Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing

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

***Abstract*—Graph partitioning and repartitioning have been widely used by scientists to parallelize compute- and data-intensive simulations. However, existing graph (re)partitioning algorithms usually assume homogeneous communication costs among partitions, which contradicts the increasing heterogeneity in inter-core communication in modern parallel architectures and is further exacerbated by increasing dataset sizes (i.e., Big Data). To resolve this, we propose an architecture-aware graph repartitioner, called ARAGONLB. ARAGONLB considers the heterogeneity in both inter- and intra-node communication while rebalancing the load. Our experimental study with a turbulent combustion simulation dataset shows that ARAGONLB can result in up to 60% improvement against existing architecture-agnostic graph repartitioners (which assume uniform communication costs among partitions), and the improvement becomes more significant as the number of computation steps, the number of partitions, or the size of the interconnect increase.**

***Keywords*-Architecture-Aware, Topology-Aware, Graph Repartitioning, Dynamic Load Balancing, Scientific Computing**

I. INTRODUCTION

There is no doubt that we are awash with data and this affects all aspects of life, from the private lives of (socially networked) individuals, to how businesses operate, to how governments make decisions, to modern scientific analysis and exploration [1], which in turn creates a lot of technical challenges [2]. In the scientific domain, Big Data is often generated by giant instruments, such as CERN's Large Hadron Collider[1] or the Large Synoptic Survey Telescope[2], or through simulation on parallel computing infrastructures, such as cosmological simulations or simulations of turbulent combustion [3].

In this paper, we focus on compute- and data-intensive applications (such as scientific simulations) that execute in parallel computing infrastructures. The communication and computation patterns of such applications are usually represented as a *vertex- and edge-weighted graph*, which needs to be periodically *repartitioned* to make sure the data communication and computing resources are properly utilized. In such a graph, each vertex corresponds to a unit of computation work: the *vertex weight* represents the amount of computation associated with the vertex, whereas the *vertex size* corresponds to the amount of data the vertex represents. Furthermore, an edge between two vertices means that data needs to be communicated between them; the amount of data communicated is reflected by the *edge weight*.

Due to the sheer scale of these graphs, an initial partitioning of the graph and an assignment of the partitions to different computing cores are usually performed at the beginning of the simulation to parallelize the computation. This aims to equalize the load across cores, while minimizing the data communication cost among partitions. However, typical scientific simulations run for multiple steps. During each step, a domain-specific 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 the outgoing edges of the vertex, send messages to its neighbors to be processed in the next step, or even modify the structure of the graph.

Clearly, the load distribution and the data communication patterns of these applications do not remain static throughout the simulation. Hence, if this dynamism is left unchecked, the quality of the initial partitioning will continuously degrade, leading to (a potentially significant) load imbalance and additional communication overhead. In fact, the bigger the dataset the bigger the magnitude of the problem because of the increasing communication volume. Thus, *the graph needs to be repartitioned periodically to rebalance the load while minimizing the data communication and migration cost.*

In the literature, graph partitioning [4], [5], [6], [7] and repartitioning [8], [9], [10], [11] have been well-studied. However, the state-of-the-art graph repartitioners, such as Zoltan [8] and Parmetis [10], usually assume uniform communication and uniform migration costs among partitions while repartitioning. That is, they simply assume that the communication and migration cost is linearly proportional only to the amount of data communicated and migrated.

However, *this uniform-cost assumption is no longer valid*, as modern parallel computing architectures often consist of multiple compute nodes interconnected by a network, resulting in nonuniform inter-node communication costs due to their varying locations and link contention. In fact, even within a compute node, the inter-core communication cost is also variable, due to the complex, multi-level memory hierarchy. As the core count per die area and the number of cache hierarchy levels continuously increase, such heterogeneity is further amplified and will only become worse in the future, if not addressed. Thus, existing architecture-agnostic graph

[1]http://home.web.cern.ch/topics/large-hadron-collider
[2]http://www.lsst.org/lsst/

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

TABLE I: Sample inter-node communication cost on a $4*4*4$ 3D-torus interconnect starting from one node

Number of Hops	1	2	3	4	5	6
Number of Nodes	1	6	15	20	15	6

repartitioners may lead to sub-optimal performance, which could be quite significant for Big Data, i.e., as the datasets, and hence the amount of data communication, become bigger. Several recent works have been proposed to cope with this heterogeneity, but they either may cause high migration cost [12], [13] or consider the heterogeneity only partially [14].

Contributions: To address the needs of compute- and data-intensive scientific applications in the era of Big Data, we propose an architecture-aware graph repartitioner, ARAGONLB, which considers both inter- and intra-node communication costs while repartitioning (Section II-III). It is a two-level repartitioner, with distinct schemes and cost models for inter- and intra-node repartitioning (Section IV-V). Our experimental study shows that ARAGONLB can outperform the state-of-the-art (but architecture-agnostic) repartitioners by up to 60%, and the improvement becomes bigger as the number of computation steps, the number of partitions, or the size of interconnect increases (Sections VI-VII).

II. ARAGONLB

Overview: ARAGONLB, short for *ARchitecture-Aware Graph repartitiONer for Load Balancing*, is a two-level hierarchical repartitioner, that performs *inter-* and *intra-node* repartitioning. Its design is driven by the following observations:

1. Communication costs between different nodes (i.e., *inter-node communication costs*) vary greatly due to the interconnection network. For example, on a $4*4*4$ 3D-torus interconnect the distance of different compute nodes to the same node ranges from 1 up to 6 hops as illustrated by Table I. With bigger data sets, contention on the network links will further increase the variability.
2. Communication costs between cores in the same node (i.e., *intra-node communication costs*) also vary a lot because of the multilevel cache hierarchy. For example, in the architecture described by Figure 1, cores sharing a L3 cache communicate faster than cores sharing no caches.
3. The internal node architecture (i.e., memory hierarchy) has a greater impact on the intra-node communication cost than on the intra-node migration cost. This is because not all data needed in the migration phase is in the caches (especially since the migration is performed after executing the repartitioning algorithm) and so the different levels of cache sharing are not going to be that important. Thus, intra-node repartitioning should first focus on minimizing the impact of intra-node topology on communication cost and then the migration cost.
4. The network interconnect affects both the inter-node communication cost and the inter-node migration cost, demanding a graph repartitioner, which considers the communication and migration cost at the same time while repartitioning.

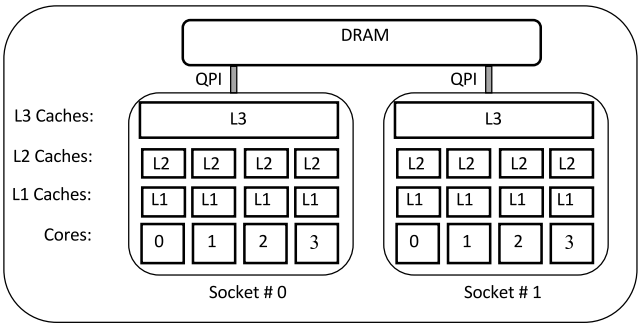

Fig. 1: Example Compute Node Architecture

Given the above observations, the goal of *inter-node repartitioning* is to rebalance the load across compute nodes while minimizing the inter-node communication and migration costs. The latter is achieved by minimizing the number of hops each data item needs to traverse, by grouping together vertices that communicate a lot. In contrast, the goal of *intra-node repartitioning* is to equalize the load assigned to each compute node across its different cores while minimizing the intra-node communication and migration cost, by co-locating vertices communicating a lot to cores sharing more cache levels.

In this paper, we assume that compute nodes used for parallel computation have the same core count and memory hierarchies for ease of implementation, and that the number of partitions assigned to each node equals the core count.

III. PROBLEM STATEMENT

Architecture-Aware Graph Repartitioning (AAGR): Let

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

be an unbalanced partitioning of graph $G = (V, E)$ with n parts. *AAGR* aims to compute a new partitioning P' that satisfies the following objectives: (a) balances the load; (b) minimizes the communication cost among partitions; and (c) minimizes the migration cost between P and P'. A partitioning is said to be balanced if

$$w(P_i) < (1 + \varepsilon) * \bar{w} \quad (2)$$

where $w(P_i)$ is the aggregated weight of vertices in P_i, ε is the user-defined imbalance tolerance ($\varepsilon = 0.05$ indicates that we allow up to 5% load imbalance among partitions), while $\bar{w}$ is the average partition weight. The communication cost of P' 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 the number of computation steps carried out between two consecutive rebalancing steps, $w(e)$ is the edge weight, and $c(P'_i, P'_j)$ is the unit communication cost between the two cores holding P'_i and P'_j. Existing (re)partitioners

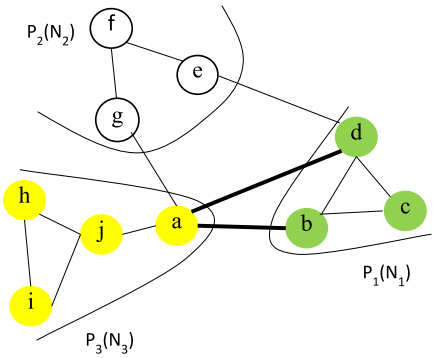

Fig. 2: Old Decomposition

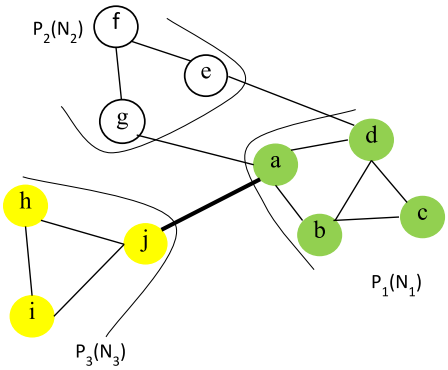

Fig. 3: Better Decomposition

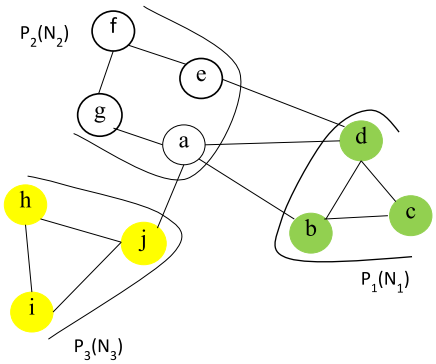

Fig. 4: Best Decomposition

usually assume $c(P'_i, P'_j) = 1$. The migration cost between P and P' is defined as:

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

where $vs(v)$ is the vertex size and $c(P_i, P'_j)$ is the unit migration cost between the two cores holding P_i and P'_j.

IV. Inter-Node Graph Repartitioning

Inter-node repartitioning consists of three phases: (1) A *regrouping phase* in which AraGonLB regroups partitions currently assigned to the same compute node into a single partition; (2) A *repartitioning phase* where AraGonLB repartitions this regrouped graph into balanced parts using existing topology-agnostic graph repartitioners, such as Zoltan [8] and Parmetis [10]; and (3) A *refinement phase* where the decomposition produced by the previous phase is modified according to the current mapping of partitions to compute nodes and the relative inter-node communication costs via a topology-aware refinement algorithm, to further reduce the communication and migration cost. We named this refinement algorithm TopoFM and present it next.

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

TopoFM Overview TopoFM is an iterative algorithm and is a variant of the FM algorithm [15]. Their input is two partitions of the k-way decomposition, the current mapping of partitions to compute nodes, and the relative communication costs among compute nodes. During each iteration, TopoFM tries to find a single vertex, v, such that moving it from its current partition to the alternative partition would lead to a maximal gain, $g(v)$. The gain is defined as the reduction in the communication and migration cost. This process is repeated until all vertices are moved once or the decomposition cannot be further improved after a certain number of vertex movements. Since TopoFM can only refine one partition pair at a time, it is repeatedly applied to all partition pairs sequentially. Although we have parallelized the refinement phase, we do not include it here due to space limitations.

TABLE II: Inter-node Communication Cost Matrix

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

Motivating Example Before we dive into the details of TopoFM, we first go through a simple motivating example. Let us assume that the graph in Figure 2 captures the computation and communication pattern of an application. For simplicity, we assume that all weights and sizes of the graph are 1. Originally, the graph is partitioned into 3 partitions, and partition P_i is assigned to compute node N_i for the parallel execution of the application. Vertices of the same color belong to the same partition, whereas the relative communication costs among N_1, N_2, and N_3 are shown in Table II.

A topology-agnostic repartitioner (i.e., assuming uniform network communication costs) could repartition the decomposition of Figure 2 into the one of Figure 3, reducing the number of edges among partitions from 4 to 3. However, if we consider the case where all network costs are not equal, i.e., we want to make our repartitioner *architecture- and topology-aware* (e.g., using the communication costs from Table II), then the decomposition in Figure 3 can be further improved by moving vertex a to P_2 (Figure 4). Even though the movement increases the communication cost between P_1 and P_2 by 1, it actually reduces the communication cost between a and its neighbors in P_3 by 5, since the network cost between N_1 and N_3 is 6, while that of N_2 and N_3 is 1. For the same reason, moving a to P_2 also decreases the migration cost of a by 5, since vertex a originally belonged to N_1.

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

Algorithm 1: TopoFM

Data: Two balanced partitions (P_i, P_j), partition assignment A, inter-node communication cost matrix c

```
orderedList ← {}
unmarkVertices(P_i, P_j)
computeInitialGain(P_i, H_1, A, c)
computeInitialGain(P_j, H_2, A, c)
while exists unmarked vertex and useless move number ≤ LIMIT do
    heap = FMHeapSelection(P_i, P_j, H_1, H_2)
    v = heapGetMaxGainVertex(heap)
    mark(v)
    append(v, orderedList)
    updateNborGain(P_i, P_j, H_1, H_2, v, A, c)
applyMove(P_i, P_j, orderedList)
```

respectively, and P_i be the partition that v currently belongs to. We define the gain of moving v in terms of its impact on the communication between P_i and P_j as:

$$g_{std}(v) = \alpha * (d^j_{ext}(v) - d^i_{int}(v)) * d(N_i, N_j) \tag{5}$$

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

$$d^j_{ext}(v) = \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_j \text{ and } i\neq j} w(e) \tag{6}$$

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

$$d^i_{int}(v) = \sum_{e=(v,u)\in E \text{ and } v\in P_i \text{ and } u\in P_i} w(e) \tag{7}$$

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

$$g_{topo}(v) = \alpha * \sum_{\substack{e=(v,u)\in E \\ \text{and } v\in P_i \text{ and } u\in P_k \\ \text{and } k\neq i \text{ and } k\neq j}} w(e)*(d(N_i, N_k) - d(N_j, N_k)) \tag{8}$$

Here, N_k is the compute node where P_k belongs.

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

$$g_{mig}(v) = vs(v) * (d(N_i, N_k) - d(N_j, N_k)) \tag{9}$$

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

$$g(v) = g_{std}(v) + g_{topo}(v) + g_{mig}(v) \tag{10}$$

TopoFM Implementation Algorithm 1 presents the basic idea of TopoFM. The input to TopoFM includes two balanced partitions $(P_i,\ P_j)$ of the k-way decomposition, the current assignment of partitions to compute nodes, A, and the relative inter-node communication costs, c. First, TopoFM unmarks all vertices of P_i and P_j, indicating that no vertex has been moved. Second, it computes the initial gain of these vertices and inserts vertices having edges connecting to the other partitions, referred as boundary vertices, into corresponding heaps (one heap per partition), which are sorted by the gain.

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

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

Moreover, although TopoFM is topology-aware, it is also topology-independent since it only requires that the relative inter-node communication costs are available, and its implementation is similar to that of standard FM algorithms, like [16]. The key differences are as follows.

1. In standard FM algorithms, $g(v) = \alpha*(d^j_{ext}(v) - d^i_{int}(v))$, which is unaware of the communication heterogeneity.
2. The refinement between P_i and P_j of standard FM only needs to consider moving boundary vertices of P_j/P_i with respect to P_i/P_j. However, TopoFM needs to consider boundary vertices of P_i/P_j with respect to all partitions.
3. In standard FM algorithms, refinement only happens between partition pairs that communicate. In contrast, TopoFM needs to refine all partition pairs even though no communication occurs between the partition pair.
4. Unlike TopoFM, standard FM algorithms usually select max gain vertices alternatively from P_i and P_j. Thus, our maximal gain vertex selection policy has a greater potential to improve the decomposition while satisfying the balance requirement, and speed up the convergence of a good decomposition.

Complexity Analysis Since the differences between TopoFM and standard FM algorithms as outlined above do not cause any increase in complexity, TopoFM has the same complexity as standard FM algorithms, $O(|E'|)$, where $|E'|$ is the number of edges belonging to the partition pair or having one vertex that ends in either partition. Thanks to the balanced reparti-

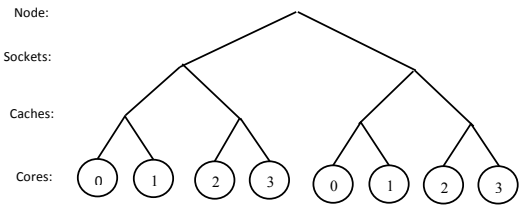

Fig. 5: Topology Tree

tioning phase, $|E'|$ can be roughly approximated by $\frac{|E|}{n} + c$, where n, $|E|$, and c are the number of partitions, the number of edges of the graph and a constant. Thus, the time complexity of TopoFM and the entire refinement phase is $O(2 * \frac{|E|}{n})$ and $O(\binom{n}{2} * 2 * \frac{|E|}{n}) = O(n * |E|)$, respectively.

V. Intra-Node Graph Repartitioning

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

HierCacheLB HierCacheLB first models the topology of each compute node as a tree, like [12]. For example, the tree in Figure 5 denotes a compute node with two quad-core sockets, where two cores in the same socket share a cache, like the one in Figure 1. Then, HierCacheLB partitions the subgraph assigned to the compute node hierarchically according to the tree. This automatically minimizes the communication volume across tree nodes at each level. At the end of each level's partitioning, HierCacheLB remaps the new decomposition to the old one to maximize the amount of data in place.

For instance, assume that we want to repartition a subgraph assigned to a compute node modeled by Figure 5. HierCacheLB will first partition the subgraph into two balanced partitions while minimizing the edge cuts, which approximately equalizes the load across the sockets while minimizing the inter-socket communication cost. Then, HierCacheLB remaps these two partitions to the old decomposition to minimize the migration cost. This step is recursively applied to the next level until it reaches the leaf level.

The remapping phase can be done in an efficient way using the Hungarian algorithm [17], because the topology tree is small. The algorithm takes as input a cost matrix, M, where $M[i][j]$ denotes the migration cost of assigning partition i of the subgraph to the jth socket/cache/core of the compute node. Along with this input, the Hungarian algorithm will output an assignment of partitions to sockets/caches/cores of the node with minimal migration cost.

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

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

$$\sum_{i=1}^{n} M[i][A[i]] * 2 * D_{L_n} + \alpha * \sum_{i=1}^{n} \sum_{j=i+1}^{n} C[i][j] * c(A[i], A[j]) \tag{11}$$

where n is the number of subgraph partitions, while $c(A[i], A[j])$ is the communication cost of $64B$ data within a compute node ($64B$ is the typical cache line size), which is approximated by the access latency to the first cache level shared by core $A[i]$ and $A[j]$. The inter-socket communication cost and the intra-node migration cost of $64B$ data within a compute node are both approximated by 2 times of the access latency to the highest cache level. Although FlatCacheLB needs to explore all possible combinations to figure out the optimal assignment, this is feasible since in practice each compute node only has dozens of cores at most.

TABLE III: Original Combustion Simulation Dataset

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

TABLE IV: Synthetic Datasets

Graph	Num. of Partitions	Degree of Imbalance[3]	Required Compute Nodes
G8	8	2.51	1
G64	64	2.81	8
G128	128	2.82	16
G256	256	2.85	32
G512	512	2.98	64

VI. Evaluation Setup

Workload For our experimental study, we used data provided by the authors of [3]. The dataset (Table III) is a 26-degree mesh, which models the computation and communication pattern of the large eddy simulation (LES) of Sandia Flame D [18]. Out of this dataset, we constructed 5 synthetic graphs

[3]The ratio of the maximal partition load to the average partition load.

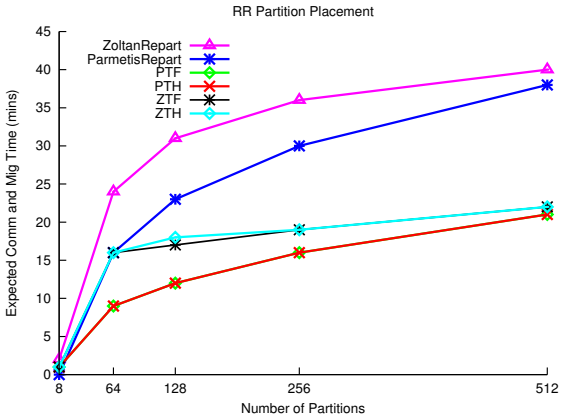

Fig. 6: Varying Num. of Partitions (RR)

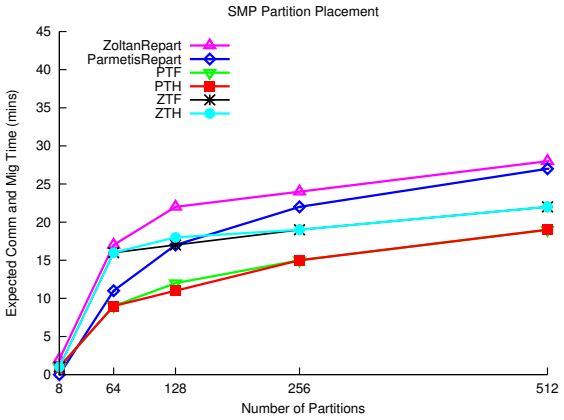

Fig. 7: Varying Num. of Partitions (SMP)

(Table IV) in order to evaluate a range of workloads. For each new graph, we first randomized its edge weights to between 20% and 50% of the sum of the pairwise vertex sizes, and then partitioned the graph into balanced parts using the partitioner from [8]. Later, 20% of the partitions are selected and the weights and sizes of vertices in these partitions are randomly increased to between 1.5 and 7.5 times of their original values, to simulate load fluctuations of the simulation.

Architecture The evaluation was performed with a simulated supercomputer, whose compute nodes are interconnected by a 3D-torus interconnect (5*5*5 by default). Each compute node has 2 quad-core sockets with shared L3 caches and private L1/L2 caches. Partitions of each graph are mapped to cores of the supercomputer at the beginning of each experiment as follows. We first sort the randomly allocated compute nodes by their x coordinates, then by their y coordinates and finally by their z coordinates. Then, we either map partitions of each graph to cores sequentially starting from cores of the first compute node as in the SMP policy[4] or place sequential partitions to the compute node next in the list following the RR policy[4]. In the end, the partitioned graph along with the mapping of partitions to cores and the relative inter-node communication costs serves as the input to repartitioners. The relative inter-node communication costs were estimated by the number of hops (the Manhattan Distance) among compute nodes. We need to clarify here that we do not aim to simulate a full-featured supercomputer. Instead, we just want to evaluate the impact of inter- and intra-node topology on graph repartitioners.

Algorithms We compared ARAGONLB with two widely used architecture-agnostic repartitioners: ZoltanRepart [8] and ParmetisRepart [10]. For ARAGONLB, we evaluated 4 different combinations of our inter- and intra-node repartitioners (Table V). For intra-node repartitioning, we only considered the partitioner from [8] because Parmetis [10] kept failing due to an error originating from its code with our dataset. Supposedly, both inter- and intra-node repartitioning can be

[4]http://www.nics.tennessee.edu/computing-resources/kraken/mpi-tips-for-cray-xt5

TABLE V: Four flavors of ARAGONLB

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

TABLE VI: Cache Access Latencies

Cache	L1	L2	L3
Latency (ns)	1	7	15

any (re)partitioners. All the results presented are the means of 5 trials with 8 MPI processes on an 8-core machine with our simulated architecture. Initially, the graph was evenly distributed across processes for parallel repartitioning.

Metrics The quality of a decomposition in terms of the expected communication and migration cost is defined by Equation 3 and 4. Throughout the evaluation, the cost of communicating or migrating $64B$ data among compute nodes is approximated by 2 times the access latency to the highest cache level weighted by the number of hops. In contrast, the communication cost of $64B$ data between two cores of the same compute node was estimated by the access latency to their first shared cache level. In cases where cores of the same node share no caches, we used 2 times the access latency to the highest cache level as an approximation. The same process was used for the intra-node migration cost. Table VI shows the cache access latencies used.

VII. EVALUATION RESULTS

Varying Number of Partitions (Figures 6 & 7) Our first experiment investigated ARAGONLB's robustness to graphs (Table IV) of varying partitions with $\alpha = 500$ (number of computation steps). The results in Figures 6 & 7 indicate that if only a few compute nodes are needed, Parmetis may obtain decompositions of similar quality as ARAGONLB (i.e., PTF/PTH/ZTF/ZTH) and sometimes even better, especially when ARAGONLB uses Zoltan for its inter-node repartitioning

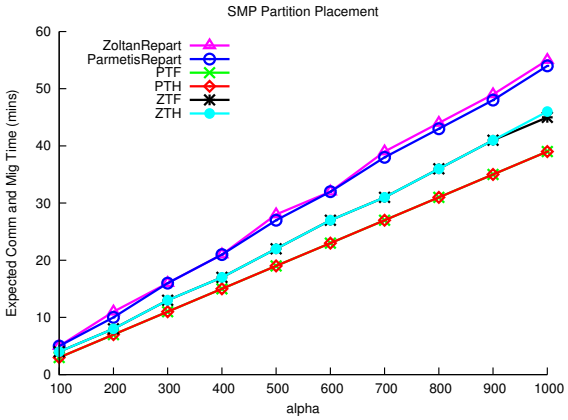

Fig. 8: Num. of Computation Steps

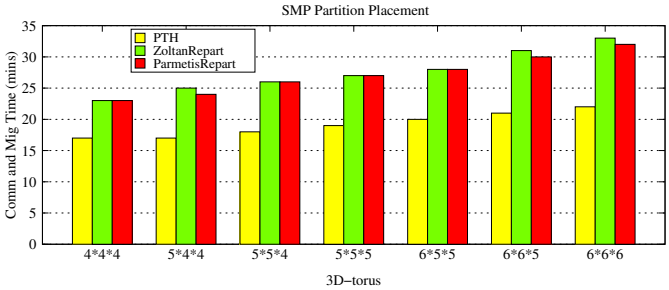

Fig. 9: Different Sized 3D-torus

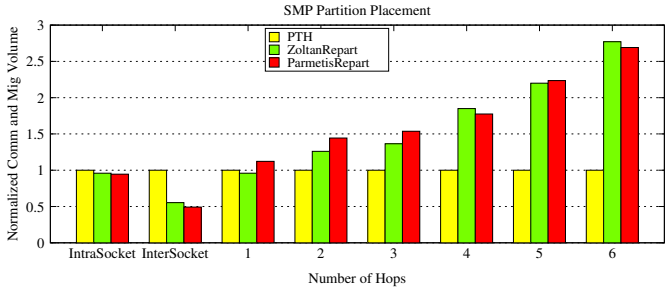

Fig. 10: Normalized communication and migration volume distribution in terms of the number of hops each byte travels.

(i.e., ZTF/ZTH). We believe this was caused by the limited heterogeneity among a small number of compute nodes allocated on a relative small sized 3D-torus (providing limited refinement space for TopoFM). In contrast, with more compute nodes, ARAGONLB can outperform Zoltan and Parmetis by up to 60% and 46%, respectively, and the improvement became bigger as the number of partitions increased (due to the increasing heterogeneity). Since scientific computation usually requires hundreds of compute nodes, we believe that our approach will be beneficial for most applications.

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

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

Varying Number of Computation Steps (Figure 8) This experiment evaluated the influence of α values (number of computation steps) using G512. The results in Figure 8 show that PTF/H and ZTF/H improved the decomposition quality by around 30% and 17%, respectively. We also observed that the improvement became more evident as α increased. As was the case in our previous experiment, the results of PTF/PTH were similar and always outperformed those of ZTF/ZTH. As such, for the rest of this paper, we only present the results of PTH against Zoltan and Parmetis. The reason we prefer PTH over PTF is that PTH does not require any quantitative information about the cache architecture (i.e., cache access latency).

Varying Sized 3D-Torus (Figure 9) This experiment evaluated the impact of different-sized interconnects using G512 with $\alpha = 500$. Figure 9 shows that PTH outperformed Zoltan and Parmetis by 26%-32%, and that the expected communication and migration time of PTH increased slower than that of Zoltan and Parmetis, as the size of interconnect increased, indicating PTH's robustness to increasing heterogeneity.

Also, we expect a bigger difference between PTH and its competitors if the evaluation was carried out with a real application on a real supercomputer. This is because in our simulation-based evaluation, we did not consider the contention for the memory bandwidth and interconnect links, which usually plays a critical role in communication cost. This contention would further favor PTH due to its ability to reduce the communication cost (and the resulting contention). We expect the network link contention (and therefore the difference of PTH with the state-of-the-art) to increase further, as the size of the datasets becomes bigger.

Breakdown of Communication and Migration Volume (Figure 10) To pinpoint the source of the improvement, we computed the breakdown of the communication and migration volume of G512 decompositions output by different algorithms over the different network distances (i.e., number of hops that the data needs to travel) with $\alpha = 500$. Figure 10 presents the overall communication and migration volume distribution in terms of the number of hops each byte traversed, normalized to that of PTH[5]. As shown, PTH produced lower inter-node volume than Zoltan and Parmetis in all cases, and the reduction in inter-node volume became more significant as

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

the number of hops increased. All these reductions added together contributed to around 30% and 35% reduction in the overall inter-node volume of PTH against Zoltan and Parmetis, respectively. The reduction is mainly due to the ability of TopoFM in grouping the most communication-heavy vertices to nodes as close as possible and the design of the two-tier architecture offering more freedom for inter-node repartitioning to group the frequently-communicating vertices into a single partition.

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

TABLE VII: Degree of Imbalance

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

Degree of Imbalance (Table VII) In terms of the imbalance degree, PTH produced decompositions slightly better than Zoltan and Parmetis. The average imbalance degree of different algorithms over all the experiments we ran and their standard deviations are presented in Table VII.

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

VIII. Related Work

Parallel Graph Repartitioning Although there exists a lot of graph (re)partitioners, such as Metis [6], Parmetis [10], Scotch [7], and Zoltan [8], only Parmetis and Zoltan support *parallel repartitioning*. However, they are oblivious of the nonuniform communication costs. Although [19], a Metis variant, consider the heterogeneity, it is a *sequential graph partitioner* and no implementation details about how it takes the heterogeneity into account are presented.

Topology-Aware Dynamic Load Balancing Paper [12] proposes two architecture-aware dynamic load balancers for computation-/communication-bound applications. However, the communication-bound one only considers minimizing the communication cost while ignoring the migration cost, and the computation-bound one prioritizes the communication cost over the migration cost. As we mentioned, inter-node communication and migration cost should be considered at the same time, while intra-node communication cost should be prioritized over intra-node migration cost. HwTopoLB [13] and NucoLB [14] are two other architecture-aware dynamic load balancers. However, HwTopoLB has the same problem as the computation-bound rebalancer of paper [12], whereas NucoLB only considers the nonuniform inter-node communication and migration costs while ignoring the asymmetric intra-node communication and migration costs.

IX. Conclusions

In this paper, we proposed an architecture-aware graph repartitioner, AraGonLB, that is particularly suited for data- and compute-intensive applications on modern parallel computing infrastructures, i.e., typical Big Data scientific applications. AraGonLB considers both the inter-node interconnect and the intra-node computer architecture (i.e., memory hierarchy) during repartitioning. For compute-intensive applications that are also data-intensive, such considerations are extremely crucial. In fact, we showed that AraGonLB outperforms the state-of-the-art (Parmetis and Zoltan) by up to 60% using data derived from a real dataset.

Acknowledgments We thank Peyman Givi, Patrick Pisciuneri, Medhi Nik, Levent Yilmaz, Esteban Meneses, and the anonymous reviewers for their help. This work was funded in part by NSF awards CBET-1250171 and OIA-1028162.

References

[1] T. Hey, S. Tansley, and K. Tolle, eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009.
[2] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi, "Big Data and Its Technical Challenges," *Commun. ACM*, July 2014.
[3] P. Pisciuneri, S. L. Yilmaz, P. Strakey, and P. Givi, "An Irregularly Portioned FDF Simulator," *SIAM J. Sci. Comput.*, 2013.
[4] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, 2000.
[5] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high performance scientific simulations*. Army HPC Research Center, 2000.
[6] "METIS." http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.
[7] "SCOTCH." http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/.
[8] "Zoltan." http://www.cs.sandia.gov/zoltan/.
[9] U. V. Catalyurek *et al.*, "A repartitioning hypergraph model for dynamic load balancing," *Journal of Parallel and Distributed Computing*, 2009.
[10] "Parmetis." http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
[11] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *Supercomputing 2000*.
[12] E. Jeannot *et al.*, "Communication and Topology-aware Load Balancing in Charm++ with TreeMatch," in *IEEE Cluster 2013*.
[13] L. L. Pilla *et al.*, "A topology-aware load balancing algorithm for clustered hierarchical multi-core machines," *Future Generation Computer Systems*, 2013.
[14] L. L. Pilla *et al.*, "A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems," in *ICPP, 2012*.
[15] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation, 1982*.
[16] C. Schulz, *Scalable parallel refinement of graph partitions*. PhD thesis, Karlsruhe Institute of Technology, May 2009.
[17] S. Micali and V. V. Vazirani, "An $O(v|v|c|E|)$ algoithm for finding maximum matching in general graphs," in *Foundations of Computer Science, 1980*.
[18] "Sandia National Laboratories, TNF Workshop Website, Piloted Jet Flames." http://www.sandia.gov/TNF/pilotedjet.html, 2014.
[19] I. Moulitsas and G. Karypis, "Architecture aware partitioning algorithms," in *ICA3PP, 2008*.