Distributed Memory Graph Representation for Load Balancing Data: Accelerating Data Structure Generation for Decentralized Scheduling

Vinicius Freitas, Alexandre Santana, Márcio Castro, Laércio Lima Pilla

Distributed Memory Graph Representation for Load Balancing Data: Accelerating Data Structure Generation for Decentralized Scheduling

Vinicius Freitas[⋆,†], Alexandre de L. Santana[⋆], Márcio Castro[†], Laércio L. Pilla[‡]
[⋆]ECL, Federal University of Santa Catarina (UFSC) — Florianópolis, Brazil
[†]LaPeSD, Federal University of Santa Catarina (UFSC) — Florianópolis, Brazil
[‡] LRI, Univ. Paris-Sud – CNRS — Orsay, France
{vinicius.mctf,alexandre.limas.santana}@posgrad.ufsc.br
marcio.castro@ufsc.br, pilla@lri.fr

May 23, 2019

Abstract

In this paper, we propose a *Distributed Graph Model* (DGM) and data structure to enable communication-aware heuristics in distributed load balancers (LBs). DGM is motivated by the desire to maintain and use information related to the affinity between tasks (their communication) in order to improve data locality while scheduling tasks in a distributed fashion to avoid the centralization overhead. Results show that DGM is able to achieve speedups of up to 50.4x with 40 virtual cores, when compared to a centralized graph representation with the same purpose. Additionally, we propose a proof-of-concept distributed scheduler that uses DGM, named *Edge Migration*, and its implementation in the *Charm++* parallel programming model. These results show that, although the communication analysis is much faster with DGM, it is still the most relevant overhead in distributed LBs. We also observe that Edge Migration has a decision time in the same order of magnitude as other communication-unaware decentralized algorithms. Thus, DGM can be used in communication-aware distributed LBs to improve load balancing decisions with a small impact in the overall LB performance.

Index Terms — High Performance Computing; Load Balancing; Distributed Algorithms; Scheduling; Communication Graph.

1 Introduction

High Performance Computing (HPC) deals with constant power demand and efficiency issues as it moves towards Exascale Computing [1]. HPC machines operate at their peak efficiency when the use of resources is well-spread and synchronizations between parallel tasks are correctly handled. The process of assigning *work units* (or *tasks*) to *processing elements* (PEs) is well-known and documented in scientific computing, and is referred to as Global Scheduling [2], or Load Balancing; which is regarded as an NP-Complete problem [3].

Scientific applications that are iterative in nature may take a long time to execute, even with perfect load balancing. Some of these applications, such as molecular dynamics [4], or wave propagation simulations [5], present irregular workloads, that vary with time. This means that even an apparently optimal task assignment may lead to imbalanced scenarios in the future, causing parallel machines to run inefficiently. These situations demand for a new mapping of work, a process of dynamic load balance.

The process of load balancing may be performed in multiple ways. However, work in the literature suggests three more relevant takes of the problem in the state of the art: (i) topology-aware load balancers (LBs), which seek to optimize the work unit distribution based on the system architecture; (ii) distributed and hierarchical LBs, which are more limited in access to information, but attempt to leverage on the parallel nature of HPC platforms to accelerate the scheduling process; and (iii) graph partitioning techniques that model the application as a graph, in which weighted vertices represent tasks and weighted edges represent communication among them, and apply partitioning algorithms in order to balance the load.

After migrating multiple tasks, dynamic LBs may cause undesired communication overheads. Although applications tend to have an initial task distribution that optimizes their communication based on a geometric decomposition [4, 6, 7], the cost of communications can be considerably increased when task migrations are needed to balance the load. Thus, LBs should minimize the number of task migrations when possible, avoiding extra communication overheads. Moreover, LBs should not incur in decision times (i.e., the time they take to decide a new mapping of tasks to PEs) that degrade the application

performance, overshadowing the benefits of load balancing.

We believe that communication-aware distributed LBs are the key to achieve the desired performance in the parallel platforms leading to the Exascale Era. In this paper, we present the following contributions to the state-of-the-art distributed LBs:

1. A distributed graph data structure to represent application load and communication information named *Distributed Graph Model* (DGM); and

2. A proof-of-concept implementation of a distributed LB, named *Edge Migration*, that makes use of the previously mentioned data structure to perform dynamic task rescheduling in the `Charm++` runtime system [8].

We show that our distributed graph data structure achieved much higher performance than the one available in `Charm++`, being up to 4× faster with 8 PEs; and up to 50.4×, 42.5×, and 21.5× with 40 PEs, depending on the communication topology. Moreover, we show that our proof-of-concept distributed LB achieves faster load balancing decisions than other communication-aware LBs considered in our study.

The remainder of our paper is divided as follows. Section 2 presents a background on load balancing approaches and discusses the state of the art. Section 3 presents our distributed graph data structure for load balancing. Section 4 presents our proof-of-concept LB. Section 5 presents the performance evaluation of our contributions. And finally, Section 6 concludes this work.

2 Background and Related Work

As HPC platforms increase in size, so does the potential of load imbalance. In recent years, a number of parallel runtime systems (RTSs) have implemented static and dynamic scheduling policies, mostly focused on shared memory platforms [9]. However, as applications seek scalability in distributed memory scenarios, they need their own optimal way to map and remap work to resources.

In Section 2.1, we present recent efforts in development of scheduling and dynamic load balancing approaches for HPC. Then, in Section 2.2, we discuss the limitations of topology-specific algorithms (as well as topology discovery overheads), and techniques used to represent application and machine communication as graphs.

2.1 Efforts in Load Balancing

We may divide the behavior of load balancing algorithms in three main categories, as depicted in Fig. 1. In this

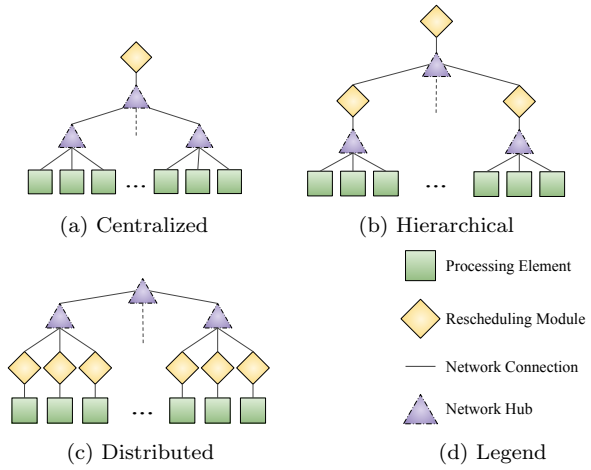

Figure 1: Three different load balancing mechanism organizations.

representation, *computing resources* (or PEs) communicate through *network hubs*, and use *rescheduling modules* (or LBs) to perform *static* or *dynamic* load balancing. In the former case, the application must be partitioned and communication edges established by the LBs before execution. In the latter case, on the other hand, the application is already executing, and LBs gather information to perform dynamic migration of work units.

Fig. 1a exhibits the behavior of *Centralized LBs*. They use absolute system information to perform load balancing, but may incur in high overheads and frequent bottlenecks, due to its centralized nature. Meanwhile, Fig. 1b represents *Hierarchical* (or *Multilevel*) *LBs*, that tend to use different load balancing heuristics in different hierarchy levels [10]. Many of these strategies still need to start or finish as centralized approaches to gather their data, which may lead to undesired bottlenecks. Finally, Fig. 1c represents completely *Distributed LBs*, which focus on balancing local resources, without global information.

Due to their decentralized nature, *Distributed* mechanisms are the most scalable of the presented approaches. Traditionally, these algorithms used a *diffusive* approach [11], which consists of iteratively sending work to neighbors that carry a lighter workload. This leads to *refinement-based* approaches, which attempt to send work units to underloaded resources using probabilistic distributions (e.g., Grapevine [12]) or accumulating work to mitigate communication increases after migration (e.g., PackDrop [13]).

Recent efforts on *Centralized* and *Hierarchical* LBs consist mostly of machine topology-aware heuristics [14]. In Non-Uniform Memory Access (NUMA) machines, centralized approaches that focus on topology and NUMA-factor have been used in different levels to perform asymp-

totically optimal scheduling of tasks to resources (e.g., NuCo, HwTopo and Hierarchical [15]). In the same kind of environment, decentralized Work Stealing (WS) mechanisms have been used to decrease migration overheads with distance-aware WS [16]. Meanwhile, TreeMatch attempts to match application communication graphs and Fat Tree machine topologies to balance application workload [17].

Weighted-Hop and Max-Congestion [18] focus on optimizing metrics that are more commonly available in distributed memory environments. The former attempts to minimize the *total hop count*, while the latter aims at a minimal *maximum message congestion*. Both strategies were designed in a centralized fashion, but as they are an initial greedy graph partition followed by a refinement-method, they may be adapted to work in a hierarchical (or multilevel) fashion. Following the multilevel approach, some scheduling heuristics exploit multiple RTSs (e.g., `Charm++` and OpenMP [19]) to refine load balancing in irregular applications.

2.2 Information Gathering and Representation

Although the use of machine topology information positively impacts the application performance when scheduling tasks to resources, this approach also has its limitations. For instance, most of the aforementioned algorithms (NuCo and HwTopo [15], TreeMatch [17], Distance-Aware [16], and Max-Congestion [18]) are topology-specific, so their heuristics may not be portable to different machine configurations. Additionally, the overhead of discovering and generating topology data structures may be limiting in very large environments, especially since recent works already present high costs when applied to modern multi-core architectures, such as Intel Knights Landing [20].

Precise representations of machine topology and application communication usually come in the form of graphs [18, 21] or hypergraphs [22]. Weighted-Hop and Max-Congestion, for instance, are based on graph partitioning techniques. Zoltan [22] uses a multilevel hypergraph partitioning scheme to perform both scheduling and dynamic load balancing of parallel applications. ParMETIS [21] is a multilevel implementation of METIS *k-way* graph partitioning, which can also be used to optimize task assignment based on application communication [23].

Geometric decomposition and *over-decomposition* approaches have proven to improve the communication and load balancing in several parallel applications [7, 4]. They are usually applied in the initial scheduling, while dynamic LBs focus on refining previous assignments of tasks to resources. With this in mind, we envision that the precise (hyper)graph representation of application communication may help schedulers that attempt to preserve previously assigned geometric partitions, which should have already optimized the communication.

3 Distributed Graph Model

Distributed memory parallel systems are some of the most scalable platforms in HPC today. Due to their elasticity, increasing the amount of resources an application can use to execute is simple, and may lead to great performance gains. However, as parallel environments move from *shared* to *distributed* memory, the communication costs of applications rapidly increase, mostly because of higher overheads in cluster nodes exchanging messages in comparison with straight memory access. Decomposing applications with these environments in mind demands a minimal amount of remote communication in order to minimize message-exchanging overheads.

Performing communication-aware load balancing in the aforementioned models require some dynamic information about the application communication. This means that, in order to reschedule work, parallel systems need to monitor the messages exchanged during application runtime. As extra information may help load balancers, it is still important to have low overheads when reassigning application workload, since this process also incurs in overheads. Fortunately, in geometrically decomposed applications, the initial distribution of work usually attempts to optimize communication among work units, which may be preserved by dynamic load balancers.

With the aforementioned concepts in mind, LBs should fulfill the following objectives to achieve the best possible performance:

1. Evenly distribute work units to PEs to diminish application makespan;
2. Attempt to preserve most of the initial work scheduling to avoid undesired communication overheads;
3. Perform work reassignment as fast as possible to minimize load balancing overheads.

In this section, we propose a decentralized communication representation model called *Distributed Graph Model* (DGM). This approach intends to portray the communication scheme locally for each resource, while still avoiding most of the centralization effort that may incur in a high overhead.

3.1 Definition

DGM intends to organize application communication and task loads in a completely decentralized fashion, using the information the RTS provides.

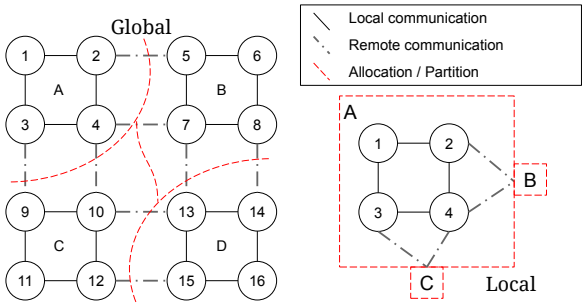

Figure 2: Graphical Representation of Global (left) and Local (right) perspectives of our Distributed Graph Model. Weights are omitted for simplicity.

Consider an original centralized application graph represented as $G = (V, E, W_V, W_E)$, where V represents the vertices (tasks), E their directed edges (communication), W_V represent the vertex weights (load of the tasks), and W_E is the edge weights (volume of communication). This graph will be represented as p disjoint sub-graphs distributed over p resources, such that $\bigcup_{i=1...p} G_i = G$.

$$G_i = (V_i, V_i^o, E_i, E_i^o, W_{Vi}, W_{Ei}, W_{Ei}^o) \quad (1)$$

In DGM, each local sub-graph G_i can be described by the components illustrated in Equation 1. V_i contains the vertices local to this partition ($V_i = \{v \mid v \in V \wedge v \notin V_j, \forall j \neq i\}$), E_i contains the internal edges ($E_i = \{(u,v) \mid u,v \in V_i\}$), and W_{Vi} and W_{Ei} represent the weights of local vertices and edges, respectively.

DGM also preserves information related to the edges among different sub-graphs. Consider an outgoing edge $(u,v) \in E$, where $u \in V_i$ and $v \notin V_i$. Instead of keeping copies of all outgoing edges and external vertices, we summarize this information with new sets of artificial vertices, edges, and edge weights V_i^o, E_i^o, and W_{Ei}^o, respectively.

A vertex $v_j^o \in V_i^o$ represents all vertices in V_j that are a tail of an edge coming from vertices in V_i. Using this idea, we can define $E_i^o = \{(u, v_j^o) \mid \exists\, (u,v) \in E, u \in V_i \wedge v \in V_j\}$ as the set of edges going outside of the partition. Finally, the weight of an outgoing edge $(u, v_j^o) \in E_i^o$ is equal to the sum of the weights off all edges from u to vertices in V_j, as illustrated in Equation 2.

$$w_{(u,v_j^o)} = \sum_{(u,v)\in E, v\in V_j} w_{(u,v)} \quad (2)$$

A graphical representation of our model is portrayed in Fig. 2. The left side of the illustration represents the original application decomposition in the form of tasks (numbered vertices), communication between them (straight and dashed gray edges), and assigned resources (A-D), divided by a red dashed line, representing graph partitions. Meanwhile, the right side represents the local DGM representation in resource A, where edges that reach remote resources direct to virtual vertices (v_B^o and v_C^o).

3.2 Implementation Details

In order to optimize the use and construction of DGM, we have chosen some specific data structures to store the information, and techniques to produce them. We assume a situation with two parameters: (i) G, which is a local graph in the format of an unsorted array, describing communication in the form of a pair ($from$, to), and the existence of n vertices, and a total of m edges; and (ii) V, an array of vertex loads (with n elements). This representation has high costs when it comes to finding specific edges, especially if it is not sorted in any way. We also assume that each PE will build their local representation in a distributed fashion.

The data structures in which we intend to store our local graph are:

- $\mathcal{T}$: A *hash*-table with key equal to a task identifier (id) and storing its $load$. The notation to access element α in this table will be denoted as: $\mathcal{T}_\alpha$ in our algorithms;
- $\mathcal{F}$: A map of sets, where the key is the identifier of a neighbor PE, and the sets have the ids of all tasks that communicate with this neighbor. We call this concept a *frontier*, and the frontier of the local PE with i will be denoted as $\mathcal{F}_i$;
- $\mathcal{H}$: An inverted *heap* with a tuple (id, $load$), sorted by $load$ (lower to higher). This will be a stub for our inner edge representation at this point[1].

We utilize this set of attributes to ensure a fast access to elements needed in the load balancing step ($O(1)$). Additionally, these data structures have a considerably cheap cost to be generated ($O(n\ log\ n + m)$ in the worst case), and should take a space of $2n$ to store, since every data slice will be represented once on $\mathcal{T}$, and another time on each of the other data structures. In future implementations, the cost of considering the inner edges may increase the overheads related to DGM. That is why, for this implementation, since we only need the outer edges, we have aggregated the inner vertices in a cost-saving fashion.

Algorithm 1 describes the process of generating a local representation of system information with inputs G and V, described above. For simplicity, the function call and data-field access notations used in Algorithm 1 is detailed in Table 1, data-fields src, id, and $weight$ refer to vertex resource, id and load, respectively. The first step of the

[1]Our proposed load balancing strategy, presented in Section 4, focuses on the use of *migration frontiers*, so the internal representation is not as important for us at this point.

Algorithm 1: Create Local Graph

Input: G, communication graph as an array of edges; V, list of local vertices with data fields *src*, *id* and *weight*; P_{self}, local resource.

Output: $\mathcal{T}$, table of vertices; $\mathcal{F}$, set of neighbor frontiers; $\mathcal{H}$, heap of remaining vertices.

1 $\mathcal{T} \leftarrow \varnothing, \quad \mathcal{F} \leftarrow \varnothing, \quad \mathcal{H} \leftarrow \varnothing, \quad U \leftarrow \varnothing$
2 **foreach** $v \in V$ **do**
3 $\quad \mathcal{T}_v \leftarrow (v_{id},\ v_{weight})$
4 **foreach** $(u, v) \in G$ **do**
5 $\quad$ **if** $v_{src} \neq P_{self}$ **then**
6 $\quad\quad \gamma \leftarrow v_{src}, \quad \mathcal{F}_\gamma \leftarrow \mathcal{F}_\gamma \cup \{u\}$
7 $\quad\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{\mathcal{F}_\gamma\}, \quad U \leftarrow U \cup \{u\}$
8 $\mathcal{H} \leftarrow V \setminus U, \quad \mathcal{H} \leftarrow HeapSort(\mathcal{H})$

Table 1: Distributed Algorithms Notation

Notation	Meaning
v_w	Access to data-field w of vertex v.
$\mathcal{A}_\alpha$	Access to data structure $\mathcal{A}$ at index α.
$\delta \leftarrow f()$	Assign the output of function $f()$ to variable δ.
$f() \rightarrow b$	Calling function $f()$ in remote resource b.
$g() \Rightarrow B$	Calling *multicast* function $g()$ on each element of B.
$\delta \leftarrow (g(a) \Rightarrow B)$	Assign the result of *multicast* $g(a)$ to δ. Usually used in *reduction* operations.

algorithm is to populate the table of tasks (lines 2 and 3). Then, we parse through the graph (G), seeking the pairs (u, v), in which one of the elements belongs to a remote resource (our notation was simplified in Algorithm 1, lines 4-7). Whenever a remote communication is detected, the remote resource (γ) has the local vertex (u) added to its frontier ($\mathcal{F}_\gamma$). Additionally, u is added to a temporary set of tasks U, which will be used to separate remote communication tasks of local ones. Finally, $\mathcal{H}$ receives the local vertices that did not have remote communication (line 8), and has its values sorted in ascending order.

4 Communication-Aware Load Balancing

In this section we propose a distributed load balancing algorithm as a proof-of-concept for the use and performance analysis of DGM. Our execution environments portray one instance of our distributed scheduler per PE, and these instances communicate with each other asynchronously to execute the scheduling algorithm. The dynamic load balancing steps occur after a synchronization of all PEs, followed by a pause in the application runtime. The RTS should then provide the algorithm with execution data and remap work units once the new mapping is given by the schedulers.

Algorithm 2: Edge Migration Scheduler

Input: G, V as in Algorithm 1; P, list of system resources, where P_{self} refers to the local resource.

Output: $\mathcal{M}$, changes in work assignment in the form $\{(v_i, P_j)\}$.

1 $\mathcal{M} \leftarrow \varnothing$
2 $\mathcal{T}, \mathcal{F}, \mathcal{H} \leftarrow$ CreateLocalGraph(G, V, P_{self}) `// Alg. 1`
3 $\beta \leftarrow$ (AverageLoadReduction$(load_{set}(\mathcal{T})) \Rightarrow P$)
4 **if** $load_{set}(T) < \beta$ **then** `// Underloaded, Eq. 3`
5 $\quad P_\gamma \leftarrow$ ChooseNeighbor$(P, \mathcal{F})$ `// Equation 4`
6 $\quad$ RequestLoad$(P_{self}, load_{set}(\mathcal{T})) \rightarrow P_\gamma$ `// Alg. 3`
7 — Wait Completion —
8 Submit$(\mathcal{M}) \Rightarrow P$

The idea behind our approach is to first optimize communication, and afterwards the distribution of load. Although the second is the most relevant for overall system balance, since we want to preserve communication as much as possible, we prioritize the migration of *communication edges* (frontiers), that is, tasks that already communicate with remote resources. Our asynchronous distributed algorithms will follow the notation described in Table 1.

4.1 Distributed Edge Migration

We have followed a pull-based approach in the development of our novel scheduling policy (Algorithm 2). Thus, the *Edge Migration Algorithm* works from underloaded to overloaded resources, much like Work-Stealing policies. The inputs to this policy are the same workload and communication information shown in Algorithm 1 (G, V), and a list of system resources (P). The output is a mapping of changes in work assignment, in the form of a set of pairs $(task, new\ resource)$, $\mathcal{M}$.

Algorithm 2 starts by creating the local graph, described in Algorithm 1 (line 2). Then, it performs a reduction to find the average load of resources in the system ($\overline{x}$). This result is multiplied by a threshold t, in the following form:

$$\beta = \overline{x} \times (1 - t)$$

to achieve the value of β attributed in line 3. The objective behind this is to determine a lower threshold, where we start considering resources underloaded. We also assume that during this reduction operation the scheduler is able to learn the load of its neighbor resources, which is necessary for future steps of the algorithm.

The next step is to use Equation 3 to determine if the resource is underloaded (line 4). If this is the case, then we must choose a neighbor, based on Equation 4, from which the scheduler will request extra work (lines 5 and 6). The chosen neighbor should be the one with maximum load, prioritizing those that share a communication frontier with the current resource. Load requesting and donation processes are explained further in Algorithm 3.

$$load_{set}(A) = \sum_{a \in A} a_{weight} \tag{3}$$

$$\text{ChooseNeighbor}(P, \mathcal{F}) = P_\gamma \mid \gamma \in \arg\max_{\mathcal{F}_\gamma \in \mathcal{F}} load_\gamma \tag{4}$$

At this point, we launch a Quiescence Detection process, which will wait until no more requests have to be answered (line 7). Finally, all remapping of workload is done in Algorithm 3 and attributed to $\mathcal{M}$, so all we need to do is commit the changes performed by each scheduling entity (line 8).

The parameters of Algorithm 3 are the requesting resource's *id* and *load* (P_γ and $load_\gamma$). It is important to note that it will be executed in the remote resource chosen in lines 5-6 of Algorithm 2. Moreover, the RequestLoad process manages data from variables defined in the main execution branch of the scheduler. Collateral effects take place in $\mathcal{M}, \mathcal{T}, \mathcal{F}, \mathcal{H}$ and β for future calls of the algorithm.

The first step of Algorithm 3 is to determine the load it should migrate to the requesting resource P_γ (φ). This is done through Equation 5. Then, it must extract φ from the available workload contained in $\mathcal{T}$, using the supporting data structures $\mathcal{F}_\gamma$ and $\mathcal{H}$. It removes tasks from the $\mathcal{F}_\gamma$ until it reaches the target load φ or the frontier is exhausted. If the frontier is exhausted, subsequent tasks will be removed from the inner vertices ($\mathcal{H}$) until φ is achieved in $\mathcal{T}$.

$$\text{DetermineLoad}(x, y, z) = \begin{cases} x - z, & \text{if } \frac{x+z}{2} > y \\ x - y, & \text{otherwise} \end{cases} \tag{5}$$

Once the migrating tasks are defined (L in line 2) a message is sent to the requesting resource to update its expected load after migrations (line 3). Then, $\mathcal{M}$ is updated in order to account for the new migrations. Finally, if the load of tasks in the local resource has led to an underloaded state ($load_{set}(\mathcal{T}) < \beta$ in line 5), the resource must choose a new neighbor to request extra work (lines 6 and 7). Additionally, since this is a pull-based scheduler, a stop criteria must be added as a maximum number of requests in line 5 of Algorithm 3. This kind of criteria is important to guarantee that the strategy will finish in a timely fashion, and not get into request-donation (or *stealing*, in Work-Stealing schedulers) cycles [24].

Algorithm 3: Request Load

Input: $P_\gamma, load_\gamma$, remote resource and its load.
Data: Local variables in Algorithm 2: $\mathcal{M}, \mathcal{T}, \mathcal{F}, \mathcal{H}, \beta$.

```
1 φ ← DetermineLoad(load_set(T), β, load_γ) // Eq. 5
2 L ← ExtractLoad(φ, F_γ, T, H)
3 Confirm(load_set(L)) → P_γ
4 M ← M ∪ {(∃v ∈ L, P_γ)}
5 if load_set(T) < β then      // Underloaded resource
6 |   P_γ ← ChooseNeighbor(P, F)      // Equation 4
7 |   RequestLoad(P_self, load_set(T)) → P_γ // Alg. 3
```

4.2 Implementation

To implement our strategy using DGM, we have chosen the `Charm++` runtime system [8]. This is a message-driven parallel programming environment, which portrays a dynamic load balancing framework among its features. Additionally, it also presents benchmarks and implementations of other load balancing algorithms, with which we may compare our approach. Communications performed by our scheduler were implemented as asynchronous messages (entry methods) in the runtime system, and we have used built-in *Reduction* and *Quiescence Detection* utilities to implement the AverageLoadReduction and Wait Completion functions, respectively, previously presented in Algorithm 2. It is important to maintain most of the algorithm behavior asynchronous to maximize performance, so these two functions are the only synchronization steps in algorithm runtime.

5 Performance Evaluation

DGM is a novel load balancing data representation that was implemented within our Edge Migration load balancer in `Charm++`. To assess the performance of our model and proposed scheduling policy, we have compared our data structure to the standard `ObjGraph` data structure provided with the RTS's load balancing framework, which we will refer to as *Charm Graph*. Moreover, we have compared the Edge Migration scheduling to other communication-aware and distributed strategies available in `Charm++`.

Our evaluation methodology divides this section in two parts: (i) Section 5.1, which evaluates the data structure generation times; and (ii) Section 5.2, which evaluates the load balancing time and application performance. The description of our experimental environment is portrayed in Table 2. Both platforms used `Charm++` version 6.8.1, `GCC` 5.4.0, and were executed with simultaneous multithreading (*hyperthreading* or SMT) turned on.

We carried out experiments with *LB Test*, which is a

Table 2: Platforms Description

Characteristics	Kaby	Tesla
# of CPUs	4 (UMA)	2×10 (NUMA)
CPU model	Intel Core i7-7700	Intel Xeon E5-2640
CPU Freq.	3.60GHz	2.40GHz
RAM	8GB@1200GHz	128GB@1333GHz ECC
OS	Linux Mint 18.2	Ubuntu 16.04

well-known benchmark shipped with the `Charm++` RTS and is largely adopted by the `Charm++` community. *LB Test* is able to simulate different communication topologies among tasks and workloads to evaluate the efficiency and viability of load balancing algorithms. The user is able to specify the limits of a uniform distribution of task workloads, as well as the number of work units. Additionally, its communication topologies replicate those of real-word applications, focusing from very low to high communication volumes. These topologies may be portrayed as a one, two, or three dimensional torus networks that could be seen as directed graphs in which every vertex has the same in and out degree of one, two, or three, respectively. We call these schemes *Ring*, *Mesh2D*, and *Mesh3D*. The fourth possible topology is a *Random Graph* that has 1% of all possible edges, which represents a more communication intensive scenario than any of the other three.

5.1 Graph Generation

The centralized Charm Graph consists of an array of vertices, each of which contains an array of ingoing and outgoing edges. The asymptotic generation time of Charm Graph should be of $O(n + m)$ reads and writes (for each 1 read, 1 write) for n vertices and m edges. Additionally, the cost to build this graph aggregates the time necessary to perform a reduction of all load and communication data, which is collected independently in each vertex during runtime. This adds a considerable communication overhead ($O(log\ k)$, for k PEs), which is completely avoided by our distributed approach during data structure generation time.

We have compared the time needed to generate the Charm Graph and the DGM after 40 iterations of the *LB Test* benchmark. In this experiment, the benchmark had a total of 12K tasks, ranging from 60μs to 4,120μs in duration. We varied this configuration through the 3 most structured configurations of the benchmark, Ring, Mesh2D, and Mesh3D. This experiment was executed 20 times with 40 threads (one per virtual PE) on the *Tesla* platform, and we have observed a maximum 5.08% standard deviation from the mean time.

Results of this experiment are portrayed in Figure 3. These results show DGM achieved *speedups* of 21.5x, 42.5x, and 50.4x for Ring, Mesh2D and Mesh3D, respectively. This acceleration reflects both the parallelism of our solution and the power of the distributed approach, which does not require centralized information. Another reason for this superlinear speedup is due to better use of memory resources in the parallel machine. While a centralized approach stresses all cache levels in a single PE at once, the distributed one will generate a lighter demand on the cache of each individual PE.

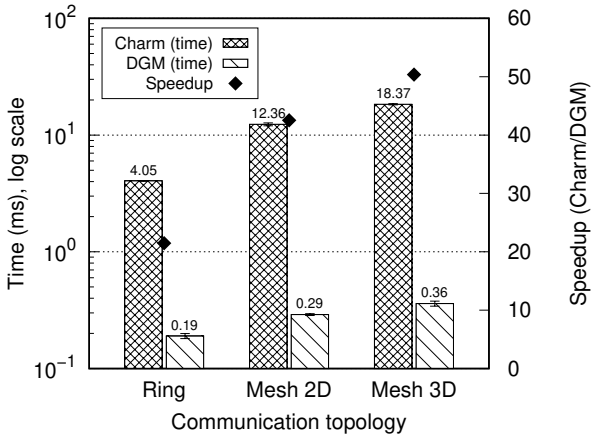

Figure 3: Graph data structure generation time in *LBTest* on **Tesla**.

5.2 Load Balancing

We have used three standard `Charm++` load balancers in order to evaluate the performance of our Edge Migration approach, as well as a control dummy algorithm, in order to assess the data structure overhead of centralized strategies. We give below a brief description of these LBs:

1. *RefineComm* is a centralized communication-aware strategy that attempts to minimize the number of migrations from overloaded to underloaded processors, while taking communication between tasks into account [10];

2. *GreedyComm* is a centralized communication-aware strategy that assigns the most loaded tasks to the least loaded PEs, prioritizing those that it communicates with [10];

3. *Distributed* is the `Charm++` implementation of the Grapevine distributed algorithm [12]. It uses a gossip protocol to assess partial system information and then, performs a diffusive transfer of load, from overloaded to underloaded PEs. Distributed uses a probabilistic distribution to make educated random choices of migration targets, guaranteeing higher chances of choosing the least loaded PEs;

Table 3: Results of *LB Test* with multiple LB algorithms on **Kaby** ($\overline{x}$ is the mean, σ is the standard deviation).

Load Balancer		Application Time (s)	LB Strategy Time (s)
EdgeMigration	$\overline{x}$	409.29	0.15
	σ	5.28	0.07
GreedyComm	$\overline{x}$	527.56	3.18
	σ	1.12	0.15
RefineComm	$\overline{x}$	478.02	10.96
	σ	8.97	24.18
Distributed	$\overline{x}$	393.53	0.001
	σ	6.02	0.000
Dummy	$\overline{x}$	394.59	0.60
	σ	1.43	0.06

4. *Dummy*, our placeholder data structure overhead load balancer. It will simply generate the Charm Graph and finish execution.

We have compared the performance of these LBs with 150 iterations of the *LB Test* benchmark, performing load balance every 20 iterations, totalizing 7 LB calls per execution. The benchmark had a total of 21K tasks, ranging from 1μs to 1,000μs in duration. In this experiment, we have only evaluated the most communication intensive topology, the Random Graph, which generated a total of 4,409,790 communication edges. This experiment was executed 10 times with 8 threads on the *Kaby* platform.

Table 3 shows the overall *application time* and *LB strategy time* in seconds. The application time refers to the average time one execution of the application takes to perform the 150 iterations, while the LB strategy time refers to the time of one load balancing call, from the time the application is synchronized to the moment it resumes execution. Results obtained with Dummy suggest that, due to the parallelism limitations of *Kaby*, the load imbalance was not so relevant as to cause considerable impacts to the application time. Disregarding the synchronization and data structure generation overheads in Dummy (7×0.6s), we see that even Distributed was not as effective as not performing any load balancing at all. Nevertheless, when comparing Edge Migration to the other 2 communication-aware schedulers (RefineComm and GreedyComm), we noticed that it achieved the best execution times.

When observing the load balancing strategy time, Edge Migration overcomes the centralized approaches, and even the Dummy LB. However, Distributed is still much faster in comparison. Since both Edge Migration and Distributed are decentralized approaches, this suggests that the data structure generation time has a relevant impact in this regard. Table 4 shows the breakdown of the LB strategy times shown in Table 3 into the necessary time to *build the graph structure* (Charm Graph or DGM) and to take the *LB decision*. Since Distributed does not take communication into account, its does not present this graph structure generation overhead.

Table 4: Communication and data structure evaluation of *LB Test*'s Random Graph topology on **Kaby**.

Load Balancer	Build Graph Time (s)	LB Decision Time (s)
EdgeMigration	0.15	0.002
GreedyComm	0.60	2.58
RefineComm	0.60	10.36
Distributed	–	0.001

Here, we observe that in *Kaby*, with the Random Graph topology, the creation of DGM was 4x faster than Charm Graph. We also observe that, excluding the DGM generation time, the decision time of our algorithm is in the same order of magnitude as Distributed, which is expected, since both are decentralized approaches.

6 Conclusion

In this paper, we have presented a *Distributed Graph Model* (DGM) to represent application communication in distributed memory scenarios. Since many communication-aware scheduling algorithms are based on graph partitioning, this is an important movement forward in the development of informed decentralized load balancing heuristics. Results presented in Section 5.1 highlight that this distributed approach is able to outperform a centralized graph representation in every analyzed scenario, while enabling the use of communication information in decentralized load balancing algorithms.

Alongside DGM, we have presented and implemented Edge Migration, a decentralized load balancer that works as a proof-of-concept for the use of communication-awareness in distributed schedulers. DGM and Edge Migration were implemented in `Charm++`, and compared to distributed and communication-aware LBs available in this RTS. Results discussed in Section 5.2 show that Edge Migration was the most efficient among all communication-aware strategies considered in this study. Additionally, the load balancing time shows that the communication analysis to generate the data structure is accountable for great part of the overhead in decentralized strategies, while the scheduling heuristic is still able to take decisions in times similar to other decentralized approaches.

The increasing performance demand of parallel applications in HPC environments creates a need for fast, reliable and efficient schedulers. Both our experiments and the literature [10, 12, 13, 22] indicate that parallel and distributed load balancers are the best candidate to fulfill this role, especially when dynamic rescheduling is required. We believe that DGM will bring more benefits to communication-aware strategies in the future, helping

to achieve Exascale performance in distributed memory environments.

6.1 Future Work

Future work in this field includes the use of classic graph partitioning approaches from the literature such as METIS [21, 23] or SCOTCH [25] to the internal portion of DGM, instead of considering only the frontiers (in Section 3.1: $V_i \backslash V_i^o$ of G_i). The use of graph partitioning to create homogeneous clusters of communicating tasks inside DGM may be used to migrate these aggregates of tasks, which should preserve the most affinity among them [26]. This approach would enhance the naive Batch Task Migration [13], which could lead to even more efficient communication-aware scheduling.

Another ramification of this work is the use of topology information alongside communication-awareness. Although incurring in additional overhead, these topology-aware schedulers have recently shown great benefits in increasing application performance [16, 17, 18].

Acknowledgement

This work was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq).

References

[1] J. Mair, Z. Huang, D. Eyers, and Y. Chen, "Quantifying the energy efficiency challenges of achieving exascale computing," in *Proceedings of International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Shenzhen, China: IEEE/ACM, 2015.

[2] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on software engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, USA: W. H. Freeman & Co., 1979.

[4] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C. Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Seattle, USA: IEEE/ACM, 2011, pp. 61:1–61:11.

[5] R. Keller Tesser, L. Mello Schnorr, A. Legrand, F. Dupros, and P. O. A. Navaux, "Using Simulation to Evaluate and Tune the Performance of Dynamic Load Balancing of an Over-decomposed Geophysics Application," in *Proceedings of European Conference on Parallel and Distributed Computing (EuroPar)*. Santiago de Compostela, Spain: Springer, 2017, pp. 192–205.

[6] A. Bhatele and L. V. Kalé, "Heuristic-based techniques for mapping irregular communication graphs to mesh topologies," in *Proceedings of International Conference on High Performance Computing and Communications (HPCC)*. Banff, Canada: IEEE, 2011, pp. 765–771.

[7] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Çatalyürek, and K. Devine, "Exploiting geometric partitioning in task mapping for parallel computers," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*. Phoenix, USA: IEEE, 2014, pp. 27–36.

[8] B. Acun, A. Langer, E. Meneses, H. Menon, O. Sarood, E. Totoni, and L. V. Kalé, "Power, reliability, and performance: One system to rule them all," *IEEE Computer*, vol. 49, no. 10, 2016.

[9] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *Springer Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, 2018.

[10] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé, "Periodic hierarchical load balancing for large supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 25, no. 4, pp. 371–385, 2011.

[11] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 4, no. 9, 1993.

[12] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Denver, USA: ACM, 2013, pp. 15:1–15:11.

[13] V. Freitas, A. Santana, M. Castro, and L. L. Pilla, "A batch task migration approach for decentralized global rescheduling," in *Proceedings of International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Lyon, France: IEEE, 2018, pp. 49–56.

[14] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards *et al.*, "Trends in data locality abstractions for HPC systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 10, 2017.

[15] L. L. Pilla, P. O. A. Navaux, C. P. Ribeiro, P. Coucheney, F. Broquedis, B. Gaujal, and J.-F. Méhaut, "Asymptotically optimal load balancing for hierarchical multi-core systems," in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*. Singapore: IEEE, 2012.

[16] R. Al-Omairy, G. Miranda, H. Ltaief, R. Badia, X. Martorell, J. Labarta, and D. Keyes, "Dense matrix computations on numa architectures with distance-aware work stealing," *Supercomputing Frontiers and Innovations (SuperFRI)*, vol. 2, no. 1, 2015.

[17] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters:algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 4, pp. 993–1002, 2014.

[18] M. Deveci, K. Kaya, B. Uçar, and U. V. Catalyurek, "Fast and high quality topology-aware task mapping," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad, India: IEEE, 2015.

[19] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Multi-level load balancing with an integrated runtime approach," in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. Washington, USA: IEEE/ACM, 2018, pp. 31–40.

[20] B. Goglin, "On the overhead of topology discovery for locality-aware scheduling in HPC," in *Proceedings of Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. St. Petersburg, Russia: IEEE, 2017, pp. 186–190.

[21] D. Lasalle and G. Karypis, "Multi-threaded graph partitioning," in *Proceedings of International Symposium on Parallel and Distributed Processing (IPDPS)*. Boston, USA: IEEE, 2013, pp. 225–236.

[22] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. T. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*. Long Beach, USA: IEEE, 2007.

[23] A. Bhatele, S. Fourestier, H. Menon, L. V. Kalé, and F. Pellegrini, "Applying graph partitioning methods in measurement-based dynamic load balancing," Lawrence Livermore National Laboratory (LLNL), Technical Report, 2012.

[24] J. Yang and Q. He, "Scheduling parallel computations by work stealing: A survey," *International Journal of Parallel Programming (IJPP)*, vol. 46, no. 2, pp. 173–197, 2018.

[25] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel computing*, vol. 34, no. 6-8, pp. 318–331, 2008.

[26] N. Cheriere and E. Saule, "Considerations on distributed load balancing for fully heterogeneous machines: Two particular cases," in *Proceedings of International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. Hyderab, India: IEEE, 2015.