PackStealLB: A Scalable Distributed Load Balancer based on Work Stealing and Workload Discretization

Vinicius Freitas, Laércio Lima Pilla, Alexandre Santana, Márcio C Castro, Johanne Cohen

PackStealLB: A Scalable Distributed Load Balancer based on Work Stealing and Workload Discretization

Vinicius Freitas[a], Laércio L. Pilla[b,*], Alexandre de L. Santana[a], Márcio Castro[a], Johanne Cohen[b]

[a] *PPGCC, Federal University of Santa Catarina (UFSC), Florianópolis, Brazil*
[b] *LRI, Univ. Paris-Saclay – CNRS, Orsay, France*

Abstract

The scalability of high-performance, parallel iterative applications is directly affected by how well they use the available computing resources. These applications are subject to load imbalance due to the nature and dynamics of their computations. It is common that high performance systems employ periodic load balancing to tackle this issue. Dynamic load balancing algorithms redistribute the application's workload using heuristics to circumvent the NP-hard complexity of the problem However, scheduling heuristics must be fast as to avoid hindering application performance when distributing the workload on large and distributed environments. In this work, we present a technique for low overhead, high quality scheduling decisions for parallel iterative applications. The technique relies on combined application workload information paired with distributed scheduling algorithms. An initial distributed step among scheduling agents group application tasks in packs of similar load to minimize messages among them. This information is used by our scheduling algorithm, PackStealLB, for its distributed-memory work stealing heuristic. Experimental results showed that PackStealLB is able to improve the performance of a molecular dynamics benchmark by up to 41%, outperforming other scheduling algorithms in most scenarios over almost one thousand cores.

Keywords: distributed load balancing, workload discretization, work stealing

*Corresponding author
Email address: pilla@lri.fr (Laércio L. Pilla)

2010 MSC: 68W15, 68M20

1. Introduction

Scientific and industrial applications commonly make use of High Performance Computing (HPC) resources to meet their needs related to size (e.g., *to treat terabytes or petabytes of research data*), performance (e.g., *to understand the spread of an epidemic as soon as possible*), or deadlines (e.g., *to predict the weather before the day comes*). Even though HPC applications may have different characteristics, they all have to be properly scheduled on the available HPC resources to achieve their objectives. As these platforms grow in scale, so does the risk of wasting their costly resources due to load imbalance [1, 2].

Load imbalance emerges when applications' tasks are waiting for some other tasks to complete, making resources go idle. This problem can be caused, among others, by tasks with different workloads, tasks whose workloads dynamically evolve during execution, or by having computing resources take on more tasks to recover from a node failure. A natural solution to load imbalance is to periodically redistribute the tasks over the computing resources. This Periodic Load Balancing (LB) approach is seen in applications such as the Gordon Bell award-winning application NAMD [3], and it can be applied on applications based on parallel iterative methods.

The problem to decide an optimal mapping of tasks to computing resources is considered to be of NP-Hard complexity [4, 5]. Moreover, applications with periodic load balancing must employ scheduling heuristics to dynamically redistribute its workload. In this study, we showcase how commonly-used heuristics can fail to achieve good results when operating in larger scales (i.e., larger numbers of tasks or computing resources) due to issues in their scheduling decisions or due to their overhead (Section 6). Without scalable periodic load balancing solutions, the scalability of HPC applications will be compromised, leading to longer execution times and wasted computing resources.

In this work, we propose a scalable load balancing approach by combining

distributed load balancing and the discretization of application workload. Our approach reduces scheduling overhead by applying the following design: (i) execute scheduling steps in a distributed fashion; (ii) perform workload discretization to simplify decisions; (iii) group local tasks to avoid numerous fine-grained migrations and (iv) minimize the messages between scheduling actors. The discretization technique, called *packing*, extends a previous technique [6] with notions related to ϵ-Nash Equilibrium. We also present *PackStealLB*, a new distributed load balancing algorithm that uses *packing* and inherits ideas related to constrained [7, 8] and randomized [9] Work Stealing (WS) heuristics. We carried out an experimental evaluation of *PackStealLB* with *LeanMD*, a molecular dynamics benchmark based on NAMD [10], on 20 compute nodes (960 cores) of the supercomputer *Joliot-Curie SKL*[1]. Our results show the benefits of *PackStealLB* in terms of total LB overhead, reduced number of task migrations, preservation of the original locality between tasks, and reduced total application execution times. Overall, our main contributions are:

1. A method for workload discretization named *packing* that extends previous work [6];
2. A novel distributed, WS-based periodic load balancing algorithm called *PackStealLB*;
3. The implementation of *PackStealLB* in `Charm++`, a state-of-the-art runtime system that supports load balancing in distributed memory scenarios, and an experimental evaluation comparing it to other state-of-the-art algorithms [6, 11, 12].

The remainder of this work is divided as follows. Section 2 discusses related work in periodic 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 algorithm. Section 5 discusses their implementation details in

[1] *Joliot-Curie SKL* is currently the 52^{nd} world's most powerful supercomputer according to November 2019's list on `www.top500.org`.

`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 scope when used in novel decentralized load balancing solutions. WS heuristics are regarded as distributed load balancing mechanisms for *task-parallel* applications. The simpler WS scheme is to have an executor to *steal* some workload from another when the former is idle. WS schedulers may also be classified as distributed *receiver-initiated* (or *pull-based*) load balancing schemes. In other words, underloaded processing elements request load from overloaded ones. LBs, on the other hand, are periodically invoked to remap the system workload. LBs exist on different flavors regarding topology (e.g. centralized or distributed) and are commonly implemented as *sender-initiated* (or *push-based*) schemes. Periodic LBs target a different application set, when compared to WSs, such as Bulk Synchronous Parallel (BSP) [13], 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 [].

Both LB and WS approaches have their advantages. Notably, WS is often applied in task-parallel Runtime Systems (RTSs) [14], and shared memory scenarios [15, 16] (even though it is distributed in nature), although it has been used for highly unpredictable applications in distributed memory as well [17]. 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* [18, 11] (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 [6, 12], and with limited strategies proposed in this field [19], 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 [20]. In this section, we discuss recent work in periodic LBs and WS, providing a solid base for our novel approach.

2.1. Periodic Load Balancing

We divide LBs into two categories based on their behavior: *global* and *diffusive* [21]. 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 [22] 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 [23].

Other relevant hierarchical algorithms are those based on graph and hypergraph partitioning. *Zoltan* [22] uses multi-level partitioning to parallelize and schedule work. This approach can also consider past work to reschedule tasks, reducing the decision time. Meanwhile, *Scotch* [24] and *Metis* [25, 26] 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* [27] 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 [28]. The *DistributedLB* (or *Grapevine*) [12] 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 [19].

The preservation of locality is a common secondary objective in scheduling as it affects the performance of large-scale applications [29]. In this sense, strategies that take system topology into account are a promising trend in global

scheduling to preserve locality [30]. For instance, *NuCo* [31] is a load balancer that considers the different latencies at Non-Uniform Memory Access (NUMA) and network levels. Likewise, *TreeMatch* [32] does topology mapping by taking the different levels in machine topologies into consideration. Our approach at this moment differs from these works by trying to be platform-agnostic. In this case, locality is preserved by keeping tasks bundled together when migrating. This avoids adding explicit information about the machine topology to the distributed scheduling agents. Nevertheless, we see the addition of topology-awareness to our algorithms as future work.

On a different approach, *PackDropLB* [6] 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 [33], or in job scheduling with *packing-based placement* to reduce fragmentation on 3D-Torus HPC systems [34].

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 [9, 35] 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* [36], general task parallelism, and parallel loops in shared memory [37] have also benefited from WS.

Following the topology-aware approach, WS strategies have been able to greatly increase application performance. The *Feudal Work Stealing* [7] 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* [8] 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* [38] uses localized hierarchical stealing to compensate imbalance in task-parallel applications.

On a different approach, *DistWS* [15] 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* [11] 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 periodic load balancing domain, the use of WS heuristics is attractive due to their well documented past and known convergence times [20]. 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 *PackDropLB*, and we believe strategies such as *Feudal WS* [7] and *Randomized WS* have much to give in the periodic 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, *PackDropLB*, and the new algorithm *PackStealLB*.

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) where S represents an arbitrary set of tasks.

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

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. *Each machine in this notation represents a core in the HPC platform.* 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. All communication between agents is lossless.

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

$$\omega(M_j) = \omega(S_j) = \sum_{T \in S_j} \omega(T), \qquad \omega^\star = \frac{\omega(\mathcal{T})}{m} \tag{2}$$

Since optimal scheduling of parallel machines is NP-Hard, achieving a process mapping that yields $\omega^\star$ to every core is rather unrealistic. In this situation, we focus on assigning tasks to machines so that the makespan approximates $\omega^\star$. We give this value a relaxation ϵ, which should be based on the imbalance characteristics of each application. In other words, given a parameter $\epsilon > 0$, we want to find a schedule such that the load $\omega(M_i)$ of each machine M_i is less than the relaxed makespan lower bound. This objective is reflected in Equation (3).

$$\epsilon + \omega^\star > \omega(M_i). \tag{3}$$

Observe that if ϵ is large enough (e.g., greater than the load of the largest task), a schedule can be greedily computed in a centralized fashion. In this work, we want to assign tasks to machines in a **distributed** fashion, so that the makespan approximates $\omega^\star$.

We use the game-theoretic idea of achieving an ϵ-Nash Equilibrium [39], a concept widely applied to distributed algorithms. Achieving ϵ-Nash Equilibrium means that no agent in the system (in this case, our schedulers) profits from taking actions that modify the state of the system: given any two machines M_p, M_q, $\epsilon \geq |\omega(M_p) - \omega(M_q)|$.

In this paper, we target parallel applications with non-uniform and non-preemptable tasks[2] that leverage persistence-based load balancing algorithms (usually implemented in asynchronous runtime systems) to improve load balancing. The application is paused during rescheduling time, meaning that LB time is considered an overhead for the application. Thus, a load balancing strategy must run quickly to actually diminish application makespan. We also consider that each local scheduling agent executes the same LB algorithm 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.

4. PackStealLB: A Work Stealing-based Load Balancer

In this section, we present the details of our workload discretization method and load balancer *PackStealLB*. First, we discuss the characteristics and limitations of a previous algorithm (*PackDropLB*). We then present our method for workload discretization that can be applied to any distributed LB to improve

[2] Brucker [40] defines preemptable tasks (or jobs), as tasks that may be divided any number of times.

scheduling time and to preserve some of the original task locality. Finally, we detail our new distributed WS-based LB that makes use of the proposed workload discretization (packing) model and overcomes the aforementioned limitations of previous approaches.

4.1. Limitations of Previous Efforts

PackDropLB is a distributed algorithm that migrates packs of tasks from overloaded to underloaded machines introduced in a previous work [6]. Overall, *PackDropLB* executes the following steps in each LB call. First, agents exchange messages to compute the average load of the system. Then, agents are divided into two groups based on the load of their corresponding machines. *Overloaded agents* create packs of tasks based on a fixed threshold. *Underloaded agents*, on the other hand, start a Gossip Protocol [41] to propagate machine load information. After a *global barrier synchronization*, overloaded agents send packs of tasks to randomly chosen underloaded targets, which in turn can reject packs if they become overloaded. Rejected packs can be sent again by overloaded agents to other random targets. This process is repeated up to a fixed number of times or until a stop criterion is reached.

PackDropLB was our first effort on improving the scalability of state-of-the-art LBs for iterative overdecomposed parallel applications. Although it has shown promising results compared to other global and diffusive LBs, it has the following limitations: (i) information propagation (Gossip Protocol [41]) and task migrations are carried out in two separate steps, which are in turn synchronized with a global barrier; (ii) several messages sent by overloaded agents containing packs of tasks may be discarded by underloaded target agents, causing the former to waste time to find new possible target agents; (iii) inferior performance than other diffusive LBs in some applications and/or platforms [6]. Overall, issues (i) and (ii) may have a significant impact on LB overhead and scalability, especially in large-scale platforms. The latter limitation is usually the outcome of issues (i) and (ii).

4.2. Application Workload Discretization (Packing Model)

Load balancing scenarios may be described as either *discrete* or *continuous* [42]. The discrete case describes uniform non-preemptable tasks, while the continuous case describes non-uniform 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).

Recall that given a fixed parameter $\epsilon > 0$, we want to find a schedule such that the load $\omega(M_i)$ of each machine M_i is less than $\epsilon + \omega^\star > \omega(M_i)$. Taking this into account, scheduling agent j takes a subset of tasks $\mathcal{P}_j$ in S_j so that the condition in Equation (4) can be satisfied.

$$\omega^\star + \epsilon \geq \omega(M_j) - \omega(\mathcal{P}_j) \geq \omega^\star. \tag{4}$$

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* that have almost the same load ω^Γ with $\epsilon \geq \omega^\Gamma$. Since we want to make all migrating workload discrete, and our tasks are non-preemptable, each agent j must aggregate them into several packs so that each pack Γ_x respects Equation (5) according to a fixed parameter γ with $\epsilon \geq \omega^\Gamma \geq \gamma$.

$$\Gamma_x \subset \mathcal{P}_j \; and \; \omega^\Gamma + \gamma \geq \omega(\Gamma_x) \geq \omega^\Gamma \tag{5}$$

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.

We slightly adapt the bin packing problem to our context. Unlike bin packing, agent j also selects a subset of tasks that will insert into the packs. In our adaptation, we consider that the load of each pack can not be greater than $\omega^{\Gamma} + \gamma$. The tasks are considered in an arbitrary order. If a task fits inside the currently considered pack Γ (of load $\omega^{\Gamma} + \gamma$), then the task is placed inside it, and the set $\mathcal{P}_j$ is also updated. Otherwise, the current pack is closed: this means that we can not insert another task. Observe that since Γ is greater than the largest task, $\omega^{\Gamma} + \gamma \geq \omega(\Gamma) \geq \omega^{\Gamma}$.

The decision process to the creation of a new pack of tasks is as follows: if $\omega(M_j) - \omega(\mathcal{P}_j) > \omega^{\star} + \epsilon$, then a new pack is opened and the current task is placed inside it. The process then repeats the previous steps. At the end of it, all packs respect the condition in Equation (5) and $\omega^{\star} + \epsilon \geq \omega(M_j) - \omega(\mathcal{P}_j)$. Observe that we also have $\omega(M_j) - \omega(\mathcal{P}_j) \geq \omega^{\star}$. Indeed, it is enough to consider the last pack generated called Γ_{last}: $\omega(M_j) - \omega(\mathcal{P}_j \backslash \{\Gamma_{last}\}) > \omega^{\star} + \epsilon$ and $\epsilon > \omega^{\Gamma} + \gamma \geq \omega(\Gamma_{last})$.

To sum it up, if agent j has a load (strictly) greater than $\omega^{\star} + \epsilon$, then it computes a set $\mathcal{P}_j$ of tasks and a set of packs so that the conditions in Equations (4) and (5) are satisfied. These conditions also imply that the number of the packs generated by our algorithm in $\mathcal{P}_j$ is less than $\frac{|\omega^{\star} - \omega(S_j)|}{\omega^{\Gamma}}$.

4.3. Load Balancing (LB)

PackStealLB is a new distributed LB that is motivated by feudal and randomized WS [7]. The feudal aspect comes in its *information propagation*, while the randomized aspect in its *victim selection*. As in *PackDropLB* [6], *PackStealLB* also migrates packs of tasks to reduce the scheduling time and to preserve task locality.

PackStealLB progressively gathers the state of the system (the load of each machine $M_j \in \mathcal{M}$) at the same time it performs its decisions. This is possible due to the attachment of the sender local load information to the messages targeting its peers. Now, we describe how system information is broadcast during message exchanging.

Algorithm 1: *PackStealLB*, perspective of M_j

Input: A set $\mathcal{S}_j$ of local tasks for machine (agent) j and a local knowledge about the machines ($\mathcal{M}$).

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** s_j according to ω^Γ `// Equations (6) and (8)`
5 $\quad$ **if** $\omega(S_j) \geq \omega^\star + \epsilon$ **then** `// Victim case`
6 $\quad\quad$ generate packs: $\mathcal{P}_j = \{\Gamma_1, \Gamma_2, \cdots, \Gamma_{|s_j|}\}$ `// Section 4.2`
7 $\quad\quad$ **send** `HINT()` to $\arg\min_{a \in \mathcal{M}} \omega(M_a)$
8 $\quad$ **else if** $\omega^\star - \omega^\Gamma \geq \omega(S_j)$ **then** `// Thief case`
9 $\quad\quad$ **for** s_j **do send** `STEAL(0)` to $a \sim top^k(\mathcal{M})$ `// Equation (7)`
10 $\quad$ **end**
11 **end**
12 **when receive** `HINT()` *from* M_b **do**
13 $\quad$ **if** $\omega(S_j) \geq \omega^\star + \epsilon$ **then send** `HINT()` to $\arg\min_{M_a \in \mathcal{M}} \omega(M_a)$ `// Victim case`
14 **end**
15 **when receive** `STEAL(`n_b`)` *from* M_b **do**
16 $\quad n_b \leftarrow n_b + 1$
17 $\quad$ **if** $\omega(S_j) \geq \omega^\star + \epsilon$ **then** `// Victim case`
18 $\quad\quad \Gamma \leftarrow \arg\min_{\Gamma \in \mathcal{P}_j} \omega(\Gamma)$; $\quad \mathcal{P}_j \leftarrow \mathcal{P}_j \setminus \{\Gamma\}$
19 $\quad\quad$ **send** `TASKS(`Γ`)` to M_b
20 $\quad\quad$ Register the migration of $T \in \Gamma$ to M_b
21 $\quad$ **else if** $n_b > vsthreshold$ **then forward** `STEAL(`n_b`)` to $M_a \sim \mathcal{M}$
22 $\quad$ **else forward** `STEAL(`n_b`)` to $M_a \sim top^k(\mathcal{M})$
23 **end**
24 **when receive** `TASKS(`Γ`)` *from* M_b **do**
25 $\quad \mathcal{R}_j \leftarrow \mathcal{R}_j \cup \Gamma$ `// Adds tasks to the received tasks list`
26 **end**

Consider that when the algorithm begins, the agent j is not aware of the current load of the other machines. However, each time agent j *sends* a message, it also includes its load information $\omega(M_j)$ with the message. If agent 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.

PackStealLB is described in Algorithm 1, following a simple and standardized notation for distributed algorithms [43]. It is split into two main parts. The first part (`INIT` on lines 1–11) describes the initial calculations and role de-

termination (*victim* or *thief*). The initial flow begins by issuing a reduction in order to assess the average system load (line 3). Then, it calculates the number of steals s_j for thief agent j using Equation (6).

$$s_j = \left\lceil \frac{\omega^\star - \omega(S_j)}{\omega^\Gamma} \right\rceil \tag{6}$$

Pack load ω^Γ is fixed so that $\epsilon \geq \omega^\Gamma$ and is defined for experimentation by Equation (8) in Section 5.

After that, agents take roles of either *victims* or *thieves* depending of their load (stated as `Victim case` and `Thief case`): agent j is a *victim* if its load ($\omega(S_j)$) is greater than $\omega^\star + \epsilon$. Agent j is a *thief* if its load is less than $\omega^\star - \omega^\Gamma$. Observe that *thief* agents have $s_j \geq 1$ and *victims* have $s_j \leq -1$.

After determining their role, victims will assemble their packs and send `HINT` messages in order to warn a potential thief. A potential thief is determined by choosing one machine in $\mathcal{M}$ with the lowest known load (line 7) uniformly at random. This choice is performed using only the local information of the victims (i.e., victim j does not know the load of the machines of $\mathcal{M}$ that have not yet communicated with it, and thus, victim j does not consider these machines).

When an agent j receives a `HINT` (lines 12–14), it stores relevant information about its peers (especially their loads). Additionally, if agent 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 9), where a is a possible victim for j. First, it sends a `STEAL` message containing the number of attempts n_b previously done ($n_b = 0$ at the beginning). At each time an agent receives this message, n_b is increased by 1. If the agent is not a victim, then the `STEAL` message is forwarded. The n_b information is used to determine if the victim selection will be *constrained* or *randomized*.

The standard choice is the *constrained selection*, which picks uniformly at random one of the k most loaded machines in $\mathcal{M}$, as determined in Equation (7). Since each agent j can perform this selection using its local information, it can

not know the load of all machines. For sake of simplicity, it has no estimation of it, and we assume that we do not consider these machines or we can assume that their load is equal to 0. We denoted by $top_j^k(\mathcal{M})$ the set of the k most loaded machines among $\mathcal{M}$ for agent j. Formally,

$$top_j^k(\mathcal{M}) \text{ is a subset of } \mathcal{M} \text{ where its cardinality is equal to} k \ |top_j^k(\mathcal{M})| = k, \\ \text{and } \forall M_i \in \mathcal{M} \setminus top_j^k(\mathcal{M}), M_\ell \in top_j^k(\mathcal{M}), \omega(M_i) \leq \omega(M_\ell) \quad (7)$$

As shown in line 21, a configurable threshold value ($vsthreshold$) is used to check if constrained selection is not working very well. Once n_b surpasses $vsthreshold$, agent j will then use randomized victim selection, simply choosing a possible machine index uniformly at random.

When receiving a `STEAL` message from machine M_b, a victim j will send the load contained in the first element of their list of packs, $\mathcal{P}_j$ (lines 17–19). Additionally, agent j must register that the tasks in Γ will migrate to M_b (line 20), 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 21 and 22 portray the two distinct aforementioned *victim selection* behaviors (randomized and informed selection, respectively).

Finally, once agent j receives a `TASKS` message, meaning agent 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 (line 25).

Asynchronous distributed algorithms often need synchronization mechanisms to perform certain operations. Although *PackStealLB* 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* [44], meaning that when no agent has messages in its queue, they have been synchronized.

4.3.1. Convergence and Complexity

Assume that at some step, a machine M_j is a victim: $\omega(M_j) > \epsilon + \omega^\star$. By the definition of $\omega^\star (= \dfrac{\omega(\mathcal{T})}{m})$ corresponding to a mean, there exists a machine

M_i, having its load less than $\omega^\star$. Thus, machine M_j is a thief and it should be sending a `STEAL` message or waiting for an answer to a `STEAL` message. So, we can define the following property:

Property 1. *When no more messages are sent, each machine M_i has load $\omega(M_i) < \omega^\star + \epsilon$.*

Now, we compute the complexity in terms of messages. The migration process boils down to two parts: (1) the search for a victim initiated by a thief, and (2) the migration of tasks from the victim to the thief.

First, we count the total number of communications corresponding to task migrations. To perform this, we will focus on agent j having $\omega(M_j) > \omega^* + \epsilon$ at the beginning. It implies that agent j is a victim. So, agent j creates a set of packs so that the conditions in Equations (4) and (5) are satisfied. During that, only tasks in $\mathcal{P}_j$ migrate. Thus during this process, its load is greater than ω^* (from Equation (4)). So, we have:

Property 2. *Let j be an agent. If $\omega(M_j) > \omega^* + \epsilon$, then its load is greater than ω^*. Moreover, agent j does not become a thief during the algorithm.*

To be precise, a victim j creates at most $s_j = (\lfloor \frac{\omega(M_j) - \omega^*}{\omega^\Gamma} \rfloor)$ packs at the beginning. Thus, there is at most s_j migration communication corresponding to j's tasks. In total, an upper bound of migration communication is

$$\sum_{j \text{ is victim}} s_j \leq \sum_{j \text{ is victim}} \frac{\omega(M_j) - \omega^*}{\omega^\Gamma}$$

Since $\sum_{j \text{ is victim}} \omega(M_j) < m \cdot \omega^*$, we obtain the following property:

Property 3. *During the entire process, there are at most $\mathcal{O}(\frac{(m-1)\cdot\omega^*}{\omega^\Gamma})$ communications devoted to task migrations.*

Observe that this bound is tight: we can consider the case where only one machine has all the tasks.

Second, we count the total number of communications corresponding to the search for a victim initiated by a thief. This number mainly depends on the knowledge of each machine.

At the beginning, a thief j knows only its own load. Thus, it randomly sends a `STEAL` message to a machine M_b adding its own knowledge (its load and its belief of the load of others) and so on, until the message reaches a victim. When a thief j receives an answer to this message, the message contains the route of `STEAL` messages. Thus, thief j can deduce that all machines except the sender of the message are victims, but thieves. Property 2 implies that these machines can not become a victim. Thus, when thief j knows that agent ℓ is not a victim, then thief j has no incentive to send a `STEAL` message to machine ℓ. Moreover, when thief j sends the next `STEAL` message, this message also contains the information about the fact that agent ℓ is a thief. So, if this message is forwarded, then it is not forwarded to ℓ.

Property 4. *During the entire process, given the fact that agent ℓ is not a victim, then agent ℓ receives only once the* `STEAL` *message initiated by agent j.*

Since the number of not victim agents is upper bounded by m, we have:

Property 5. *During the entire process, there are at most $\mathcal{O}(m^2)$* `STEAL` *messages corresponding to a retransmission of a* `STEAL` *message.*

Observe that a migration of tasks is triggered by a thief. So the total number of messages corresponding to an initial `STEAL` message is equal to the total number of messages devoted to task migrations. So we can conclude that:

Property 6. *During the entire process, there are at most $\mathcal{O}(m^2 + \frac{(m-1)\cdot\omega^*}{\omega^{\Gamma}})$ messages.*

5. Implementation

We implemented *PackStealLB* in `Charm++` [45] using its distributed load balancing framework[3,4]. `Charm++` is one of the most receptive runtime systems

[3] `Charm++` is available at: `http://charm.cs.illinois.edu/software`.

[4] *Packing* schemes and LBs are available at: `https://github.com/viniciusmctf/packing-schemes/tree/packs_2019-v1`.

for new LB strategies, especially in distributed memory systems [46], which allows us to pair it up with applications already existing in the environment. `Charm++`'s load balancing framework commonly follows a sequence of steps as it: (1) pauses the application, (2) organizes execution statistics for the schedulers, (3) computes a new schedule, (4) migrates tasks, and then (5) resumes application execution. This framework is applied to NAMD [3] with success, for instance. Nevertheless, periodic load balancing can also be done in parallel to the application's execution in general.

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 (8).

$$\epsilon = \omega^\star \times \xi, \qquad \omega^\Gamma = \epsilon \times \delta, \qquad \gamma = \omega^\Gamma \times \xi \tag{8}$$

Also detailed in Equation (8) is the γ relaxation factor, which is used to define pack size in Section 4.2, Equation (5).

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 (9).

$$\xi \leftarrow 0.05, \qquad \delta \leftarrow 0.4 \tag{9}$$

The seeded neighborhood of *PackStealLB* (initial known $\mathcal{M}$ in Algorithm 1) of a given scheduling agent j was predetermined as being its *right-hand neighbor*, which is given by $\{M_r\}$ for $r = (j+1)\ mod\ m$. `Charm++` attribute indices to each computational resource. In a nutshell, every core in a processing node is assigned an integer in ascending order. This process is repeated for every processing node so that every core has an unique id. Indeed, in homogeneous clusters, it is possible to use the indices to determine if a given core is local or remote. 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.

Finally, we set the victim selection strategy threshold to $vsthreshold = \frac{m}{4}$. The value is based on an empirical study carried out on the platform used to assess the performance of *PackStealLB* (Section 6).

6. Performance Evaluation

We performed an experimental evaluation of our workload discretization model and *PackStealLB* to evaluate the scalability of the algorithms when handling a molecular dynamics benchmark on a supercomputer [47]. This evaluation includes a comparison of *PackStealLB* to LBs from the state of the art (listed in Table 1), and with a *Baseline* execution using the `Charm++`'s *DummyLB*, which captures statistics but performs no actual load balancing. In this section, we first provide an overview of the metrics, statistical methods, experimental environment, and details of the molecular dynamics benchmark. This is followed by an analysis of the results obtained in our experiments.

Table 1: Brief description of other LBs used in the experiments.

LB	Type	Short description
GreedyLB	Global	Optimizes load distribution, not communication. Assigns tasks to cores using a Longest Processing Time (LPT) first policy [48].
RefineLB	Global	Attempts to minimize the number of migrations. Migrates tasks from overloaded to underloaded resources only [11].
DistributedLB	Diffusive	Push-based strategy that uses probabilistic transfer of load to choose task receivers [12]. Gathers system information with a *gossip* protocol.
PackDropLB	Diffusive	Push-based strategy that migrates packs of tasks from overloaded to underloaded machines [6]. Gathers system information with a *gossip* protocol.

6.1. Experimental Methodology and Environment

We first present the metrics and statistical methods used to compare *PackStealLB* with other LBs. Then, we describe our experimental environment. Finally, we give a brief overview of the molecular dynamics benchmark and we discuss our experimental design. The raw result files and scripts for their analysis are available in [47].

6.1.1. Metrics and Statistical Methods

Our methodology involves the evaluation of three factors:

1. *Total execution time (makespan):* how long an application takes to execute;
2. *Load balancing invocation time:* the time between invoking the LB and resuming the application after migrations;
3. *Useful application time*: total execution time, excluding the load balancing invocation times.

Minimizing the application total 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.

All comparisons of these metrics are based on a confidence interval threshold of 95% (significance of 5%) for the different statistical methods used. We start our comparisons by organizing the samples (e.g., total execution times for different LB and problem sizes), and then checking if they follow normal distributions (Kolmogorov-Smirnov test). If we do not reject the null hypothesis in any tests (i.e., all p-values > 0.05), then we use parametric methods for our comparisons (Welch Two Sample t-test). Otherwise, we move to nonparametric methods (Mann-Whitney U test, or Wilcoxon signed-rank test for dependent samples). For all these methods, a p-value < 0.05 means that we reject the null hypothesis that the compared versions perform the same, meaning that they perform differently.

6.1.2. Experimental Environment

We carried out the experiments on the *Joliot-Curie SKL* supercomputer[5]. 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. In our experiments, we employed 20 compute nodes for a total of 960 cores.

Joliot-Curie SKL runs on Red Hat Enterprise Linux 7.6, loading Open-MPI 2.0.4, and `C/C++` Intel 17.0.6.256 modules. `Charm++` version 6.9.0 was installed in the machine using the build target `mpi-linux-x86_64` and option `--with-production`. Both `Charm++` and the molecular dynamics benchmark were compiled with the `-O3` flag.

[5]Detailed specifications of Joliot-Curie (SKL Irene) available at: `http://www-hpc.cea.fr/en/complexe/tgcc-JoliotCurie.htm`

6.1.3. Molecular Dynamics Benchmark and Experimental Design

We selected the molecular dynamics benchmark *LeanMD* for our experiments. *LeanMD* is based on (and performs core computations of) the Gordon Bell award-winning application NAMD [10], providing a realistic scenario to measure the impact of novel LBs.

The parallel implementation of *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[6], 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 1.15 to 3.08 millions of atoms.

Our experimental evaluation was carried out with 20 repetitions for each parameter combination (input size and load balancing algorithm). More specifically, as *Joliot-Curie SKL* 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 is to measure the capability of LBs in dealing with varying problem sizes in a large scale. This benchmark creates n particles per cell following Equation (10).

$$n = 100 + \frac{cell\_id \times 150}{X \times Y \times Z}, \tag{10}$$

[6] *LeanMD is available at:* `http://charmplusplus.org/miniApps/`

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 also leads to higher imbalance in the application as the input size increases.

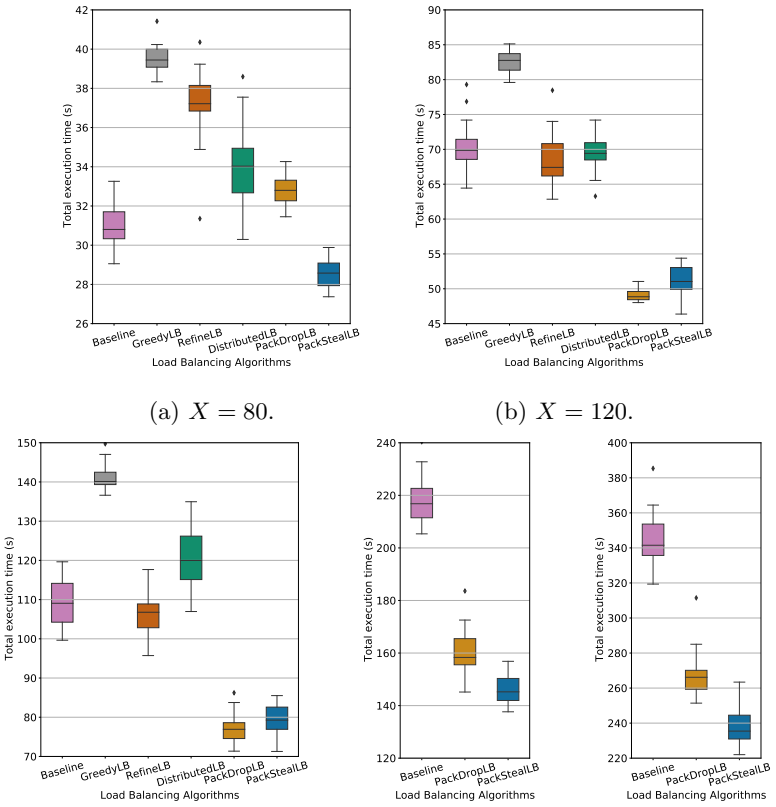
6.2. Result Analysis and Discussion

The total execution times of *LeanMD* with different input sizes and load balancers are illustrated as boxplots[7] in Figure 1. Only the *Baseline*, *PackDropLB* and *PackStealLB* were executed for input sizes 240 and 320 due to the increasing time it takes to run *LeanMD* for larger sizes, and to the performance results seen for the other LBs with smaller input sizes.

At a first glance, the results portrayed in Figure 1 show that *PackStealLB is the only LB to consistently reduce the total execution time of LeanMD compared to the baseline*. *PackDropLB* is a close second, failing only to outperform the baseline for the scenario with $X = 80$. *RefineLB* and *DistributedLB* achieved performances close to the baseline or worse than it, and *GreedyLB* always increased the total execution time of *LeanMD*, proving itself to be a bad fit for this situation.

In order to verify if the performance of the algorithms were statistically different, we applied Welch's t-test to some pairs of samples. This test was chosen because all samples followed normal distributions. The comparisons of *PackStealLB* or *PackDropLB* to the baseline all resulted in p-values < 0.05, so we can conclude that their execution times are actually different from the baseline as first suspected. Using the same test, we also verify that *PackStealLB* outperforms *PackDropLB* for $X = \{80, 240, 320\}$, it is outperformed by *PackDropLB* for $X = 120$, and that no difference can be seen between them for $X = 160$ (p-value $= 0.165$). Finally, there is no statistical difference between the perfor-

[7] 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) $X = 80$. (b) $X = 120$.

(c) $X = 160$. (d) $X = 240$. (e) $X = 320$.

Figure 1: Boxplots of the total execution times of *LeanMD* with different input sizes and load balancers on *Joliot-Curie SKL*. Sizes indicate a variation of the application dimension parameter X. Each figure has a vertical axis starting at its own value to emphasize performance differences.

mances measured for *RefineLB* or *DistributedLB* to the baseline for $X = 120$ (p-values = 0.107 and 0.402, respectively), and for *RefineLB* to the baseline for $X = 160$ (p-value = 0.153).

Figure 1 also indicates that both *PackStealLB* and *PackDropLB* have a greater impact to *LeanMD*'s total execution time as its input size increases. We can observe the speedup achieved over the baseline with load balancing for the different input sizes in Figure 2. *PackStealLB* achieved speedups of 1.09, 1.38, 1.38, 1.47, and 1.41 for increasing input sizes. Starting on $X = 120$, *PackStealLB*

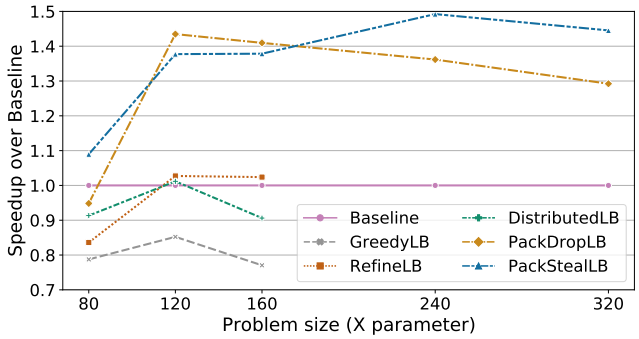


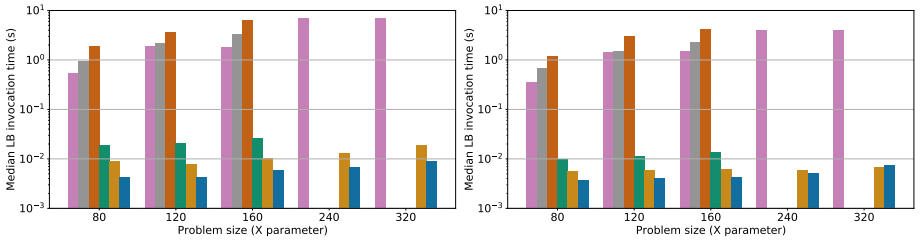
Figure 2: LeanMD speedups with different load balancers.

and *PackDropLB* lead to significantly better results than competing LBs, which emphasize their scalability and capability to handle load imbalance.

As mentioned earlier, the total execution time of *LeanMD* can be decomposed into two parts: (1) the *load balancing invocation times*, which act as an overhead during the execution of the application; and (2) the *useful application time*, which represents the actual time computing the molecular dynamics solution. We first focus on the load balancing invocation times in Figure 3. The figures display the median load balancing invocation time for each of the three calls by LB and problem size. These times are shown in log scale due to their differences of multiple orders of magnitude. We chose the median for these samples because some of them did not come from normal distributions (p-values < 0.05).

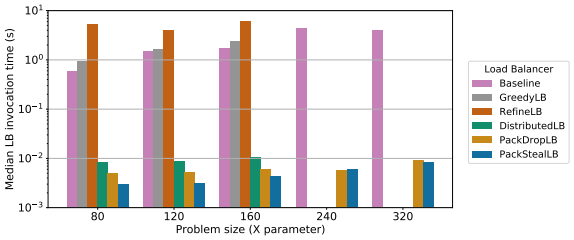
The results displayed in Figure 3 highlight multiple aspects of the LBs. Some of these aspects are as follows:

1. The first load balancing invocation usually takes longer than the other ones for all scenarios. The differences between the first and second invocation are significant for most scenarios (Wilcoxon signed rank tests with p-values < 0.05) with the exceptions of the baseline for $X = 160$ (p-value $= 0.062$), *PackDropLB* for $X = 160$ (p-value $= 0.067$), and *PackStealLB* for $X = 240$ and $X = 320$ (p-value $= 0.232$ and 0.191, respectively). As this behavior is seen even for the baseline execution, we can conclude



(a) LB invocation times (1st call).

(b) LB invocation times (2nd call).



(c) LB invocation times (3rd call).

Figure 3: Median load balancing invocation times for the first, second, and third LB calls during application execution. Data is displayed in log_{10} scale.

that these times are affected by the behavior of the application or even by `Charm++`'s LB framework. In this situation, comparisons between LBs must be limited only to invocation times of the same type (first, second, or third).

2. We can observe the high overhead of the centralized synchronization and data organization for the baseline execution. As it invokes `Charm++`'s *DummyLB*, it takes from 0.35 s up to 7.15 s for one LB call.

3. We can notice that *GreedyLB* and *RefineLB* take longer than the baseline, but *RefineLB* takes the longest. *RefineLB*'s longer times come from its more demanding decisions and not from a larger number of migrations. For instance, *RefineLB* migrates only a few hundred tasks on each LB invocation, while *GreedyLB* migrates almost all tasks (hundreds of thousands) every time.

4. The efficiency of diffusive algorithms is evident when compared to centralized approaches for this kind of large-scale application and platform. The

differences span from two to three orders of magnitude.

5. We can see the difference between *packing*-based algorithms (such as *PackStealLB*) and *DistributedLB*. For instance, Figure 3a shows that *PackStealLB* takes from 4.27 to 6.05 ms on its LB calls for problem sizes from $X = 80$ to $X = 160$, while *DistributedLB* takes from 19.43 to 26.05 ms. Additionally, *PackStealLB*'s invocation times are statistically different from the ones of *PackDropLB*. The only exceptions happen for the second and third LB calls for $X = \{240, 320\}$ (Mann-Whitney U tests with p-values > 0.05).

When the LB invocation times are subtracted from the total execution times, what remains are the useful application times. The useful application times for the different tested scenarios are presented in Table 2. Each line represents one LB, and each column shows the average useful times for a given problem size. The average time was chosen here because all samples follow normal distributions.

The results in Table 2 extend the insights from the total execution times in Figure 1 in a few ways. For instance, besides *PackStealLB*, *RefineLB* is also able to improve *LeanMD*'s performance for all tested problem sizes. This highlights the importance of keeping a low overhead when using periodic LB algorithms, as all gains from a better task distribution are erased by the LB invocation times of *RefineLB*. Still, *PackStealLB* achieves useful application times that are clearly better for $X = \{120, 160\}$, which emphasizes the quality of its scheduling decisions. On the opposite sense, we see that *DistributedLB* achieves worse times than the baseline for $X = 120$. This shows that just having a diffusive algorithm does not lead to performance improvements automatically. Finally, we see that *GreedyLB* always increases the useful application times, even though one would expect the application to become well-balanced with it. Given that the main difference between *GreedyLB* and *RefineLB* is how much of the original mapping they preserve, reasoning follows that the main performance issue being generated by *GreedyLB* comes from its disregard to the original task

Table 2: Average useful application times in seconds for different LBs and problem sizes.

	Problem sizes				
LB	**80**	**120**	**160**	**240**	**320**
Baseline	29.382	65.287	102.999	201.139	325.936
GreedyLB	36.865	77.158	133.254	—	—
RefineLB	28.688	56.395	89.500	—	—
DistributedLB	34.014	69.520	120.045	—	—
PackDropLB	32.769	49.035	77.174	160.049	266.891
PackStealLB	28.535	51.097	78.936	146.021	238.611

locality. Conversely, the preservation of locality is an important characteristic of our *packing* scheme, which helps explain how *PackStealLB* and *PackDropLB* outperform the other algorithms.

In conclusion, these experimental results show how *PackStealLB* and our *packing* scheme achieve their objectives of improving the scheduling process, as we see that: (i) they reduces the total and useful execution times, and scale to large platforms and input sizes; (ii) they achieve smaller load balancing times than other diffusive algorithms; and (iii) they also preserve some of the original locality of the application with their small number of groups of tasks migrating together.

7. Conclusion

In this paper we have developed the idea of workload discretization (*packing*) for periodic LB, and presented *PackStealLB*, a new diffusive scheduler that employs this technique. *PackStealLB* is a pull-based LB that employs WS heuristics such as constrained and randomized victim selection [7, 15].

We implemented *PackStealLB* in the `Charm++` RTS and ran experiments with a molecular dynamics benchmark (*LeanMD*). We have compared *PackStealLB* with our previous push-based LB (*PackDropLB*) as well as with multiple LBs

available in `Charm++` [12] in the *Joliot-Curie SKL* supercomputer.

The results of our experiments have shown *PackStealLB* as the most effective LB among the tested algorithms. *PackStealLB* achieved speedups of up to 1.49 over the total execution time of the baseline (a dummy LB that only collects statistics of the application), and of up to 1.39 when considering only the useful application time (execution time without any LB overhead). It also achieved a speedup of up 1.12 over the second best algorithm in the experiments, *PackDropLB*.

The success of *PackStealLB* comes from a combination of factors. *PackStealLB* was able to balance the application's load without disturbing much of its original locality thanks to its *packing* scheme, whereas other algorithms (*GreedyLB* and *DistributedLB*) increased the useful application time due to their disregard for locality. Additionally, its low overhead (a few milliseconds for every LB invocation) enables the application to benefit from the improved task distribution over its whole execution (tens or hundreds of seconds). This is in sharp contrast to what was seen for *RefineLB* whose LB invocation times hid any benefits coming from its scheduling decisions. Finally, when compared to the also *packing*-based *PackDropLB*, *PackStealLB*'s WS-based heuristics were shown to perform similarly or better for large problem sizes. This emphasizes that *PackStealLB*'s gains were not only due to its diffusive nature or workload discretization, but also from its own scheduling heuristics.

Our results lead us to believe that the development of new and effective distributed load balancers is indeed crucial for future applications and parallel systems. In this scenario, we believe that studying established distributed scheduling algorithms is a path that has still to be explored to achieve exascale grade scheduling. Developing distributed schedulers targeting HPC applications based on concepts such as *Deterministic Load Balancing* [49] and *Selfish Load Balancing* [39] is part of the work we have in mind moving forward. Additionally, we intend to develop new packing strategies that leverage the task communication pattern and the network topology in order to place tasks that communicate more often (or exchange more data) in the same pack whenever

possible. Finally, we plan to evaluate *PackStealLB* with a broad range of workloads and real-world applications to determine its best usage scenarios.

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).

This work was granted access to the HPC resources of TGCC under the allocation 2018 - AP010610724 made by GENCI (*Grand Equipement National de Calcul Intensif*).

[1] H. Khaleghzadeh, R. R. Manumachu, A. Lastovetsky, A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms, IEEE Transactions on Parallel and Distributed Systems 29 (10) (2018) 2176–2190.

[2] A. Cabrera Prez, A. Acosta, F. Almeida, V. Blanco, A dynamic multi-objective approach for dynamic load balancing in heterogeneous systems, IEEE Transactions on Parallel and Distributed Systems Early Access (2020) 1–13.

[3] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C. Phillips, 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), IEEE/ACM, Seattle, USA, 2011, pp. 61:1–61:11.

[4] M. R. Garey, D. S. Johnson, " strong " np-completeness results: Motivation, examples, and implications, J. ACM 25 (3) (1978) 499–508. doi:10.1145/322077.322090.

[5] J. Lenstra, A. R. Kan, P. Brucker, Complexity of machine scheduling problems, in: P. Hammer, E. Johnson, B. Korte, G. Nemhauser (Eds.), Studies in Integer Programming, Vol. 1 of Annals of Discrete Mathematics, Elsevier, 1977, pp. 343 – 362. doi:https://doi.org/10.1016/S0167-5060(08)70743-X.

[6] V. Freitas, A. Santana, M. Castro, 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), IEEE, Lyon, France, 2018, pp. 49–56.

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

[8] Q. Chen, M. Guo, Contention and locality-aware work-stealing for iterative applications in multi-socket computers, IEEE Transactions on Computers 67 (6) (2018) 784–798.

[9] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.

[10] J. C. Phillips, Gengbin Zheng, S. Kumar, L. V. Kale, Namd: Biomolecular simulation on thousands of processors, in: SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, 2002, pp. 36–36, Gordon Bell Award winning work for special accomplishment. doi:10.1109/SC.2002.10019.

[11] J. Lifflander, S. Krishnamoorthy, 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), ACM, Delft, NL, 2012, pp. 137–148.

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

[13] R. da Rosa Righi, R. de Quadros Gomes, V. F. Rodrigues, C. A. da Costa, A. M. Alberti, L. L. Pilla, P. O. A. Navaux, Migpf: Towards on self-organizing process rescheduling of bulk-synchronous parallel applications, Future Generation Computer Systems 78 (2018) 272 – 286.

[14] W. Lee, E. Slaughter, M. Bauer, S. Treichler, T. Warszawski, M. Garland, 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, SC, IEEE Press, Piscataway, NJ, USA, 2018, pp. 34:1–34:13.

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

[16] R. Al-Omairy, G. Miranda, H. Ltaief, R. Badia, X. Martorell, J. Labarta, D. Keyes, Dense matrix computations on numa architectures with distance-aware work stealing, J. Supercomputing Frontiers and Innovations (JSFI) 2 (1).

[17] C. F. Joerg, B. C. Kuszmaul, Massively parallel chess, in: Proceedings of the DIMACS Parallel Implementation Challenge, 1994.

[18] H. Menon, N. Jain, G. Zheng, L. Kalé, Automated load balancing invocation based on application characteristics, in: International Conference on Cluster Computing (CLUSTER), 2012, pp. 373–381. doi:10.1109/CLUSTER.2012.61.

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

[20] N. Gast, G. Bruno, A mean field model of work stealing in large-scale systems, ACM SIGMETRICS Performance Evaluation Review (PER) 38 (1) (2010) 13–24. doi:10.1145/1811099.1811042.

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

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

[23] G. Zheng, A. Bhatelé, E. Meneses, L. V. Kalé, Periodic hierarchical load balancing for large supercomputers, International Journal of High Performance Computing Applications (IJHPCA) 25 (4) (2011) 371–385.

[24] C. Chevalier, F. Pellegrini, Pt-scotch: A tool for efficient parallel graph ordering, Parallel computing 34 (6-8) (2008) 318–331.

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

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

[27] M. Deveci, K. Kaya, B. Uçar, U. V. Catalyurek, Fast and high quality topology-aware task mapping, in: Proceedings of International Parallel

and Distributed Processing Symposium (IPDPS), IEEE, Hyderabad, India, 2015.

[28] M. H. Willebeek-LeMair, A. P. Reeves, Strategies for dynamic load balancing on highly parallel computers, IEEE Transactions on Parallel and Distributed Systems (TPDS) 4 (9).

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

[30] 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) 28 (10).

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

[32] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, G. Zheng, Communication and topology-aware load balancing in charm++ with treematch, in: International Conference on Cluster Computing (CLUSTER), 2013, pp. 1–8. doi:10.1109/CLUSTER.2013.6702666.

[33] P. H. Penna, A. T. A. Gomes, M. Castro, P. D.M. Plentz, H. C. Freitas, F. Broquedis, J.-F. Méhaut, A comprehensive performance evaluation of the BinLPT workload-aware loop scheduler, Concurrency and Computation: Practice and Experience (CCPE) (2019) e5170Early view. doi:10.1002/cpe.5170.

[34] K. Li, M. Malawski, 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. doi:10.1109/ISPDC.2017.11.

[35] J. Yang, Q. He, Scheduling parallel computations by work stealing: A survey, International Journal of Parallel Programming (IJPP) 46 (2) (2018) 173–197.

[36] J. Lifflander, S. Krishnamoorthy, 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), IEEE, Piscataway, NJ, USA, 2014, pp. 857–868.

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

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

[39] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. W. Goldberg, Z. Hu, R. Martin, Distributed selfish load balancing, SIAM Journal on Computing 37 (4) (2007) 1163–1181.

[40] P. Brucker, Scheduling Algorithms, 3rd Edition, Springer-Verlag, Berlin, Heidelberg, 2001.

[41] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database

maintenance, in: Proceedings of Symposium on Principles of Distributed Computing (PODC), ACM, Vancouver, Canada, 1987.

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

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

[44] M. P. Wellman, W. E. Walsh, Distributed quiescence detection in multiagent negotiation, in: Proceedings International Conference on MultiAgent Systems, 2000, pp. 317–324. doi:10.1109/ICMAS.2000.858469.

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

[46] 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, D. S. Nikolopoulos, A taxonomy of task-based parallel programming technologies for high-performance computing, Springer Journal of Supercomputing 74 (4) (2018) 1422–1434.

[47] L. L. Pilla, V. Freitas, Performance results of LeanMD on the Joliot-Curie supercomputer using different load balancing algorithms (Jun. 2020). doi:10.5281/zenodo.3878953.
URL https://doi.org/10.5281/zenodo.3878953

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

[49] P. Berenbrink, R. Klasing, A. Kosowski, F. Mallmann-Trenn, P. Uznański, Improved analysis of deterministic load-balancing schemes, ACM Trans. Algorithms (TALG) 15 (1) (2018) 10:1–10:22. doi:10.1145/3282435.