PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm

Angen Zheng
University of Pittsburgh
Pittsburgh, PA, USA
anz28@cs.pitt.edu

Alexandros Labrinidis
University of Pittsburgh
Pittsburgh, PA, USA
labrinid@cs.pitt.edu

Patrick Pisciuneri
University of Pittsburgh
Pittsburgh, PA, USA
php8@pitt.edu

Panos K. Chrysanthis
University of Pittsburgh
Pittsburgh, PA, USA
panos@cs.pitt.edu

Peyman Givi
University of Pittsburgh
Pittsburgh, PA, USA
pgivi@pitt.edu

ABSTRACT

With the explosion of large, dynamic graph datasets from various fields, graph partitioning and repartitioning are becoming more and more critical to the performance of many graph-based Big Data applications, such as social analysis, web search, and recommender systems. However, well-studied graph (re)partitioners usually assume a homogeneous and contention-free computing environment, which contradicts the increasing communication heterogeneity and shared resource contention in modern, multicore high performance computing clusters. To bridge this gap, we introduce PARAGON, a *parallel architecture-aware* graph partition refinement algorithm, which mitigates the mismatch by modifying a given decomposition according to the nonuniform network communication costs and the contentiousness of the underlying hardware topology. To further reduce the overhead of the refinement, we also make PARAGON itself architecture-aware.

Our experiments with a diverse collection of datasets showed that on average PARAGON improved the quality of graph decompositions computed by the de-facto standard (hashing partitioning) and two state-of-the-art streaming graph partitioning heuristics (deterministic greedy and linear deterministic greedy) by 43%, 17%, and 36%, respectively. Furthermore, our experiments with an MPI implementation of Breadth First Search and Single Source Shortest Path showed that, in comparison to the state-of-the-art streaming and multi-level graph (re)partitioners, PARAGON achieved up to 5.9x speedups. Finally, we demonstrated the scalability of PARAGON by scaling it up to a graph with 3.6 billion edges using only 3 machines (60 physical cores).

1. INTRODUCTION

It is well-known that graph (re)partitioning has been extensively studied in the area of scientific simulations [14, 34]. Yet, its importance is continuously increasing due to the explosion of large graph datasets from various fields, such as the World Wide Web, Protein Interaction Networks, Social Networks, Financial Networks, and Transportation Networks. This has led to the development of graph-specialized parallel computing frameworks, e.g., Pregel [21], GraphLab [19], and PowerGraph [13].

Pregel, as a representative of these computing frameworks, embraces a vertex-centric approach where the graph is partitioned across multiple servers for parallel computation. Computations are often divided into a sequence of *superstep*s separated by a global synchronization barrier. During each superstep, a user-defined function is computed against each vertex based on the messages it received from its neighbors in the previous step. The function can change the state and outgoing edges of the vertex, send messages to the neighbors of the vertex, or even add or remove vertices/edges to the graph.

Traditional Graph Partitioners Clearly, the distribution of the graph data across servers may impact the performance of target applications significantly. *Graph partitioning* has been studied for decades [14, 34], attempting to provide a good partitioning of the graph data, whereby both the skewness and the communication (edge-cut) among partitions are minimized as much as possible, in order to minimize the total response time for the entire computation. However, classic graph partitioners such as METIS [23] and Chaco [7] do not scale well with large graphs.

Streaming Graph Partitioners Streaming graph partitioners (e.g., DG/LDG [39], arXiv'13 [11], and Fennel [42]) have been proposed in order to overcome the scalability challenges of classic graph partitioners, by examining the graph incrementally. One of the main shortcomings of these approaches is that they also assume uniform network communication costs among partitions as classic graph partitioners do. That is, they all assume that the communication cost is proportional only to the amount of data communicated among partitions. *This assumption is no longer valid in modern parallel architectures due to the increasing communication heterogeneity* [47, 8]. For example, on a $4 * 4 * 4$ 3D-torus interconnect, the distance to different nodes starting from a single node varies from 0 to 6 hops.

Architecture-Aware Graph Partitioners Architecture-aware graph partitioners [24, 8, 46] have been proposed to improve the mapping of the application's communication patterns to the underlying hardware topology. Chen et al. [8] (SoCC'12) took architecture-awareness a step further, by making the partitioning algorithm itself partially aware of the communication heterogeneity. However, both [8] and [24] (ICA3PP'08) are built on top of existing heavyweight graph partitioners, namely, METIS [23] and PARMETIS [30], which

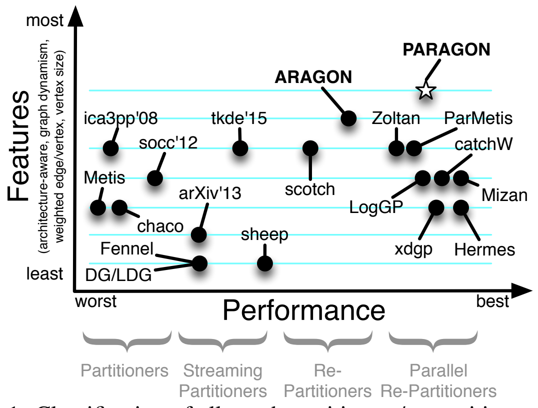


Figure 1: Classification of all graph partitioners/re-partitioners according to their features vs performance profile.

are known to be the best graph partitioners and repartitioners in terms of partitioning quality but have poor scalability. Finally, although Xu et al. [46] (TKDE'15) proposed a lightweight architecture-aware streaming graph partitioner, the partitioner may lead to suboptimal performance for dynamic graphs [43].

Traditional Graph Repartitioners Most real-world graphs are often non-static, and continue to evolve over time. Because of this *graph dynamism*, both the quality of the initial partitioning and the mapping of the application communication pattern to the underlying hardware topology will continuously degrade over time, leading to (a potentially significant) load imbalance and additional communication overhead. Considering the sheer scale of real-world graphs, repartitioning the entire graph from scratch using [46, 8, 24], even in parallel, is often impractical, either because of the long partitioning time or the huge volume of data migration the repartitioning may introduce. To address this, several graph repartitioning algorithms have been proposed, such as Zoltan [1, 6] and PARMETIS [30, 33]. Although they are able to greatly reduce the data migration cost, they are all architecture-agnostic and do not scale well with massive graphs.

Parallel/Lightweight Graph Repartitioners Parallel lightweight graph repartitioners (e.g., CatchW [37], xdgp [43], Hermes [26], Mizan [17], arXiv'13 [11], and LogGP [45]) have been proposed to improve the performance and scalability of graph repartitioning. Instead of seeking an optimal partitioning at once, these algorithms adapt the graph decomposition to changes efficiently by incrementally migrating vertices from one partition to another based on some local heuristics. However, they are all oblivious of the nonuniform network communication costs among partitions.

Limitations of the State-of-the-Art Despite the plethora of graph partitioners and repartitioners (Figure 1), the current state-of-the-art is suffering from two main problems:

- Graph (re)partitioners either consider architecture-awareness (for CPU/network heterogeneity) or consider performance (i.e., parallel/lightweight implementation), but never both. This is illustrated in Figure 1, where the top-right corner is empty (except for PARAGON, which is presented in this paper).
- No existing graph (re)partitioner considers the issue of *shared resource contention* in modern multicore high performance computing (HPC) clusters. Shared resource contention is a well-known issue in multicore systems and has received a lot of attention in system-level research [15, 41].

Our prior work We have previously presented an architecture-aware graph repartitioner, ARAGONLB [48]. Although ARAGONLB considers the communication heterogeneity for target applications, it disregards the issue of shared resource contention, and the repartitioning itself is not architecture-aware. Moreover, the refinement algorithm that ARAGONLB uses to improve the mapping of the application communication pattern to the underlying hardware topology requires the entire graph to be stored in memory by a single server, which is infeasible for large graphs. Furthermore, the refinement algorithm is performed sequentially, which may become a performance bottleneck. Finally, ARAGONLB assumes that compute nodes used for parallel computation have the same number of cores and memory hierarchies, which may not always be true.

Contributions In this paper, we present PARAGON, which overcomes both limitations of the current state-of-the-art graph repartitioners by extending ARAGONLB in the following aspects.

1. We separate the refinement algorithm, ARAGON, from ARAGONLB as an independent component, and develop a parallelized version of ARAGON, PARAGON, for large graphs (Section 3, 4, 5). We further reduce the overhead of PARAGON by making it aware of the nonuniform network communication costs (explained in Section 2.1).
2. We identify and consider the issue of shared resource contention in modern HPC clusters for graph partitioning (Section 2.2 & 6).
3. We perform an extensive experimental study of PARAGON with a diverse set of 13 datasets and two real-world applications, demonstrating the effectiveness and scalability of PARAGON (Section 7).

2. MOTIVATION

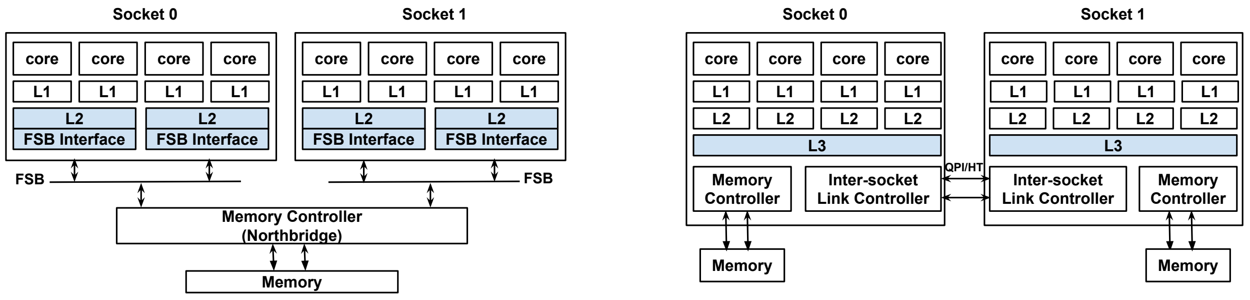
In this section we explain the importance of architecture-awareness (i.e., communication heterogeneity and shared resource contention) for efficient graph (re)partitioners.

2.1 Communication Heterogeneity

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 [27, 25], a predominant messaging standard for HPC applications, intra-node communication is implemented via shared memory/cache [16, 5], while inter-node communication needs to go through the network interface. Additionally, both inter-node and intra-node communication are themselves nonuniform.

Nonuniform 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 [28], 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.

Nonuniform Intra-Node Network Communication Communication among cores of the same compute node is also nonuniform because of the complex memory hierarchy. Communication among



(a) Uniform Memory Access (UMA) Architecture (b) Nonuniform Memory Access (NUMA) Architecture

Figure 2: Example Architectures of Modern Compute Nodes

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 2a, 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 2b, 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 2a) or the inter-socket link controller (Figure 2b).

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

2.2 Intra-Node Shared Resource Contention

As mentioned above, MPI intra-node communication is implemented via shared memory, which can either be user-space or kernel-based [16, 5]. Current MPI implementations often use the former for small messages and the latter for large messages. The user-space approach requires two memory copies. The sender first needs to load the send buffer into its cache and then write the data to the shared buffer (which may require loading the shared buffer block into the sender's cache first). Then, the receiver reads the data from the shared memory (which may demand loading the shared memory block and receiving buffer into the receiver's cache first). For kernel-based approaches, the receiver first loads the send buffer directly to its cache with the help of the OS kernel. Then, the receiver writes the data to the receiving buffer (which may require loading the receiving buffer into its cache first). Clearly, kernel-based approaches reduce the number of memory copies to be one, mitigating the traffic on the memory subsystem. However, it demands a trap to the kernel on both the sender and receiver, making it inefficient for small messages. As can be seen, intra-node communication generates lots of memory traffic and cache pollution, which may saturate the memory subsystem if we put too much communication within each compute node. This issue is further amplified by the increasing contentiousness of the shared resources in modern multicore systems. Table 1 summarizes the resources that different cores may have to compete for when they are communicating with each other for the architectures presented in Figures 2a and 2b. The

Table 1: Intra-Node Shared Resource Contention

Cores/Resources		Sharing		Contention		
Core Groups		Socket	LLC	LLC	FSB/QPI(HT)	Memory Controller
UMA Fig. 2a	G1	✓	✓	✓	✓	✓
	G2	✓			✓	✓
	G3					✓
NUMA Fig. 2b	G1	✓	✓	✓		✓
	G2				✓	

summary is based on whether the cores are on the same socket and whether they share the last level cache (LLC).

Take-away *Focusing solely on placing neighboring vertices as close as possible is not sufficient to achieve superior performance. In fact, putting too much communication within each compute node may even hurt the performance due to the traffic congestion on memory subsystems.* Counter-intuitively, offloading a certain amount of intra-node communication across compute nodes may sometimes achieve better performance. This is because inter-node communication is often implemented using Remote Direct Memory Access (RDMA) and rendezvous protocols [40], which allow a compute node to read data from the memory of another compute node without involving the processor, cache, or operating system of either node, thus alleviating the traffic on memory subsystems and cache pollution. Additionally, it is reported in [3] that modern RDMA-enabled networks can deliver comparable network bandwidth as that of memory channels. This requires us to examine the impact of multi-core architecture on graph partitionings more carefully, especially for small HPC clusters, since the network may no longer be the bottleneck.

3. PROBLEM STATEMENT

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

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

and let M be the current assignment of partitions to servers, where P_i is assigned to server $M[i]$. The server can be either a hardware thread, a core, a socket, or a machine.

Architecture-aware graph partition refinement 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 refinement should also minimize the data migration cost among partitions. Also, to ensure balanced load distribution in terms of the computation requirement, the refinement should keep the skewness of the partitioning as small as possible.

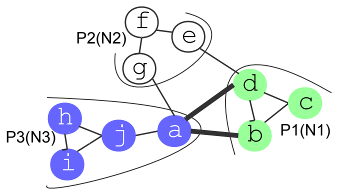


Figure 3: Old Decomposition

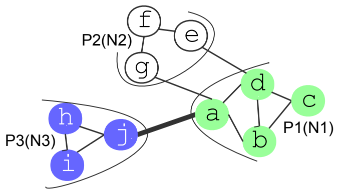


Figure 4: Better Decomposition

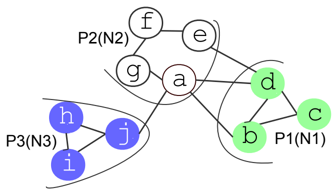


Figure 5: Best Decomposition

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

Figure 6: Relative Network Communication Costs

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

$$comm(G,P) = \alpha * \sum_{\substack{e=(u,v)\in E \\ \text{and } u\in P_i \text{ and } v\in P_j \text{ and } i\neq j}} w(e) * c(P_i,P_j) \quad (2)$$

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, $w(e)$ is the edge weight, indicating the amount of data communicated along the edge per superstep, 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 $c(P_i,P_j)=1$.

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

$$mig(G,P,P') = \sum_{\substack{v\in V \\ \text{and } v\in P_i \text{ and } v\in P'_j \text{ and } i\neq j}} vs(v) * c(P_i,P'_j) \quad (3)$$

where $vs(v)$ is the vertex size, reflecting the amount of application data represented by v, and P' denotes the partitioning after being refined/repartitioned.

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

$$skewness(G,P) = \frac{\max\{w(P_1), w(P_2), \cdots, w(P_n)\}}{\frac{\sum_{i=1}^{n} w(P_i)}{n}} \quad (4)$$

where $w(P_i)=\sum_{v\in P_i} w(v)$ with $w(v)$ denoting the vertex weight (i.e., the computation requirement of the vertex).

Self Architecture-Awareness In fact, the refinement algorithm itself should be architecture-aware (during its execution), since the refinement may also result in a lot of communication.

4. OUR PRIOR WORK: ARAGON

ARAGON is a serial, architecture-aware graph partition refinement algorithm proposed by us in [48]. It is a variant of the Fiduccia-Mattheyses (FM) algorithm [12]. It tries to reduce the application communication cost by modifying the current decomposition according to the nonuniform network communication costs of the underlying hardware topology. Each time it takes as input two partitions of the n-way decomposition and the relative network communication costs among partitions. For each input partition pair, it attempts to improve the mapping of the application communication pattern to the underlying hardware topology by iteratively moving vertices between them. During each iteration, it tries to find a single vertex such that moving it from its current partition to the alternative partition would lead to a maximal gain, where the gain is defined as the reduction in the communication and migration cost. Upon each movement of a vertex, v, it also updates the gain of v's neighbors of the partition pair. This process is repeated until all vertices are moved once or the decomposition cannot be further improved after a certain number of vertex movements. Since ARAGON can only refine one partition pair at a time, it is repeatedly applied to all partition pairs sequentially.

The gain of moving vertex v from its current partition, P_i, to its refinement partner, P_j, is defined as:

$$g^{i,j}(v) = g^{i,j}_{std}(v) + g^{i,j}_{topo}(v) + g^{i,j}_{mig}(v) \quad (5)$$

Here, $g^{i,j}_{std}(v)$ considers the impact of the movement on the communication between P_i and P_j, defined as:

$$g^{i,j}_{std}(v) = \alpha * (d_{ext}(v,P_j) - d_{ext}(v,P_i)) * c(P_i,P_j) \quad (6)$$

where $d_{ext}(v,P_i)$ denotes the amount of data v communicates with vertices of partition P_i, formally defined as

$$d_{ext}(v,P_i) = \sum_{\substack{e=(v,u)\in E \\ \text{and } v\in P_i \text{ and } u\in P_j \text{ and } i\neq j}} w(e) \quad (7)$$

The second term of Equation 5, $g^{i,j}_{topo}(v)$, considers the impact of the movement on the communication between v and its neighbors in other partitions in addition to P_i and P_j. We define it as:

$$g^{i,j}_{topo}(v) = \alpha * \sum_{\substack{k=1 \\ \text{and } k\neq i \text{ and } k\neq j}}^{n} d_{ext}(v,P_k) * (c(P_i,P_k) - c(P_j,P_k)) \quad (8)$$

The third term of Equation 5, $g^{i,j}_{mig}(v)$, considers the impact of the movement on migration cost, which is defined as:

$$g^{i,j}_{mig}(v) = vs(v) * (c(P_i,P_k) - c(P_j,P_k)) \quad (9)$$

where P_k is the owner of v in the original decomposition. The current owner of v, P_i, may be different from its original owner, P_k, due to the refinement.

Example In the decomposition shown in Figure 3, we have a graph with unit weights and sizes and is initially distributed across 3 machines: N_1, N_2, and N_3. The relative network communication costs among partitions are shown in Figure 6. Clearly, the number of edges among partitions goes from 4 in Figure 3, to 3 in Figure 4. In fact, if we assume uniform network communication costs among partitions, Figure 4 would be the optimal decomposition of the graph. However, if we consider the case where all network costs are not equal (as in Figure 6), then the decomposition in Figure 4 can be further improved by moving vertex a to P_2 (Figure 5). Even though moving vertex a from P_1 to P_2 increases the communication cost between P_1 and P_2 by 1, it actually reduces the communication cost between a and j by 5, since the relative network communication cost between P_1 and P_3 is 6, while that of P_2 and P_3 is 1. For the same reason, moving a to P_2 also decreases the migration cost of a by 5, since vertex a was originally in P_3.

5. PARAGON

Motivation Clearly, one 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 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). In our prior work [48], ARAGON goes for another extreme, where all servers send their local partitions to a single server that is responsible for the refinement of all partition pairs. By doing this, ARAGON only needs to send the entire graph over network once, significantly reducing the communication traffic. One drawback of this approach is that it requires the server to store the entire graph in memory. Another issue is that the server can easily become a performance and scalability bottleneck.

Overview Based on the observation above, PARAGON takes a middle point of the two extremes, 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 1 describes the main idea of PARAGON. During refinement, each server 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 partitions into groups such that each group can be refined independently on different servers and the selection of group servers (Line 4–6). The group servers take responsibility of 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, group servers will start to do the refinement of each group independently (Line 8–13). After finishing the refinement of its group, 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).

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 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, 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 partitions. Fortunately, we find that random grouping along with the *shuffle refinement* (the remedy technique presented below) works quite well.

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.

Algorithm 1: 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 $P_i \in pg$ do
11         foreach $P_j \in pg$ do
12             if $i \neq j$ then
13                 AragonRefinement($P_i, P_j, c$)
14     shuffleRefinement($pg$)
15     vertexLocationUpdate($pg$)
16 physicalDataMigraton($P_l$)
```

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 [1]. 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[1] 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.

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

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

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.

Degree of Refinement Parallelism 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 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.

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, 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:

$$\sum_{P_i \in g} ps[i] * c(P_i, P_s) * (1 + \frac{\sigma(s)}{drp}) \qquad (10)$$

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 group servers are selected, the master node will broadcast the group servers of all groups to all 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.

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 [35] 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 find 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, [35] has presented a solution to parallellize the standard FM algorithms [12]. 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 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.

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 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:

$$\min_{m \in [1,n]} \sum_{i=1 \text{ and } i \neq m}^{n} c(P_i, P_m) \qquad (11)$$

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, 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.

Physical Data Migration To support efficient distributed computation, we also provide a basic migration service for graph workloads. Considering that physical data migration is highly application-dependent, the migration service only takes responsibility for the redistribution of the graph data itself. It is the users who are responsible for the migration of any application data associated with each vertex. That is, the users should save the application context before using our migration service and restore the context afterwards. For example, in breadth first search, each vertex is usually associated with a value indicating its current distance to the source vertex. Users need to keep track of the distance value of each vertex while migrating. For complicated workloads, users can exploit the migration service provided by Zoltan [1] to simplify the migration.

6. CONTENTION AWARENESS

So far, we have presented how we parallelize ARAGON. In this section, we will cover how we make PARAGON aware of the issue of shared resource contention in multicore systems. We know that, guided by a given network communication cost matrix, PARAGON is able to gather neighboring vertices as close as possible, and that the contention is caused by the fact that we put too much communication within the compute nodes. Hence, to avoid serious intra-node shared resource contention, we can simply 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. In our implementation, we refine the intra-node communication costs as follows:

$$c(P_i, P_j) = c(P_i, P_j) + \lambda * (s_1 + s_2) \tag{12}$$

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. Clearly, if $\lambda = 0$, PARAGON will only consider the communication heterogeneity, and $\lambda = 1$ 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. Considering the impact of both 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.

7. EVALUATION

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 7.1). We then validate the effectiveness of PARAGON using two real-world graph workloads: Breadth-First Search (BFS) [4] and Single-Source Shortest Path (SSSP) [20], which we implemented using MPI (Section 7.2). Finally, we demonstrate the scalability of PARAGON via a billion-edge graph (Section 7.3).

Datasets Table 2 describes the datasets used. By default, the graphs were (re)partitioned with vertex weights (i.e., computational requirement) set to be their vertex degree, with vertex sizes (i.e., amount of the data of the vertex) set to be 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. 2). Unless explicitly specified, all the graphs were initially partitioned by DG (deterministic greedy heuristic), a state-of-the-art streaming graph partitioner [39], across cores of the compute 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 partitions. For fairness, DG/LDG were extended to support vertex- and edge-weighted graphs.

Platforms We evaluated PARAGON on two clusters: PittMPICluster [32] and Gordon supercomputer [28]. PittMPICluster had a flat network topology, with all 32 compute nodes connected to a single switch via 56GB/s FDR Infiniband. On the other hand, the Gordon network topology was a 4x4x4 3D torus of switches connected via 8GB/s QDR Infiniband with 16 compute nodes attached to each switch. Table 3 depicts the compute node configuration of both clusters. The results presented were the means of 5 runs, except the

Table 2: Datasets used in our experiments

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	Description
wave [38]	156,317	2,118,662	2D/3D FEM
auto [38]	448,695	6,629,222	3D FEM
333SP [10]	3,712,815	22,217,266	2D FE Triangular Meshes
CA-CondMat [2]	108,300	373,756	Collaboration Network
DBLP [18]	317,080	1,049,866	Collaboration Network
Email-Eron [2]	36,692	183,831	Communication Network
as-skitter [2]	1,696,415	22,190,596	Internet Topology
Amazon [2]	334,863	925,872	Product Network
USA-roadNet [9]	23,947,347	58,333,344	Road Network
PA-roadNet [2]	1,090,919	6,167,592	Road Network
YouTube [18]	3,223,589	24,447,548	Social Network
com-LiveJournal [2]	4,036,537	69,362,378	Social Network
Friendster [2]	124,836,180	3,612,134,270	Social Network

Table 3: Cluster Compute Node Configuration

Node Configuration	PittMPICluster (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

execution of SSSP on Gordon (Section 7.2) and the scalability test (Section 7.3).

Network Communication Cost Modelling The relative network communication costs among partitions (cores) were approximated using a variant of the osu_latency benchmark [29]. 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 [25] on Gordon and OpenMPI 1.8.6 [27] on PittMPICluster. MVAPICH2 and OpenMPI were two different MPI implementations available on the clusters.

7.1 MicroBenchmarks

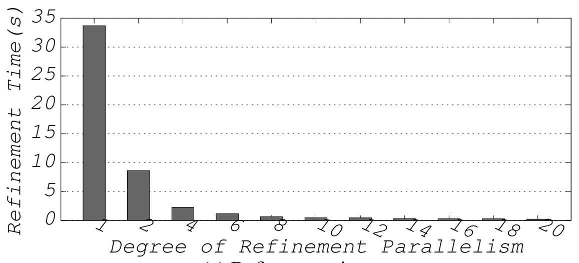
7.1.1 Varying Degree of Parallelism

Configuration In this experiment, we examined the impact of the degree of parallelism in terms of both 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 2 compute nodes of PittMPICluster, and then applied PARAGON to the decompositions with varying degree of refinement parallelism but with shuffle refinement disabled.

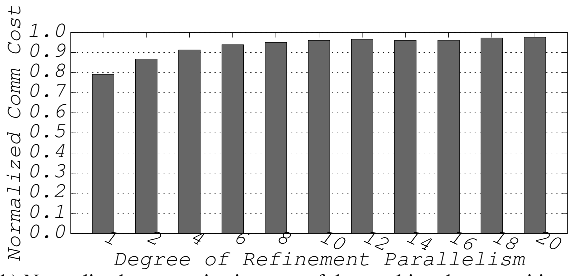
Results (Figures 7a & 7b) Figure 7a plots the runtime of PARAGON on the com-lj dataset for various degrees of parallelism. As expected, the higher the degree of parallelism, the faster the refinement would finish, and PARAGON significantly reduced the refinement time of ARAGON (PARAGON with degree of parallelism of 1). However, the speedup was achieved at the cost of higher communication cost of the resulting decompositions (Figure 7b). 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 cases when compared to the initial decompositions.

7.1.2 Impact of Shuffle Refinement

Configuration 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 8 to 15).



(a) Refinement time



(b) Normalized communication cost of the resulting decompositions

Figure 7: Refinement time and communication costs of the com-lj decompositions after being refined with varying degree of refinement parallelism on two 20-core compute nodes. The communication costs presented were normalized to that of the initial decomposition.

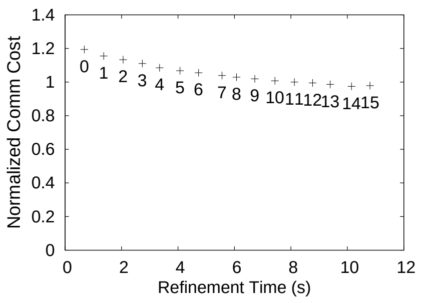


Figure 8: Y-axis corresponds to the communication costs of the com-lj decompositions after being refined with varying number of shuffle refinement times on two 20-core compute nodes when they were normalized to that of the decompositions refined by ARAGON; X-axis denotes the corresponding refinement time; the labels on each data point were the number of refinement times.

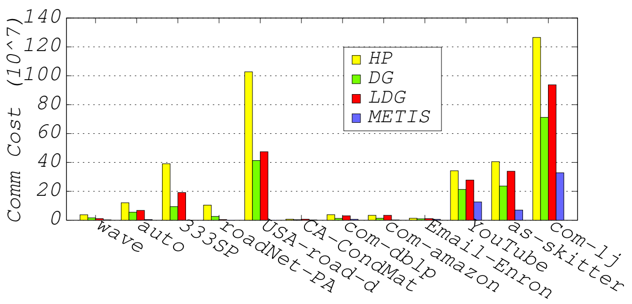


Figure 9: Communication cost of the initial decompositions computed by HP, DG, LDG, and METIS across cores of two 20-core compute nodes for a variety of graphs.

Results (Figure 8) Figure 8 shows the corresponding refinement time and the normalized communication costs of resulting decompositions with the decompositions computed by ARAGON as the baseline. As shown, PARAGON (with shuffle refinement enabled) 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).

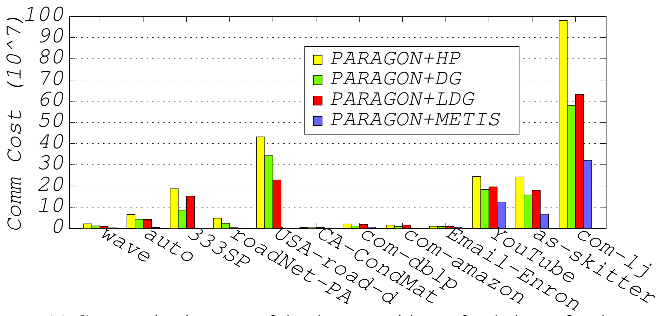
7.1.3 *Impact of Initial Partitioners*

Configuration 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 [39]; and (c) METIS, a state-of-the-art multi-level graph partitioner [23]. The graphs were initially partitioned across 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.

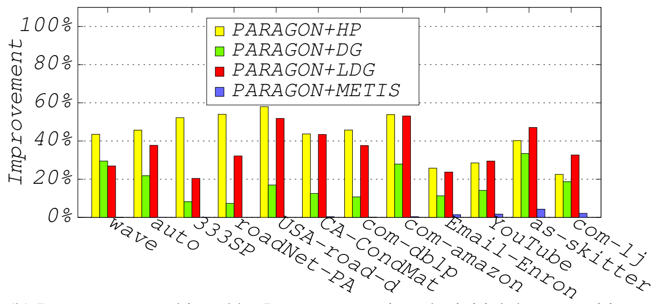
Quality of the Initial Decompositions (Figure 9) Figure 9 denotes the communication cost of the initial decompositions computed by HP, DG, LDG, and METIS for a variety of graphs. As anticipated, METIS performed the best and HP the worst. However, METIS is a heavyweight serial graph partitioner, making 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 [42] 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 [39]. 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.

Quality of the Resulting Decompositions (Figures 10a & 10b) Figures 10a and 10b 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 7.2).

Refinement Overhead (Figures 11a & 11b) We also noticed that the quality of the initial decomposition impacted the refinement overhead greatly. Figures 11a and 11b plot the migration cost (Eq. 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.

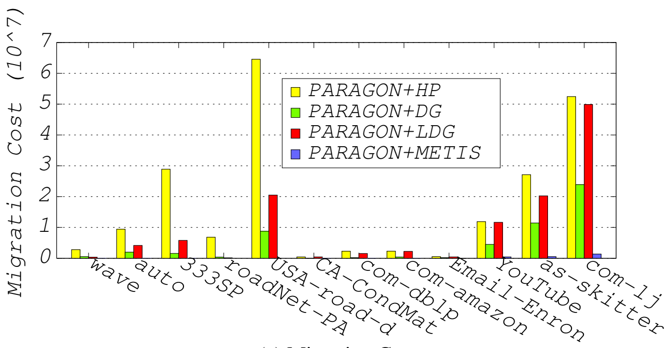


(a) Communication cost of the decompositions after being refined.

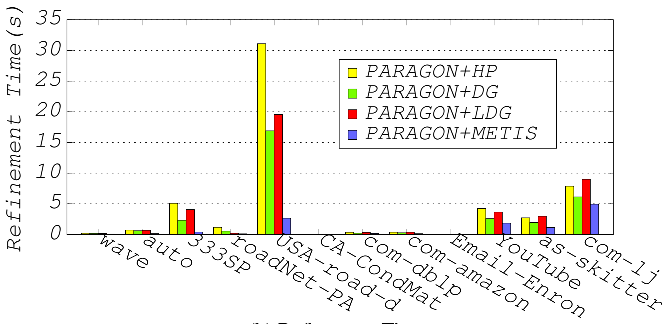


(b) Improvement achieved by PARAGON against the initial decomposition.

Figure 10: 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.



(a) Migration Cost



(b) Refinement Time

Figure 11: 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.

7.2 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 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 (8 for YouTube and as-skitter dataset and 16 for com-lj dataset) messages sent by each MPI rank to the same destination into a single one.

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 PittMPICluster, 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 (56GB vs 8GB) 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 PittMPICluster. Hence, we fixed λ to be 1 on PittMPICluster and 0 on Gordon for the experiment.

Job Execution Time (Tables 4 & 5) Tables 4 and 5 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 [30], 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 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 PittMPICluster, respectively. If we time the improvements by the number of MPI ranks (48 for Gordon and 64 for PittMPICluster), the improvements were more remarkable. Yet, the overhead PARAGON exerted (the sum 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 (edge-cut). Also, if we compare the execution time of BFS/SSSP on both clusters, we would find that the speedup PARAGON achieved by increasing the number of cores from 48 to 60 was much higher than that of DG. 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.

Communication Volume Breakdown (Figures 12 & 13) 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),

Table 4: BFS Job Execution Time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
PittMPICluster			
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 5: SSSP Job Execution Time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
PittMPICluster			
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)

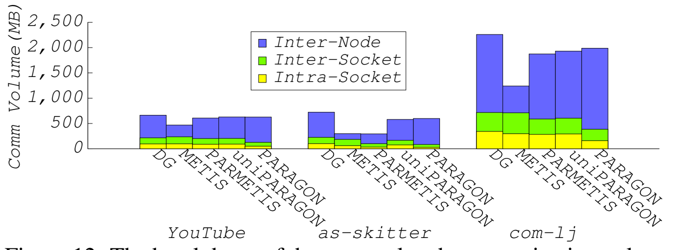


Figure 12: The breakdown of the accumulated communication volume across all supersteps for BFS on PittMPICluster.

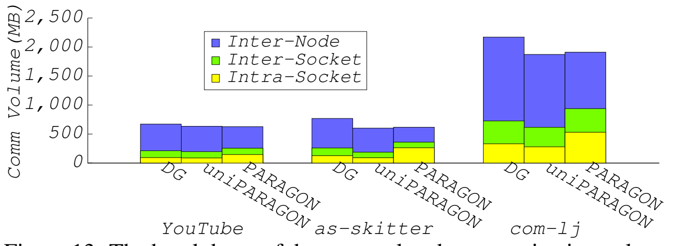


Figure 13: The breakdown of the accumulated communication volume across all supersteps for BFS on Gordon.

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 supersteps for BFS here.

As shown in Figures 12 (for PittMPICluster) and 13 (for Gordon), Paragon and uniParagon have much lower remote communication volume than DG in all 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 PittMPICluster), which was expected given our choice for λ. It is worth mentioning that on PittMPICluster, 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.

Graph Dynamism (Figure 14) 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 14 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.

7.3 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 PittMPICluster with the degree of refinement parallelism, the number of shuffle refinement times, and the message grouping size set to 10, 10, and 256, respectively.

Results (Figures 15 & 16) Figures 15 and 16 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).

8. RELATED WORK

Graph partitioning and repartitioning are receiving more and more attention in recent years due to the proliferation of large graph datasets. In this section, we categorize existing approaches of graph (re)partitioners into three types: (a) heavyweight, (b) lightweight, and (c) streaming, which are presented next.

Heavyweight Graph (Re)Partitioning Graph partitioning and repartitioning has been studied for decades (e.g., Metis [23], ParMetis [30], Scotch [36], Chaco [7], and Zoltan [1]). These graph (re)partitioners are well-known for their capability of producing high-quality graph decompositions. However, they usually require full knowledge of the entire graph for (re)partitioning, making them scale poorly against

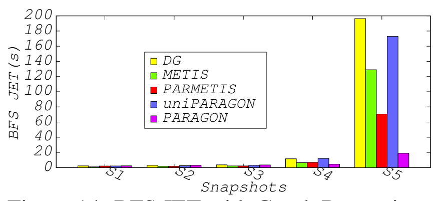


Figure 14: BFS JET with Graph Dynamism

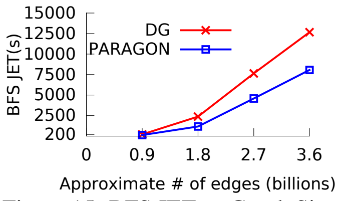


Figure 15: BFS JET vs Graph Size

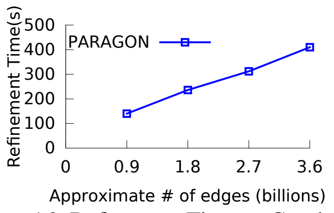


Figure 16: Refinement Time vs Graph Size

large graphs even if performed in parallel. Furthermore, they are all architecture-agnostic. Although [24], a METIS variant, considers the communication heterogeneity, it is a *sequential static graph partitioner*, which is inapplicable for massive graphs or dynamic graphs. Several recent works [48, 8] have been proposed to cope with the heterogeneity and dynamism. However, they are also too heavyweight for massive graphs because of the high communication volume they generate. As a consequence, they are not appropriate for online graph repartitioning in large-scale distributed graph computation. Furthermore, they disregard the issue of resource contention in multicore systems.

Lightweight Graph Repartitioning As a result of the shortcomings of heavyweight graph (re)partitioners, many lightweight graph repartitioners [37, 43, 26, 17, 45] have been proposed. They efficiently adapt the partitioning to changes by incrementally migrating vertices among partitions based on some heuristics (rather than repartitioning the entire graph). Nevertheless, they are not architecture-aware. Also, many of them assume uniform vertex weights and sizes, and some [43, 26] even assume uniform edge weights, which may not always be true.

In fact, work [17] is a Pregel-like graph computing engine, which migrates vertices based on runtime characteristics of the workload (i.e., # of message sent/received by each vertex and response time) instead of the graph structure (i.e., the distribution of vertex neighbors, edge weights, and vertex sizes). Paper [45] also presents a repartitioning system that 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).

Recently, a novel distributed graph partitioner, Sheep [22], has been proposed for large graphs. It is similar in spirit to METIS. That is, 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 outperforms 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 for larger numbers of partitions. However, Sheep is unable to deal with both weighted and dynamic graphs, and it is architecture-agnostic.

Streaming Graph Partitioning Recently, a new family of graph partitioning heuristics, streaming graph partitioning [39, 11, 42], has been proposed for online graph partitioning. They are able to produce partitionings comparable to the heavyweight graph partitioner, METIS, within a relative short time. However, they are architecture-agnostic. Although [46] has presented a streaming graph partitioner with awareness of both compute and communication heterogeneity, it may lead to suboptimal performance in the presence of graph dynamism.

Vertex-Cut Graph Partitioning Several vertex-cut graph partitioners [44, 31, 13] were also proposed to improve the performance of distributed graph computation. Vertex-cut solutions partition the graph by assigning edges of the graph across partitions instead of vertices. It has been shown that vertex-cut solutions reduce the communications with respect to edge-cut ones, especially on power-law graphs. However, it also has to deal with the issue of communication heterogeneity and the issue of shared-resource contention, since vertices appearing in multiple partitions need to communicate with each other during the computation. Nevertheless, its discussion is beyond the scope of this paper.

Overview of Related Work Table 6 visually classifies the state-of-the-art graph (re)partitioners according to algorithm and graph properties. In terms of *algorithm properties*, we characterize each approach as to whether it (a) runs in parallel and (b) is architecture-aware (i.e., CPU heterogeneity, network cost non-uniformity, and resource contention). In terms of *graph properties*, we characterize each approach as to whether it can handle graphs with (a) dynamism, (b) weighted vertices (i.e., nonuniform computation), (c) weighted edges (i.e., nonuniform data communication), and (d) vertex sizes (i.e., nonuniform data sizes on each vertex).

9. CONCLUSIONS

In this paper, 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 nonuniform network communication costs and consideration of the contentiousness of the underlying hardware. 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.

10. ACKNOWLEDGMENTS

We would like to thank Jack Lange, Albert DeFusco, Kim Wong, Mark Silvis, and the anonymous reviewers for their valuable help on the paper. This work was funded in part by NSF awards CBET-1250171 and OIA-1028162.

11. REFERENCES

[1] http://www.cs.sandia.gov/zoltan/.

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

[3] C. Binnig, U. Çetintemel, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian, and S. B. Zdonik. The End of Slow Networks: It's Time for a Redesign. *CoRR*, 2015.

[4] A. Buluç and K. Madduri. Parallel Breadth-First Search on Distributed Memory Systems. *CoRR*, abs/1104.4518, 2011.

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

[6] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *J Parallel Distr Com*, 2009.

[7] http://www.sandia.gov/~bahendr/chaco.html.

Table 6: State-of-the-art Graph (Re)Partitioners

Name/Reference	Algorithm Properties				Graph Properties			
	Parallel	Architecture-Aware			Dynamism	Weighted		Vertex Size
		CPU	Network	Contention		Vertex	Edge	
Graph Partitioners								
METIS [23]						✓	✓	
ICA3PP'08 [24]		✓	✓			✓	✓	
Chaco [7]						✓	✓	
DG/LDG [39]/Fennel [42]					Yes/No			
arXiv'13 [11]					✓			
TKDE'15 [46]		✓	✓		Yes/No	✓	✓	
SoCC'12 [8]			✓			✓	✓	
Sheep [22]	✓							
Graph Repartitioners								
PARMETIS [30]	✓				✓	✓	✓	✓
Zoltan [1]	✓				✓	✓	✓	✓
Scotch [36]					✓	✓	✓	✓
CatchW [37]	✓				✓	✓	✓	
xdgp [43]	✓				✓	✓		
Hermes [26]	✓				✓	✓		
Mizan [17]	✓				✓	✓	✓	
LogGP [45]	✓				✓	✓	✓	
ARAGON [48]			✓		✓	✓	✓	✓
PARAGON	✓		✓	✓	✓	✓	✓	✓

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

[9] http://www.dis.uniroma1.it/challenge9.

[10] http://www.cc.gatech.edu/dimacs10/.

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

[12] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC*, 1982.

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

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

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

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

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

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

[19] 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.

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

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

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

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

[24] I. Moulitsas and G. Karypis. Architecture aware partitioning algorithms. In *ICA3PP*, 2008.

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

[26] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, 2015.

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

[28] https://portal.xsede.org/sdsc-gordon.

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

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

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

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

[33] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *SC*, 2000.

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

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

[36] http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/.

[37] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, 2013.

[38] http://staffweb.cms.gre.ac.uk/~wc06/partition/.

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

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

[41] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

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

[43] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *CoRR*, 2013.

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

[45] N. Xu, L. Chen, and B. Cui. LogGP: a log-based dynamic graph partitioning method. *VLDB*, 2014.

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

[47] C. Zhang, X. Yuan, and A. Srinivasan. Processor affinity and MPI performance on SMP-CMP clusters. In *IPDPSW*, 2010.

[48] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing. In *BigGraphs*, 2014.