Scalable Load Balancing: Distributed Approaches and the Packing Model
Vinicius Freitas, Laércio Lima Pilla, Alexandre Santana, Marcio Castro, Johanne Cohen

To cite this version:
Vinicius Freitas, Laércio Lima Pilla, Alexandre Santana, Marcio Castro, Johanne Cohen. Scalable Load Balancing: Distributed Approaches and the Packing Model. 2019. hal-02405735
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 $\epsilon$-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-
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 PackStea;
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 [2] (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 central-
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 [29] 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 $T$ of $n$ tasks : $T = \bigcup_{i=1}^{n} \{T_i\}$. 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,

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

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

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

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

Definition 1. ($\epsilon$-relaxed order $\geq$). Given any two machines $M_p, M_q$, their relaxed difference in load is as stated in Equation (3).

$$\omega(M_p) \geq \omega(M_q) \iff \omega(M_p) > \omega(M_q) + \epsilon \quad \text{(3)}$$

From Equation (3), the definitions of $\epsilon$, $\geq$ and $\geq$ follow trivially.

We want to assign tasks to machines in a distributed fashion, so that the makespan approximates $\omega^*$. This follows the game-theoretic idea of achieving an $\epsilon$-Nash Equilibrium [22], a concept widely applied to distributed algorithms. Achieving $\epsilon$-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 $\epsilon$ 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 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 $P_j$ in $S_j$ that contains tasks so that $\omega(M_j) - \omega(P_j) \leq \omega^*$. This subset $P_j$ corresponds to the set of tasks the agent $j$ intends to migrate. Moreover, $P_j$ is subdivided into disjoint sets of tasks called packs, $\Gamma$, with total load $\omega^x$ 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 $\gamma$-relaxed equality among themselves, following Equation (4).

$$\forall \Gamma_x, \Gamma_y \in \mathcal{P} \mid \omega(\Gamma_x) \leq \omega(\Gamma_x) \leq \omega^x$$

Equation (4) also means that the load of all packs, regardless of machine, are similar. We denote this load as $\omega^x$.

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 $\omega^*$ 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 ($P_j$), 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.

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.   $\mathcal{R}_j \leftarrow \emptyset$; $\mathcal{P}_j \leftarrow \emptyset$
3.   reduce from $M$: $\frac{\omega(\Gamma)}{m} \rightarrow \omega^*$
4.   calculate $\omega^x$ // Equation (7) in Section 5
5.   if $\omega(S_j) > \omega^*$ then
6.     generate packs: $\mathcal{P}_j$ // See Section 4.1
7.   else
8.     gossip information: $\omega(S_j)$
9.   end
10. --- Wait Barrier ---
11. $[\mathcal{M}] \leftarrow \{ [\mathcal{M}] \subseteq \mathcal{M} \mid \forall M_i \in [\mathcal{M}], M_i \not\in \omega^* \}$
12. while $\mathcal{P}_j \neq \emptyset$ do
13.   $\Gamma \rightarrow p \mid p \in \mathcal{P}_j$; $\mathcal{P}_j \leftarrow \mathcal{P}_j \setminus \{\Gamma\}$
14.   send TASKS ($\Gamma$) to $M_s \sim [\mathcal{M}]$
15. end
16. when receive TASKS ($\Gamma$) from $M_s$ do
17.   if $\omega(S_j) + \omega^x \leq \omega^* + \epsilon$ then
18.     $\mathcal{R}_j \leftarrow \mathcal{R}_j \cup \{\Gamma\}$
19.     send ACK (true, $\Gamma$) to $M_s$
20.   else
21.     send ACK (false, $\Gamma$) to $M_s$
22. end
23. end
24. when receive ACK (bool, $\Gamma$) from $M_s$ do
25.   if bool=true then
26.     Register the migration of $T \in \Gamma$ to $M_s$
27.   else
28.     send TASKS ($\Gamma$) to $M_i \sim [\mathcal{M}]$
29. end
30. end
```

In the second step (ii), overloaded machines will create a list of underloaded machines $[\mathcal{M}]$ (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 $M_i \in \mathcal{M}$) 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-
Pack load $\omega^f$ is defined by Equation (7) in Section 5. After that, schedulers take roles of either thieves or victims depending on 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 $\omega$, 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 $s_j$ STEALS 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 $n_b$ usage 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).

$$top^k(\mathcal{M}) = \{M \in \mathcal{M} \mid \forall M_i \in \mathcal{M} \setminus \mathcal{M}^k, M_j \in \mathcal{M}^k, \omega(M_i) \leq \omega(M_j)\}$$

Once $n_b$ surpasses $\frac{m}{2}$, 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 $\Gamma$ 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 [48].

2. Charm++ is available at: http://charm.cs.illinois.edu/software
3. Packing schemes and LBs are available at: https://github.com/viniciusmct/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 

\[
\text{cores, the seed may be a topology-aware hint. For instance, if each cluster node has c right-hand neighbor}
\]

\[
\text{termined as being its }
\]

\[
\text{in Algorithm 2) of a given scheduling agent j}
\]

\[
\text{ω gap between }
\]

\[
\text{However, the larger the packs, the larger is their potential load, since it will mitigate more of the algorithm complexity.}
\]

\[
\text{attempt local neighbors in j which is an in-node neighbor of }
\]

\[
\text{ξ erating the algorithm.}
\]

\[
\text{middle ground between optimizing the balance and accel-}
\]

\[
\text{pack narrowing factor}
\]

\[
\text{Meanwhile, the }
\]

\[
\text{in this scenario and is used by other schedulers in }
\]

\[
\text{Charm++’s native synchronization mechanisms to perform the reduction operation and the quiescence de-}
\]

\[
\text{CkReduction and CkStartQD).}
\]

In Section 4 we used some variables whose values must be set beforehand. Ideally, \( \epsilon \) 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 \( \omega^j (Γ^j) \) and the ideal \( \omega^j \). So, smaller packs tend to be tighter, which leads to higher quality in load balancing.

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

\[
\epsilon = \omega^j × ξ, \quad \omega^j = \epsilon × δ, \quad γ = \omega^j × ξ
\]

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 \( \omega^j \), 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 (5).

\[
ξ ← 0.05, \quad δ ← 0.4
\]

The seeded neighborhood of PackSteal (initial known \( M \) in Algorithm 2) of a given scheduling agent \( j \) was pre-determined 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 \( r = c × \left\lfloor \frac{j}{c} \right\rfloor + (j + 1) \mod c \), 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.

#### 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 multicores-linux64 --with-production. This machine was locally available for our experiments, allowing us to explore mul-

<table>
<thead>
<tr>
<th>LB</th>
<th>Type</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>Diffusive</td>
<td>Push-based strategy that uses probabilistic transfer of load to choose task receivers [18]. Gather system information with a gossip protocol.</td>
</tr>
<tr>
<td>Refine</td>
<td>Global</td>
<td>Attempts to minimize the number of migrations. Migrates tasks from overloaded to underloaded resources only [28].</td>
</tr>
<tr>
<td>Greedy</td>
<td>Global</td>
<td>Optimizes load distribution, not communication. Assigns tasks to cores using a Longest Processing Time (LPT) first policy [17].</td>
</tr>
</tbody>
</table>

| LB Test | Diffusive | Push-based strategy that uses probabilistic transfer of load to choose task receivers [18]. Gather system information with a gossip protocol. |
| LeanMD  | Global    | Attempts to minimize the number of migrations. Migrates tasks from overloaded to underloaded resources only [28].                                    |
| Master  | Global    | Optimizes load distribution, not communication. Assigns tasks to cores using a Longest Processing Time (LPT) first policy [17].                        |

#### 6.2 Experimental Results

| LB Test | Diffusive | Push-based strategy that uses probabilistic transfer of load to choose task receivers [18]. Gather system information with a gossip protocol. |
| LeanMD  | Global    | Attempts to minimize the number of migrations. Migrates tasks from overloaded to underloaded resources only [28].                                    |
| Master  | Global    | Optimizes load distribution, not communication. Assigns tasks to cores using a Longest Processing Time (LPT) first policy [17].                        |

### Table 1

Brief description of other LBs used in the experiments.
multiple execution scenarios. The second is the supercomputer Joliot-Curie. 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 ($c \in \{ \text{Ring}, \text{2D Mesh}, \text{3D Mesh}, \text{Random Graph} \}$), 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.


We discuss experiments in Tesla with the 12k and 18k tasks configurations for all communication patterns, and 24k for the 3D Mesh pattern only.[4] 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 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.

<table>
<thead>
<tr>
<th>Communication</th>
<th>Ring</th>
<th>2D Mesh</th>
<th>3D Mesh</th>
<th>Random Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB Frequency</td>
<td>40</td>
<td>70</td>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>Baseline</td>
<td>77.879</td>
<td>67.717</td>
<td>59.966</td>
<td>80.585</td>
</tr>
<tr>
<td>PackDrop</td>
<td>76.822</td>
<td>69.894</td>
<td>65.862</td>
<td>76.541</td>
</tr>
<tr>
<td>PackSteal</td>
<td>43.711</td>
<td>42.405</td>
<td>44.387</td>
<td>47.773</td>
</tr>
<tr>
<td>Distributed</td>
<td>72.650</td>
<td>63.646</td>
<td>65.384</td>
<td>63.840</td>
</tr>
<tr>
<td>Greedy</td>
<td>47.687</td>
<td>47.219</td>
<td>54.662</td>
<td>57.758</td>
</tr>
<tr>
<td>Refine</td>
<td>74.532</td>
<td>70.508</td>
<td>66.323</td>
<td>77.984</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Communication</th>
<th>Ring</th>
<th>2D Mesh</th>
<th>3D Mesh</th>
<th>Random Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB Frequency</td>
<td>40</td>
<td>70</td>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>Baseline</td>
<td>117.316</td>
<td>104.095</td>
<td>90.035</td>
<td>125.677</td>
</tr>
<tr>
<td>PackDrop</td>
<td>113.691</td>
<td>105.736</td>
<td>96.996</td>
<td>110.315</td>
</tr>
<tr>
<td>PackSteal</td>
<td>63.872</td>
<td>65.569</td>
<td>68.252</td>
<td>71.264</td>
</tr>
<tr>
<td>Distributed</td>
<td>77.000</td>
<td>93.719</td>
<td>104.811</td>
<td>71.733</td>
</tr>
<tr>
<td>Greedy</td>
<td>58.805</td>
<td>58.039</td>
<td>57.618</td>
<td>62.670</td>
</tr>
<tr>
<td>Refine</td>
<td>114.581</td>
<td>108.585</td>
<td>99.792</td>
<td>116.192</td>
</tr>
</tbody>
</table>

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

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, signaling 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. 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 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\textsuperscript{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/](http://charmplusplus.org/miniApps/)
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):

$$n = 100 + \frac{\text{cell} \_ \text{id} \times 150}{X \times Y \times Z}, \quad (9)$$

where $\text{cell} \_ \text{id}$ 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. 5 show boxplots 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. 5 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 $\leq 0.05$), 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 $\leq 0.05$). Finally, PackDrop also outperforms Distributed for $X = 80$ (p-value = 0.018), even though their execution times look similar.

Fig. 5 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. 5. 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
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× and 1.12× 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 1−ε−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× 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 and Selfish Load Balancing 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-Prlnt Program (grant number 88881.310783/2018-01).

REFERENCES


