vertex migration during each adaptation superstep (Line 2–7) and only require one physical vertex migration at the end of the whole reparitioning process (Line 9). However, we do need an update of the vertex location in each adaptation superstep (Line 5), because each vertex needs to know the up-to-date locations of its neighbors for the convergence check as well as the execution of the next adaption superstep.

To support efficient vertex location update, we choose one MPI process as the root.

The root maintains an array to keep track of the locations of all the vertices, with $A[i]$ specifying the location of vertex v_i. Note that we could potentially increase the number of root processes and let each root process take responsibility for a small range of vertices, in cases where one root process could not hold the locations of all the vertices in memory. Then, each MPI process takes advantages of the MPI one-sided data communication [77] to update the locations of vertices at the end of Phase-1 vertex migration. MPI one-sided data communication allows each MPI process to directly read/write a dedicated memory region of the root process potentially more efficiently than regular two-sided data communication (e.g., MPI collective and point-to-point operations).

Once the root process has the up-to-date vertex locations, it will broadcast the locations to all the processes, such that each partition can update the location information for its neighboring list. To avoid memory exhaustion, we divide the vertex neighbor location update into multiple rounds. Within each round, the root process will only broadcast locations of vertices within a fixed size contiguous range. Correspondingly, the MPI processes will only update the locations of their vertex neighbors of the specified range in each round. For example, for a graph with $|V|$ vertices, we may choose to only update the locations of vertices whose IDs are in the range of $[i * \frac{|V|}{R}, (i+1) * \frac{|V|}{R})$ in the ith round. Here, R is the total number of rounds required and $i \in [0, R)$. We also note that the performance of MPI one-sided data communication may vary greatly for different MPI implementations. Towards this, PLANAR+ also provides a fall back solution that is proposed in our previous work [22] for the vertex location update.

4.4.3.2 Optimizing Network Communication Cost Measurement

The effectiveness of PLANAR/PLANAR+ relies on a fairly accurate measurement of the relative network communication costs among the computing elements. This section describes how we measure the relative network communication costs among the computing elements.

Possible Solutions: Clearly, one straightforward approach would be measuring the relative network communication costs for all the communication peers quantitatively. The measurement for a single pair can be achieved via a sequence of pingpong messages with varying message sizes like benchmark [66] does. The problem of such a solution is that the

measurement may take very long time to finish. For example, to measure the relative network communication costs among processes running on five 20-core machines (one process per core), we have to measure the costs for $\frac{100*99}{2} = 4950$ pairs. Assuming the measurement of a single pair takes 1s, 4950 pairs would take more than an hour, leading to significant resource waste.

On the other hand, we could also measure the relative network communication costs qualitatively using the knowledge of the network topology. For example, we could assign a cost of 1 for process pairs running on cores of the same CPU socket, a cost of 2 for process pairs running on cores of different CPU sockets but on the same machine, a cost of 3 for process pairs running on cores of different machines but connected to the same switch, and a cost of 4 for process pairs running on cores of different machines connected via two switches. Although the measurement is extremely fast, the information about the network topology may not always be available. Besides, manually assigning the cost also compromises the accuracy.

Proposed Solution: Towards this, PLANAR+ proposes a hybrid approach where we still measure the relative network communication costs quantitatively but with the help of a minimum amount of topology information to speed up the measurement. Specifically, we first categorize the relative network communication costs into three types: *inter-node, inter-socket, and intra-socket* network communication costs. Inter-node network communication costs denote the communication costs among processes running on cores of different machines, inter-socket network communication costs correspond to the communication costs among processes running on cores of the same machine but on different CPU sockets, and intra-socket communication costs are the communication costs among processes running on cores of the same CPU socket.

With such a categorization, we only need to select one process as a representative for the processes running on cores of the same machine to measure the relative inter-node network communication costs (the relative network communication costs among the selected processes). Similarly, to measure the relative inter-socket network communication costs, we only have to pick one process as a representative for all the processes running on cores of the same CPU socket. Note that the measurement of inter-socket costs of different machines

can be performed in parallel. For processes running on cores of the same CPU socket, we only measure the cost for one process pair and use the cost for all the process pairs running on the same CPU socket. Clearly, our proposed approach only requires the knowledge of the machine architecture. Such information can be easily obtained via the HwLoc [78] library.

Benefits of the Proposed Solution: By measuring the relative network communication costs in this way, we significantly reduce the number of communication peers we have to measure. For our previous example with 100 processes running on five 20-core machines (10 cores per CPU socket), we only have to measure 25 pairs in total: $\frac{5*4}{2} = 10$ pairs for the inter-node network communication costs, 5 pairs for the inter-socket network communication costs (one pair per machine), and 10 pairs for the intra-socket network communication costs (1 pair per socket). An additional benefit of this is that with such a significant decrease in the number of pairs to be measured, we can spend more time in measuring the per pair network communication costs to further increase the accuracy.

4.4.3.3 Optimizing Vertex Gain Computation

In addition to making the measurement of the relative network communication costs faster, measuring the network costs in this way also provides us a way to speed up our Phase-1a vertex migration (Algorithm 4 & Section 4.4.1.1). Algorithms 8 and 9 show how PLANAR implements Line 3 of Algorithm 4. As illustrated, the time complexity for a vertex, v, to find its optimal migration destination is $O(d(v)+n^2)$, where $d(v)$ and n, respectively, denote the degree of the vertex and the number of partitions. This is because the computation of $d_{ext}(v, P_k)$ (Line 2–4 of Algorithm 8) takes $O(d(v))$ time and the optimal migration destination selection (Line 5–9 of Algorithm 8) takes $O(n^2)$.

As can be seen, computing the gain of moving v from its current partition P_i to P_j (Algorithm 9) always takes $O(n)$ time regardless of where P_j may be. This is suboptimal. For example, in cases where P_i and P_j are assigned to cores of the same CPU socket, the amount of data that v communicates with any other partitions has no impact on the gain of moving v from P_i to P_j. This is because, in such cases, $c(P_i, P_k)$ and $c(P_j, P_k)$ are exactly the same for any P_k other than P_i and P_j. Because of this, the difference between $comm(v, P_i)$

Algorithm 8: Phase-1a: Migration Destination Selection

```
Data: v, c
//implementation of Equation 4.15
foreach u ∈ {Neighbors of v} do
    //P_k is the partition that u currently belongs to
    d_ext(v, P_k) += weight of edge (u, v)
foreach Partition P_j where j ∈ [1, n] do
    //P_i is the partition v that currently belongs to
    g^{i,j}(v) = VertexGainComputation(v, P_i, P_j, d_ext, c)
    if g^{i,j}(v) is greater than maxGain then
        update maxGain and the optimal migration destination
return maxGain and the optimal migration destination
```

Algorithm 9: PLANAR: Vertex Gain Computation

```
Data: v, P_i, P_j, d_ext, c
comm(v, P_i) = 0
comm(v, P_j) = 0
//implementation of Equation 4.14
foreach Partition P_k where k ∈ [1, n] do
    comm(v, P_i) += d_ext(v, P_k) * c(P_i, P_k)
    comm(v, P_j) += d_ext(v, P_k) * c(P_j, P_k)
//implementation of Equation 4.17
gain = (comm(v, P_i) - comm(v, P_j)) * α - mig(v, P_i, P_j)
return gain
```

and $comm(v, P_j)$ becomes

$$(d_{ext}(v, P_j) - d_{ext}(v, P_i)) * c(P_i, P_j) \tag{4.21}$$

In other words, we reduce the computation of $comm(v, P_i)$ to

$$d_{ext}(v, P_j) * c(P_i, P_j) \tag{4.22}$$

and $comm(v, P_j)$ to

$$d_{ext}(v, P_i) * c(P_i, P_j) \tag{4.23}$$

both of which can be computed in $O(1)$ time.

Similarly, in cases where P_i and P_j are on cores of the same machine but of different CPU sockets, the amount of data that v communicates with partitions that are residing on other machines has no impact on the gain of moving v from P_i to P_j, since $c(P_i, P_k)$ and

$c(P_i, P_k)$ are exactly the same for any P_k that are on other machines. In other words, we only have to consider the partitions that are assigned to the same machine as P_i and P_j for the gain computation, reducing the computation of $comm(v, P_i)$ to

$$d_{skt}(v, s_j) * c_{skt}(s_i, s_j) + (d_{skt}(v, s_i) - d_{ext}(v, P_i)) * c_{skt}(s_i, s_i) \tag{4.24}$$

and of $comm(v, P_j)$ to

$$d_{skt}(v, s_i) * c_{skt}(s_i, s_j) + (d_{skt}(v, s_j) - d_{ext}(v, P_j)) * c_{skt}(s_j, s_j) \tag{4.25}$$

Here, s_i represents the CPU socket that P_i is assigned to, whereas $c_{skt}(s_i, s_j)$ represents the relative network communication costs the among processes running on cores of socket s_i and s_j. Term $d_{skt}(v, s_i)$ denotes the amount of data that v communicates with partitions that are assigned to cores of socket s_i. Clearly, with the help of d_{skt} this can be computed in $O(1)$ time. Here, we assume that each machine has two CPU sockets for the ease of presentation, but our solution obviously handles arbitrary number of CPU sockets. It is worth mentioning that if P_i and P_j are on the same CPU socket, the difference between Eq. 4.24 and 4.25 is the same as that of Eq. 4.22 and 4.23.

In fact, even if P_i and P_j are assigned to cores of different machines, we can still reduce the time complexity from $O(n)$ to $O(m)$, where m represents the number of machines used. The rationale behind this is that P_i has the same relative network communication costs to all the partitions that are residing on a single machine. That is, we can reduce the computation of $comm(v, P_i)$ to

$$\sum_{k=1 \text{ and } k \neq i}^{m} d_{mach}(v, M_k) * c_{mach}(M_i, M_k) \tag{4.26}$$

where $d_{mach}(v, M_k)$ represents the amount of data that v communicated with partitions of machine M_k, M_i denotes the machine which P_i is assigned to, and $c_{mach}(M_i, M_k)$ is the relative network communication cost between machine M_i and machine M_k. Note that we still need to consider the communication cost that v incurs among partitions that are assigned to machine M_i (Eq. 4.24).

Algorithm 10 presents the optimized implementation of the vertex gain computation process in PLANAR+. The new algorithm takes as input four extra parameters: d_{skt}, d_{mach},

Algorithm 10: PLANAR+: Vertex Gain Computation

```
Data: v, P_i, P_j, d_ext, d_skt, d_mach, c, c_skt, c_mach
comm(v, P_i) = 0
comm(v, P_j = 0
//optimized implementation of Equation 4.14
if P_i and P_j are on different machines then
    foreach Machine M_k where k ∈ [1, m] do
        comm(v, P_i) += d_mach(v, M_k) * c_mach(M_i, M_k)
        comm(v, P_j) += d_mach(v, M_k) * c_mach(M_j, M_k)
comm(v, P_i) += the value of Equation 4.24
comm(v, P_j) += the value of Equation 4.25
//implementation of Equation 4.17
gain = (comm(v, P_i) - comm(v, P_j)) * α - mig(v, P_i, P_j)
return gain
```

c_{skt}, and c_{mach}. The former two can be easily computed in the same way as d_{ext} with the knowledge of the partition to socket/machine mapping, whereas the latter two are readily available from the way the relative network communication costs are measured. Although the optimization only reduces the time complexity by a constant factor, the improvement is still non-negligible considering that we have to compute the gain for each boundary vertex once per adaptation superstep, and the repartitioning consists of multiple adaptation supersteps.

4.4.4 PLANAR+: Evaluation

4.4.4.1 Setup In this section, we first evaluate the effectiveness of PLANAR+ in improving the quality of the partitionings (Section 4.4.4.2). We then validate the scalability of PLANAR+ with respect to the number of partitions using two billion-edge graphs (Section 4.4.4.3). Lastly, we access the effectiveness of PLANAR+ using an MPI implementation of PageRank on two billion-edge graphs (Section 4.4.4.4).

Algorithms We compared PLANAR+ to (a) three architecture-aware graph repartitioners: ARAGON [21], PARAGON [22], and PLANAR [23], (b) a state-of-the-art streaming graph partitioner, LDG [8], and (c) UNIPLANAR+, the architecture-agnostic version of PLANAR+, serving as a representative of existing lightweight graph repartitioners.

Datasets Table 4.7 describes the datasets used. By default, the graphs were (re)partitioned with both the vertex weights (i.e., computational requirement) and vertex sizes (i.e., amount

of the data of the vertex) set to their vertex degree. Their edge weights (i.e., amount of data communicated) were set to 1. Vertex degree is a good approximation of the computational requirement and the migration cost of each vertex, while an edge weight of 1 is a close estimation of the communication pattern of PageRank. Considering the communication cost is more important than migration cost, all the experiments were performed with $\alpha = 10$ (Eq. 4.2). Unless explicitly specified, the graphs were initially partitioned by the linear deterministic greedy heuristic, LDG [8], across cores of the machines used (one partition per core). The partitionings were then improved by the repartitioners until convergence. During the (re)partitioning, we allowed up to 2% load imbalance among the partitions. Noted that (a) LDG was extended to support vertex- and edge-weighted graphs for fair comparison; (b) vertices of the graphs were presented to LDG in the BFS order; and (c) PARAGON was performed with the degree of parallelism set to $\frac{1}{4}$ of the number of partitions and the number of shuffle refinement times of 10.

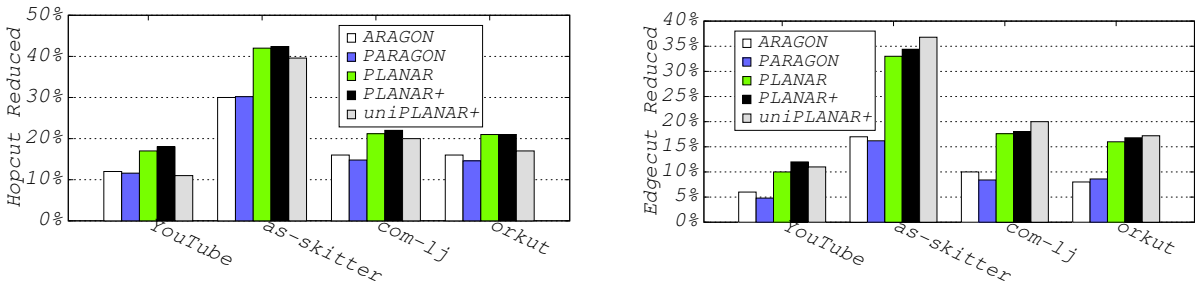
Platforms We evaluated PLANAR+ on a local cluster we had at the University of Pittsburgh: MPICluster [65]. MPICluster had a flat network topology, where all the 32 compute nodes were connected to a single switch via 56Gbps FDR Infiniband. Table 4.8 depicts the compute node configuration of the cluster. All results presented were the means of 5 runs.

MPI Libraries The specific MPI implementation used in our experiments was OpenMPI 1.10.2 [46]. During the evaluation, we bound each MPI rank (process) to a core using the options provided by OpenMPI 1.10.2.

4.4.4.2 Partitioning Quality

Configuration In this experiment, we access the effectiveness of PLANAR+ in improving the quality of the partitionings. Towards this, we first partitioned some datasets of Table 4.7 into 40 partitions using LDG across cores of two 20-core machines. Then, we examined the quality of the resulting decompositions. In particular, we measured the quality of a partitioning in terms of (a) hopcut (Eq. 4.2), (b) edgecut (the number of edges across partitions), (c) vertex migration ratio (percentage of vertices migrated), and (d) skewness (Eq. 4.4).

Results in terms of hopcut and edgecut (Figures 4.36a and 4.36b) Figures 4.36a



(a) hopcut (b) edgecut

Figure 4.36: Percentage of hopcut and edgecut reduced by the repartitioners over the decompositions initially generated by LDG.

and 4.36b, respectively, present the percentage of the hopcut and edgecut reduced by the repartitioners on a variety of datasets, when compared with the initial decompositions generated by LDG. Interestingly, we observed similar patterns among ARAGON, PARAGON, PLANAR, and PLANAR+ in terms of both hopcut and edgecut, where PLANAR+ was always the winner whereas PARAGON always performed the worst. In the best case, PLANAR+ reduced the hopcut and edgecut of the initial decomposition by 42.4% and 34.4%, respectively. The reason why PLANAR+ was slightly better than PLANAR was probably because of the changes we made to the vertex migration probability in our Phase-1a vertex migration as well as the effect of full repartitioning. The greedy nature we had in PLANAR and PLANAR+'s vertex migration policy was probably responsible for their superiority over ARAGON and PARAGON. The reason why ARAGON was slightly better than PARAGON was because PARAGON was a parallel version of ARAGON, where we traded the quality of the resulting decompositions for scalability. The decompositions computed by PARAGON was quite comparable to ARAGON in terms of both edgecut and hopcut.

As expected, UNIPLANAR+ outperformed PLANAR and PLANAR+ in terms of edgecut, but was beaten by them in terms of hopcut. The rationale behind this is that UNIPLANAR+ is architecture-agnostic whereas PLANAR and PLANAR+ are architecture-aware. Architecture-aware graph repartitioners focus more on minimizing the hopcut even at the cost of increasing edgecut. Nevertheless, the decompositions computed by PLANAR and PLANAR+ were still much better than the initial decompositions in terms of edgecut in spite of being architecture-

aware. What we did not expect was that ARAGON and PARAGON were always outperformed by UNIPLANAR in terms of hopcut. This was probably caused by the fact that the partitionings output by UNIPLANAR+ had significantly lower edgecut than that of ARAGON and PARAGON.

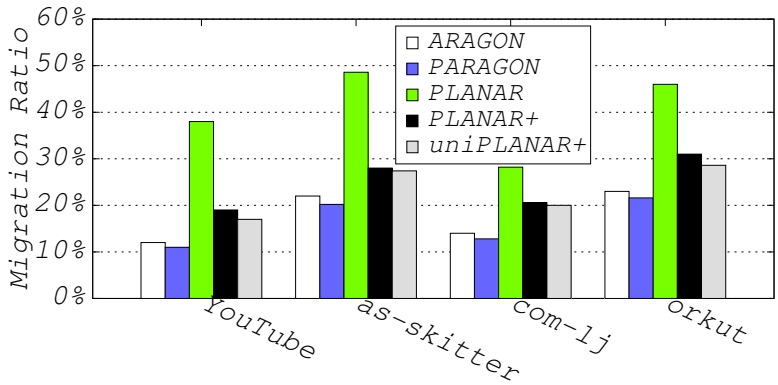


Figure 4.37: Percentage of vertices migrated the repartitioners

Results in terms of vertex migration ratio (Figure 4.37) Another item of interest in this experiment was the percentage of the vertices migrated by the repartitioners. Figure 4.37 shows the corresponding vertex migration ratio of different repartitioners. Note that the ratio of PLANAR reported here was the accumulated vertex migration ratio across its adaptation supersteps. As expected, PLANAR had the highest vertex migration ratio in all the cases, since it had a physical vertex migration phase in each of its adaptation superstep. As a result, a vertex may be migrated multiple times before it was moved to its final optimal destination. It is also expected that PLANAR+ had much lower vertex migration ratio than PLANAR, since it eliminated the need of per adaptation superstep physical vertex migration. Specifically, PLANAR+ reduced the vertex migration ratio of PLANAR by up to 20%. We also noticed that both ARAGON, PARAGON, and UNIPLANAR had lower migration ratio than that of PLANAR and PLANAR+. This was also reasonable considering the fact that they achieved much lower improvement in terms of hopcut (Figure 4.36a).

Results in terms of partition skewness (Table 4.14) We also examined the skewness of the resulting decompositions computed by the repartitioners. Table 4.14 shows the corresponding results when the repartitioners were executed with degree of imbalance tolerance of 1.02. As shown, none of the repartitioners was able to guarantee the exact load balance,

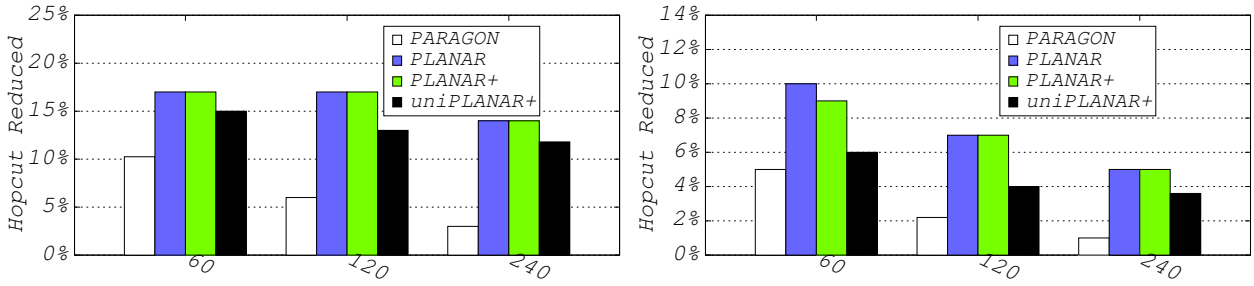
Table 4.14: Skewness of the resulting decompositions

Algorithm/Dataset	YouTube	as-skitter	com-lj	orkut
Aragon	1.010	1.040	1.010	1.010
Paragon	1.002	1.044	1.002	1.000
Planar	1.022	1.026	1.020	1.020
Planar+	1.056	1.038	1.020	1.020
uniPlanar+	1.022	1.066	1.020	1.020

but in most cases they were able to provide the approximate load balance.

4.4.4.3 Scalability Study

Configuration In this experiment, we examined the behavior of the repartitioners as the number of partitions increased. Towards this, we initially partitioned the Friendster and Twitter dataset of Table 4.7 across cores of three up to twelve 20-core machines using LDG. We then applied the repartitioners to the decompositions to improve their quality.

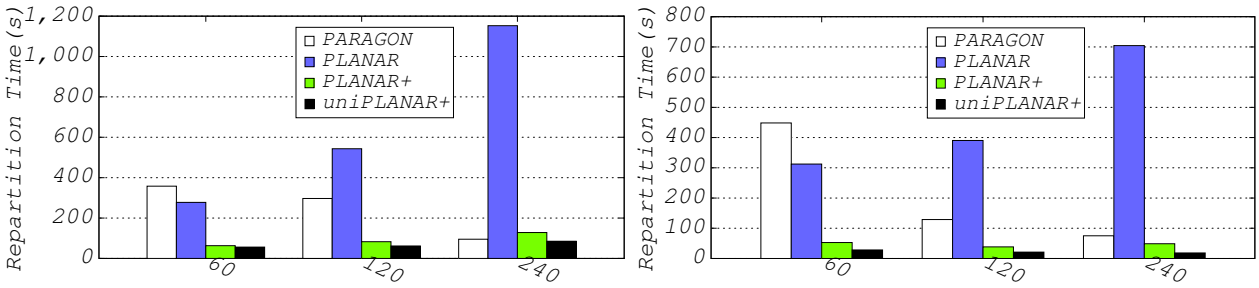


(a) Friendster (b) Twitter

Figure 4.38: Percentage of hopcut reduced after running the repartitioners over the decompositions with varying number of partitions.

Results in terms of hopcut (Figures 4.38a and 4.38b) Figures 4.38a and 4.38b show the percentage of the hopcut reduced by the repartitioners against the initial decompositions with varying number of partitions. As expected, Planar and Planar+ were always better than Paragon and uniPlanar+. Also, Planar+ was almost as good as Planar in all the cases yet being much faster (Figures 4.39a & 4.39b). We also noticed that as the number

of partitions increased, the improvement achieved by PARAGON seemed to drop much more significantly than that of PLANAR and PLANAR+. Although we can slow down the trend by increasing the number of shuffle refinement times of PARAGON, it would increase the overhead of repartitioning. Nevertheless, the improvement was still non-negligible, if we consider the absolute number of the hopcut reduced.

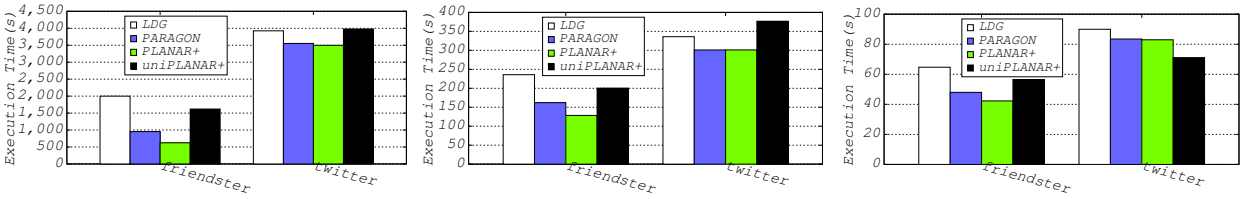


(a) Friendster (b) Twitter

Figure 4.39: Repartition time of the repartitioners over the decompositions with varying number of partitions.

Results in terms of repartition time (Figures 4.39a and 4.39b) We also report the repartition time of each algorithm in Figures 4.39a and 4.39b. As shown, in terms of the repartition time, PLANAR+ was always better than PLANAR. This was because PLANAR+ is an optimized implementation of PLANAR. The optimizations included: the memorization technique that PLANAR+ used to avoid repeated vertex gain computation, the elimination of per adaptation step physical vertex migration, and the use of the hardware topology knowledge to speed up the process of vertex gain computation. Specifically, PLANAR+ speeded up the repartitioning process of PLANAR by up to 9x without compromising the quality of the partitioning.

We also observed that PLANAR+ had lower repartition time than that of PARAGON in spite of producing better partitionings. One of the reasons behind this was that PLANAR+ required less data communication than PARAGON for repartitioning. Another reason for this was that PLANAR+ had higher degree of parallelism than that of PARAGON. The maximal degree of parallelism that PARAGON can have was $\frac{1}{2}n$, whereas the degree of parallelism that PLANAR+ had was always n. Here, n was the number of partitions. The reason why the gap was closing up as n increased was because as n increased the improvement achieved by



(a) Grouping size of 256 (b) Grouping size of 512 (c) Grouping size of 1024

Figure 4.40: PageRank execution time on Friendster and Twitter datasets with varying message grouping sizes.

PARAGON had a tendency to drop greatly, making its execution terminate earlier. This also explained why PARAGON was faster than PLANAR sometimes.

We also observed that UNIPLANAR was always faster than others. This was expected as the vertex gain computation process of UNIPLANAR+ was simpler than the architecture-aware ones. Another thing worth mentioning here was that regardless of the increasing number of partitions, the overhead of PLANAR+ and UNIPLANAR+ remained quite stable in comparison to that of PLANAR. This further highlighted the effectiveness of the optimizations we made to PLANAR+.

4.4.4.4 Real-World Workload (PageRank)

Configuration In this section, we evaluated PLANAR+ with an MPI implementation of PageRank on the Friendster and Twitter datasets. To this end, we first partitioned the graphs across cores of MPICluster using LDG. Then, we ran PageRank on the decompositions that were improved by the repartitioners. During the execution of PageRank, we grouped multiple messages sent by each MPI rank to the same destination into a single one. Given the superiority of PLANAR+ over PLANAR, we would only present the results of PLANAR+ with its competitors in this section.

Resource Contention Modelling To capture the impact of resource contention, we ran a profiling experiment for PageRank with the datasets on the cluster by increasing λ gradually from 0 to 1. Interestingly, we found that intra-node shared resource contention was more critical to the performance on the cluster. Hence, we fixed λ to be 1 throughout the experiment.

Results in terms of PageRank execution time (Figures 4.40a to 4.40c) Figures 4.40a to 4.40c present the resulting execution time of PageRank (20 iterations) with different message grouping sizes, when the datasets were partitioned across six 20-core machines. As expected, PLANAR+ outperformed others in almost all the cases, and architecture-aware solutions (PARAGON and PLANAR+) performed better than architecture-agnostic ones (LDG and UNIPLANAR+) in most of the cases.

When compared with LDG, PLANAR+ reduced the workload execution time, respectively, by 68%, 45%, and 34% on the Friendster dataset for message grouping size of 256, 512, and 1024, and, respectively, by 10%, 10%, and 7% on the Twitter dataset for message grouping size of 256, 512, and 1024. In comparison to UNIPLANAR+, PLANAR+ reduced the workload execution time, respectively, by 61%, 35%, and 25% on the Friendster dataset for message grouping size 256, 512, and 1024, and, respectively, by 12% and 20% on the Twitter dataset for message grouping size of 256 and 512. One thing worth mentioning here is that PLANAR+ reduced the execution time of all the computing elements (120 cores) by this much. This is essentially equivalent to many hours of CPU time saving.

We also noted that as the message grouping size increased, the improvement achieved by PLANAR+ against LDG and UNIPLANAR tended to decrease. 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 intra-node data communication gradually decreased. This indicates that architecture-aware solutions (PARAGON and PLANAR+) work better with workloads dominated with a large number of small message exchanges. On the other hand, because of the weakening impact of the intra-node data communication, the importance of reducing edgecut increased. This also explains the reason why UNIPLANAR+ performed better than PLANAR+ for message group size of 1024 on the Twitter dataset.

Another interesting thing here was that even though UNIPLANAR reduced the edgecut of the decompositions computed by LDG greatly, it was still outperformed by LDG sometimes, especially with smaller message grouping size. This was because with smaller message grouping size the contention on the memory subsystems can impact the performance greatly. Consequently, lower edgecut did not always lead to better performance.

Table 4.15: PageRank communication volume breakdown in GB

	Friendster			Twitter		
	Intra-Socket	Inter-Socket	Inter-Node	Intra-Socket	Inter-Socket	Inter-Node
LDG	27.1	29.2	303	37.2	40.6	416
PARAGON	19.6	23.9	310	30.7	35.6	424
PLANAR+	16.6	20.8	272	27.8	32.5	409
UNIPLANAR+	23.2	25.5	263	35.5	38.3	397

Results in terms of PageRank communication volume (Table 4.15) To further confirm the effectiveness of PLANAR+ in avoiding contention (reducing intra-node data communication), we also report the breakdown of the communication volume for the execution of PageRank in Table 4.15. Note that message grouping size did not change the amount of data communicated by the execution of PageRank. It only impacted the number of messages exchanged. As shown, PLANAR+ had the lowest intra-node communication volume. When compared with LDG, it, respectively, reduced the intra- and inter-socket communication volume by 38% and 28% on the Friendster dataset, and by 25% and 19% on the Twitter dataset. In comparison with UNIPLANAR+, it, respectively, reduced the intra- and inter-socket communication volume by 28% and 18% on the Friendster dataset, and by 21% and 15% on the Twitter dataset. Another thing worth mentioning here was that PLANAR+ also had much lower overall communication volume than others and much lower inter-node communication volume than LDG and PARAGON.

4.4.5 Section Summary

In this work, we presented a lightweight architecture-aware graph repartitioner, PLANAR+, for large dynamic graphs. PLANAR+ can not only efficiently respond to graph dynamism by migrating vertices among the partitions, but can also improve the mapping of the application communication pattern to the underlying hardware topology. PLANAR+ only requires a small amount of local information plus a minimal amount of global coordination for repartitioning, making it quite feasible for large-scale, graph-based big data applications.

Considering the size of real-world graphs, features like being lightweight, architecture-aware, and workload-aware (which are all present in PLANAR+) are absolutely essential for online repartitioners. Our evaluation confirmed PLANAR+'s superiority in terms of repartitioning time (up to 9x speedup against PLANAR), performance improvement (up to hours reduction in the CPU time), and scalability (up to two billion-edge graphs).

5.0 SKEW-RESISTANT GRAPH PARTITIONING

As demonstrated in the previous chapters, there are dozens of graph partitioners, from the "classic" ones, like [6, 4, 32, 34], to new, *(re)streaming* graph partitioners, like [8, 10, 13, 9], which address the scalability challenge of partitioning the graphs. However, despite the large amount of work so far (including our own works: ARGO, ARAGON, PARAGON, PLANAR, and PLANAR+), *largely overlooked are the effects of different types of skewness* on the performance of distributed graph computation. In particular, we distinguish between two types: algorithmic skewness and structural skewness, which we explain next.

Algorithmic Skewness Current graph partitioners all assume that a balanced partitioning of the graph is equivalent to an even load distribution. Put simply, they all assume that vertices of the graph are always active during the computation. This is true for *always-active-style* graph algorithms, like PageRank. However, for *traversal-style* graph algorithms, like Breadth-First Search (BFS) and Single-Source Shortest Path (SSSP), only a subset of vertices are explored in each superstep. As a result, vertices active in the same superstep may be concentrated into a few partitions by existing graph partitioners, leading to load imbalance, resource underutilization, and contention on the network interface. One way to avoid this *algorithmic skewness* is to migrate vertices dynamically based on some system metrics [40, 26, 42]. However, this is too late and the migration is not cost-free. Migrating a vertex to a new partition requires migrating both its edge list and its associated application data plus an update of the vertex location.

Structural Skewness Existing graph partitioners often do not care about what vertices each partition will have. As a result, high-degree vertices may be concentrated into a few partitions, causing a new type of imbalance, *structural skewness*. This is because high-

degree vertices are often the computation and communication hotspots given their large neighborhood. Unfortunately, graphs from various important domains are scale-free, where the vertex degree-distribution asymptotically follows a power law distribution [63, 64].

Side-Effect of Algorithmic and Structural Skewness Another side effect of the skewness on modern multicore machines is that it may lead to contention for the shared resources in the memory subsystems, especially when the partitions that contain most of the active vertices are assigned to the cores of the same machine for parallel processing. This is because intra-node data communication (the communication among cores of the same machine) is often implemented via shared memory [48, 49], requiring additional data copies. Thus, having too much data communication among partitions that are residing on the same machine may lead to serious cache pollution and therefore contention for the shared last level cache, front side bus, and memory controller (which has been experimentally demonstrated in our previous chapters).

Contributions To address the needs of efficient distributed graph computation, we make the following contributions in this work:

1. To better understand the skewness issue, we experimentally demonstrate the runtime characteristics of two classic traversal-style-graph workloads (Section 5.1.1) and their predictability using real-world graphs (Section 5.1.2).
2. We introduce the idea of multi-label graph partitioning (MLGP) (Section 5.2) and an application of MLGP to do skew-resistant graph partitioning (Section 5.3).

5.1 TRAVERSAL-STYLE GRAPH WORKLOAD CHARACTERIZATION

In this section, we motivate our work by examining the runtime characteristics of two representative traversal-style graph workloads: BFS and SSSP. It is well known that traversal-style graph workloads only explore a subset of the vertices of the graph in each superstep [40]. Thus, a balanced partitioning of the entire graph cannot always guarantee an even load distribution over all the supersteps. In particular, we are interested in the runtime char-

acteristics of such workloads on scale-free and small-world graphs. This is because many real-world graphs are scale-free yet small-world and current graph partitioners may lead to serious structural skewness.

5.1.1 Active Vertex Distribution Across Supersteps (Table 5.1)

Configuration In this experiment, we examined the runtime characteristics of BFS and SSSP on the Orkut dataset. Orkut is a social network run by Google [53] for people across the world to discuss their common interests. The dataset used is a subset of the Orkut user population (around 11.3% at the time crawled by A. Mislove et. al. [54]). The degree distribution of the dataset follows a power-law distribution with average and maximal vertex degree equal 76.281 and 33,313, respectively. The maximal diameter of the dataset is 10 with the effective diameter of 5.4489.

In the experiment, the graph was partitioned across six 20-core machines using three different techniques with one partition per core. The techniques examined included: (a) METIS, a well-known multilevel graph partitioner [32]; (b) LDG, a state-of-the-art streaming graph partitioner [8]; and (c) reLDG, a state-of-the-art restreaming graph partitioner [9].

Table 5.1: Active vertex distribution across supersteps of BFS & SSSP execution with one randomly selected source vertex

com-orkut	# of Active Vertices	
Supersteps	BFS	SSSP
0	1	1
1	72	45
2	5,871	4,663
3	215,425	297,943
4	1,753,891	1,421,993
5	1,088,870	1,229,917
6	8,242	117,496
7	69	383
8	0	0

Results Table 5.1 presents the number of vertices that are active in each superstep for the execution of BFS/SSSP with one randomly selected source vertex. As shown, only a subset of the vertices were active in each superstep, and the execution exhibited highly skewed active vertex distribution across supersteps. The top-3 supersteps with largest fraction of active vertices covered around 96% of vertices of the graph. This was expected for *small-world* and *scale-free* graphs. Small-world graphs are known to have low diameter. Consequently, the execution of BFS/SSSP on such graphs usually ends in a few supersteps, causing a large number of vertices to be visited per superstep. On the other hand, the scale-free property allows the number of vertices active in each superstep to be expended and shrink exponentially. As a result, a majority of vertices were visited in very few supersteps. These supersteps were also the top-3 most time-consuming supersteps.

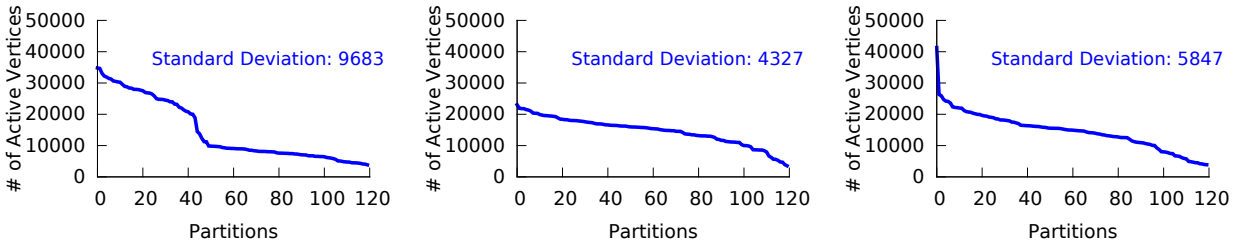
We observed similar results for the execution of BFS/SSSP on the partitionings computed by METIS, LDG, and reLDG. This was because (1) the execution of BFS/SSSP on the partitionings all started from the same randomly selected source vertex; and (2) the way the graph was distributed across partitions only affected the amount of data communication performed by BFS/SSSP (but not the algorithm characteristics).

Take-away *To achieve superior performance, we should offer differentiated partitioning for vertices that are active in the peak supersteps. That is, we should focus more on reducing the edgecut of vertices that are active in the peak supersteps and balancing the load of the peak supersteps.*

5.1.2 Active Vertex Distribution Across Partitions (Fig. 5.1 & 5.2)

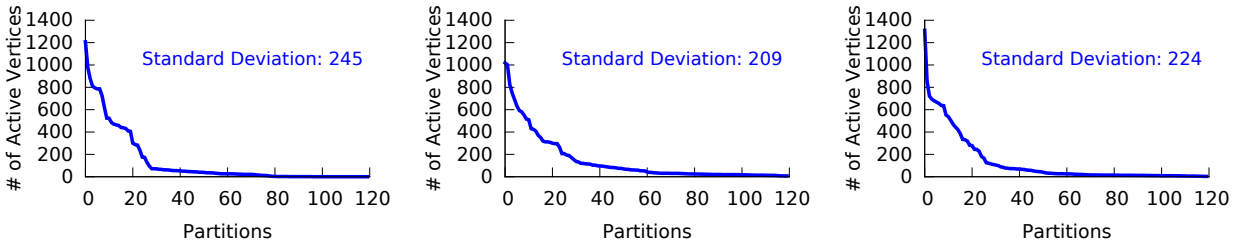
Configuration This experiment examined the corresponding *active vertex* and *active high-degree* vertex distribution across partitions for the execution of BFS/SSSP on the partitionings. We treated the top 1% vertices as the high-degree ones. For brevity, we only showed the results of BFS in Figures 5.1 and 5.2 for the most time-consuming superstep (Step 4 of Table 5.1).

Results As can be seen, the execution of BFS on the partitionings computed by METIS, LDG, and reLDG all exhibited highly skewed active vertex and active high-degree vertex



(a) METIS (b) LDG (c) reLDG

Figure 5.1: BFS *active vertex* distribution across partitions for the most time-consuming superstep (Step 4 of Table 5.1) on com-orkut dataset with one randomly selected source vertex. The distribution was measured, when the graph was partitioned across six 20-core machines with one partition per core.



(a) METIS (b) LDG (c) reLDG

Figure 5.2: BFS *active high-degree vertex* distribution across partitions for the most time-consuming superstep (Step 4 of Table 5.1) on com-orkut dataset with one randomly selected source vertex. The distribution was measured, when the graph was partitioned across six 20-core machines with one partition per core.

distribution across partitions, especially the distribution of high degree vertices (around half of the partitions have nearly zero active high-degree vertices). This may lead to potential significant load imbalance and thus resource underutilization as well as contention on both the network interface and memory subsystems. Another interesting result was that the decomposition computed by METIS had the largest skewness followed reLDG next to it. This was somehow expected considering the fact that METIS tends to produce partitionings of the highest quality, while LDG performed the worst among the three. Put simply, METIS

and reLDG were better than LDG in grouping tightly connected vertices together, leading to higher chance of load imbalance. This also explains the reason why simple partitioning techniques (e.g., hashing partitioning) may sometimes perform better than those well-studied ones.

Take-away *We should consider the characteristics of both the target workload and the graph structure while partitioning.*

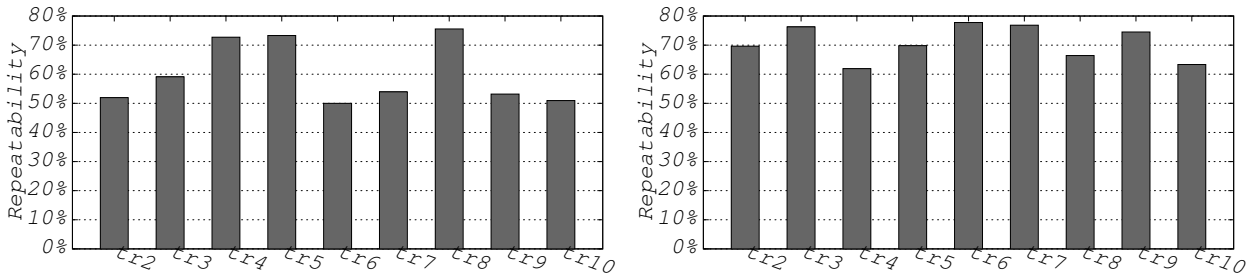
5.1.3 Workload Predictability (Fig. 5.3 & 5.4)

Given the above observations, one may wonder if we could incorporate such characteristics into the partitioning process, such that both the algorithmic and structural skewness are minimized. Towards this, we kept track of vertices that were active in each superstep for ten distinct executions of BFS/SSSP on the Orkut dataset. Each such execution was performed with one randomly selected source vertex on the dataset, when it was partitioned into 60 partitions. Then, we examined the repeatability of the execution traces. Considering the highly skewed active vertex distribution across supersteps, we only considered the top-3 most time-consuming supersteps for repeatability computation. We defined the repeatability of execution trace tr_1 with respect to tr_2 as:

$$repeat(tr_1, tr_2) = \frac{\sum_{i=1}^{3} \max_{j=1,2,3} |s_{tr_1(i)} \cap s_{tr_2(j)}|}{\sum_{i=1}^{3} |s_{tr_1(i)}|} \tag{5.1}$$

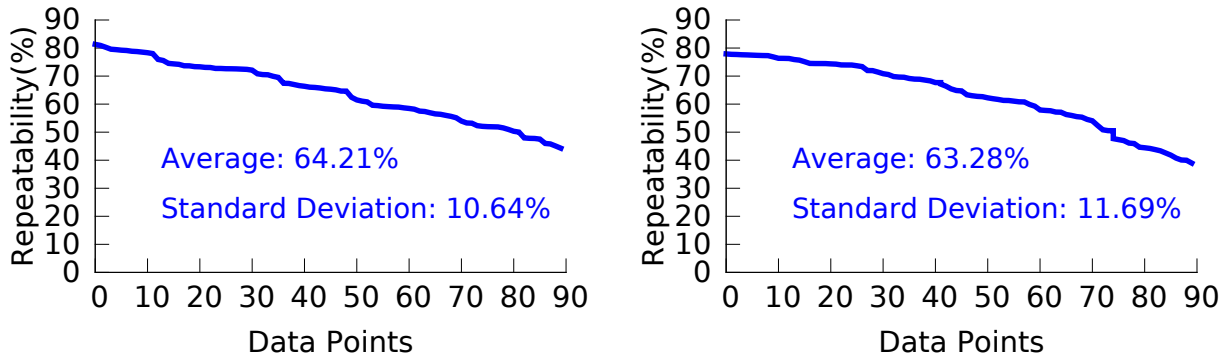
where $s_{tr_1(i)}$ denotes the set of vertices that are active in the ith most time-consuming superstep of trace tr_1. The execution trace repeatability indicates the degree of overlap among the traces. It should be noted that this was a conservative estimation, because $s_{tr_1(i)}$ may overlap with multiple supersteps of execution trace tr_2. Yet, we only considered the superstep that overlaps $s_{tr_1(i)}$ the most.

Figure 5.3 shows the repeatability of tr_1 with respect to different traces collected for the execution of BFS and SSSP, and Figure 5.4 plots the distribution of the trace repeatability across all the trace pairs. As shown, the runtime characteristics of both BFS and SSSP on the Orkut dataset were actually quite predictable. On average, around 60% of vertices are always active in the same supersteps for two distinct executions of BFS/SSSP with one



(a) BFS Execution Trace Repeatability (b) SSSP Execution Trace Repeatability

Figure 5.3: Repeatability of BFS and SSSP execution trace: tr_1 with respect to different traces on the Orkut dataset.



(a) BFS Repeatability Distribution (b) SSSP Repeatability Distribution

Figure 5.4: Distribution of BFS and SSSP execution trace repeatability across all trace pairs.

randomly selected source vertex. The relatively high repeatability can be explained by the *wave* access pattern of the traversal-style graph workloads. That is, once the superstep active vertex set of two distinct executions of the workload intersects, the number of vertices active in the same time period (superstep) will become larger and larger (up to a certain point), especially if we hit a high-degree vertex. This is because all the neighbors of the vertices in the current common active vertex set will become active in the next superstep. Note that we observed similar results for the execution of BFS/SSSP on partitionings computed by METIS, LDG, and reLDG.

Take-away *The execution trace of the traversal-style graph workloads on many small-world*

and scale-free graphs can be used as a representative of the runtime characteristics of the target workloads. This provides us an opportunity to leverage the runtime characteristics of the target workload into the partitioning process (using the execution trace).

5.2 MULTI-LABEL GRAPH PARTITIONING

In this section, we first introduce the *Multi-Label Graph Partitioning (MLGP)* problem as well as a streaming-based implementation of such a graph partitioner that could be used to do *Skew-Resistant Graph Partitioning*.

5.2.1 Problem Statement

Let $G = (V, E, L)$ be a graph with labels on vertices, where V is the set of vertices, E is the set of edges, and $L = \{L_1, L_2, ...L_m\}$ is the set of labels associated with vertices in V. Each vertex is associated with a binary label vector, indicating if the corresponding label exists on the vertex. MLGP aims to minimize the communication cost among the partitions under the constraint (1) that each partition is balanced; (2) and that vertices of each partition follow a user-defined distribution in terms of their labels. The quality of the partitioning (communication cost) is defined as:

$$comm(G, MLGP) = \sum_{\substack{e=(u,v)\in E \text{ and} \\ u\in P_i \text{ and } v\in P_j \text{ and } i\neq j}} w(e) \tag{5.2}$$

where $w(e)$ is the edge weight, indicating the amount of data communication between the vertex pair.

Constraint 1 can be formally defined as:

$$\sum_{v\in P_i} w(v) \leq C(P_i) \text{ for i} \in [1, k] \tag{5.3}$$

where k corresponds to the number of partitions we want, $w(v)$ is the vertex weight (indicating the computational requirements of the vertex), and $C(P_i)$ denotes the partition capacity.

As for *Constraint 2* (the vertex distribution of each partition), we are particularly interested in distributing vertices of the same labels evenly across partitions, which can be formulated as:

$$\sum_{v^l \in P_i} w(v^l) \leq C^l(P_i) \text{ for i} \in [1, k] \tag{5.4}$$

where v^l denotes vertices that have label L_l, whereas $w(v^l)$ and $C^l(P_i)$, respectively, corresponds to the vertex weight and the partition capacity of P_i for l-labelled vertices. In other words, we want each partition to eventually have a similar vertex distribution to the original graph in terms of their labels. In case of vertices of the graph do not have any labels, Constraint 1 is self-included in Eq. 5.4. Sometimes, we may only want to apply Constraint 2 to a subset of $|V|$ while guaranteeing the rest of vertices do not violate Constraint 1.

5.2.2 Streaming-Based Implementation

5.2.2.1 Graph Partitioning Model MLGP follows the same graph partitioning model proposed by [8, 10]. In this model, vertices arrive at the partitioner in 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 k partitions based on the placements of vertices previously arrived. The placement of the vertex never changes once it is assigned to a partition. [8] presents a variety of heuristics for the placement of vertices, among which the linear deterministic greedy (LDG) performs the best. LDG tries to assign a vertex, v, to a partition, P_i, that maximizes:

$$(1 - \frac{w(P_i)}{C(P_i)}) \sum_{e=(u,v) \in E \text{ and } u \in P_i} w(e) \tag{5.5}$$

Intuitively, LDG aims to place the vertex to the partition having the largest number of its neighbors but penalizes the partition based on its current load.

5.2.2.2 Streaming Heuristic For the streaming-based implementation of MLGP, we change the vertex assignment rule to maximize the following objective for each vertex v:

$$l\_factor(P_i) * \sum_{e=(u,v) \in E \text{ and } u \in P_i} w(e) \tag{5.6}$$

where $l\_factor(P_i)$ is used to penalize the partitions based on their vertex label distribution. We formally defined it as:

$$l\_factor(P_i) = \begin{cases} \min\limits_{\forall l \in [1,m]} \{1 - \lambda(P_i, l)\} & \text{if } \exists l, \lambda(P_i, l) > 1 \\ \sum_{l=1}^{m} lv[l](1 - \lambda(P_i, l)) & \text{otherwise} \end{cases} \tag{5.7}$$

where lv is the binary label vector associated with each vertex and $\lambda(P_i, l) = \frac{w^l(P_i)}{C^l(P_i)}$, with $w^l(P_i)$ denoting the aggregated weights of l-labelled vertices that have been assigned to P_i. Thus, $\lambda(P_i, l)$ represents the degree of skewness for partition P_i in terms of l-labelled vertices. In other words, if $\lambda(P_i, l)$ is smaller than 1, we could put more l-labelled vertices to P_i. The smaller the value is, the more we could put and vice verse. Since we need to consider the for all the labels of the vertex, we choose to sum them together. On the other hand, $\lambda(P_i, l)$ greater than 1 means that P_i is overloaded in terms of l-labelled vertices and that we should avoid putting any l-labelled vertices to P_i. Since each vertex may have multiple labels and the placement of a vertex to a partition may not always satisfy the balance constraint for all its labels, we choose to penalize the label overloaded the most.

5.2.2.3 Restreaming Model MLGP loads vertices of the graph in blocks and streams each in-memory block multiple passes instead of streaming the entire graph multiple passes as the current restreaming graph partitioners do. In this way, we can enjoy the benefits of restreaming partitioning model but avoiding loading the graph from disk multiple times. The default value of the number of restreaming passes is two in our implementation. By default, we treat 2^{19} vertices that are stored contiguously in the file system as a block.

5.3 SKEW-RESISTANT GRAPH PARTITIONING

In this section, we first introduce SARGON, an application of MLGP to prevent the algorithmic and structural skewness of traversal-style graph workloads. Then, we outline a few other possible use cases that MLGP can be applied to.

5.3.1 MLGP: Traversal-Style Graph Workloads

5.3.1.1 Avoiding Algorithmic Skewness To guarantee that the load of the traversal-style graph workloads is evenly distributed in every superstep, SARGON models it as a MLGP problem, in which SARGON only needs to divide the entire execution time into finite time periods, and associates each vertex with a label vector. The label vector indicates the time periods in which the vertex is active. Given the relatively high predictability of the runtime characteristics of BFS and SSSP on the datasets of interest (Section 5.1), SARGON uses the supersteps as the natural time periods and obtains the label vector from the execution trace. With the augmented label information, MLGP will automatically split vertices active in the same superstep evenly across partitions while keeping the communication among the partitions as small as possible, thus eliminating algorithmic skewness.

In fact, SARGON only applies MLGP to vertices of the peak supersteps, while ordinary graph partitioning heuristic (LDG) to rest of the vertices. This is because if the number of vertices active in a superstep is large, the computation time will probably dominate over the superstep execution time. Even if the target workload is communication-intensive, the concentration of a large amount of active vertices into a few partitions may lead to serious contention on the network interfaces or memory subsystems, making balanced load distribution very critical. On the other hand, if the number active vertices in a superstep is small, the communication cost will become the dominant factor of the superstep, making reducing the communication cost more important. By default, SARGON only applies MLGP to supersteps whose active vertex set has more than 1% of the vertices of the graph.

5.3.1.2 Avoiding Structural Skewness Considering the relatively small number of high-degree vertices and vast disparity in the vertex weights the graph may have, SARGON avoids structural skewness by simply assuming that all high-degree vertices are active in a single additional superstep. By doing this, MLGP will attempt to distribute high-degree vertices evenly across partitions. At the same time, the labels that high-degree vertices originally have can serve as a way to penalize partitions that have a large number of vertices that are active at the same time with the high-degree ones. This also means that high-degree

vertices originally active in the same superstep will have a smaller chance to be put together.

5.3.2 MLGP: Multiphase Graph Workloads

In addition to the traversal-style graph workloads, some workloads may further organize each of their supersteps into multiple phases, and each phase processes a different part of the graph based on the result of the previous phase (synchronization is often required between phases). Thus, these workloads may still belong to always-active-style graph workloads, except that vertices may be active in different phases of the supersteps. Clearly, a static partitioning of the graph using existing graph partitioners will not work well, since they provide no guarantee on how the vertices of each phase are distributed across partitions.

As a result, users often have to repartition the graph at the beginning of each phase to ensure an even load distribution. However, the repartitioning has non-negligible overhead. In addition to the time taken to compute the new partitioning, repartitioning also requires migrating vertices (and its associated application state) from one partition to another, rebuilding the graph structure after migration, as well as an update of the vertex locations. What is even worse is that repartitioning has to be performed in every superstep.

Instead, if the phases during which the vertices are active are predictable, we can assign each vertex a label vector indicating the phases that it is active in, and partition the graph using MLGP. In this way, we can postpone/eliminate the need of repartitioning and yet guarantee that the partitioning holds well over time. We can even replace the binary label vector with a weight vector, indicating the computation requirement of the vertex in each phase to further balance the load.

5.3.3 MLGP: Graph Database Partitioning

Although the focus of this work is offline batch processing, it would also be interesting to see what types of interactive queries on graph databases have similar type of predictable runtime characteristics (as the one demonstrated in Section 5.1). Offline batch processing usually involves the computation on vertices of the entire graph, whereas online query processing often only involves the computation on a subset of the vertices. Workload characterization of

interactive queries is more challenging, because interactive queries usually have parametrized constraints on vertex or edge attributes.

Nevertheless, some workloads may naturally exhibit repeatable runtime characteristics but in different forms. For example, vertices of the graph may have different degree of hotness/popularity, in which we would like each partition of the graph to have a mix of hot and cold data for even load distribution. Also, vertices of the graph may have seasonal access patterns or vertices of the same geo-location usually have a tendency to be active in the same time period (in the diurnal form). In these cases, we would like vertices that are accessed in the same season or of the same geo-location to be evenly distributed across partitions. By doing this, each partition will have a better chance of holding well over time without constantly be overloaded. All these can be achieved by partitioning the graph using the idea of MLGP.

5.4 EVALUATION

In this section, we first evaluate the effectiveness of SARGON in terms of the skewness and partitioning quality (Section 5.4.2), and then validate the effectiveness of SARGON using two representative traversal-style graph workloads: Breadth-First Search and Single-Source Shortest Path (Section 5.4.3), and finally conclude our evaluation with a scalability study (Section 5.4.4). Both BFS and SSSP were implemented using MPI [58] based on the idea presented in [59, 60]. The specific MPI implementation we used in the experiment was OpenMPI 1.8.6 [46].

5.4.1 Setup

Baselines We compared SARGON to three different graph partitioners: (a) METIS, a well-known multilevel graph partitioner [32]; (b) LDG, a state-of-the-art streaming graph partitioner [8]; and (c) reLDG, a state-of-the-art restreaming graph partitioner [9]. For reLDG, we set the number of restreaming passes to 2.

Datasets Table 4.7 describes the datasets used. All the datasets were undirected, except the Twitter dataset but was treated as undirected. Note that the datasets were all scale-free and small-world graphs. During the experiments, 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 and SSSP, 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. For the partitioning, we allowed up to 2% imbalance among the partitions.

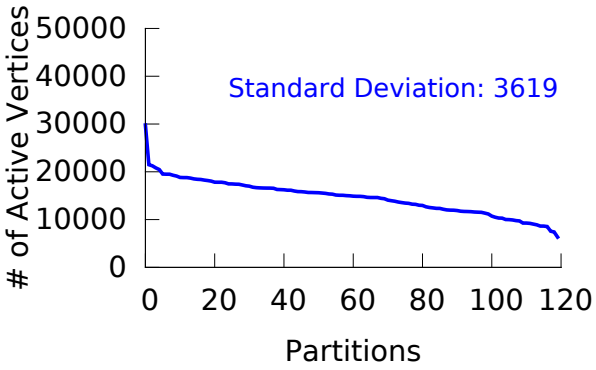
Evaluation Platform The experiments were performed on a 32-node university cluster [65]. The cluster had a flat network topology, where all the compute nodes were connected to a single switch via FDR Infiniband. Table 4.8 depicts the compute node configuration of the cluster.

5.4.2 Microbenchmarks

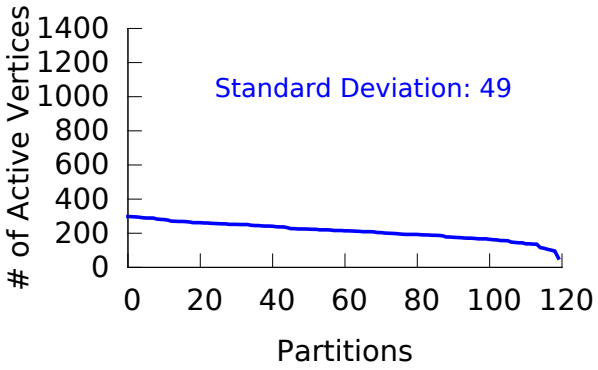
5.4.2.1 Effectiveness in terms of Skewness

Configuration This experiment assessed the effectiveness of Sargon on com-orkut dataset using one of the BFS execution traces that we collected in Section 5.1. In addition to the labels indicating the supersteps that each vertex was active in, we also appended an additional label to each high-degree vertex as illustrated in Section 5.3.1 to avoid structural skewness. We then examined the distribution of the active vertices across partitions for an execution of BFS on the partitioning computed by Sargon with one randomly selected source vertex.

Results (Figure 5.5) Figure 5.5 plots the active vertex and active high-degree vertex distribution for the most time consuming superstep. By comparing it to Figures 5.1 and 5.2, we can conclude that Sargon balanced the distribution of both active vertex and active high-degree vertex much better than Metis, LDG, and reLDG, especially the distribution of high-degree vertices. Specifically, Sargon reduced the standard deviation of the active vertex and active high-degree vertex distribution by up to 62% and 80%, respectively.



(a) Active Vertex Distribution on SARGON Partitionings



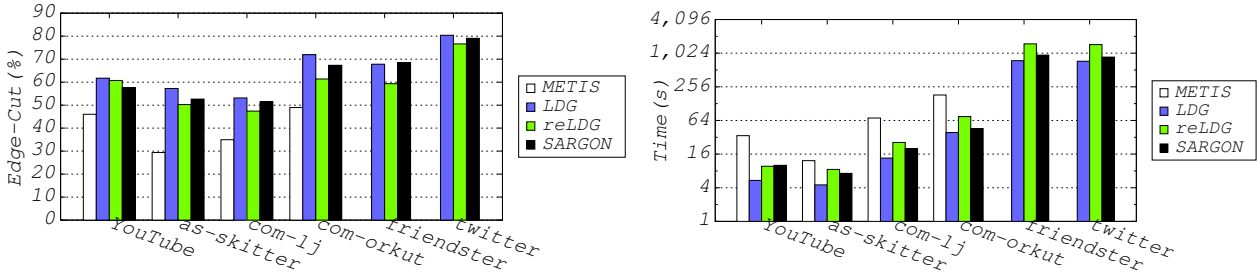
(b) Active High-Degree Vertex Distribution on SARGON Partitionings

Figure 5.5: Active (high-degree) vertex distribution across partitions for the most time-consuming superstep of a BFS execution on com-orkut dataset with one randomly selected source vertex. The distribution was measured when the dataset was partitioned across six 20-core machines with one partition per core.

5.4.2.2 Effectiveness in terms of Partitioning Quality

Configuration Another aspect of interest was the quality (Eq. 5.2) of the resulting partitionings output by SARGON and the partitioning overhead. Thus, we partitioned some datasets of Table 4.7 across six 20-core machines and examined the percentage of the edges that were cut (Figure 5.6a) as well as the overhead of partitioning (Figure 5.6b). Note that the overhead reported included the cost of loading and partitioning the graph as well as the cost of sending vertices to the assigned partitions. The execution trace collection overhead was not included in SARGON's partitioning overhead, since we assumed that the execution traces were available beforehand and we only need to collect the trace of each target workload once for each specific graph dataset.

Results (Figure 5.6) As expected, METIS performed the best in terms of partitioning quality but the worst in terms of partitioning overhead. This was because METIS requires more information about the graph for partitioning, whereas LDG, reLDG, and SARGON are streaming graph partitioners. The reason why SARGON sit in between LDG and reLDG in



(a) Partitioning Quality (b) Partitioning Overhead

Figure 5.6: The quality of the partitionings computed by different partitioners over a variety of graphs, as well as the corresponding partitioning overhead (in log scale). The datasets presented were partitioned across six 20-core machines with one partition per core.

terms of both the quality and the overhead was that SARGON only requires loading the graph from disk once (lowering the partitioning overhead in comparison to reLDG) but streams each vertex block in memory twice (lowering the edgecut when compared with LDG). In fact, SARGON was expected to have higher edgecut, since it focuses more on avoiding algorithmic and structural skewness. Regardless, most of the decompositions computed by SARGON still had lower edgecut than that of LDG. Note that METIS failed to partition the Friendster and Twitter dataset.

5.4.3 Real-World Workloads (BFS & SSSP)

Configuration This experiment evaluated the effectiveness of SARGON using BFS and SSSP on the com-orkut dataset, when it was partitioned across three 20-core compute nodes (one partition per core). Note that SARGON partitioned the graph along with a trace of a single BFS/SSSP execution on the graph with one randomly selected source vertex. Given the long execution time of BFS and SSSP on the dataset we grouped multiple messages sent by a single MPI rank (process) to the same destination into a single one.

Results (Table 5.2) Table 5.2 presents the BFS and SSSP execution time on the dataset with 100 randomly selected source vertices and different message grouping sizes. As shown, even though the partitionings computed by SARGON had higher edgecut than that of METIS

Table 5.2: BFS and SSSP execution time in seconds on com-orkut dataset with varying message grouping size

Workloads	BFS			SSSP		
Message Grouping Size	64	128	256	64	128	256
METIS	1459	260	73.24	30,784	6,152	727
LDG	2027	396	116	37,418	5,293	870
reLDG	1114	317	83.26	27,099	2,921	677
SARGON	857	238	63.28	21,643	2,426	431

and reLDG (Section 5.4.2), SARGON consistently outperformed LDG and reLDG thanks to its capability of avoiding algorithmic and structural skewness. In comparison to METIS, LDG, and reLDG, SARGON speeded up the execution of BFS and SSSP by up to 2.36 and 2.53 times, respectively.

We also noticed both METIS and reLDG performed better than LDG in most cases. This was probably because METIS and reLDG produced decompositions of lower edgecut than LDG. What we did not expect was that METIS was outperformed by reLDG in many cases even though its decompositions had lower edgecut. We attributed this to the fact that the decompositions computed by METIS had highly skewed active (high-degree) vertex distribution across partitions (Section 5.1.1).

Interestingly, for the BFS execution, reLDG outperformed METIS only if the message grouping size was small enough (when the message grouping size equaled 64). This was because the smaller the message grouping size was the more the messages were communicated, which in turn put more contention on the network interface and memory subsystems and therefore exacerbated the performance impact of skewness. This was further confirmed by the observation that the smaller the message grouping size was, the longer the execution of BFS/SSSP took.

The reason why reLDG was always better than METIS for the execution of SSSP in the experiment was because the execution of SSSP required more data communication than that of BFS. Consequently, in spite of the increasing message grouping size, there would

still be a large number of message exchanges, calling for skew-resistant graph partitioners to avoid both the network and memory contention. This also indicates that SARGON is more suitable for workloads with a large number of small message exchanges and larger graphs. The latter was attributed to the fact that as the size of the graph increased, the amount of data communication would also increase regardless of the message grouping size.

5.4.4 Scalability Study

5.4.4.1 Scalability in terms of Graph Size

Configuration This experiment investigated the scalability of SARGON as the size of the graph increased. Towards this, we first generated six additional datasets by sampling the edge set of the Friendster and Twitter dataset. Then, we examined the BFS execution time on the datasets when they were partitioned across three 20-core machines (with 10 randomly selected source vertices and message grouping size of 512). Note that METIS failed to partition the datasets.

Table 5.3: BFS execution time in seconds with 10 randomly selected source vertices on varying sized graphs

Dataset	Friendster				Twitter			
# of Edges (Billion)	0.9	1.8	2.7	3.6	0.98	1.96	2.94	3.92
LDG	34.01	158	623	1,239	45.65	460	1,092	2,219
reLDG	34.24	132	480	1,171	54.91	403	1,217	2,499
SARGON	26.96	137	392	933	38.53	275	924	1,982

Results (Table 5.3) Table 5.3 shows the corresponding BFS execution time on varying sized graphs. As can be seen, SARGON outperformed LDG and reLDG in almost all the cases. In comparison to LDG and reLDG, SARGON speeded up the execution of BFS by up to 1.67 and 1.46 times, respectively. The speedup remained quite stable regardless of the increasing graph size.

Interestingly, we noticed that reLDG was outperformed by LDG in many cases, especially on the execution of BFS on the Twitter dataset, even though the decompositions computed by reLDG had lower edgecut. This was probably because reLDG tended to

produce decompositions of higher skewness than those of LDG (Section 5.1.1). The fact that the Twitter dataset had higher average vertex degree and higher variation in its vertex degree distribution than that of Friendster dataset further aggravated the performance impact of the skewness.

5.4.4.2 Scalability in terms of # of Partitions

Configuration This experiment inspected the effectiveness of SARGON as the number of partitions increased. Towards this, we first partitioned the original Friendster and Twitter datasets across three up to ten 20-core machines (one partition per core) and then examined the BFS execution time on the partitionings (with 10 randomly selected source vertices and message grouping size of 512).

Table 5.4: BFS execution time in seconds with 10 randomly selected source vertices on varying number of partitions

Datasets	Friendster			Twitter		
# of Partitions	LDG	reLDG	SARGON	LDG	reLDG	SARGON
60	1,239	1,171	933	2,219	2,499	1,982
80	444	318	285	973	771	706
100	148	189	126	264	258	230
120	103	103	71.48	133	172	127
140	85.27	127	69.36	150	147	117
160	58.39	59.32	57.72	70.64	83.30	91.27
180	48.53	54.24	40.00	50.75	54.69	48.84
200	40.35	32.95	34.21	56.48	61.24	44.21

Results (Table 5.4) Table 5.4 shows the corresponding results as the number of partitions increased. As shown, SARGON performed better than LDG and reLDG in almost all the cases in spite of the increasing number of partitions. When compared with LDG and reLDG, SARGON speeded up the execution of BFS by up to 1.55 and 1.49 times, respectively. Consistent with our previous observations, reLDG was better than LDG in many cases. However, it did got beat by LDG in some cases, further highlighting the importance of skew-awareness. The reason why the improvement achieved by SARGON gradually become

smaller was because as the number of partitions increased the impact of skewness was also mitigated due to the reduced work per core (partition). However, the improvement was still non-negligible, since it reduced the execution time of all the computing elements (60 up to 200 cores) by this much.

5.5 CHAPTER SUMMARY

In this chapter, we introduced the multi-label graph partitioning problem and an application of such idea to avoid the skewness of traversal-style graph workloads by being aware of the characteristics of the target workload and the structure of the graph. We also demonstrated the effectiveness and scalability of our proposed solution, Sargon, on many real-world graphs of varying sizes (up to 3.9 billion edges) and varying number of partitions.

6.0 CONCLUSIONS AND FUTURE WORK

6.1 MAIN CONTRIBUTIONS

This thesis considered the well-known graph partitioning problem. We claim that the computation performed on the partitionings computed by existing graph partitioning algorithms does not efficiently utilize modern HPC infrastructures. This impedes the efficiency of computing infrastructure as well as the scalability of the target workload. We advocate for *architecture- and workload-aware* graph partitioning to enable efficient distributed graph computation.

We first investigated the performance impact of modern HPC infrastructures on distributed graph workloads. As a result of this study, we identified two important factors one should consider when partitioning the graphs: (1) the non-uniform network communication costs of the underlying computing infrastructures; and (2) the contention for the shared hardware resources on the memory subsystems of modern HPC clusters. We also provided a holistic view on: (a) why we have to be aware of the characteristics of modern HPC infrastructures for distributed graph workloads; and (b) to what extent these characteristics may impact the performance of distributed graph workloads.

To avoid such negative performance impact, we proposed an **architecture- and workload-aware graph partitioning algorithm,** Argo, for efficient distributed graph computation on static graphs. Argo follows the same streaming model proposed by other graph partitioners. In this model, vertices arrive at the partitioner in a certain order along with their adjacency lists. The partitioner decides the placement of each arrived vertex to one of the partitions permanently based on the placements of the vertices previously arrived. The key novelty of Argo lies in making the vertex placement aware of (a) the non-uniform network

communication costs of the underlying computing infrastructures; and (b) the contentiousness of the memory subsystems of modern HPC clusters. We also make ARGO aware of the runtime characteristics of the target workload by encoding such information into the vertex and edge weights of the graph.

We then presented four new **graph repartitioning algorithms:** ARAGON**,** PARAGON**,** PLANAR**, and** PLANAR+ for efficient distributed graph computation on dynamic graphs. They all attempt to adapt the current partitioning to the changes in the graph by migrating vertices among the partitions. The migration is only allowed if the gain of moving the vertex from its current partition to an alternative partition is positive. The gain of migrating a vertex is defined as the reduction in the communication cost incurred by the vertex during the computation. One of the key contributions of this thesis is that we make the vertex gain computation process aware of both the communication heterogeneity and the contentiousness of the underlying computing infrastructures.

Out of the four new algorithms, ARAGON is a centralized solution with the assumption that the graphs are small enough to be held in the memory of a single machine, whereas PARAGON is a parallel version of ARAGON designed for median-sized graphs. PLANAR and PLANAR+ overcome the drawbacks of PARAGON by scaling it to even larger graphs and by increasing the degree of parallelism of the repartitioning algorithm. PLANAR+ further reduces the overhead of PLANAR, by introducing an efficient way of modeling the communication heterogeneity and contentiousness. This, in turn, enables an optimized vertex gain computation. Making the partitioning algorithms scale efficiently against large graphs is another key contribution of the thesis.

Tables 6.1 and 6.2 provide a brief summary for the four proposed architecture- and workload-aware graph repartitioners. Table 6.1 summarizes our proposed repartitioners in terms of (a) the expected size of the graphs that each algorithm can handle; (b) the optimization objective; (c) the maximum and average hopcut/edgecut reduced by each algorithm when compared with the initial partitioning on the YouTube, as-skitter, com-lj, and com-orkut datasets; and (d) the maximum and average percentage of vertices migrated. Table 6.2 summarizes the repartitioners in terms of (a) the largest graph evaluated; (b) the largest number of partitions evaluated; and (c) the maximum speedup achieved against LDG for

the execution of PageRank (20 Iterations) on the partitionings of the Friendster dataset as well as the actual CPU time saved. The CPU time saved is defined as:

$$CPUTimeSaving = (JETSaving - repartTime) * n \tag{6.1}$$

Here, $JETSaving$, $repartTime$, and n denote the reduction in the workload execution time, the time taken by the repartitioners to compute the partitioning, and the number of computing elements (cores) used, respectively.

Table 6.1: A summary of the Proposed Graph Repartitioners: Part1

Algorithms	Desired Graph Size	Optimization Objectives		Hopcut Reduced (Figure 4.36a)		Edgecut Reduced (Figure 4.36b)		Vertex Mig. Ratio (Figure 4.37)	
		Edgecut or Hopcut	Mig. Cost	Max	Avg.	Max	Avg.	Max	Avg.
ARAGON	Small	Hopcut	✓	30.0%	18.5%	17.0%	10.2%	23.0%	17.7%
PARAGON	Median-Sized			30.2%	17.8%	16.2%	9.5%	21.6%	16.4%
PLANAR	Large			42.0%	25.3%	33.0%	19.1%	48.6%	40.1%
PLANAR+	Large			42.4%	25.8%	34.4%	17.8%	31.0%	24.6%
UNIPLANAR+	Large	Edgecut		39.6%	21.8%	36.8%	18.7%	28.6%	24.2%

Table 6.2: A summary of the Proposed Graph Repartitioners: Part2

Algorithms	Largest Graph Evaluated		Largest # of Partitions Evaluated	Evaluation of PageRank on Friendster with 120 Partitions (Figure 4.40a)	
	$\|V\|$	$\|E\|$		Max Speedup	CPU Time Saved
ARAGON	3M	234M	40	NA	NA
PARAGON	124M	3.6B	240	2.09	25h
PLANAR				3.2	27h
PLANAR+					43h
UNIPLANAR+				1.23	10h

Finally, we looked at the problem of **skew-resistant graph partitioning** for graph workloads with predictable runtime characteristics. Towards this, we studied the runtime characteristics of two representative traversal-style graph workloads: BFS and SSSP. Based on the study, we proposed the idea of multi-label graph partitioning (MLGP) and an application of this idea is to do skew-resistant graph partitioning.

6.2 MAIN IMPACT

The main impact of this thesis is that we identified an important aspect that has been ignored by the current graph processing community, that is, the performance impact of the underlying HPC infrastructures, especially the contentiousness of the memory subsystems, on distributed graph computation. In fact, this is also a blind spot for general distributed computation, where people often assume that the network is the bottleneck.

In particular, we made an in-depth analysis about the factors that one should consider while (re)partitioning the graph for distributed graph computing, namely, the non-uniform network communication costs (Section 2.2.1) and the contention on the memory subsystems (Section 2.2.2). We also experimentally demonstrated (1) that the network may not always be the bottleneck in modern HPC clusters (Section 2.2.3); and (2) that the contention on the memory subsystems can impact the performance of distributed graph computation significantly (Section 2.2.3). Based on our analysis and our observations, we showed that even with simple managed graph (re)partitioning we can achieve significantly better performance (Chapters 3 & 4). All these observations will enable the graph processing community to rethink the design of graph (re)partitioning algorithms and even the design of distributed graph computing frameworks.

6.3 DISCUSSION AND FUTURE WORK

Note that we did not claim that we have solved the problem of architecture- and workload-aware graph partitioning. Instead, the most important contribution of this thesis is a demonstration of the importance of architecture-awareness (heterogeneity and contentiousness) for distributed graph computation on modern HPC clusters as well as a set of possible solutions. In the rest of this section, we will discuss the limitations of our proposed solutions. Addressing these limitations is left as future work.

Time Complexity Comparison Ideally, we would like to include a formal comparison of the time complexity for different graph (re)partitioners. However, for distributed graph (re)partitioners, the (re)partitioning overhead may rely more on the amount of communication required by the (re)partitioning algorithm. As a result, the time complexity may not always be a good indicator to look at. The fact that many graph (re)partitioners simply do not have such complexity analysis in their published papers further increases the difficulty of comparison. Additionally, the memory usage of the graph (re)partitioners is another important factor one should consider while choosing the (re)partitioners. Many graph (re)partitioners simply do not work with large graphs either because it takes too much time to compute a partitioning or it consumes too much memory. To a certain point, we have made an indirect comparison of this in Table 2.1. The lightweight property we talked about covers all the three dimensions: time complexity, space complexity, and data communication.

Heterogeneity-Awareness Modeling Clearly, all our proposed architecture-aware graph (re)partitoiners require a fairly accurate measurement of the relative network communication costs among the computing elements. Although PLANAR+ has provided an efficient way to measure the costs, its effectiveness was only evaluated on a relatively small cluster with a simple network topology in our experimental study. For larger clusters with complicated network topologies, the solution may not work well. In addition to this, the variability of the relative network communication costs may increase, as the size of the cluster and the complexity of the network topology increase. Nevertheless, both PARAGON and PLANAR+ allow customized solutions for relative network communication cost measurement.

Contention-Awareness Modeling Another limitation of our proposed solutions is that we require users to do some profiling of the target workload on the computing infrastructures to determine the ideal λ value, the degree of contentiousness. The parameter λ depends on many factors, such as the characteristics of the graph dataset that the workload is operating on, the characteristics of the graph algorithm, as well as the characteristics of the underlying computing infrastructures. As a result, the profiling may become an extra burden for the users. In addition to this, the set of computing elements used for profiling may be different from the set of computing elements used for actual computation in many cases, and users may not have control over this. Thus, one of the things people could look at is to automate this profiling process (using some machine learning techniques for example).

Workload-Awareness Modeling Currently, we encode the characteristics of the target workload into weights/sizes of the vertices and edges for (re)partitioning. Each vertex/edge of the graph can only be assigned with a single weight/size. However, vertices of the graph may have different computation and communication requirements in different time periods of the computation. Thus, a single weight/size could not always accurately reflect the requirements. This could also be a part of the future work people could explore. In fact, the modeling of the computation and communication requirements of the workload is also not an easy task, especially for scientific simulations.

Additionally, we use the knowledge of the execution traces for skew-resistant graph partitioning. However, the solution we adopted in Sargon is somewhat simplistic. A more complete solution (e.g., a machine learning model to predict the characteristics) is required to increase the potential benefit/generality of Sargon.

Number of Partitions In the current implementation of our proposed solutions, especially Paragon, Planar, and Planar+, we assume that the number of partitions remains the same as int the initial partitioning. However, users may sometimes want to repartition the graph into a different number of partitions. To the best of our knowledge, a large body of existing graph repartitioners, including ours, do not support this. This could also be an interesting problem to look at. Along these lines, it would also be interesting to examine the problem of how to determine the optimal number of partitions for a given workload and a

given graph dataset.

In addition to this, all the experimental studies of our proposed solutions were designed and performed with the assumption of one partition per core in mind. Sometimes, it may make sense to do over decomposition, that is, assigning more than one partitions to a core. Theoretically, all our proposed solutions can also work in the over decomposition case. However, it has not been thoroughly evaluated.

Vertex-Cut Based Graph Partitioning The focus of this thesis is the edgecut-based graph partitioning problem, where vertices of the graph are distributed across partitions by cutting edges if needed. However, it would also be interesting to investigate the architecture- and workload-aware vertex-cut based graph partitioning problem, where edges of the graph are assigned to partitions by cutting vertices if needed.

7.0 BIBLIOGRAPHY

[1] 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*, pp. 135–146, 2010.

[2] 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.

[3] 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*, pp. 17–30, 2012.

[4] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

[5] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *Proceedings of the 1995 International Conference on Parallel Processing*, pp. 113–122, 1995.

[6] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[7] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[8] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1222–1230, 2012.

[9] J. Nishimura and J. Ugander, "Restreaming graph partitioning: simple versatile algorithms for advanced balancing," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1106–1114, 2013.

[10] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pp. 333–342, 2014.

[11] I. Moulitsas and G. Karypis, "Architecture aware partitioning algorithms," in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 42–53, 2008.

[12] 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 Third ACM Symposium on Cloud Computing*, p. 3, 2012.

[13] N. Xu, B. Cui, L.-n. Chen, Z. Huang, and Y. Shao, "Heterogeneous Environment Aware Streaming Graph Partitioning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 6, pp. 1560–1572, 2015.

[14] J. Xue, Z. Yang, S. Hou, and Y. Dai, "When computing meets heterogeneous cluster: Workload assignment in graph computation," in *Big Data (Big Data), 2015 IEEE International Conference on*, pp. 154–163, 2015.

[15] Mellanox, "InfiniBand and Mellanox usage on the November 2016 TOP500 list." http://www.mellanox.com/solutions/hpc/top500.php, 2016.

[16] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The End of Slow Networks: It's Time for a Redesign," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 528–539, 2016.

[17] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 283–294, 2011.

[18] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248–259, 2011.

[19] 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 *Parallel and Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, 2010.

[20] A. Zheng, A. Labrinidis, P. K. Chrysanthis, and J. Lange, "Argo: Architecture-Aware Graph Partitioning," in *Big Data (Big Data), 2016 IEEE International Conference on*, pp. 284–293, 2016.

[21] A. Zheng, A. Labrinidis, and P. K. Chrysanthis, "Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing," in *Big Data (Big Data), 2014 IEEE International Conference on*, pp. 78–85, 2014.

[22] A. Zheng, A. Labrinidis, P. Pisciuneri, P. K. Chrysanthis, and P. Givi, "Paragon: Parallel Architecture-Aware Graph Partitioning Refinement Algorithm," in *19th International Conference on Extending Database Technology*, pp. 365–376, 2016.

[23] A. Zheng, A. Labrinidis, and P. K. Chrysanthis, "Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Repartitioning," in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pp. 121–132, 2016.

[24] A. Zheng, P. Pisciuneri, A. Labrinidis, P. K. Chrysanthis, J. Lange, and P. Givi, "Planar+: Parallel Lightweight Architecture-Aware Graph Repartitioning," *Under Submission*, 2017.

[25] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[26] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 169–182, 2013.

[27] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[28] 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 *European Conference on Parallel Processing*, pp. 451–462, 2014.

[29] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[30] M. Sarwat, S. Elnikety, Y. He, and G. Kliot, "Horton: Online query execution engine for large distributed graphs," in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pp. 1289–1292, 2012.

[31] D. Yan, J. Cheng, T. Ozsu, F. Yang, Y. Lu, J. C. Lui, Q. Zhang, and W. Ng, "A general-purpose query-centric framework for querying big graphs [innovative systems and applications]," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 564–575, 2016.

[32] G. Karypis *et al.*, "METIS: Serial Graph Partitioning and Fill-reducing Matrix Ordering." http://glaros.dtc.umn.edu/gkhome/metis/metis/overview, 1995.

[33] K. Schloegel *et al.*, "Parmetis: Parallel graph partitioning and sparse matrix ordering library." http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview, 2000.

[34] F. Pellegrini *et al.* http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/, 2016.

[35] B. Hendrickson and R. Leland, "Chaco: Algorithms and Software for Partitioning Meshes." http://www.sandia.gov/~bahendr/chaco.html, 1995.

[36] U. Catalyurek *et al.*, "Parallel Partitioning, Load Balancing and Data-Management Services." http://www.cs.sandia.gov/zoltan/, 2013.

[37] L. M. Erwan, L. Yizhong, and T. Gilles, "(Re) partitioning for stream-enabled computation," *arXiv:1310.8211*, 2013.

[38] D. Margo and M. Seltzer, "A Scalable Distributed Graph Partitioner," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1478–1489, 2015.

[39] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "xdgp: A dynamic graph processing system with adaptive partitioning," *arXiv preprint arXiv:1309.1049*, 2013.

[40] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 553–564, 2013.

[41] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic partitioning for distributed social network graph databases," in *18th International Conference on Extending Database Technology*, pp. 25–36, 2015.

[42] N. Xu, L. Chen, and B. Cui, "LogGP: a log-based dynamic graph partitioning method," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1917–1928, 2014.

[43] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed Power-law Graph Computing: Theoretical and Empirical Analysis," in *Advances in Neural Information Processing Systems 27*, pp. 1673–1681, 2014.

[44] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-Based Partitioning for Power-Law Graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 243–252, 2015.

[45] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, "GrapH: Heterogeneity-Aware Graph

Computation with Adaptive Partitioning," in *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pp. 118–128, 2016.

[46] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "OpenMPI: A High Performance Message Passing Library." http://www.open-mpi.org/, 2004.

[47] J. Liu *et al.*, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE." http://mvapich.cse.ohio-state.edu/, 2004.

[48] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Limic: Support for high-performance mpi intra-node communication on linux cluster," in *Parallel Processing, 2005. ICPP 2005. International Conference on*, pp. 184–191, 2005.

[49] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis," in *Parallel Processing, 2009. ICPP'09. International Conference on*, pp. 462–469, 2009.

[50] Gordon, "SDSC Gordon cluster." https://portal.xsede.org/sdsc-gordon, 2016.

[51] HPE, "RDMA protocol: improving network performance." http://h21007.www2.hpe.com/portal/download/files/unprot/c00589475.pdf, 2016.

[52] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[53] Orkut, "Orkut Community Archive." https://orkut.google.com/en.html, 2014.

[54] A. Mislove, *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.

[55] H. McCraw *et al.*, "Performance Application Programming Interface." http://icl.cs.utk.edu/papi/, 2012.

[56] 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 *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 32–39, 2006.

[57] Intel, "Intel Data Direct I/O Technology." http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2012.

[58] Wikipedia, "Message Passing Interface (MPI)." https://en.wikipedia.org/wiki/Message_Passing_Interface, 2017.

[59] A. Buluç and K. Madduri, "Parallel Breadth-First Search on Distributed Memory Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 65, 2011.

[60] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 281–292, 2014.

[61] J. Leskovec and A. Krevl, "Stanford Large Network Dataset Collection." http://snap.stanford.edu/data, 2014.

[62] J. Kunegis, "Konect: the koblenz network collection." http://konect.uni-koblenz.de/networks/, 2013.

[63] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, pp. 251–262, 1999.

[64] E. Papalexakis, B. Hooi, K. Pelechrinis, and C. Faloutsos, "Power-Hop: A Pervasive Observation for Real Complex Networks," *PloS one*, vol. 11, no. 3, p. e0151027, 2016.

[65] SAM, "The HPC Cluster at the University of Pittsburgh." http://core.sam.pitt.edu/MPIcluster, 2016.

[66] O. S. U. NOWLAB, "osu_latency Benchmark." http://mvapich.cse.ohio-state.edu/benchmarks/, 2016.

[67] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation, 1982. 19th Conference on*, pp. 175–181, 1982.

[68] C. Schulz, *Scalable parallel refinement of graph partitions*. PhD thesis, Karlsruhe Institute of Technology, May 2009.

[69] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, G. Zheng, *et al.*, "Communication and Topology-aware Load Balancing in Charm++ with TreeMatch," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pp. 1–8, 2013.

[70] S. Micali and V. V. Vazirani, "An $O(v|v|c|E|)$ algorithm for finding maximum matching in general graphs," in *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pp. 17–27, 1980.

[71] P. Pisciuneri, S. L. Yilmaz, P. Strakey, and P. Givi, "An Irregularly Portioned FDF Simulator," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C438–C452, 2013.

[72] Sandia, "Sandia National Laboratories TNF Workshop Website, Piloted Jet Flames." http://www.sandia.gov/TNF/pilotedjet.html, 2016.

[73] Kraken, "Kraken Cray XT5." http://www.nics.tennessee.edu/computing-resources/kraken, 2014.

[74] C. Walshaw, "Chris Walshaw Collection." http://staffweb.cms.gre.ac.uk/~wc06/partition/, 2000.

[75] D. A. Bader *et al.*, "10th DIMACS Challenge." http://www.cc.gatech.edu/dimacs10/, 2012.

[76] C. Demetrescu *et al.*, "9th DIMACS Challenge." http://www.dis.uniroma1.it/challenge9, 2006.

[77] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur, "High performance MPI-2 one-sided communication over InfiniBand," in *Cluster Computing and the Grid, 2004. IEEE International Symposium on*, pp. 531–538, 2004.

[78] Open MPI Team, "Portable Hardware Locality (hwloc)." http://www.open-mpi.org/projects/hwloc/, 2016.