Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Repartitioning

Angen Zheng, Alexandros Labrinidis, Panos K. Chrysanthis

Department of Computer Science, University of Pittsburgh
{anz28, labrinid, panos}@cs.pitt.edu

***Abstract*—Graph partitioning is an essential preprocessing step in distributed graph computation and scientific simulations. Existing well-studied graph partitioners are designed for static graphs, but real-world graphs, such as social networks and Web networks, keep changing dynamically. In fact, the communication and computation patterns of some graph algorithms may vary significantly, even across their different computation phases. This means that the optimal partitioning changes over time, requiring the graph to be repartitioned periodically to maintain good performance. However, the state-of-the-art graph (re)partitioners are known for their poor scalability against massive graphs. Furthermore, they usually assume a homogeneous and contention-free computing environment, which is no longer true in modern high performance computing infrastructures.**

In this paper, we introduce PLANAR, a parallel lightweight graph repartitioner, which does not require full knowledge of the graph and incrementally adapts the partitioning to changes while considering the heterogeneity and contentiousness of the underlying computing infrastructure. Using a diverse collection of datasets, we showed that, in comparison with the de-facto standard and two state-of-the-art streaming graph partitioning heuristics, PLANAR improved the quality of graph partitionings by up to 68%, 46%, and 69%, respectively. Furthermore, our experiments with an MPI implementation of Breadth First Search and Single Source Shortest Path showed that PLANAR achieved up to 10x speedups against the state-of-the-art streaming and multi-level graph (re)partitioners. Finally, we scaled PLANAR up to a graph with 3.6 billion edges.

I. INTRODUCTION

This work targets graph-based, *communication-intensive* big data applications, such as large-scale scientific simulations (e.g., Combustion Simulations [28]) and distributed graph computation using Pregel-like graph computing engines [17]. Graph (re)partitioning has been widely used in scientific simulations for decades [11], [27], while the use of graph (re)partitioning in the latter is receiving more and more attention recently [34], [36], [10], [32], [14], [37], [22]. The computation and communication patterns of such applications are inherently, or can be modeled as, a graph. They often divide their computations into a sequence of *supersteps* 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 superstep. The function can change the state and the outgoing edges of the vertex, send messages to its neighbors, or add or remove vertices/edges to the graph.

The graph can be assigned a weight and a size for each vertex to indicate the computational requirement and the amount of data represented by the vertex. Also, the amount of data communicated between each neighboring vertex pair can be used as the corresponding edge weights. Thus, *a balanced partitioning of the graph is equivalent to distributing the load evenly across compute nodes, whereas minimizing the number of edges crossing partitions minimizes the communication among neighboring vertices in different partitions*.

Existing well-studied graph partitioners like METIS [18] and CHACO [6] are designed for static graphs, but real-world graphs, such as social networks and Web/semantic networks, are inherently dynamic and evolve continuously over time. If the dynamism is left unchecked, the quality of the partitioning will continuously degrade, leading to load imbalance and additional data communication. Furthermore, real-world systems often dynamically increase or shrink their capacity in response to load fluctuations, demanding the graph to be repartitioned into a different number of partitions dynamically. Put simply, *the graph needs to be frequently repartitioned to adapt to graph structural, load distribution, and environmental changes*.

Unfortunately, given the sheer scale of real-world graphs, repartitioning the entire graph. even in parallel, like state-of-the-art repartitioners (e.g., ZOLTAN [1], [5], PARMETIS [26], [30], and SCOTCH [31]), is costly in terms of both time and space. Besides, existing repartitioners are known to have poor scalability. Recently, several *streaming* graph partitioners (e.g., *DG*/*LDG* [34], Fennel [36], and arXiv'13 [10]) have been proposed, which can produce relatively good partitionings in a quite short time for both static and dynamic graphs. Nevertheless, they may result in sub-optimal performance for dynamic graphs. Consequently, several *lightweight* repartitioners (e.g., arXiv'13 [10], CatchW [32], Mizan [14], xdgp [37], and Hermes [22]) that do not require full knowledge of the graph were proposed. However, they all assume uniform vertex weights and sizes, and some ([10], [37], [22]) also assume uniform edge weights. These assumptions are often unrealistic, since weights and sizes of real-world graphs are almost always nonuniform. For example, in social networks, high-degree vertices often have significantly higher computational requirement and migration costs than low-degree ones. Moreover, edge weights are often algorithm-dependent. Hence, *to maintain good performance, we need a lightweight adaptive graph repartitioner for massive dynamic graphs with nonuniform weights and sizes*.

Like existing heavyweight repartitioners, current streaming and lightweight solutions also assume uniform network communication costs among partitions while repartitioning. However, modern parallel architectures, like supercomputers,

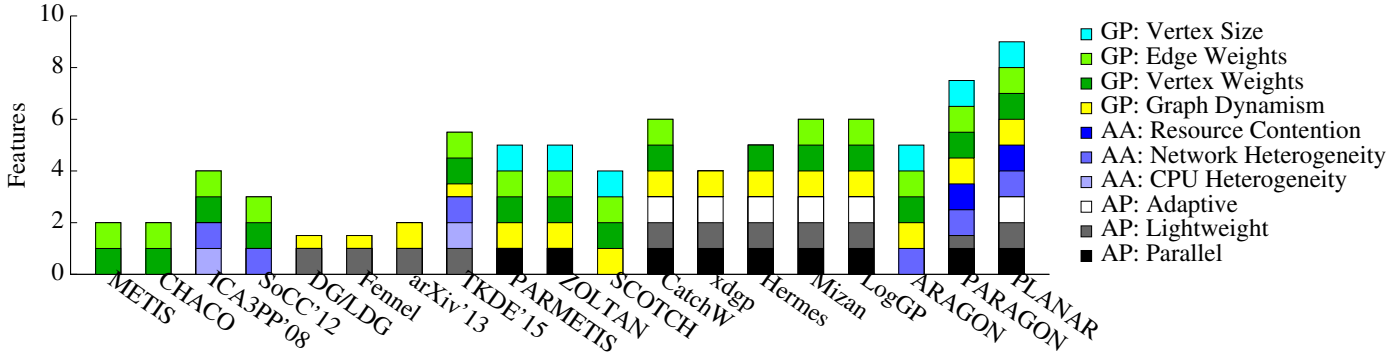


Fig. 1: Overview of the state-of-the-art graph (re)partitioners showing which graph properties (GP) they support, whether they are architecture-aware (AA), and what algorithm properties (AP) they have.

usually comprise hundreds of compute nodes linked with a network, resulting in nonuniform network communication costs among compute nodes (*inter-node communication*) because of their varying locations and link contention. In cloud computing environments, the uneven network bandwidth is another contributing factor to the heterogeneity [7]. In fact, the communication costs among cores of the same compute node (*intra-node communication*) are also nonuniform, where cores sharing more cache levels typically communicate faster. Additionally, inter-node communication is often an order of magnitude slower than intra-node communication.

Hence, existing architecture-agnostic graph (re)partitioners (which assume uniform network communication costs) may lead to sub-optimal performance, which could be quite significant for large-scale distributed computation. This issue has drawn a lot of attention in the past few years (e.g., ICA3PP'08 [20], SoCC'12 [7], ARAGON [40], and TKDE'15 [39]). Nevertheless, the former three share the same fate as that of heavyweight repartitioners, since they are built on top of them, whereas the last one may lead to sub-optimal performance in the presence of graph dynamism as streaming graph partitioners [34], [36].

Additionally, none of the existing work considers the issue of shared resource contention in modern multi-core systems. Shared resource contention has received heated attention in system-level research [12], [35]. Although PARAGON [41], a parallel version of ARAGON, considers both the contentiousness and communication heterogeneity, it requires global knowledge of the entire graph for repartitioning, limiting its scalability. Therefore, *we need a lightweight architecture-aware graph repartitioner to adapt the partitioning to the changing world such that both the contentiousness and the amount of data communicated and migrated among partitions having high network communication costs are minimized. In fact, even though the graphs are static, we still need an architecture-aware graph (re)partitioner to improve the mapping of the application communication pattern to the underlying hardware topology.*

Summary of the state-of-the-art (Figure 1) We summarize the state-of-the-art graph (re)partitioners in Figure 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., nonuniform computation), (c) weighted edges (i.e., nonuniform data communication), and (d) vertex sizes (i.e., nonuniform 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, (b) is lightweight, and (c) is adaptive. It is worth pointing out that the current state-of-the-art is either architecture-aware OR parallel, adaptive, and lightweight, but no one approach (except for PLANAR, our proposed solution) combines all.

Contributions To address the needs of efficiently parallelizing graph-based big data applications, we make the following contributions:

1. We report how the architecture-awareness and graph dynamism impact application performance (Section II).
2. We formally define the desired properties of graph repartitioners needed to address the challenges identified, namely (a) run in parallel, (b) be lightweight, (c) be adaptive, and (d) be architecture-aware (Section III).
3. We present, PLANAR, a parallel lightweight graph repartitioner (Section IV), which efficiently adapts the decomposition to graph dynamism by incrementally migrating vertices among partitions with the awareness of the communication heterogeneity and contentiousness of the underlying computing infrastructures.
4. We perform an extensive evaluation of PLANAR using many real-world datasets (Section VI). The results show the effectiveness and scalability of PLANAR compared to the de facto standard and the state-of-the-art.

II. DYNAMISM & ARCHITECTURE-AWARENESS

Configuration (Table I) To motivate the need to consider architecture-awareness and graph dynamism, we run an experiment using the YouTube dataset (a collection of YouTube users and their friendship connections over a period of 225 days [19]). We split the dataset into 5 snapshots (Table I), As shown in the table, snapshot S_i denotes the set of users and their connections appearing during the first $45 * i$ days.

We then ran Breadth First Search (BFS) on snapshot S_1, on two 20-core compute nodes on our evaluation platform

TABLE I: YouTube Growth Dataset

Snapshots	$\|V\|$	$\|E\|$	Description
S_1	1,138,499	6,135,216	45 days
S_2	1,606,185	9,966,724	90 days
S_3	1,952,292	12,032,134	135 days
S_4	2,455,644	15,969,462	180 days
S_5	3,223,589	24,447,548	225 days

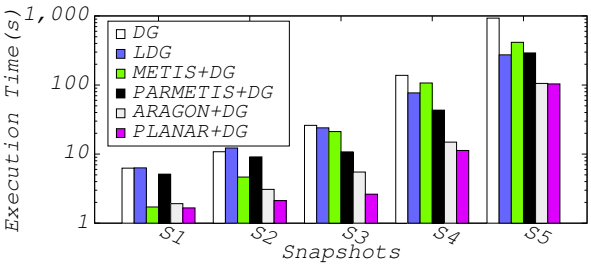


Fig. 2: BFS Job Execution Time

(Section VI-B). $S1$ was (re)partitioned across each core using 5 techniques: (a) Deterministic Greedy (*DG*) and Linear Deterministic Greedy (*LDG*), two state-of-the-art streaming graph partitioning heuristics [34]; (b) METIS, a state-of-the-art, multi-level graph partitioner [18]; (c) PARMETIS, a state-of-the-art, multi-level graph repartitioner [26], (d) our prior work, ARAGON, a centralized architecture-aware graph partition refinement algorithm [40]; and (e) PLANAR (presented here). For PARMETIS, ARAGON, and PLANAR, S_1 was initially partitioned by *DG*, after which the partitioning was further improved by them. Vertices of $\{S_{i+1} - S_i\}$ were injected into the system using *LDG* for *LDG* and *DG* for others, whenever BFS finished its computation for 15 randomly selected source vertices. The injection also triggered the execution of PARMETIS, ARAGON, and PLANAR on the decomposition.

Results (Figure 2) In Figure 2, we plot the execution time of BFS (in log-scale) for 15 randomly selected source vertices on each snapshot as the graph evolves. Clearly, architecture-awareness and the ability to handle graph dynamism are critical to system performance. ARAGON and PLANAR outperformed architecture-agnostic approaches (i.e., *DG*, *LDG*, METIS, and PARMETIS) by up to 91%. Note that ARAGON is a centralized graph partition refinement algorithm, making it infeasible for large scale distributed online graph repartitioning.

Take-away *To maintain superior performance, we should continuously adapt the partitioning to the changes while considering the nonuniform network communication costs and the contentiousness of the underlying computing infrastructures.*

III. PROBLEM STATEMENT

In this paper, we would like to identify a graph repartitioner that has the following four properties: (a) runs in parallel, (b) is lightweight, (c) is adaptive, and (d) is architecture-aware.

Parallel A graph repartitioner is considered *parallel* if the repartitioning is performed in a parallel/distributed fashion. This is essential, because real-world, graph-based big data applications often require the use of parallel computing infrastructures, where the graph is distributed across a set of machines for parallel computation. Thus, to avoid massive data communication, the repartitioning has to be performed in parallel using the same set of machines across which the graph has been distributed. Such parallelism also allows the repartitioner to exploit the power of parallel computing and to complete the repartitioning faster.

Lightweight A repartitioner is said to be *lightweight* if it only relies on a small amount of information about the graph structure for repartitioning. In contrast, if a repartitioner requires full knowledge of the entire graph (has to access all edges) while repartitioning, such as PARMETIS [26], it is *heavyweight* because of the heavy network traffic it generates. Additionally, the repartitioning should perform well in terms of both time and memory requirements.

Adaptive/Incremental A repartitioner is *adaptive* if it improves the partitioning in an incremental way over time, rather than seeking an optimal partitioning at once by costly repartitioning the entire graph.

Architecture-Aware Let $G = (V, E)$ be a graph with V and E as its vertex and edge set, P be a partitioning of G with n partitions:

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

and M be the current assignment of partitions to servers with P_i assigned to server $M[i]$. Each server can either be a hardware thread, a core, a socket, or a machine. *Architecture-aware* graph repartitioning aims to compute a new partitioning P' of G that satisfies the following objectives: (a) balances the load; (b) minimizes the communication among partitions; and (c) minimizes the migration cost between P and P'. A partitioning is said to be *balanced* if

$$w(P_i) < (1 + \varepsilon) * \frac{\sum_{j=1}^{n} w(P_j)}{n} \quad (2)$$

where $w(P_i)$ is the aggregated weight of vertices in P_i, and ε is the user-defined imbalance tolerance. The communication cost of a partitioning is defined as:

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

where α is a parameter specifying the relative importance between communication and migration cost, $w(e)$ is the edge weight, and $c(P_i, P_j)$ is the relative network communication cost between P_i and P_j. Existing graph (re)partitioners usually assume $c(P_i, P_j) = 1$, which usually fails to reflect the reality of modern computing hardware. Thus, to minimize $comm(G, P)$, we should gather vertices communicating a lot of data as close as possible and minimize the number of edges crossing partitions having high network communication costs. The migration cost of a repartitioning is defined as:

$$mig(G, P, P') = \sum_{\substack{v\in V \text{ and} \\ v\in P_i \text{ and } v\in P'_j \text{ and } i\neq j}} vs(v) * c(P_i, P'_j) \quad (4)$$

where $vs(v)$ is the vertex size. 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

Algorithm 1: Planar Overview

```
Data: P_l, c, σ, τ
if the partitioning has not converged then
    // Phase-1 (Section IV-A & IV-B)
    LogicalVtxMigration(P_l, c, &pv)
    // Phase-2 (Section IV-C)
    PhysicalVtxMigration(P_l, pv)
    // Convergence Check (Section IV-D)
    CheckPartitionConvergence(σ, τ);
```

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.

IV. PLANAR

PLANAR, (*Parallel Lightweight Architecture-aware Adaptive graph Repartitioning*), is a lightweight graph repartitioner designed for massive, dynamic graphs. Rather than costly repartitioning the entire graph at once, PLANAR adapts the current partitioning in the presence of changes by incrementally migrating vertices among partitions, while considering the non-uniformity of network communication costs. Algorithm 1 presents PLANAR at a high level. It is triggered whenever there are enough changes in the graph or imbalance among 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. 3) between two consecutive adaptations is within a user-defined threshold σ after τ consecutive adaptation supersteps (Section IV-D).

Each such adaptation step has two phases: *logical vertex migration phase* (Phase-1) and *physical vertex migration phase* (Phase-2). Phase-1 attempts to improve the decomposition by logically migrating vertices among 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 IV-C) 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 IV-A) tries to improve the decomposition in terms of communication cost as much as possible. Phase-1b (Section IV-B) aims to improve the decomposition in terms of load distribution without significantly increasing the communication cost of the decomposition output by Phase-1a.

A. Phase-1a: Minimizing Communication Cost

In this phase, each server runs an instance of Algorithm 2 *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.

Algorithm 2: 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
```

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:

$$comm(v, P_i) = \alpha * \sum_{k=1 \text{ and } k \neq i}^{n} d_{ext}(v, P_k) * c(P_i, P_k) \quad (5)$$

where $d_{ext}(v, P_k)$ represents the amount of data that v communicates with vertices of P_k, which is further defined as:

$$d_{ext}(v, P_k) = \sum_{e=(u,v) \in E \text{ and } u \in P_k} w(e) \quad (6)$$

The cost of migrating v from its current partition P_i to another partition P_j is defined as:

$$mig(v, P_i, P_j) = vs(v) * c(P_i, P_j) \quad (7)$$

Hence, the gain of migrating v from P_i to P_j is:

$$g^{i,j}(v) = comm(v, P_i) - comm(v, P_j) - mig(v, P_i, P_j) \quad (8)$$

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.

Migration Destination Selection Example (Figures 3–6) Consider a decomposition given by Figure 3 with three partitions and unit weights and sizes, and the relative network communication costs among partitions as shown in Figure 6. 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) and to P_2 (Figure 5) 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 over Figure 5 due to its lower edge-cut (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 3, 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

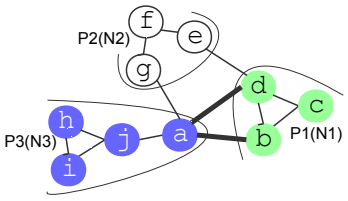


Fig. 3: Old Decomposition

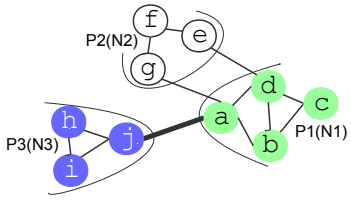


Fig. 4: Better Decomposition

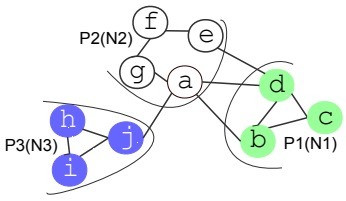


Fig. 5: Best Decomposition

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

Fig. 6: Relative Network Communication Costs

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. Towards this, we first split space $[0, \max_{u \in P_l} g^{l,d}(u)]$ into k equal sized regions, where d denotes the optimal migration destination of u. Then, a boundary vertex is migrated to its optimal destination with a probability of $i * \frac{1.0}{k}$, if its gain is in the ith region, where $i \in [1, k]$ and $k = 100$. 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 2 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, $|V_l|$ denotes the number of (boundary) vertices of each partition. The time complexity of Algorithm 2 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 $O(|E_l|+n^2 * |V_l|)$.

B. 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 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?

1) Question #1: How much to move: To resolve our first question, we first compute the amount of work that needs to be moved out from each overloaded partition:

$$Q(P_i) = w(P_i) - TC(P_i) \tag{9}$$

Algorithm 3: 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 $TC(P_i)$ denotes the maximal load that P_i can have. $TC(P_i) = (1+2\%) * \frac{\sum_{i=1}^{n} w(P_i)}{n}$ by default. Clearly, $-Q(P_i)$ corresponds to the remaining capacity of P_i.

Architecture-Aware Quota Allocation Algorithm 3 describes how PLANAR distributes the remaining capacity of each underloaded partition across 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 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 servers. Here, we take a simple yet effective approach (line 7), where $quota[i][j]$ is distributed across 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.

Algorithm 4: Phase-1b: Vertex Migration

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

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

$$pg(P_i, P_j) = \sum_{v \in P_i} g^{i,j}(v) \tag{10}$$

Each server only needs to consider migrating boundary vertices of 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 3 can be run in parallel on each server without coordination with other nodes. The time complexity of Algorithm 3 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 3 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 boundaries of overloaded partitions to all possible migration destinations), and another $n*n$ matrix (for the quota allocation result).

2) Question #2: What to move: Given the quota allocation, each overloaded server knows how much work it should migrate to each underloaded partition. Along with the sorted heap we maintain while computing the potential gain, we can easily figure out the vertices to migrate and their optimal migration destinations, which is described by Algorithm 4. Clearly, Algorithm 4 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.

C. 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 [16]. 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 paper. 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 [1]).

D. Convergence

To avoid unnecessary execution of PLANAR at the beginning of each superstep, we check if the partitioning converges and discontinue PLANAR if it is. 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 supersteps. 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 VI-A).

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 situation 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[1]), 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 graph during each of PLANAR's convergence supersteps is not drastic. This is a reasonable assumption, since repartitioning is performed in a periodic manner in real-world scenarios.

V. CONTENTION AWARENESS

We found that gathering neighboring vertices as close as possible does not always lead to better performance [41]. This

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

is because many parallel programming models like MPI [23], [21], the de facto messaging standard for HPC applications, often implement intra-node communication (the communication among cores of the same compute node) via shared memory/cache [13], [4]. Thus, putting too much communication within each compute node may result in serious contention for the shared resources (i.e., last level cache, memory controller, front-side buses, or the inter-socket links) in modern multicore systems, having an adversarial impact on performance.

Fortunately, we can avoid the issue if we properly reflect this trade-off in our cost model, by penalizing intra-node network communication costs through the introduction of a penalty score [41]. 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. Hence, we simply refine the intra-node communication costs as follows:

$$c(P_i, P_j) = c(P_i, P_j) + \lambda * (s_1 + s_2) \quad (11)$$

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 $\lambda = 0$, PLANAR will only consider the communication heterogeneity, and $\lambda = 1$ 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.

VI. EVALUATION

In this section, we first evaluate the sensitivity of PLANAR to (a) its two important parameters (Section VI-A) and (b) varying input decompositions computed by different initial partitioners (Section VI-B). We then validate the effectiveness of PLANAR using two graph workloads: Breadth-First Search (BFS) [3] and Single-Source Shortest Path (SSSP) [16] (Section VI-C). Finally, we demonstrate the scalability of PLANAR using a billion-edge graph (Section VI-D). Towards this, we implemented the two workloads and a prototype of PLANAR using MPI [23], [21].

Datasets Table II 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. 3). Unless explicitly specified, the graphs were initially partitioned by the deterministic greedy heuristic, *DG* [34], 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 that DG/LDG were extended to support vertex- and edge-weighted graphs for fair comparison.

TABLE II: Datasets used in our experiments

Dataset	$\|V\|$	$\|E\|$	Description
wave [33]	156,317	2,118,662	2D/3D FEM
auto [33]	448,695	6,629,222	3D FEM
333SP [9]	3,712,815	22,217,266	2D FE Triangular Meshes
CA-CondMat [2]	108,300	373,756	Collaboration Network
DBLP [15]	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 [8]	23,947,347	58,333,344	Road Network
roadNet-PA [2]	1,090,919	6,167,592	Road Network
YouTube [15]	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 III: Cluster Compute Node Configuration

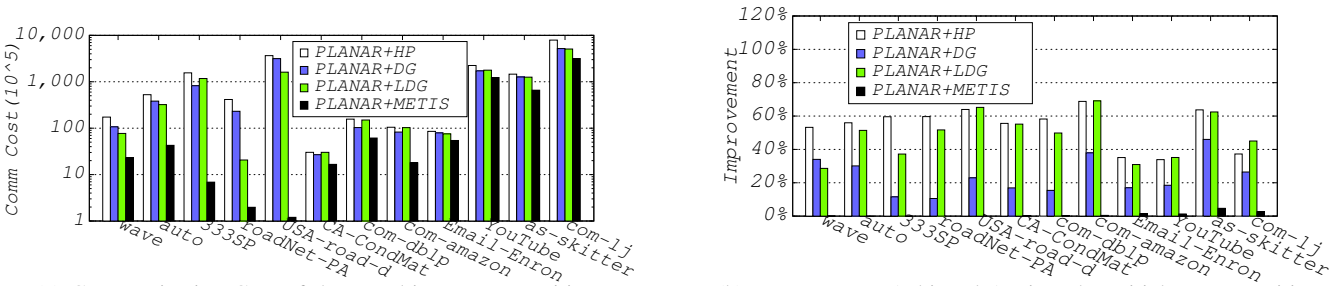
Node Configuration	PittMPICluster (Intel Haswell Processor)	Gordon (Intel Sandy Bridge Processor)
Sockets	2	2
Cores	20	16
Clock Speed	2.6 GHz	2.6 GHz
L3 Cache	25 MB	20 MB
Memory Capacity	128 GB	64 GB
Memory Bandwidth	65 GB/s	85 GB/s

Platforms We evaluated PLANAR on two clusters: PittMPICluster [29] and Gordon supercomputer [24]. PittMPICluster 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 III 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 partitions (cores) were approximated using a variant of osu_latency benchmark [25]. To ensure the accuracy of the cost matrix, we bound each MPI rank (process) to a core using options provided by OpenMPI 1.8.6 [23] on PittMPICluster and MVAPICH2 1.9 [21] on Gordon. OpenMPI and MVAPICH2 were two different MPI implementations available on the clusters.

A. Parametter Selection

Configuration This experiment studied the sensitivity of PLANAR to its two critical parameters: σ and τ (Section IV-D). 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).



(a) Communication Cost of the Resulting Decompositions

(b) Improvement Achieved Against the Initial Decompositions

Fig. 12: 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.

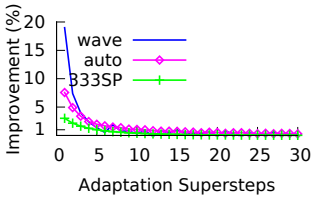


Fig. 7

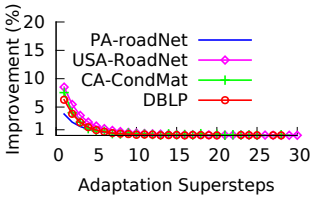


Fig. 8

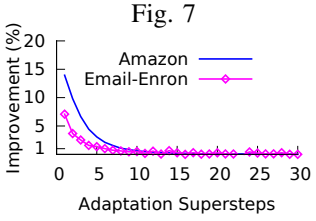


Fig. 9

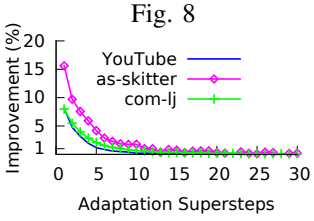


Fig. 10

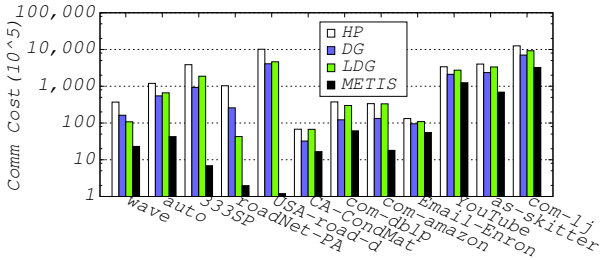


Fig. 11: Communication costs of the initial decompositions partitioned by *HP*, *DG*, *LDG*, and METIS into 40 partitions.

Results Figures 7 to 10 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 as-skitter 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.

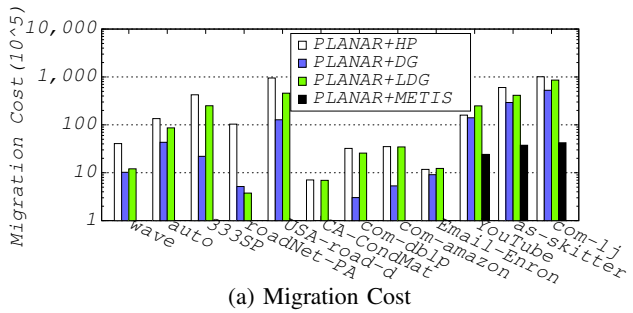
B. Microbenchmarks

Configuration This experiment examined the effectiveness of PLANAR in terms of partitioning quality (Eq. 3 and 4), 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 [34]; and METIS is a state-of-the-art multi-level graph partitioner [18]. The graphs were initially partitioned across two 20-core compute nodes on PittMPICluster.

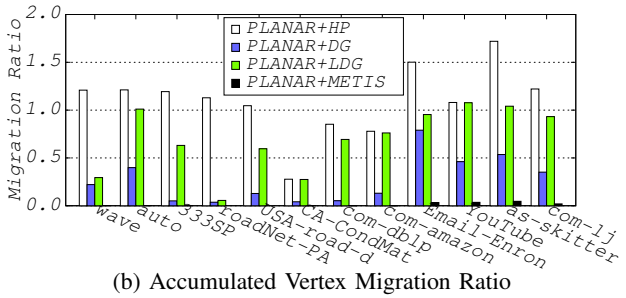
Quality of the Initial Decompositions (Figure 11) Figure 11 presents the initial communication costs of the decompositions computed by *HP*, *DG*, *LDG*, and METIS for a variety of graphs in log-scale. As expected, METIS performed the best and *HP* was the worst. However, METIS is a heavyweight serial graph partitioner, making it infeasible for large-scale distributed graph computation either as an initial partitioner or as an online repartitioner (repartitioning from scratch). It was reported in [36] 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 [34]. This was probably because the 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.

Quality of the Resulting Decompositions (Figures 12a & 12b) Figures 12a and 12b, 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 decompositions 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 VI-C).

Migration Cost (Figures 13a & 13b) In the experiment, we also examined the migration cost introduced by PLANAR in terms of Eq. 4 and the accumulated vertex migration ratio (# of vertices migrated as a percentage of the entire graph) across all adaptation supersteps. Figures 13a and 13b 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



(a) Migration Cost



(b) Accumulated Vertex Migration Ratio

Fig. 13: Overhead of the adaptation on varying initial decompositions computed by *HP*, *DG*, *LDG*, and METIS into 40 partitions.

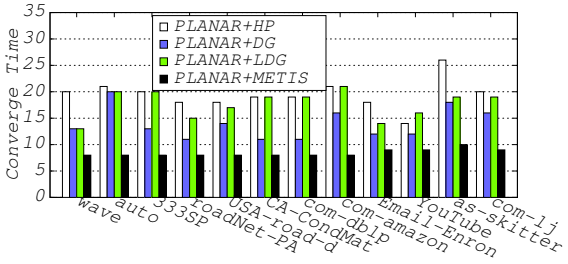


Fig. 14: PLANAR converge time in terms of supersteps

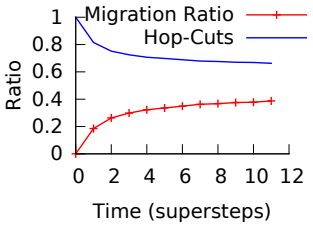


Fig. 15: wave

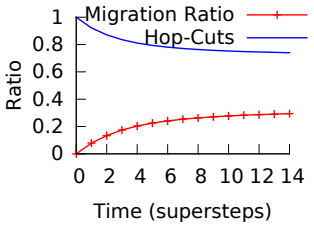


Fig. 16: com-lj

because each vertex may be migrated multiple times during the adaptation. 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 14) Another item of interest in this experiment is the average number of supersteps PLANAR took to converge (Figure 14). As presented, for graph decompositions that have limited improvement space, PLANAR only took around 8 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 in the convergence check phase to further reduce the overhead of the adaptation.

Convergence Process (Figures 15 & 16) 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 hop-cuts across supersteps. Figures 15 and 16 show the accumulated vertex migration ratio and the normalized hop-cuts (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 hop-cuts in the first 5 adaptation supersteps, which were also the places where most vertices got migrated.

C. Real-World Applications (BFS & SSSP)

Configuration This experiment evaluated PLANAR using BFS and SSSP on 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 (8 for the YouTube and as-skitter dataset and 16 for the com-lj dataset) messages sent by each MPI rank to the same destination into a single one. 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 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 (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 PittMPICluster. Hence, we fixed λ to be 1 on PittMPICluster and 0 on Gordon for our experiments.

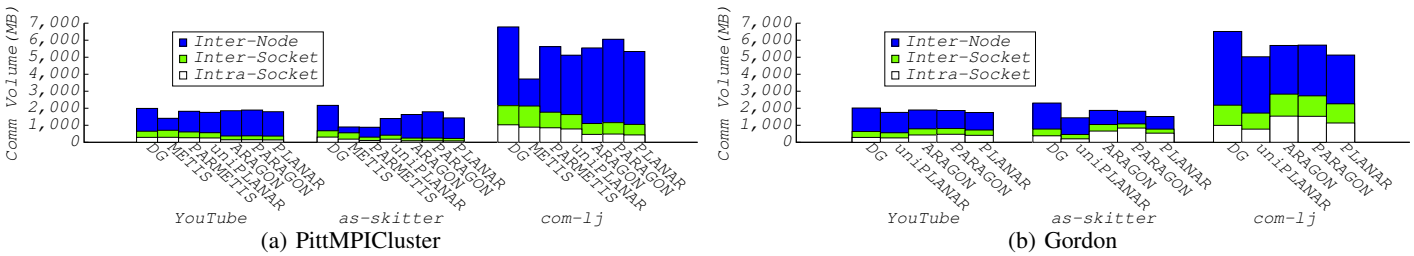
Job Execution Time (Tables IV & V) Tables IV and V show the execution time of BFS and SSSP with 15 randomly selected source vertices on the three datasets. The job execution time is defined as: $JET = \sum_{i=1}^{n} SET(i)$, where n corresponds to the number of supersteps the job has, while $SET(i)$ 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 a long time to complete (even hours for large graphs).

TABLE IV: BFS Job Execution Time (s)

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

Algorithm/Dataset	YouTube	as-skitter	com-lj
PittMPICluster			
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) PittMPICluster (b) Gordon

Fig. 17: The communication volume breakdown of SSSP on both clusters.

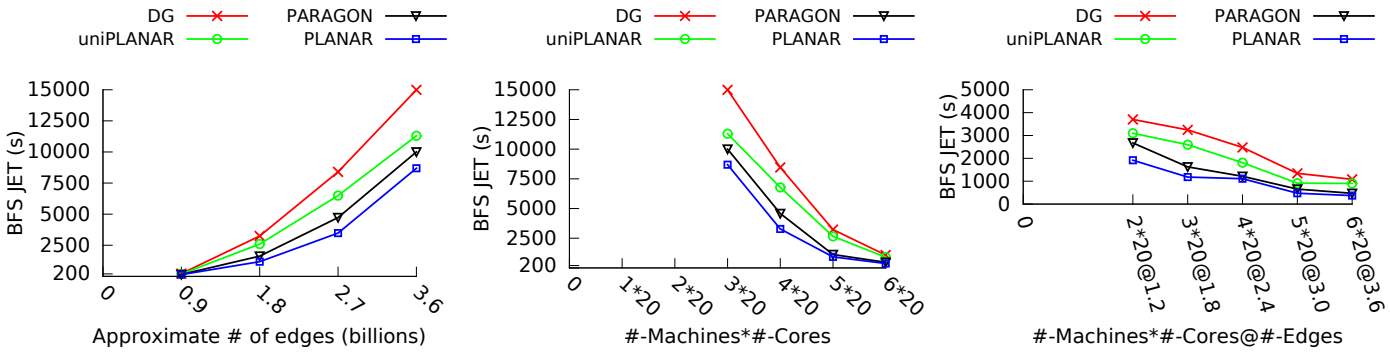
As expected, PLANAR beat *DG*, PARMETIS, and UNIPLANAR in almost all cases. Compared to *DG*, PLANAR reduced the execution time of BFS and SSSP on Gordon by up to 69% and 57%, respectively, and by up to 90% and 88% on PittMPICluster, respectively. So, in the best case, PLANAR is 10 times better than *DG*. Yet, the overhead PLANAR exerted (the sum 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 (edge-cut). 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.

Communication Volume Breakdown (Figures 17a & 17b) To further confirm our observations, we also measured 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 supersteps for SSSP on both clusters here.

As shown in Figures 17a and 17b, 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 PittMPICluster and lowest inter-node communication volume on Gordon. It should be noticed that on PittMPICluster 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 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.

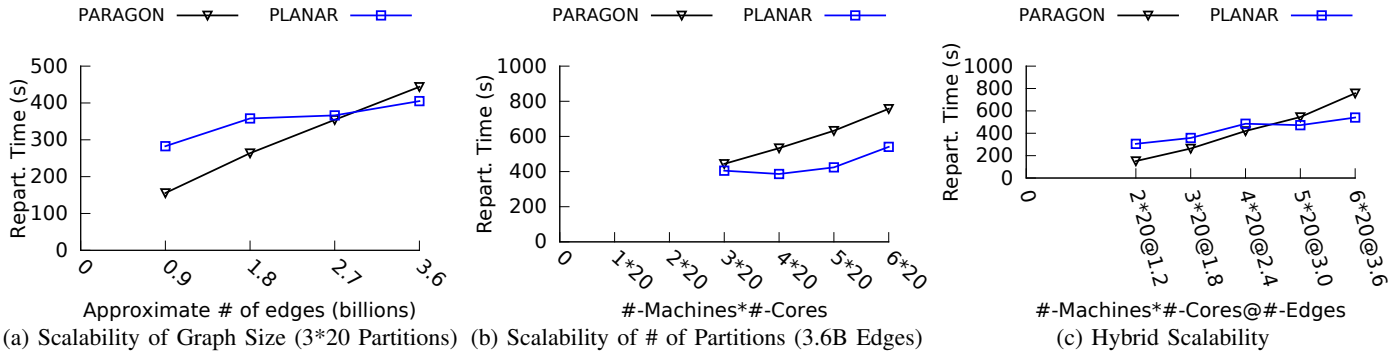
D. Billion-Edge Graph Scaling

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 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 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 PittMPICluster with BFS message grouping size set to 256. We would only present the results of *DG*, PARAGON, UNIPLANAR, and PLANAR, since



(a) Scalability of Graph Size (3*20 Partitions) (b) Scalability of # of Partitions (3.6B Edges) (c) Hybrid Scalability

Fig. 18: BFS Job Execution Time (JET)



(a) Scalability of Graph Size (3*20 Partitions) (b) Scalability of # of Partitions (3.6B Edges) (c) Hybrid Scalability

Fig. 19: Repartitioning Time

METIS, PARMETIS, and ARAGON failed to (re)partition the graphs even for the smallest graph of this experiment, due to their heavyweight nature.

Results (Figures 18 & 19) Figures 18 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 cases. We also noticed that in Setup 1 (Figure 18a), 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 PittMPICluster. 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 machines (6*20 cores) not just one.

Figure 19 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 setups. The reason why the 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 adaptation supersteps, the overhead introduced by PLANAR in each adaptation superstep would be very small.

VII. Related Work

Graph (re)partitioners are widely used to support scientific simulation and large-scale distributed graph computation in parallel computing infrastructures. We organize the existing graph (re)partitioners into three categories: (a) heavyweight, (b) lightweight, and (c) streaming, which are presented next.

Heavyweight Graph (Re)Partitioning Graph (re)partitioning have been extensively studied (i.e., METIS [18], PARMETIS [26], SCOTCH [31], CHACO [6], and ZOLTAN [1]), but only PARMETIS and ZOLTAN support *parallel* graph (re)partitioning. However, neither of them are architecture-aware. Although [20], a METIS variant, considers the communication heterogeneity, it is a *sequential static graph partitioner*, which is inapplicable for large dynamic graphs. Several recent works [40], [7] have been proposed to cope with heterogeneity and dynamism. However, they are too heavyweight for massive graphs because of the high communication volume they generate while (re)partitioning.

Lightweight Graph Repartitioning Many lightweight graph repartitioners [32], [37], [22], [14], [38] have been proposed for efficiently adapting the partitioning to changes by incrementally migrating vertices among partitions based on some heuristics. Nevertheless, they are architecture-agnostic. Also, many of them assume uniform vertex weights and sizes, and some [37], [22] even assume uniform edge weights. Moreover, they migrate vertices under the constraint that the load is evenly distributed in a single phase. In contrast, PLANAR splits

vertex migration into 2 phases, in which we try to minimize the communication cost without considering the balancing requirement first and then focus on balancing the load.

Streaming Graph Partitioning A new family of graph partitioning heuristics, streaming graph partitioning [34], [36], [10], has been proposed recently for online graph partitioning. They can produce partitionings comparable to METIS within a relative short time. However, they are not architecture-aware. Although [39] has presented a streaming graph partitioner that is aware of both CPU and communication heterogeneity, it has the same issue with dynamic graphs as *DG*/*LDG*. Furthermore, unlike PLANAR, which strikes to minimize the communication cost under the constraint that the load is balanced, [39] aims to balance load (the computation time plus communication time) as a whole. Article [10] also proposed an architecture-agnostic adaptive graph repartitioner for dynamic graphs. Instead of optimizing both the communication cost and the skewness during each adaptation superstep as PLANAR does, it chooses to optimize one heuristic at a time with a probability of 0.5.

VIII. CONCLUSION

In this paper, we presented a lightweight architecture-aware graph repartitioner, PLANAR, for large dynamic graphs. PLANAR can not only efficiently respond to graph dynamism by incrementally migrating vertices among 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 adaptive, lightweight, and architecture-aware (which are all present in PLANAR) are absolutely essential for online repartitioners. Our evaluation confirmed PLANAR's superiority in terms of performance improvement (up to 10x speedup) and scalability (up to 3.6 billion edges).

ACKNOWLEDGMENTS

We would like to thank Peyman Givi, Patrick Pisciuneri, Mark Silvis, and the anonymous reviewers for their help. This work was funded in part by NSF awards CBET-1250171 and OIA-1028162.

REFERENCES

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

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

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

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

[5] 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.

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

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

[8] 9th DIMACS Challenge. http://www.dis.uniroma1.it/challenge9.

[9] 10th DIMACS Challenge. http://www.cc.gatech.edu/dimacs10/.

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] A. Mislove, "Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems," Ph.D. dissertation, Rice University, 2009.

[20] I. Moulitsas and G. Karypis, "Architecture aware partitioning algorithms," in *ICA3PP*, 2008.

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

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

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

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

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

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

[27] P. Pisciuneri, A. Zheng, P. Givi, A. Labrinidis, and P. Chrysanthis, "Repartitioning Strategies for Massively Parallel Simulation of Reacting Flow (Abstract)," *Bull. Am. Phys. Soc.*, 2015.

[28] P. Pisciuneri, S. L. Yilmaz, P. Strakey, and P. Givi, "An Irregularly Portioned FDF Simulator," *SIAM J. Sci. Comput.*, 2013.

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

[30] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *Supercomputing*, 2000.

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

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

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

[34] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *SIGKDD*, 2012.

[35] 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.

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

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

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

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

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

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