Scalable Load Balancing: Distributed Approaches and the Packing Model

Vinicius Freitas, Laércio Lima Pilla, Alexandre Santana, Marcio Castro, Johanne Cohen

Scalable Load Balancing: Distributed Approaches and the Packing Model

Vinicius Freitas, Laércio L. Pilla, Alexandre de L. Santana, Márcio Castro, and Johanne Cohen

Abstract—Periodical load balancing heuristics are employed in parallel iterative applications to assure the effective use of high performance computing platforms. Work stealing is one of the most widely used load balancing techniques, but it is not the most friendly for iterative applications. Optimal mapping of tasks to machines, while minimizing overall makespan, is regarded as an NP-Hard problem; so suboptimal heuristics are used to schedule these tasks in feasible time. Among the existing approaches, distributed load balancers are the most scalable for iterative applications and have much to profit from work stealing. In this work, we propose the discretization of application workload for load balancing, as well as two distributed load balancers: PackDrop, which is based on constrained work diffusion; and PackSteal, which is based on work stealing. Our algorithms group tasks in batches before migration, creating packs of homogeneous load to make scheduling decisions in an informed and timely fashion. Our results show that PackSteal and PackDrop enhanced our molecular dynamics benchmark performance by up to 41% and 29%, respectively, on our largest evaluated scale. Moreover, PackSteal is consistently the most effective in 8 of 9 evaluated scenarios, compared to PackDrop and other load balancing algorithms.

Index Terms—C.1.2.e Load balancing and task assignment, C.1.4.d Scheduling and task partitioning, C.2.4 Distributed Systems, D.2.8.b Performance measures

1 Introduction

High Performance Computing (HPC) applications are solving some of the most important scientific and industrial problems of our time. Many of these applications are based on parallel iterative methods to solve large-scale problems in different research domains, such as Molecular Dynamics (MD) [1], Wave Propagation [2], N-Body simulations [3], and Dense Matrix computations [4]. As HPC platforms grow bigger [5], the workload of these applications must be precisely scheduled on the available resources, avoiding the waste of costly large scale platforms [6].

Load imbalance emerges in parallel applications when resources go idle, waiting for others to complete their tasks (workload units). This problem is accentuated in dynamic scenarios, where the workload of an application evolves during run time. These applications require periodical remapping of their tasks to use resources efficiently. Since finding the optimal mapping of tasks in parallel machines is an NP-Hard problem [7], [8], a number of different techniques to compute solutions in a feasible time have been proposed.

Many data-parallel and iterative simulations tend to follow the *principle of persistence*, which states that the behavior of a parallel application *tends* to persist over time [9]. Since the workload is dynamic, but tends to persist, making Periodical Load Balancing (LB) decisions to remap work from overloaded to underloaded machines is possible, and crucial to maximize application performance. However, these periodical scheduling steps incur in undesired overhead, so employed strategies must be fast and scalable to reduce application makespan.

Usually, dynamic LBs for iterative parallel applications are classified as *global* or *diffusive*. *Global schedulers* attempt to balance the application from a centralized point, or in a hierarchical fashion [10]. These strategies tackle the problem with approaches that include list scheduling [11], graph partitioning algorithms [12], [13], and topology-aware algorithms [14], [15], [16], among others. *Diffusive schedulers* execute from multiple decentralized points, taking work away from overloaded machines [17], [18], [19]. Completely decentralized scheduling algorithms are explored in different scenarios, such as Work Stealing (WS) [20], [21] and selfish load balancing [22], [23]. Still, even though the diffusive approach tends to portray better scalability, there are not many LBs using this approach [24], leaving much to be explored in order to handle load imbalance on the ever-growing HPC platforms.

In this work, we propose efficient, decentralized load balancing algorithms based on the discretization of application workload. The technique, called *Packing*, extends a technique from a previous work [19] with notions related to ϵ-Nash Equilibrium. We present two asynchronous and decentralized LB algorithms: a reintroduction of *PackDrop* [19], and the new *PackSteal* based on constrained [25], [26] and randomized [20] WS heuristics. Our experimental evaluation shows the benefits of *packing*-based decentralized LB algorithms, namely faster LB execution time, reduced number of task migrations, preservation of the original locality between tasks, and reduced application execution times. Overall, our main contributions in this work are:

1) A method for workload discretization named *Pack-*

- *V. Freitas, A. Santana and M. Castro are with PPGCC, Federal University of Santa Catarina (UFSC), Florianópolis, Brazil. Contact: marcio.castro@ufsc.br*
- *V. Freitas, L. L. Pilla and J. Cohen are with LRI, Univ. Paris-Sud – CNRS, Orsay, France. Contact: pilla@lri.fr*

*Manuscript received ***** **, 20XX; revised ***** **, 2020.*

ing that extends previous work [19];
2) A reintroduction of the *PackDrop* [19] algorithm in this new context;
3) A novel asynchronous, distributed, WS-based LB algorithm called *PackSteal*;
4) The implementation of *PackSteal* and *PackDrop* in `Charm++`, a state-of-the-art runtime system that supports load balancing in distributed memory scenarios, and an experimental evaluation comparing them to other state-of-the-art algorithms.

The remainder of this work is divided as follows. Section 2 discusses related work in periodical LB and WS. Section 3 presents the distributed model considered in this work and its notations. Section 4 presents our *packing* model and load balancing algorithms. Section 5 discusses the implementation details of these LB algorithms in `Charm++`. Section 6 presents our performance evaluation and results. Finally, Section 7 concludes this paper.

2 Related Work

State-of-the-art WS and LB techniques differ in both behavior and applicability when used to develop novel decentralized load balancing algorithms. WS heuristics are regarded as distributed load balancing mechanisms for *task-parallel* applications, usually issuing *steals* when a parallel machine goes idle or is in the imminence of going idle. WS schedulers may also be classified as distributed *receiver-initiated* (or *pull-based*) load balancing schemes, where underloaded machines request load from overloaded ones. LBs, on the other hand, are invoked periodically to remap work in the system. Usually implemented as centralized or distributed *sender-initiated* (or *push-based*) schemes, periodical LBs have different applications in mind, such as Bulk Synchronous Parallel (BSP) [27], iterative, and other *data-parallel* applications. This is especially efficient when using persistence-based techniques, which allows the scheduler to more accurately predict the load of the tasks it is moving [28].

Both LB and WS approaches have their advantages. Notably, WS is often applied in task-parallel Runtime Systems (RTSs) [29], and shared memory scenarios [3], [4] (even though it is distributed in nature), although it has been used for highly unpredictable applications in distributed memory as well [30]. LBs, on the other hand, have been widely applied to both shared and distributed memory scenarios, but due to their periodic nature, they are often applied to applications that follow the principle of persistence [9] (i.e., applications with dynamic workload that tends to change slowly over time). Although they have been traditionally implemented as centralized schedulers, LB performance points towards distributed approaches [18], [19], and with limited strategies proposed in this field [24], WS emerges as a source of inspiration for novel LB policies. This is specially interesting considering the convergence properties of WS in large-scale scenarios [31]. In this section, we discuss recent work in periodical LBs and WS, providing a solid base for our novel approach.

2.1 Periodical Load Balancing

We divide LBs into two categories based on their behavior: *Global* and *Diffusive* [10]. The *Global* approach uses centralized or hierarchical schedulers in order to balance load. This approach aggregates relevant system information, allowing precise understanding of the application state [12] with a potentially high overhead, especially in large scale overdecomposed applications. Hierarchical algorithms usually try to divide the underlying system in different locality levels, and schedule tasks on each level with a different strategy to reduce overheads [32].

Other relevant hierarchical algorithms are those based on graph and hypergraph partitioning. `Zoltan` [12] uses multilevel partitioning to parallelize and schedule work. This approach can also consider past work to reschedule tasks, reducing the decision time. Meanwhile, `Scotch` [33] and `Metis` [13], [34] use classical graph partitioning techniques such as *Dual Recursive Bipartitioning* and *k-way* coarsening to map tasks. In a similar fashion, `Weighted-Hop` and `Max-Congestion` [15] use topology information in order to enhance other classical graph partitioning algorithms when scheduling tasks.

The *Diffusive* approach, on the other hand, follows the classic *greedy* algorithms principle: *optimize locally to optimize globally*. These strategies try to solve the imbalance issue from a much narrower scope by using local information only [17]. The *Distributed* (or *Grapevine*) [18] scheduler uses probabilistic transfer of load and high levels of parallelism to achieve a balanced state of the system, scaling much better than global load balancing strategies. Although these strategies have shown high scalability, they are still scarce in the state of the art [24].

Locality-awareness is an important factor to achieve performance in large-scale applications [35]. In this sense, strategies that take system topology into account are a promising trend in global scheduling to preserve locality [36]. However, most of these are still topology-specific [14], [16]. For instance, *NuCo* and *HwTopo* [14], [37] are load balancers for Non-Uniform Memory Access (NUMA) machines, which model distance between source and destination tasks in order to determine whether it is worth migrating them among NUMA nodes. *TreeMatch* [16] has hierarchical and distributed versions, and was originally designed for *fat-tree* machine architectures. As it can be observed, none of these algorithms are able to achieve locality-awareness in a platform-agnostic fashion.

On a different approach, *PackDrop* [19] attempts to preserve the affinity among tasks by grouping them in resources prior to migration but completely disregards the network topology. This kind of workload grouping has been used in shared-memory scenarios as well with *BinLPT* in *OpenMP* loop scheduling [38], or in job scheduling with *packing-based placement* to reduce fragmentation on 3D-Torus HPC systems [39].

2.2 Work Stealing Schedulers

WS schedulers are inherently distributed. In WS, independent scheduling agents manage resources in a parallel system, and may take roles of: (i) *thieves*, which attempt to dynamically remap work to underloaded (or idle) resources, trying to manage workload so tasks are constantly available to be computed; or (ii) *victims*, which are targets chosen by thieves to have their tasks *stolen*. WS schedulers are

commonly applied to dynamic and imbalanced applications [20], [21] that cannot afford a stable work decomposition, but may be applied to any parallel application decomposable as a Direct Acyclic Graph (DAG). This way, applications decomposed in models like *fork/join* [40], general task parallelism, and parallel loops in shared memory [41] have also benefited from WS.

Following the topology-aware approach, WS strategies have been able to greatly increase application performance. The Feudal Work Stealing [25] approach shares system information as tasks are stolen and attempts to select victims both in local and remote work groups, which increases task locality (e.g., a scheduler that manages a subgroup of cores may attempt to migrate work within its own subgroup of cores or from remote cores to it). CLAWS [26] is a contention- and locality-aware work stealing runtime for NUMA architectures, which takes care of task migration, reducing remote memory accesses. In a similar fashion, ADWS [42] uses localized hierarchical stealing to compensate imbalance in task-parallel applications.

On a different approach, DistWS [3] uses application task affinity instead of topology-awareness by selecting tasks that are more favorable for migration (i.e., have less data to be copied) on steal attempts in distributed shared-memory machines. Retentive Work Stealing [28] tries to apply the benefits of Persistence-based Load Balancing into a WS model in a distributed memory MPI environment. This approach uses a persistence model in iterative applications, in which, instead of rescheduling all of the workload every iteration, resources keep a list of processed tasks that is used as a seed for the next iteration, improving the balance as the application is executed, and performing work stealing when appropriate.

2.3 Discussion

As distributed schedulers rise as solutions in the periodical load balancing domain, the use of WS heuristics is attractive due to their well documented past and known convergence times [31]. However, distributed strategies must be adapted to achieve harmony with balancing load, locality, and quickly computing a new mapping. We aimed to achieve this with our *packing* scheme with *PackDrop*, and we believe strategies such as Feudal WS [25] and Randomized WS have much to give in the periodical scheduling scenario (if correctly adapted). In the next sections, we present how we aim to achieve this harmony by first explaining our distributed scheduling model, and then explaining the *packing* model, *PackDrop*, and the new algorithm *PackSteal*.

3 Scheduling Model and Notation

A parallel application may be described as a set $\mathcal{T}$ of n tasks : $\mathcal{T} = \bigcup_{i=1}^{n}\{T_i\}$. The load of a task T_i is denoted by $\omega(T_i)$. For the sake of simplicity, we extend the notion of load to sets of tasks. We also assume that the load of an empty set is equal to 0; both are shown in Equation (1).

Let T be an arbitrary set of tasks,

$$\omega(T) = \sum_{T_i \in T} \omega(T_i), \qquad \omega(\varnothing) = 0 \quad (1)$$

We also consider a set $\mathcal{M}$ of m identical machines : $\mathcal{M} = \bigcup_{j=1}^{m}\{M_j\}$, and $n \gg m$, resembling overdecomposed parallel applications. Additionally, for each machine M_j, a subset of tasks $S_j \subset \mathcal{T}$ is assigned to M_j.

Each machine M_j has a unique scheduling agent j that makes load balancing decisions. Its local view is composed of: (i) a set of tasks S_j assigned to it; and (ii) an indexed communication table of $\mathcal{M}$, containing all machine identifiers. The remaining information used in decision making has to be entirely derived from this local view.

The objective of load balancing is to minimize application makespan. The best way to do this is to distribute load evenly across machines. Equation (2) describes the load of a machine M_j and the lower bound of the makespan $\omega^\star$.

$$\omega(M_j) = \sum_{t \in S_j} \omega(t), \qquad \omega^\star = \frac{\omega(\mathcal{T})}{|\mathcal{M}|} \quad (2)$$

Since optimal scheduling of parallel machines is NP-Hard, achieving a process mapping that yields $\omega^\star$ to every core is rather unrealistic. So, we give this value a relaxation ϵ, which should be based on the imbalance characteristics of each application.

Definition 1. [*ϵ-relaxed order $\lesssim$*]. *Given any two machines M_p, M_q, their relaxed difference in load is as stated in Equation* (3)*.*

$$\omega(M_p) \lesssim \omega(M_q) \iff \omega(M_p) > \omega(M_q) + \epsilon \quad (3)$$

From Equation (3)*, the definitions of $\lesssim$, $\eqsim$ and $\gtrsim$ follow trivially.*

We want to assign tasks to machines in a distributed fashion, so that the makespan approximates $\omega^\star$. This follows the game-theoretic idea of achieving an ϵ-Nash Equilibrium [22], a concept widely applied to distributed algorithms. Achieving ϵ-Nash Equilibrium means that no agent in the system (in this case, our schedulers) profit from taking actions that modify the state of the system. Since optimization turns into an exhaustive process once the answer approaches an optimal result, we define the ϵ relaxation factor, converging faster to an equilibrium.

4 Workload Discretization and Load Balancers

Our algorithms are designed for persistence-based load balancing in asynchronous runtime systems. The application is paused during rescheduling time, meaning that LB time is considered an overhead for the application. Thus, any load balancing strategy in this situation must run quickly to actually diminish application makespan.

We split our load balancers into two different algorithms named *PackDrop* and *PackSteal*, following sender- and receiver-initiated protocols, respectively. These algorithms are executed by each local scheduling agent, following a simple and standardized notation for distributed algorithms [43]. Each of our algorithms executes from beginning to end with no interruptions. New messages execute in receiving order as their predecessors are processed. Every communication is asynchronous, so messages expect no answers.

Overall, the algorithms have two main phases: (i) workload discretization; and (ii) load balancing decision. The former is common to both algorithms and is explained in Section 4.1, whereas the latter will be explained in Section 4.2 (*PackDrop*) and Section 4.3 (*PackSteal*).

4.1 Application Workload Discretization (*Packing*)

Load balancing scenarios are often described as either *discrete* or *continuous* [23]. The discrete case describes uniform non-preemptable[1] tasks, while the continuous case describes non-uniform preemptable tasks. Our case of scheduling in parallel machines presents overdecomposed applications with non-uniform non-preemptable tasks.

Our *packing* model aims to improve the scheduling process by approximating our load balancing scenario to the *discrete* scenario. The main objectives of *packing* are: (i) to reduce the scheduling time by making it simpler to decide if tasks will be migrated or not (as the groups of tasks have all the same approximate load), and by reducing the number of messages exchanged between agents (as multiple tasks are suggested for migration in the same message); (ii) to preserve some of the original locality of the application (assuming that the original mapping already grouped together communicating tasks in the same machine).

In our model, scheduling agent j takes a subset of tasks $\mathcal{P}_j$ in S_j that contains tasks so that $\omega(M_j) - \omega(\mathcal{P}_j) \triangleq \omega^\star$. This subset $\mathcal{P}_j$ corresponds to the set of tasks the agent j intends to migrate. Moreover, $\mathcal{P}_j$ is subdivided into disjoint sets of tasks called *packs*, Γ, with total load ω^Γ each. Since we want to make all migrating workload discrete, and our tasks are non-preemptable, agents must aggregate them into packs that respect a γ-*relaxed equality* among themselves, following Equation (4).

$$\forall \Gamma_x, \Gamma_y \subset \mathcal{P} \mid \omega(\Gamma_x) \triangleq \omega(\Gamma_y) \triangleq \omega^\Gamma \tag{4}$$

Equation (4) also means that the load of all packs, regardless of machine, are similar. We denote this load as ω^Γ. As our focus lies on the discretization of the overloading tasks (i.e., those that make a machine overloaded), we use a greedy approximation algorithm to solve the bin packing problem of assigning tasks to *packs* for further migration.

4.2 PackDrop: A Sender-initiated Load Balancer

PackDrop is a sender-initiated, informed strategy that attempts to migrate packs from overloaded to underloaded machines. Algorithm 1 presents *PackDrop*, which is divided in two steps: (i) information propagation (lines 2–9 in `INIT`); and (ii) pack exchanging (`INIT` after the synchronization barrier in line 10, `TASKS` and `ACK`).

In the first step (i), the average load in the system $\omega^\star$ is first reduced. At this point, agents know if they have to take an overloaded, or underloaded role (line 5). While overloaded agents will start packing their tasks ($\mathcal{P}_j$), underloaded ones will start a Gossip Protocol [45] (line 8) to propagate machine load information, $\omega(S_j)$. This allows the overloaded machines to make educated decisions to send packs later on.

1. Brucker [44] defines preemptable tasks (or jobs), as tasks that may be divided any number of times.

Algorithm 1: *PackDrop*, perspective of M_j

```
   Input:
   S_j− set of local tasks.
   M− set of system machines.
1  when receive INIT() do
2      R_j ← ∅;   P_j ← ∅
3      reduce from M: ω(T)/m → ω*
4      calculate ω^Γ   // Equation 7 in Section 5
5      if ω(S_j) ≶ ω* then
6          generate packs: P_j   // See Section 4.1
7      else
8          gossip information: ω(S_j)
9      end
10     —— Wait Barrier ——
11     ⌊M⌋ ← {⌊M⌋ ⊆ M | ∀M_i ∈ ⌊M⌋, M_i ≲ ω*}
12     while P_j ≠ ∅ do
13         Γ ← p | p ∈ P_j;        P_j ← P_j \ {Γ}
14         send TASKS(Γ) to M_i ∼ ⌊M⌋
15     end
16 end
17 when receive TASKS(Γ) from M_s do
18     if ω(S_j) + ω^Γ ≲ ω* + ε then
19         R_j ← R_j ∪ {Γ}
20         send ACK(true, Γ) to M_s
21     else
22         send ACK(false, Γ) to M_s
23     end
24 end
25 when receive ACK(bool, Γ) from M_s do
26     if bool=true then
27         Register the migration of T ∈ Γ to M_s
28     else
29         send TASKS(Γ) to M_i ∼ ⌊M⌋
30     end
31 end
```

In the second step (ii), overloaded machines will create a list of underloaded machines $\lfloor \mathcal{M} \rfloor$ (line 11), from which they will pick targets for their packs uniformly at random (operator $\sim$, in line 14). Underloaded machines will only accept packs that will not make them overloaded (line 18). Rejected packs will be will be sent again by the overloaded machine to a new random target (line 29).

Sending a rejected pack again is only attempted for a fixed number of times. Since the algorithm is asynchronous, we cannot measure convergence during load balance, which means the ending must be local as well. Hence, a fixed stop criteria is defined to avoid livelocks.

4.3 PackSteal: A Receiver-initiated Load Balancer

PackSteal is a new receiver-initiated LB that is motivated by feudal and randomized WS [25]. The feudal aspect comes in its information propagation, while the randomized aspect in its *victim selection*.

PackSteal progressively gathers the state of the system (the load of each machine $M_a \in \mathcal{M}$) at the same time it performs its decisions. This is possible due to the attachment of the sender local load information to the messages target-

Algorithm 2: *PackSteal*, perspective of M_j

Input:
$\mathcal{T}^j-$ set of local tasks.
$\mathcal{M}-$ local knowledge of the state of the machines in the system.

1 **when receive** `INIT()` **do**
2 $\quad\mathcal{R}_j \leftarrow \varnothing; \quad \mathcal{P}_j \leftarrow \varnothing$
3 $\quad$**reduce** from $\mathcal{M}$: $\frac{\omega(\mathcal{T})}{m} \rightarrow \omega^\star$
4 $\quad$**calculate** ω^Γ and s_j `// Equation 7 and 5`
5 $\quad$**if** $\omega(S_j) \gtrdot \omega^\star$ **then** `// Victim case`
6 $\quad\quad$generate packs: $\mathcal{P}_j$ `// See Section 4.1`
7 $\quad\quad$**send** `HINT()` to $\arg\min_{a \in \mathcal{M}} \omega(M_a)$
8 $\quad$**else if** $\omega(S_j) \lessdot \omega^\star$ **then** `// Thief case`
9 $\quad\quad$**for** s_j **do** `// See Equation 6`
10 $\quad\quad\quad$**send** `STEAL(0)` to $a \sim top^k(\mathcal{M})$
11 $\quad\quad$**end**
12 $\quad$**end**
13 **end**
14 **when receive** `HINT()` *from* M_b**do**
15 $\quad$**if** $\omega(S_j) \gtrdot \omega^\star$ **then** `// Victim case`
16 $\quad\quad$**send** `HINT()` to $\arg\min_{M_a \in \mathcal{M}} \omega(M_a)$
17 $\quad$**end**
18 **end**
19 **when receive** `STEAL(`n_b`)` *from* M_b**do**
20 $\quad n_b \leftarrow n_b + 1$
21 $\quad$**if** $\omega(S_j) \gtrdot \omega^\star$ **then** `// Victim case`
22 $\quad\quad\Gamma \leftarrow \arg\min_{\Gamma \in \mathcal{P}_j} \omega(\Gamma); \quad \mathcal{P}_j \leftarrow \mathcal{P}_j \setminus \{\Gamma\}$
23 $\quad\quad$**send** `TASKS(`Γ`)` to M_b
24 $\quad\quad$Register the migration of $T \in \Gamma$ to M_b
25 $\quad$**else if** $n_b > \frac{m}{4}$ **then**
26 $\quad\quad$**forward** `STEAL(`n_b`)` to $M_a \sim \mathcal{M}$
27 $\quad$**else**
28 $\quad\quad$**forward** `STEAL(`n_b`)` to $M_a \sim top^k(\mathcal{M})$
29 $\quad$**end**
30 **end**
31 **when receive** `TASKS(`Γ`)` *from* M_b**do**
32 $\quad\mathcal{R}_j \leftarrow \mathcal{R}_j \cup \Gamma$
33 **end**

ing its peers. Now, we describe how to broadcast system information during message exchanging.

Consider that when the algorithm begins, the agent j is not aware of the current load of the other machines. However, each time j *sends* a message, it may also include its load information $\omega(M_j)$ with the message. If j has already attained information on the load of any of its peers, this information may also be passed with every message it sends in a *piggybacking* fashion.

PackSteal is described in Algorithm 2, which is split into two main parts. The first part (`INIT` on lines 1–13) describes the initial calculations and role determination (*victim* or *thief*). The initial flow begins by issuing a reduction in order to assess the average system load (line 3). Then, it calculates number of steals s_j for thief j using Equation (5).

$$s_j = \left\lfloor \frac{\omega^\star - \omega(S_j)}{\omega^\Gamma} \right\rfloor \tag{5}$$

Pack load ω^Γ is defined by Equation (7) in Section 5. After that, schedulers take roles of either *thieves* or *victims* depending of their load (stated as `Thief case` and `Victim case`). Observe that only thief agents have $s_j > 1$.

After determining their role, *victims* will assemble their packs and send `HINT` messages in order to warn a potential *thief*. This *thief* is determined by choosing one machine in $\mathcal{M}$ with the lowest known load (line 7). Of course, limited to the local information of $\mathcal{M}$ (i.e., machines of $\mathcal{M}$ that have not yet communicated with j will not have an ω, and thus, will not be accounted for). When an agent j receives a `HINT` (lines 14–18), it stores relevant information about its peers. Additionally, if j is a *victim*, it will send a new `HINT` to its known most probable *thief*. This informs agents of each others' states incrementally, which assists future stealing attempts.

Thief j will attempt s_j `STEAL`s to target machines a (line 10), where a is a possible victim for j. At the beginning, it starts to send a message `STEAL` containing the number of attempts n_b previously done (starts at 0). At each time a thief receives this message, n_b is increased by 1 and the `STEAL` message is forwarded. The n_b information is used to determine if the victim selection will be (i) *constrained* or (ii) *randomized*.

The standard choice is the constrained selection (i), which picks uniformly at random one of the k most loaded machines in $\mathcal{M}$, as determined in Equation (6).

$$\begin{aligned} &top^k(\mathcal{M}) \text{ is a subset of } \mathcal{M} \text{ where } |top^k(\mathcal{M})| = k, \text{and} \\ &\forall M_i \in \mathcal{M} \setminus \mathcal{M}^k, M_j \in \mathcal{M}^k, \omega(M_i) \leq \omega(M_j)\} \end{aligned} \tag{6}$$

Once n_b surpasses $\frac{m}{4}$, meaning the constrained selection is not working very well, j will then use randomized victim selection (ii), simply choosing a possible machine index uniformly at random (line 25).

When receiving a `STEAL` message, a *victim* j will send the load contained in the first element of their list of packs, $\mathcal{P}_j$ (lines 21–23). Additionally, j must register that the tasks in Γ will migrate to M_b (line 24), as the runtime system or the scheduler must perform these migrations. Meanwhile, if the receiving agent is not a *victim*, it will *forward* the steal to another possible victim. Lines 26 and 28 portray the two distinct aforementioned *victim selection* behaviors, randomized and informed selection, respectively.

Finally, once j receives a `TASKS` message, meaning j is a thief and it has received new tasks, j will add the received pack to its received tasks list ($\mathcal{R}_j$), which is used to update the current load of j. Although *PackSteal* does not require a barrier to separate different steps of the algorithm, it needs one to coordinate the end of execution and assess global information. The ending is performed via *quiescence detection* [46], meaning that when no scheduler has messages in its queue, they have been synchronized.

5 Implementation

PackDrop and *PackSteal* were implemented in `Charm++` [47] using its distributed load balancing framework[2,3]. `Charm++`

2. `Charm++` is available at: http://charm.cs.illinois.edu/software.
3. *Packing* schemes and LBs are available at: https://github.com/viniciusmctf/packing-schemes/tree/packs_2019-v1.

is one of the most receptive runtime systems for new LB strategies, especially in distributed memory systems [48], which allows us to pair it up with applications already existing in the environment. The use of `Charm++`'s load balancing framework commonly requires pausing the application while schedulers compute and remap work.

In `Charm++`, the workload is decomposed in independent and migratable virtual processors named *chares*, usually following a geometric decomposition scheme. `Charm++` is a message-driven, asynchronous RTS, meaning that work is issued when *chares* (our *tasks*) receive messages. Load balancers are also implemented as *chares*, meaning that they benefit from `Charm++`'s native synchronization mechanisms to perform the reduction operation and the quiescence detection (`CkReduction` and `CkStartQD`).

In Section 4, we used some variables whose values must be set beforehand. Ideally, ϵ is the best value for *packing* load, since it will mitigate more of the algorithm complexity. However, the larger the packs, the larger is their potential gap between $\omega(\Gamma_x)$ and the ideal ω^Γ. So, smaller packs tend to be tighter, which leads to higher quality in load balancing.

We propose to calculate ϵ as a fraction of $\omega^\star$, using a ξ factor, such that the maximum overall imbalance will be at most $\xi\%$. Additionally, we use a δ coarsening factor to make the pack size ω^Γ smaller, as described in Equation (7).

$$\epsilon = \omega^\star \times \xi, \qquad \omega^\Gamma = \epsilon \times \delta, \qquad \gamma = \omega^\Gamma \times \xi \tag{7}$$

Also detailed in Equation (7) is the γ relaxation factor, which is used to define pack size in Section 4.1, Equation (4).

The *imbalance tolerance* ξ value was determined as 0.05, meaning that we only consider that a given core is balanced when its load is in an interval of 5% to $\omega^\star$, which is plausible in this scenario and is used by other schedulers in `Charm++`. Meanwhile, the *pack narrowing factor* δ is fixed to 0.4 as a middle ground between optimizing the balance and accelerating the algorithm. ξ and δ are defined in Equation (8).

$$\xi \leftarrow 0.05, \qquad \delta \leftarrow 0.4 \tag{8}$$

The seeded neighborhood of *PackSteal* (initial known $\mathcal{M}$ in Algorithm 2) of a given scheduling agent j was predetermined as being its *right-hand neighbor*, which is given by $\{M_r\}$ for $r = (j+1) mod\ m$. Since `Charm++` generally numbers its resources in ascending order, all cores in a parallel machine will be numbered before starting to number the cores in the next machine, so there is a good chance (higher if nodes are tightly coupled) this is a nearby resource. In homogeneous clusters, this value can also be used as a topology-aware hint. For instance, if each cluster node has c cores, the seed may be $\{M_r\}$ for $r = c \times \lfloor \frac{j}{c} \rfloor + (j+1)\ mod\ c$, which is an in-node neighbor of j. This way, we would first attempt local neighbors in $\{M_r\}$ during the constrained WS phase, and global machines in the randomized one.

6 Performance Evaluation

We performed an experimental evaluation of our workload discretization scheme and its distributed LBs in two different moments. At first, we conducted extensive experiments to understand how the algorithms behave when acting on applications with different numbers of tasks, communication patterns, and different rescheduling frequencies, in a smaller but more available machine (Section 6.2). Then, we moved to experiments to evaluate the scalability of the algorithms when handling a molecular dynamics benchmark on a supercomputer (Section 6.3). In both cases, we compared *PackDrop* and *PackSteal* to LBs from the state of the art (listed in Table 1), and with a *Baseline* execution using a *Dummy* LB that performs no actual load balancing. For each evaluation, we provide first an explanation of the experimental design followed by a discussion on the results.

TABLE 1
Brief description of other LBs used in the experiments.

LB	Type	Short description
Distributed	Diffusive	Push-based strategy that uses probabilistic transfer of load to choose task receivers [18]. Gathers system information with a *gossip* protocol.
Refine	Global	Attempts to minimize the number of migrations. Migrates tasks from overloaded to underloaded resources only [28].
Greedy	Global	Optimizes load distribution, not communication. Assigns tasks to cores using a Longest Processing Time (LPT) first policy [11].

6.1 Experimental Environment and Methodology

Our methodology involves the evaluation of:

1) *Application time (makespan):* how long an application takes to execute;
2) *Load balancing time:* the time between invoking the LB and resuming the application after migrations;
3) *Total number of migrations:* the number of tasks migrated on each LB invocation.

Minimizing the application execution time is the most important objective, which is the factor that enables the execution of high scale scientific applications. Nonetheless, a low LB overhead (coming from the load balancing time and migrations) is important as it allows LBs to scale with applications as systems grow larger.

We selected the synthetic benchmark *LB Test* for the first set of experiments, and the molecular dynamics benchmark *LeanMD* for the second set. *LB Test* allows the variation of generic parameters (like communication pattern), while *LeanMD* is based on (and performs core computations of) the Gordon Bell award-winning application NAMD [49], providing a scenario to measure the impact of novel LBs. All benchmarks were compiled with the `-O3` flag.

We carried out the experiments on two different platforms. The first is *Tesla*, a NUMA machine with simultaneous multi-threading composed of two 10-core Intel Xeon E5-2640@2.4GHz CPUs. It has a total of 128GB ECC RAM memory @1333MHz, configured with Ubuntu 16.04, `GCC` 5.4.0 and `Charm++` 6.8.1 `multicore-linux64 --with-production`. This machine was locally available for our experiments, allowing us to explore mul-

TABLE 2
LB Test average execution times (in seconds) with 12k tasks and variations on communication pattern and load balancing frequency on *Tesla*. Values in bold represent the best average execution time for a given communication pattern.

Communication	Ring			2D Mesh			3D Mesh			Random Graph		
LB Frequency	**40**	**70**	**100**	**40**	**70**	**100**	**40**	**70**	**100**	**40**	**70**	**100**
Baseline	77.879	67.717	59.966	80.585	71.999	63.176	83.196	71.415	63.131	183.054	166.423	156.163
PackDrop	76.822	69.894	65.862	76.541	70.763	69.457	78.244	75.219	69.362	175.227	166.549	161.401
PackSteal	43.711	**42.405**	44.387	47.773	46.472	**45.617**	45.406	50.947	45.063	135.081	**132.808**	135.815
Distributed	72.650	63.646	65.384	69.404	65.596	70.989	60.575	64.581	72.590	171.974	165.192	159.599
Greedy	47.687	47.219	54.662	63.582	50.654	57.758	47.158	**38.748**	41.400	199.353	184.817	175.331
Refine	74.532	70.508	66.323	77.984	71.378	69.483	78.471	71.297	68.748	176.897	165.850	153.634

TABLE 3
LB Test average execution times (in seconds) with 18k tasks and variations on communication pattern and load balancing frequency on *Tesla*. Values in bold represent the best average execution time for a given communication pattern.

Communication	Ring			2D Mesh			3D Mesh			Random Graph		
LB Frequency	**40**	**70**	**100**	**40**	**70**	**100**	**40**	**70**	**100**	**40**	**70**	**100**
Baseline	117.316	104.095	90.035	125.677	105.825	98.733	127.895	111.600	97.259	362.582	330.883	313.609
PackDrop	113.691	105.736	96.996	110.315	110.551	105.694	118.081	116.345	109.178	332.470	325.321	316.252
PackSteal	63.872	65.569	68.252	71.264	74.540	71.733	72.898	77.679	**76.891**	272.080	274.124	**268.365**
Distributed	77.000	93.719	104.811	78.108	96.732	107.007	89.138	99.709	111.270	332.413	326.763	317.951
Greedy	58.805	58.039	**57.618**	62.670	65.609	**60.758**	110.191	92.025	91.020	402.275	370.815	352.992
Refine	114.581	108.585	99.792	116.192	113.332	105.088	121.097	114.039	109.467	348.633	327.295	315.318

tiple execution scenarios. The second is the supercomputer *Joliot-Curie*[4]. It contains NUMA compute nodes interconnected with EDR Infiniband. Each node features two 24-core Intel Xeon 8168@2.7GHz CPUs and 192GB ECC RAM DDR4 memory @2666MHz. *Joliot-Curie* runs on Red Hat Enterprise Linux 7.6, loading OpenMPI 2.0.4, and `C/C++` Intel 17.0.6.256 modules, and using `Charm++` 6.9.0 `mpi-linux-x86_64 --with-production`.

6.2 Evaluation with *LB Test* on *Tesla*

6.2.1 Experimental Design

The variables considered in *LB Test* were total number of tasks ($w \in \{12k, 18k, 24k\}$), communication pattern ($c \in \{Ring, 2D\ Mesh, 3D\ Mesh, Random\ Graph\}$), and load balancing frequency (i.e., how many iterations between each time an LB is called) ($r \in \{40, 70, 100\}$). All configurations of *LB Test* were executed for 300 iterations, with a sample size of 20 executions per configuration.

The communication patterns express four different ways in which tasks communicate with each other in *LB Test*. The *Ring* pattern expresses a one dimensional torus network of size n, so task T_i requests a computation to $T_{(i+1) mod\ n}$, and executes the request of $T_{(i-1) mod\ n}$. Two- and three-dimensional meshes work in a similar fashion, adding a dimension (and thus, a neighbor), for each. This way, *Ring*, *2D Mesh* and *3D Mesh* patterns model tasks with 1, 2, and 3 outgoing communication edges, respectively. The *Random Graph* pattern is more intense in the neighbor creation process, connecting 1% of all possible communication edges, generating a much more work- and communication-intensive scenario.

4. Detailed specifications of SKL Irene available at: http://www-hpc.cea.fr/en/complexe/tgcc-JoliotCurie.htm

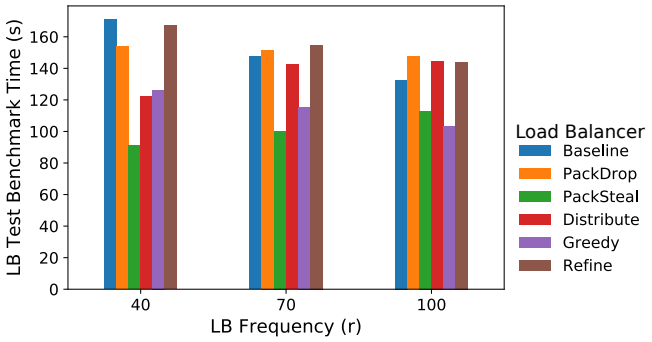


Fig. 1. Execution time impact of different LB frequencies for *LB Test* with 24k tasks on *Tesla*

We discuss experiments in *Tesla* with the 12k and 18k tasks configurations for all communication patterns, and 24k for the *3D Mesh* pattern only[5]. For each combination of workload w and communication pattern c, we tested each of the load balancing frequencies in order to measure the impact of synchronization and task migration in the application total execution time.

We have different objectives for the variation of each parameter. Workload size w is used to evaluate the LB capability of dealing with varying input sizes. Meanwhile, the communication pattern c is used to evaluate the impact of migrations and how LBs perform with different simulated workload profiles. Finally, the load balancing frequency r serves to help us find the best frequency for the different configurations and LBs, as some scenarios may benefit more from early and often remapping tasks with a low overhead, while others may make perform better when less LB calls are made.

5. Other parameter combinations for *LB Test* follow similar trends.

TABLE 4
LB Test average cumulative LB times (in milliseconds) with 12k and 18k tasks and variations on communication pattern for a load balancing frequency $r = 100$ on *Tesla*.

Simulation Size	12k tasks				18k tasks			
Communication	Ring	2D Mesh	3D Mesh	R. Graph	Ring	2D Mesh	3D Mesh	R. Graph
PackDrop	9.789	8.994	9.403	272.400	12.705	12.500	13.516	449.261
PackSteal	11.629	8.903	10.378	273.948	13.894	14.390	14.206	444.419
Distributed	20.189	19.399	22.802	276.462	33.575	26.903	28.367	455.171
Refine	46.0047	50.290	49.374	31.964	93.065	99.473	107.136	66.235
Greedy	386.191	385.690	385.171	383.413	606.654	606.569	604.582	601.671

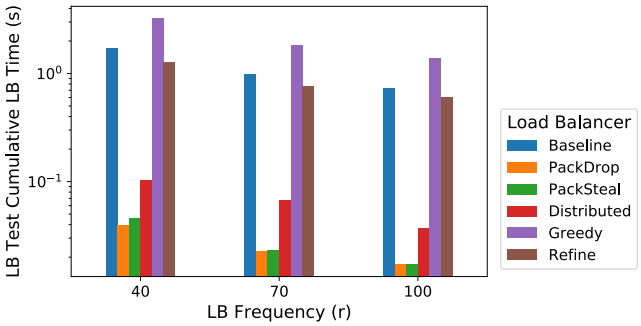


Fig. 2. Cumulative LB time impact of different LB frequencies for *LB Test* with 24k tasks on *Tesla*

6.2.2 Results Discussion

Results are portrayed in Tables 2 and 3, and in Figs. 1 and 2. Overall, *PackSteal* outperformed every other LB, with the exception of *3D Mesh* in the 12k-task scenario, and *Ring* and *2D Mesh* in 18k and 24k ones. In *Random Graph*, the most work- and communication-intensive pattern, *PackSteal* was the overall best choice for load balancing, signalizing the efficiency of its pattern-agnostic approach.

Moreover, the second best performing algorithm was *Greedy*, a centralized strategy that disregards migration and communication costs and scales poorly into larger systems. This can be verified in cumulative load balancing time (Fig. 2), as well as on the diminishing returns of issuing multiple LB calls, indicating that *Greedy* works better when called less times. *Greedy* is also known for performing numerous unnecessary migrations. Since tasks in *LB Test* are very light, migrating a lot of tasks does not incur in high overheads. However, *Greedy* is often not the best LB choice in scenarios with heavy tasks. This will be further discussed in Section 6.3.

We also observe that *Distributed* was able to outperform *PackDrop* in this synthetic scenario. Since they are both push-based strategies, we believe this is due to precision in the scheduling decisions and to their target choosing process. *Distributed* sets higher probabilities of choosing targets that are less loaded, while *PackDrop* chooses its targets uniformly at random.

In Fig. 2, we also observe that diffusive strategies display much higher performance than centralized ones. *PackSteal* is often more efficient when called multiple times ($r = 70$, compared to the $r = 100$ in the other two strategies), which leads to higher cumulative LB time. However, the improved workload distribution compensates the higher frequency for some LBs, which was the case for *PackSteal*, delivering lower application times with frequency $r = 70$.

Additionally, Table 4 shows the accumulated load balancing time of our evaluated LBs used in this experiment when executed every 100 iterations. Values show the average of the cumulative LB time per application execution with frequency $r = 100$. These results show the scalability potential of diffusive strategies, and further justifies the load discretization as it leads *PackDrop* and *PackSteal* to make their decisions consistently faster than *Distributed*. Moreover, the larger input size example, *Random Graph*, shows *Refine* as providing the faster decision time; however, checking the actual application performance when using the LB, we see that *PackSteal* is, truly, the only one able to improve the *LB Test* application performance in this case.

We further analyzed the LB frequency (r) variation *3D Mesh* scenario with 24k tasks in Fig. 2. While some strategies such as *Refine* or *Greedy* clearly show diminishing returns when called more often, *PackSteal* and *Distributed* portray significant benefits in performing this kind of execution tuning, speeding up *LB Test* by $1.46\times$ and $1.08\times$, respectively[6]. These results also show the high costs of synchronization in asynchronous applications such as the ones provided by `Charm++`. The baseline portrayed 29% performance degradation when called every 40 iterations (7 calls in total) when compared to every 100 (2 calls in total). Nonetheless, *PackSteal* still had the best performing result in this synthetic scenario.

6.3 Evaluation with *LeanMD* on *Joliot-Curie*

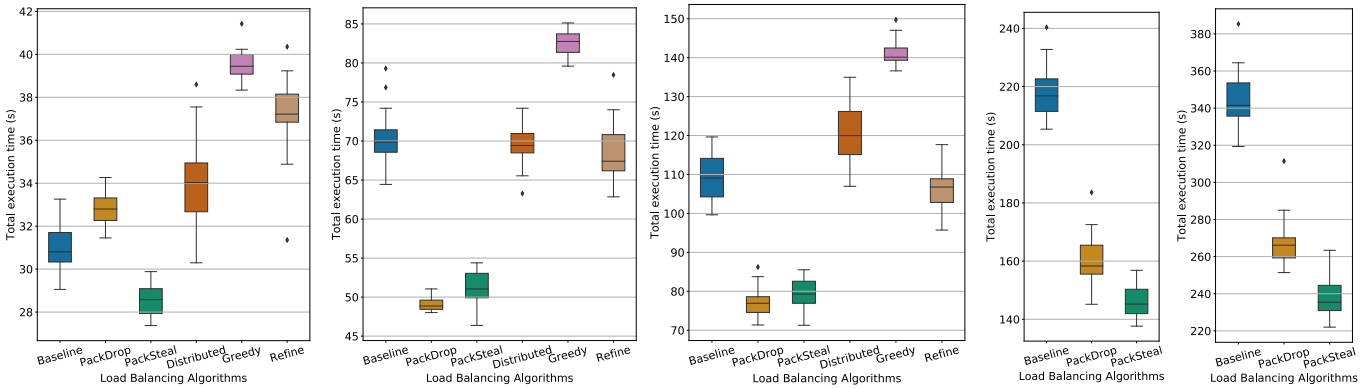
6.3.1 Experimental Design

The parallel solution implemented in *LeanMD* uses a 3D spatial decomposition approach, where the 3D space consisting of atoms is divided into cells. Our experiment used the standard *LeanMD* configurations available online[7], parameterized with $X \times 11 \times 5$ cells of dimensions $15 \times 15 \times 30$. Cells are further divided into *computes*, which are the actual *chares*. These contain multiple particles, and manage communication among them.

The parameter X was varied from sizes 80 to 320. Each execution of *LeanMD* comprises 301 iterations, executing the first load balancing call at the 40^{th} iteration and every 100 iterations after that, summing up a total of 3 LB calls. These parameter combinations generate simulations ranging from

6. Comparing the best and the worst performing averages of *LB Test* with the same LB strategy.

7. *LeanMD is available at: http://charmplusplus.org/miniApps/*



(a) X = 80. (b) X = 120. (c) X = 160. (d) X = 240. (e) X = 320.

Fig. 3. Boxplots of the execution times of *LeanMD* with different input sizes and load balancers on *Joliot-Curie*. Sizes indicate a variation of the application dimension parameter X. Each figure has its own vertical axis starting at its own value to emphasize performance differences.

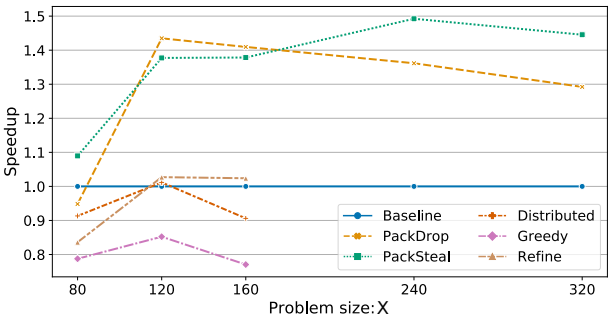


Fig. 4. LeanMD Speedups with different load balancers.

1.15 to 3.08 millions of atoms. These experiments were executed on 20 compute nodes of *Joliot-Curie* for a total of 960 cores.

Our experimental evaluation was carried out with 20 repetitions for each parameter combination (input size and load balancing algorithm). More specifically, as *Joliot-Curie* is a supercomputer with a job scheduler and multiple users at the same time, we organized our experiments in four jobs (two for $X = \{80, 120, 160\}$ and two for $X = \{240, 320\}$). Each job contains 10 repetitions for all LBs and input sizes involved. For each repetition, all pairs of input size and LB were executed in a random order with the objective of avoiding having noise from other users affecting a single LB.

The objective of executing *LeanMD* with these different input sizes follows the same line as *LB Test*, measuring the capabilities of LBs in dealing with varying input sizes but in a much larger scale. This benchmark creates n particles per cell following Equation (9).

$$n = 100 + \frac{cell\_id \times 150}{X \times Y \times Z}, \tag{9}$$

where $cell\_id$ ranges from 0 to $(X \times Y \times Z - 1)$. This way, as we scale any of the dimension parameters we allow the particles to be more spread in the simulation area. This leads to higher imbalance in the application as the input size increases.

6.3.2 Results Discussion

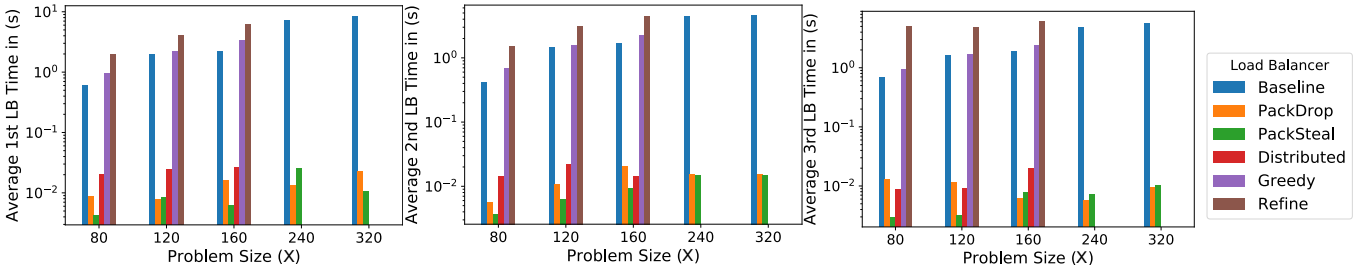
Fig. 3 show boxplots[8] representing the execution times of *LeanMD* with different input sizes and LBs. Only the *Baseline*, *PackDrop* and *PackSteal* were executed for input sizes 240 and 320 due to the increasing time it takes to run *LeanMD* for larger sizes and due to the performance results seen for the other LBs with smaller input sizes.

At a first glance, the results portrayed in Fig. 3 seem to indicate that *PackSteal* and *PackDrop* outperform the *Baseline* and other LBs, with the exception of the scenario with *PackDrop* and $X = 80$. Using a confidence threshold of 95%, we checked if their execution times were statistically different from the baseline. As all samples followed normal distributions (all Kolgomorov-Smirnov tests with p-values ¿ 0.05), we used parametric methods to compare *PackDrop* and *PackSteal* to the baseline. As all p-values < 0.05 using Welch's t-test, we can conclude that their execution times are actually different as first suspected. Using the same test, we also verify that no difference can be seen between *PackDrop* and *PackSteal* for $X = 160$ (p-value = 0.165), while *PackDrop* outperforms *PackSteal* for $X = 120$ and the opposite happens to other input sizes (all p-values ¡ 0.05). Finally, *PackDrop* also outperforms *Distributed* for $X = 80$ (p-value = 0.018), even though their execution times look similar.

Fig. 3 also indicates that our *packing*-based algorithms have a greater impact on *LeanMD*'s performance as its input size increases. We can observe the speedup achieved over the baseline with load balancing for the different input sizes in Fig. 4. *PackSteal* achieved speedups of 1.09, 1.38, 1.38, 1.47, and 1.41 for increasing input sizes, while *PackDrop* achieved speedups of 0.95, 1.44, 1.41, 1.32, and 1.29. Starting on $X = 120$, *PackSteal* and *PackDrop* lead to significantly better results than competing LBs, which emphasize their scalability and capability to handle load imbalance.

LB time in LeanMD was measured individually for each LB step. Results displayed in Fig. 5 show: (i) a great overhead on centralized synchronization, observed in the

8. Boxes extend from the 1st to 3rd quartiles of the samples. Lines represent the median values. Whiskers represent the data within 1.5 IQR from the lower or upper quartile. Points represent outliers.



(a) LB times (1st LB call). (b) LB times (2nd LB call). (c) LB times (3rd LB call).

Fig. 5. Average LB time in its multiple invocations. Data is displayed in log_{10} scale, otherwise the times of *PackDrop*, *PackSteal*, and *Distributed* cannot be seen in comparison to the others.

TABLE 5
Task migrations observed for each LB invocation in *LeanMD*, with problem size X = 160.

Load Balancer	Average # of Migrations		
	1st	2nd	3rd
PackDrop	8,942.9	9,405.1	9,516.6
PackSteal	2,704.2	827.5	618.3
Distributed	66,304.5	16,145.9	11,555.7
Greedy	35,163.5	35,162.8	35,162.5
Refine	713.9	535.7	698.4

baseline (which invokes `Charm++`'s *Dummy* LB); (ii) how efficient distributed LBs are when compared to centralized approaches for large scale applications and platforms; and (iii) the difference in LB time between our *packing*-based distributed algorithms when compared to *Distributed*. This last point can be specially observed in Fig. 5a, where, *Distributed* displays considerably higher LB times in its first call, averaging 20.2ms to 26.3ms, while *PackSteal* averages 4.3ms to 8.7ms, and *PackDrop* averages 8.0ms to 16.2ms for sizes 80 to 160.

When further analyzing the total number of migrations of each LB, we have observed that *Distributed* performs much more migrations than *Refine*, *PackDrop*, or *PackSteal*. For instance, Table 5 presents the average number of migrations for each load balancer and each LB call in *LeanMD* for input size $X = 160$. This suggests that the number of migrations is affecting the costs of communication in these scenarios, thus affecting the application's performance. In this scenario, we see that strategies that migrate less tasks become more effective, as they will distort less the predicted cost of tasks in the system.

Overall, Table 5 also shows that *Refine* is the LB that performs the least migrations, which is its purpose. Nevertheless, just performing a small number of migrations is not enough to improve the performance of *LeanMD* in these scenario, indicating that migrating more tasks together can be more beneficial that just avoiding migrations. For instance, we see in Fig. 3 that *Refine* performs equal to the baseline for input sizes 120 and 160 (Welch's t-test, p-values > 0.05).

These experimental results show how our *packing* scheme achieves its objectives of improving the scheduling process, as we see that: (i) it reduces the application's total execution time and scales to bigger platforms and input sizes; (ii) it achieves smaller load balancing times than other diffusive algorithms; and (iii) it also preserves some of the original locality of the application with its small number of groups of tasks migrating together.

7 Conclusion

In this paper we have developed the idea of workload discretization for periodical load balancing. We have also presented two diffusive schedulers that employ this technique, *PackDrop* and *PackSteal*. While *PackDrop* is push-based and uses randomized workload diffusion [19], *PackSteal* is pull-based, and employs WS heuristics such as constrained and randomized victim selection [3], [25].

In order to evaluate the benefits of *packing*-based diffusive schedulers, we implemented both LBs in the `Charm++` RTS and ran experiments with a synthetic and a molecular dynamics benchmark. We have compared *PackDrop* and *PackSteal* with multiple LBs available in `Charm++` [18] in a shared memory machine and a supercomputer.

Results showed that for highly irregular applications, *PackSteal* is one of the best approaches currently available for `Charm++`. Using a synthetic benchmark (Section 6.2), we have shown results with a total of 9 experimental profiles. *PackSteal* achieved the higher application time speedups in multiple scenarios, and was the only one to enhance application performance in the Randomized Graph communication scenario, which led to the most unpredictable system states. For this communication pattern, *PackSteal* made the application $1.13\times$ and $1.12\times$ faster in 12k and 18k tasks scenarios, respectively.

Moreover, *PackSteal* outperformed all other scheduling strategies on our two largest synthetic scenarios (18k and 24k tasks). On a smaller scale (12k tasks), *Greedy* was the only strategy able to outperform *PackSteal*. This is understandable since it is able to deliver a $\frac{4}{3}-$approximate solution, however it has been shown to scale poorly over larger systems [18], [19], as we have observed in Section 6.3. Finally, we showed the difference in performance of LBs with different LB periods, a technique we used on our previous experiment to compare the available strategies fairly.

For the molecular dynamics benchmark, *PackSteal* was the most effective LB in our larger experimental platform, especially when evaluating load balancing time. *PackSteal* was the best overall performing load balancer, being up to $5.2\times$ faster than *Distributed* [18]. When compared to

centralized schedulers, decentralized approaches tend to show faster convergence, which was observed here as well. This advocates for the development of new and effective distributed load balancers. Additionally, *PackDrop* was the second best performer LB in the MD benchmark, which indicates that *packing* is a good load balancing approach for this application class.

Our results lead us to believe that the study of established distributed scheduling algorithms is a path that has still to be more explored to achieve exascale grade scheduling. Developing distributed schedulers targeting HPC applications based on concepts such as *Deterministic Load Balancing* [50] and *Selfish Load Balancing* [22] is part of the work we have in mind moving forward. Additionally, we intend to further develop *packing schemes* with the use of local communication and topology awareness in order to create the migration packs, since in this work, as in [19], our packs are uninformed. Evaluating this scheduling strategy with different load profiles and real-world applications, in order to determine its best usage scenarios, is also a future research we intend to perform.

ACKNOWLEDGMENTS

This work was partially supported by the *Conselho Nacional de Desenvolvimento Científico e Tecnológico* – Brasil (CNPq) under the Universal Program (grant number 401266/2016-8) and by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* – Brasil (CAPES) under the Capes-PrInt Program (grant number 88881.310783/2018-01).

REFERENCES

[1] 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.

[2] C. R. Noble, A. T. Anderson, N. R. Barton, J. A. Bramwell, A. Capps, M. H. Chang, J. J. Chou, D. M. Dawson, E. R. Diana, T. A. Dunn, D. R. Faux, A. C. Fisher, P. T. Greene, I. Heinz, Y. Kanarska, S. A. Khairallah, B. T. Liu, J. D. Margraf, A. L. Nichols, R. N. Nourgaliev, M. A. Puso, J. F. Reus, P. B. Robinson, A. I. Shestakov, J. M. Solberg, D. Taller, P. H. Tsuji, C. A. White, and J. L. White, "Ale3d: An arbitrary lagrangian-eulerian multiphysics code," Lawrence Livermore National Lab. (LLNL), Tech. Rep. LLNL-TR-732040, 5 2017.

[3] J. Paudel, O. Tardieu, and J. N. Amaral, "On the merits of distributed work-stealing on selective locality-aware tasks," in *Proceedings of International Conference on Parallel Processing (ICPP)*. Lyon, France: IACC, 2013, pp. 100–109.

[4] 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," *J. Supercomputing Frontiers and Innovations (JSFI)*, vol. 2, no. 1, 2015.

[5] 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.

[6] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé, "Thermal aware automated load balancing for hpc applications," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2013, pp. 1–8.

[7] M. R. Garey and D. S. Johnson, "" strong " np-completeness results: Motivation, examples, and implications," *J. ACM*, vol. 25, no. 3, pp. 499–508, Jul. 1978.

[8] J. Lenstra, A. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," in *Studies in Integer Programming*, ser. Annals of Discrete Mathematics, P. Hammer, E. Johnson, B. Korte, and G. Nemhauser, Eds. Elsevier, 1977, vol. 1, pp. 343 – 362.

[9] H. Menon, N. Jain, G. Zheng, and L. Kalé, "Automated load balancing invocation based on application characteristics," in *International Conference on Cluster Computing (CLUSTER)*, 2012, pp. 373–381.

[10] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *International Conference on Supercomputing (ICS)*. Venice, Italy: ACM, 2012, pp. 185–194.

[11] R. Graham, E. Lawler, J. Lenstra, and A. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," in *Discrete Optimization II*, ser. Annals of Discrete Mathematics, P. Hammer, E. Johnson, and B. Korte, Eds. Elsevier, 1979, vol. 5, pp. 287 – 326.

[12] 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.

[13] 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), Livermore, US, Techinical, 2012, technical Report.

[14] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. Navaux, F. Broquedis, J.-F. Méhaut, and L. V. Kalé, "A hierarchical approach for load balancing on parallel multi-core systems," in *Proceedings of International Conference on Parallel Processing (ICPP)*. Pittsburgh, USA: IEEE, 2012, pp. 118–127.

[15] 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.

[16] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, and G. Zheng, "Communication and topology-aware load balancing in charm++ with treematch," in *International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[17] 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.

[18] 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.

[19] 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.

[20] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[21] 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.

[22] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. W. Goldberg, Z. Hu, and R. Martin, "Distributed selfish load balancing," *SIAM Journal on Computing*, vol. 37, no. 4, p. 1163–1181, Jan 2007.

[23] P. Berenbrink, T. Friedetzky, D. Kaaser, and P. Kling, "Tight & simple load balancing," in *Proceedings of International Conference on Parallel and Distributed Computing (IPDPS)*, Rio de Janeiro, BR, 05 2019, pp. 718–726.

[24] M. Lieber, K. Gössner, and W. E. Nagel, "The potential of diffusive load balancing at large scale," in *Proceedings of European MPI Users' Group Meeting (EuroMPI)*. New York, NY, USA: ACM, 2016, pp. 154–157.

[25] V. Janjic and K. Hammond, "How to be a successful thief," in *Proceedings of European Conference on Parallel Processing (EuroPar)*. Berlin, Germany: Springer, 2013.

[26] Q. Chen and M. Guo, "Contention and locality-aware work-stealing for iterative applications in multi-socket computers," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 784–798, 2018.

[27] R. da Rosa Righi, R. de Quadros Gomes, V. F. Rodrigues, C. A. da Costa, A. M. Alberti, L. L. Pilla, and P. O. A. Navaux, "Migpf: Towards on self-organizing process rescheduling of bulk-

synchronous parallel applications," *Future Generation Computer Systems*, vol. 78, pp. 272 – 286, 2018.

[28] J. Lifflander, S. Krishnamoorthy, and L. V. Kalé, "Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications," in *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. Delft, NL: ACM, 2012, pp. 137–148.

[29] W. Lee, E. Slaughter, M. Bauer, S. Treichler, T. Warszawski, M. Garland, and A. Aiken, "Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC. Piscataway, NJ, USA: IEEE Press, 2018, pp. 34:1–34:13.

[30] C. F. Joerg and B. C. Kuszmaul, "Massively parallel chess," in *Proceedings of the DIMACS Parallel Implementation Challenge*, 1994.

[31] N. Gast and G. Bruno, "A mean field model of work stealing in large-scale systems," *ACM SIGMETRICS Performance Evaluation Review (PER)*, vol. 38, no. 1, pp. 13–24, Jun. 2010.

[32] 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.

[33] 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.

[34] 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.

[35] M. Diener, S. White, L. V. Kalé, M. Campbell, D. J. Bodony, and J. B. Freund, "Improving the memory access locality of hybrid MPI applications," in *Proceedings of European MPI Users' Group Meeting (EuroMPI)*. New York, USA: ACM, 2017, pp. 11:1–11:10.

[36] 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.

[37] L. L. Pilla, C. P. Ribeiro, P. Coucheney, F. Broquedis, B. Gaujal, P. O. Navaux, and J.-F. Méhaut, "A topology-aware load balancing algorithm for clustered hierarchical multi-core machines," *Future Generation Computer Systems (FGCS)*, vol. 30, pp. 191 – 201, 2014.

[38] P. H. Penna, A. T. A. Gomes, M. Castro, P. D.M. Plentz, H. C. Freitas, F. Broquedis, and J.-F. Méhaut, "A comprehensive performance evaluation of the BinLPT workload-aware loop scheduler," *Concurrency and Computation: Practice and Experience (CCPE)*, p. e5170, 2019, early view.

[39] K. Li, M. Malawski, and J. Nabrzyski, "Reducing fragmentation on 3d torus-based hpc systems using packing-based job scheduling and job placement reconfiguration," in *Proceedings of International Symposium on Parallel and Distributed Computing (ISPDC)*, 2017, pp. 34–43.

[40] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Piscataway, NJ, USA: IEEE, 2014, pp. 857–868.

[41] M. Tchiboukdjian, V. Danjean, T. Gautier, F. Le Mentec, and B. Raffin, "A work stealing scheduler for parallel loops on shared cache multicores," in *Proceedings of European Conference on Parallel Processing Workshops (EuroParW)*. Berlin, Heidelberg: Springer, 2010, pp. 99–107.

[42] S. Shiina and K. Taura, "Almost deterministic work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. Denver, CO, US: ACM/IEEE CS, 2019.

[43] A. D. Kshemkalyani and M. Singhal, *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[44] P. Brucker, *Scheduling Algorithms*, 3rd ed. Berlin, Heidelberg: Springer-Verlag, 2001.

[45] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of Symposium on Principles of Distributed Computing (PODC)*. Vancouver, Canada: ACM, 1987.

[46] M. P. Wellman and W. E. Walsh, "Distributed quiescence detection in multiagent negotiation," in *Proceedings International Conference on MultiAgent Systems*, July 2000, pp. 317–324.

[47] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New Orleans, USA: IEEE/ACM, 2014.

[48] 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.

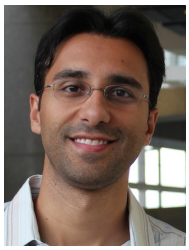
[49] J. C. Phillips, Gengbin Zheng, S. Kumar, and L. V. Kale, "Namd: Biomolecular simulation on thousands of processors," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov 2002, pp. 36–36, Gordon Bell Award winning work for special accomplishment.

[50] P. Berenbrink, R. Klasing, A. Kosowski, F. Mallmann-Trenn, and P. Uznański, "Improved analysis of deterministic load-balancing schemes," *ACM Trans. Algorithms (TALG)*, vol. 15, no. 1, pp. 10:1–10:22, Nov. 2018.

Vinicius Freitas is a student undertaking a master's degree in computer science at the Federal University of Santa Catarina (UFSC) in cooperation with LRI at Univ. Paris-Sud. He obtained his B.Sc. in computer science from UFSC in 2018. His research topics are load balancing, graph partitioning, and distributed algorithms.

Laércio L. Pilla is a CNRS Research Scientist at LRI, and member of the ParSys team. He obtained his Ph.D. from the Federal University of Rio Grande do Sul (UFRGS) and from the Univ. Grenoble Alpes in 2014. His research topics are load balancing, performance analysis and portability, and hierarchical architectures.

Alexandre de L. Santana is a Research Engineer at the Barcelona Supercomputing Center. He obtained his master's degree from the Federal University of Santa Catarina (UFSC) in 2019. His research topics are load balancing, performance portability, and runtime system decoupling.



Márcio Castro is an Adjunct Professor at the Federal University of Santa Catarina (UFSC), and head of the Distributed Systems Research Lab (LaPeSD). He obtained his Ph.D. from the Univ. Grenoble Alpes in 2012. His research topics are parallel programming models, load balancing, multicore and manycore architectures.

Johanne Cohen is a CNRS Research Director, and head of the graphs, algorithms and combinatorial research team at LRI. She obtained her Ph.D. from the University of Paris XI in 1998, and her research habilitation in 2009. Her research topics are game theory, graph theory, and distributed algorithms.