HAL
open science

A Batch Task Migration Approach for Decentralized Global Rescheduling

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

To cite this version:

Vinicius Freitas, Alexandre Santana, Marcio Castro, Laércio Lima Pilla. A Batch Task Migration Approach for Decentralized Global Rescheduling. SBAC-PAD 2018 - International Symposium on Computer Architecture and High Performance Computing, Sep 2018, Lyon, France. pp.49-56, 10.1109/CAHPC.2018.8645953 . hal-01860626

HAL Id: hal-01860626

https://inria.hal.science/hal-01860626

Submitted on 23 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Batch Task Migration Approach for Decentralized Global Rescheduling

Vinicius Freitas, Alexandre de L. Santana, Márcio Castro*, Laércio L. Pilla†‡

August 23, 2018

Abstract

Effectively mapping tasks of High Performance Computing (HPC) applications on parallel systems is crucial to assure substantial performance gains. As platforms and applications grow, load imbalance becomes a priority issue. Even though centralized rescheduling has been a viable solution to mitigate this problem, its efficiency is not able to keep up with the increasing size of shared memory platforms. To efficiently solve load imbalance today, and in the years to come, we should prioritize decentralized strategies developed for large scale platforms. In this paper, we propose our *Batch Task Migration* approach to improve decentralized global rescheduling, ultimately reducing communication costs and preserving task locality. We implemented and evaluated our approach in two different parallel platforms, using both synthetic workloads and a molecular dynamics (MD) benchmark. Our solution was able to achieve speedups of up to 3.75 and 1.15 on rescheduling time, when compared to other centralized and distributed approaches, respectively. Moreover, it improved the execution time of MD by factors up to 1.34 and 1.22 when compared to a scenario without load balancing on two different platforms.

Index terms— High Performance Computing, Global Rescheduling, Load Balancing, Performance Evaluation, Parallel Algorithms

*Vinicius Freitas, Alexandre de L. Santana, and Márcio Castro are with the Federal University of Santa Catarina (UFSC) — Florianópolis, Brazil.

†Laércio is with the Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG — Grenoble, France.

‡This work was partially supported by the Brazilian Federal Agency for the Support and Evaluation of Graduate Education (CAPES) and by the Brazilian Council of Technological and Scientific Development (CNPq), project grant 401266/2016-8.

1 Introduction

Parallel machines are at their best when the workload is evenly distributed among compute nodes, and idle time is merely a myth. Unfortunately, strong scaling applications for these platforms has been a challenge as long as they have existed. In this context, uneven workload distribution and high communication overheads are the main villains when developing parallel applications [1]. Concerns towards these problems increase as systems grow in size and performance, consuming more resources, specially power, to solve some of the most complex problems in scientific computing [2, 3].

Applications such as simulations of molecular dynamics (MD) and hydrodynamics suffer from load imbalance due to their intrinsic dynamic and iterative nature [4, 5]. Although rescheduling algorithms have been able to greatly improve the performance of these applications [4], new approaches are needed to guarantee their performance as parallel systems grow. Since mapping tasks to processing elements (PEs) is an NP-Hard problem [6], the increase in application data and platform size makes centralized rescheduling approaches inefficient. This creates a need for scalable and decentralized rescheduling approaches, avoiding both excess of data to process and network contention [7].

The two main paths to achieve scalability in global rescheduling for iterative applications are *hierarchical* and *distributed* approaches. Hierarchical load balancing explores parallelism using different approaches for fine-grain and coarse-grain steps [8]. Although scalable, hierarchical schedulers are often tied to the same limitations of centralized approaches, as data is still aggregated in parent nodes. On the other hand, distributed load balancing approaches seek to achieve scalability through multiple smaller, decentralized scheduling decisions. Albeit more scalable than hierarchical

approaches, decentralized schedulers have limited system information and often incur in higher amounts of communication.

Despite their notable effectiveness, few are the distributed strategies in the domain of global rescheduling [9, 10]. In this paper, we present the concept of *Batch Task Migration* and a novel distributed global rescheduling algorithm that applies this technique, called *PackDrop*. Our approach is based on the idea of grouping tasks in batches prior to migration decisions, decreasing the communication overhead in the scheduling decision time and preserving the locality of migrated tasks [11].

In this paper, we present the following contributions:

1. A highly scalable *Batch Task Migration* approach for distributed rescheduling algorithms;
2. A novel distributed rescheduling algorithm, *PackDrop*, using our *Batch Task Migration* approach;
3. An implementation of our algorithm in a well-known parallel programming model as well as a performance evaluation of this implementation.

The remainder of this paper is divided as follows. Section 2 discusses recent work in dynamic rescheduling of scientific applications. Section 3 presents our novel approach and the developed algorithms. Section 4 presents a complexity analysis of our distributed algorithm (*PackDrop*). Section 5 displays implementation details, execution environments and benchmarks used in this paper. Section 6 presents our performance evaluation methodology and discusses the obtained results. Finally, Section 7 presents the conclusion of this work and our plans for future research.

2 Related Work

Global rescheduling is a well-studied problem in High Performance Computing (HPC) [1, 9, 10, 12, 13, 14, 15]. Redistributing the workload among PEs of a parallel system is a way to mitigate load imbalance created by dynamic applications. This is done in order to achieve strong scalability, and thus, efficient use of computing resources. In this section, we will discuss how different approaches seek to perform load balancing in distributed systems, why they lack scalability, and how we intend to mitigate migration, communication, and scalability issues in our proposed solution.

In the centralized domain, strategies implement a variety of heuristics in order to achieve an homogeneous distribution of load. Although centralized algorithms are used the most, their sequential and data dependent approach lacks scalability, as load balancing overheads exceeds its benefits with the increasing amount of input data. Different scheduling approaches are designed to tackle this scalability problem, being *hierarchical* and *distributed* the most widespread ones.

Hierarchical algorithms approach load balancing in different granularity levels, exploring parallelism and delivering better performance [8, 16]. These strategies are able to acquire as much static system information as the centralized techniques, while exploiting system parallelism. Some hierarchical strategies have used topology-aware approaches to increase mapping affinity [12, 14]. Others rely on a hypergraph representation to precisely describe application communication patterns [13]. However, these approaches still tend to create communication bottlenecks and may incur undesired overheads to aggregate the required information for rescheduling. As parallel systems grow, the amount of system data increases, and so does the cost of querying this information, which leads hierarchical approaches to be inefficient in larger systems.

Work stealing is one of the most broadly used distributed techniques for balancing load in parallel systems [17, 18, 19]. The essence of work stealing makes it a very effective solution for highly irregular parallel and distributed applications. However, work stealing may not be as effective, since its concurrent and randomized nature may interfere with iterative application execution cycle [20].

Also in the distributed domain, diffusive techniques have been used to irradiate work in an iterative fashion among PEs [9]. Although such an approach is interesting, since it may not impact much communication costs, it may also have a high convergence time, rapidly becoming inefficient in very imbalanced scenarios. Refinement-based distributed techniques, on the other hand, are able to provide fast and efficient rescheduling decisions without knowing too much about system information [10]. The main disadvantage of these techniques is the lack of affinity in migrated tasks, diminishing task locality, and thus, increasing total communication workload.

In the loop scheduling domain, a *Bin Packing* oriented approach has been able to exploit iteration affinity by adaptively partitioning loops [21]. Due to its greedy approach, this strategy can efficiently distribute work

among chunks before scheduling, increasing the overall application performance.

In this work, we propose a new approach for the distributed rescheduling domain named *PackDrop*. We adopt an approach similar to *Bin Packing* in order to preserve task affinity and diminish communication overheads in a decentralized fashion. Thus, our novel rescheduling approach intends to take profit from both distributed and affinity oriented scheduling policies.

3 Batch Task Migration Approach

The role of the global rescheduler is to ultimately reduce the application makespan. Thus, the scheduler policy must incur low overheads as to not overshadow its benefits. Moreover, we envision that a *Batch Task Migration* approach can ensure a quick and informed remapping of tasks, basically reducing the amount of messages during the scheduling decision process. Therefore, this section is dedicated to present our *PackDrop* strategy as a distributed refinement-based technique that implements our proposed *Batch Task Migration* approach.

Overall, our approach intends to:

1. *Reduce unnecessary communication* between PEs;
2. *Exploit locality* among tasks in the same PE through grouped migrations;
3. *Accelerate the decision making process*;
4. *Reduce the application makespan*.

We will first explain two algorithms that are used by *PackDrop*: *BatchAssembly* (Algorithm 1) and *BatchSend* (Algorithm 2). Then, the complete strategy, called *PackDrop*, will be presented in Algorithm 3. The *PackDrop* algorithm will be executed on each PE and communicates with other remote *PackDrop* instances via message passing. For clarity, all symbols used in the algorithms are listed in Table 1.

Table 1: List of symbols, variables and functions.

Symbol	Meaning	Definition
v	Load threshold in %	Equation 1
$ub(load, v)$	Upper bound of *load* with threshold v	Equation 1
$load$	Compute load of the local PE	
$load_{task}(t)$	Compute load of a task t	Equation 2
$load_{set}(T)$	Load of a set (T)	Equation 2
T	Set of tasks	Equation 2
ps	Estimate number of tasks in a LT	Equation 3
ttc	Total number of tasks in the system	Equation 3
$\rightarrow$	Remote procedure call	Section 3
$\Rightarrow$	Reduction process	Section 3
$load_{avg}$	Average system load of a PE	Section 3.1
$rand(S)$	Random element of S	Section 3.2
M	Mapping of tasks	Section 3.3
P	Global set of PEs	Section 3.3
$Gossip$	Start of information propagation	Section 3.3
$pack$	Set of leaving tasks	Section 3.3
LT	List of sets of tasks leaving a PE	Algorithm 1
L	Set of tasks assemblying a batch, subset of T	Algorithm 1
s	Threshold of load in a batch of tasks	Algorithm 1
G	Target for task receiving	Algorithm 2
BG	Pairs expecting migration ack	Algorithm 2
$Send(T) \rightarrow G$	Send a set T to target G	Algorithm 2
Id_l	Local PE identifier	Algorithm 3
$TaskMap$	Call runtime system to start migrations	Algorithm 3

3.1 *BatchAssembly* Algorithm

The *BatchAssembly* algorithm is presented in Algorithm 1. It uses an estimated batch size (s), a set of local tasks (T), the current PE *load* and a threshold for PE loads (v), to create a list of leaving packs (LT). The threshold is used to calculate an upper bound of the average system load ($load_{avg}$), using Equation 1. The load of any set of tasks is given by Equation 2.

$$ub(load, v) = (1 + v) \times load \quad (1)$$

$$load_{set}(T) = \sum_{t \in T} load_{task}(t) \ \mid \ T \text{ is a set of tasks} \quad (2)$$

With this information, each PE will take the task with the smallest load within its pool, and add it to a set of tasks (L) (lines $3-5$). Then, if the sum of all tasks in the pack (L) is greater than the expected batch size (s), the batch is assembled and the strategy starts creating another one (lines $6-9$). The process is repeated until the load of the set becomes greater than the upper bound (line 2).

Any unfinished packs should be sent even if these are not complete. This is done in order to prioritize migration from overloaded PEs, even if they cannot assemble a complete batch. A PE that receives this load will not receive as much load as others, but since the PE will not overload, it should not be prejudicial to global system balance.

3.2 *BatchSend* Algorithm

The *BatchSend* algorithm is presented in Algorithm 2. The algorithm will use the LTs, produced by *BatchAssembly*, and the set of *Targets*, produced by an

Input: s, load threshold of a batch; T, set of local tasks; $load_{avg}$, average global PE load; v, imbalance tolerance ratio.
Output: LT, list of packs leaving this PE.

```
1  L ← ∅, LT ← ∅
2  while load_set(T) > ub(load_avg, v) do
3      t ← a ∈ T | a is the lower bound of T
4      T ← T \ {t}
5      L ← L ∪ {t}
6      if load_set(L) > s then
7          LT ← LT ∪ {L}
8          L ← ∅
9      end
10 end
11 LT ← LT ∪ {L}
```

Algorithm 1: *BatchAssembly*

information propagation step (*Gossip* [22]), in order to schedule packs on remote PEs. This will produce a set of expected Batch/Target (*BG*) pairs, which should be confirmed by the remote target.

For each subset $b \in LT$ (as assigned in Algorithm 1), the algorithm will select a random target from G (line 3). It will invoke a remote *Send* procedure on the target g (line 4), and register its attempt in a pair (b, g). This pair is then stored in the expecting confirmation set (*BG*) (line 5).

Input: LT, set of tasks leaving the local PE; G, set of possible migration targets.
Output: BG, set of expected migrations.

```
1  BG ← ∅
2  foreach b ∈ LT do
3      g ← rand(G)
4      Send(b) → g
5      BG ← BG ∪ {(b, g)}
6  end
```

Algorithm 2: *BatchSend*

In case of negative responses from remote *Send* procedures, Algorithm 2 may initiate another round of sends with the failed attempts so every member of LT is migrated.

3.3 *PackDrop* Algorithm

The *PackDrop* strategy is presented in Algorithm 3. For the sake of simplicity, *packs* will be a short for "set of leaving tasks" in this algorithm (recall Sections 3.1 and 3.2).

PackDrop will run individually on each PE, in a distributed fashion. It will produce a new mapping (M') using a current local mapping of tasks to PEs (M), a local load ($load$), a local PE identification (Id_l) and the global set of PEs (P). The mapping of tasks is defined as $M: T \rightarrow P$, which describes the relation of each task to its corresponding physical location. A local mapping of tasks contains only tasks that are assigned to the current PE.

The first part of the algorithm (lines $1-6$) is the information sharing and setup process. This process is done through 2 global reductions of average PE load (line 2) and global number of tasks (line 3). In this implementation we used two constants: (i) v is set to 0.05 in order to limit the imbalance at 5% (line 5); and (ii) in Equation 3, a constant is set to 2, in order to determine ps.

$$ps = 2 - \frac{|P|}{ttc} \tag{3}$$

This configuration aims to both increase communication efficiency (later explained in Section 4) and ensure precise balancing. Finally, as ps grows, so does the communication efficiency, but in order to do so, fine-grain migrations are compromised.

Next, PEs are divided between two different workflows (line 7). At this time, *overloaded* PEs will start the *BatchAssembly* algorithm (line 8), which was previously explained in Algorithm 1. Meanwhile, *underloaded* PEs will initiate a *Gossip Protocol* [22] in order to inform other elements they are willing to receive work (line 11). *Gossip* is a well-known epidemic algorithm used to spread messages on a system, providing fast convergence and near-global awareness of shared information.

Once information propagation is done, each PE must synchronize to start the remapping process (line 13). At this point, PEs will send their packs using *BatchSend* (Algorithm 2) asynchronously (line 14). After a pack is sent, PEs will accept or reject it based on their current load. This is done via a *three-way handshake*, so both parts confirm the migration.

If one or more packs were not successfully exchanged, an *overloaded* PE must attempt a new *BatchSend*, in order to achieve load balance, as specified in Section 3.2. Once PEs know their new mappings, tasks are migrated and the strategy is finished, requesting the confirmed migrations to the underlying runtime

Input: M, local mapping of tasks; $load$, local PE load; P, set of all PEs in the system; Id_l, local PE identifier.
Output: M', new mapping of local tasks.

1 $M' \leftarrow \varnothing$
2 $load_{avg} \leftarrow (AveragePeLoadReduction(load) \Rightarrow P)$
3 $ttc \leftarrow (TotalTaskCountReduction(|M|) \Rightarrow P)$
4 $ats \leftarrow \frac{load_{avg}}{ttc}$ `// Average task size`
5 $v \leftarrow 0.05$ `// 5% precision on balance`
6 $s \leftarrow ats \times ps$ `// Pack load`
7 **if** $load > ub(load_{avg}, v)$ **then**
8 | $packs \leftarrow BatchAssembly(s, T(M), load, v)$
9 **else**
10 | $packs \leftarrow \varnothing$
11 | $G \leftarrow (Gossip \rightarrow P)$ `// Targets for migration`
12 **end**
13 $---SynchronizationBarrier---$ `/* Requests are processed as they are received back */`
14 $R \leftarrow BatchSend(packs, G)$ `// Implicit synchronization in TaskMap`
15 $TaskMap(R, M, M', Id_l)$

Algorithm 3: *PackDrop*

system (RTS) on line 15. The *TaskMap* function will take care of informing the new mapping (M') to all tasks received via *Send* and removed via *BatchSend*.

PackDrop intends to remap tasks to PEs in a distributed, workload-aware fashion. This approach may be the basis for new batch task migration distributed strategies that might take other factors into account.

4 Analysis of the Algorithm

This section presents an analysis of *PackDrop* (Algorithm 3). Symbols presented in this section are available on Tables 1 and 2. The complexity of the information propagation (*Gossip*) has being evaluated as $O(log_{fout} n)$ [10], where $fout$ is the *Gossip* fanout and n is the number of PEs in the system. Here we use $fout = 2$, in order to avoid network congestion.

For the sake of simplicity, in the remainder of this analysis, the number of tasks in the system will be referred to as m, and the costs for computation and communication will be represented as p_c and c_c, respectively. We also assume $c_c > p_c$ for all concurrent scenarios, since communication costs are several orders of magnitude higher than computational costs. $C(A)$ is referred here as the complexity class of a given workload A, similar to its total cost. Unmentioned lines are assumed to have non-varying costs, and thus will not interfere in the asymptotic analysis.

Table 2: List of symbols used in the Analysis of the Algorithm.

Symbol	Meaning
$fout$	*Gossip* protocol's *fanout*
p_c	Computational (processing) base cost
c_c	Communication base cost
$C(A)$	The complexity class of a given function A
m	Number of tasks in the system
m_l	$max(\|T\|)$ in an overloaded PE

Lines 2 and 3 are global reductions, which have a well-known cost of $O(log\ n)$. Lines 8 and 11 are concurrent, so their cost will be the maximum among them:

$$max(C(BatchAssembly), C(Gossip)) \quad (4)$$

We also know that the worst case for *BatchAssembly* (Algorithm 1) is rather unrealistic, since it would assume that a single PE contains m tasks and a single task may have a load greater than the average system load, being $O(m-1)$, assuming 1 would not be migrated, asymptotically, $O(m)$.

Thus, the cost of lines 8 and 11 is:

$$max((p_c \times m),\ (c_c \times log\ n)) \quad (5)$$

and since c_c is several orders of magnitude bigger than p_c, we could assume $C(BatchAssembly) \in C(Gossip)$, which makes the complexity of these lines to $O(log\ n)$.

Finally, line 14 will have a complexity equal to the largest number of packs migrated by an overloaded PE. Let ps be the average number of tasks inside of an LT, and m_l the maximum number of tasks in a given overloaded PE. At this step, a solution without *Batch Task Migration* would have a cost of $c_c \times m_l$, while our approach will divide this complexity by ps. This is the most expensive part of Algorithm 3, and as such it is the most interesting one to optimize. Our final asymptotic complexity is:

$$C(PackDrop) = O(m_l/ps) + O(log\ n) \quad (6)$$

This shows that determining a good *ps* value is crucial to achieve the best performance with this algorithm. Higher values of *ps* will lower communication complexity, but may lead to an imprecise scheduling. In our implementation, we chose to vary the value of *ps* around a base value of 2, according to system load and characteristics, as described in Equation 3. *BatchAssembly* (Algorithm 1) stores tasks in *LT* in an increasing load order. So, even though our average pack should have around 2 tasks, as their load vary, we are able to include more tasks at a lower communication cost. This way, we attempt precise load balance, while still preserving task locality after migration.

It is important to note that different applications may react differently to different values of *ps*. This is specially related to application load variance. The *ps* factor may be fine-tuned to each specific application and platform, but we believe a reasonable, generic, approach such as the provided by Equation 3 is enough to provide balance to most applications and systems, while still harvesting the advantages of the *Batch Task Migration* approach.

5 Implementation

PackDrop was implemented as a load balancing strategy in `Charm++`[1], a parallel programming model that provides a *load balancing framework* based on migration of its parallel, message-driven objects, the *chares* [23]. *Chares* are mapped as *tasks* to PEs and the `Charm++` RTS provides the load information needed for our rescheduling strategy.

The `Charm++` RTS allows for the desired asynchronous behavior of *PackDrop*. It also provides the necessary reduction and quiescence detection mechanisms used in this implementation. Reductions are used to evaluate the total number of chares and the average load in the system, while the quiescence detection is necessary to finalize the information propagation step of the algorithm.

`Charm++` provides application-independent load balancing, which means that any application may use global rescheduling strategies implemented for this RTS. This way, any of the available applications for `Charm++` can be used to evaluate and compare our strategy to other load balancers in this RTS.

During load balancing in the `Charm++` RTS, each instance of *PackDrop* runs as an individual chare. Remote procedures in the algorithms ($\rightarrow$) were implemented as messages exchanged between objects. Reductions ($\Rightarrow$) used the RTS provided interface (`CkReduction`). The *three-way handshake*, necessary to acknowledge a *Send*, was implemented with message exchanges. Finally, synchronization barriers in our algorithms use either quiescence detection or reductions.

5.1 Benchmarks

We experimented our strategy with 2 benchmarks that are publicly available for `Charm++`. The first one is a synthetic benchmark called *LB Test*. It simulates work with a variety of communication topologies, such as ring, meshes and randomized patterns. *LB Test* is known to have a low migration cost, with light *chares*, and most of its load bound to computation, instead of communication.

The second one is a molecular dynamics application called *LeanMD*[2]. This application simulates the behavior of atoms and it mimics the force computation done in the state-or-the-art NAMD application, which was the winner of the Golden Bell Award [4]. *LeanMD* uses geometric decomposition in a three-dimensional (3D) simulation space. However, since the number of simulated atoms in each region affects the number of exchanged messages, it has an irregular and dynamic communication pattern, even though it respects the geometric distribution.

5.2 Other schedulers

We compare the performance of *PackDrop* to other strategies available on `Charm++`. More specifically, strategies that may be selected by `Charm++`'s workload-aware *Meta-balancer* [24]. An overall description of each one of these load balancing strategies is presented below:

- *Refine* is a refinement-based strategy that tries to minimize the number of migrated *tasks*, exchanging load among PEs. This strategy is specially efficient if the system imbalance is low, and may not be able to deal with high imbalance due to its limited migration approach.

[1] Available at: `https://github.com/viniciusmctf/packdrop-code/tree/SBAC-Release`

[2] Available at: `http://charmplusplus.org/miniApps`

- *Greedy* creates two heaps, one for *tasks* (max-min) and one for PEs (min-max). Then, it assigns tasks to PEs, associating the most work-heavy tasks with the least loaded PEs. This strategy provides a good load balance, but may incur in high migration overhead.
- *Distributed*, also known as *Grapevine*, is a distributed strategy based on epidemic communication and probabilistic transfer of work. This strategy has good scalability, but does not perform as well as centralized ones in smaller scenarios.
- *Dummy*, a centralized load balancer that does not remap tasks but still gathers system information as other centralized approaches. This is the representative of a scenario without load balancing.

6 Performance Evaluation

We used two platforms to evaluate the performance of *PackDrop*:

- **Platform 1:** A cluster called *Graphene* from *Grid'5000* with 4 PEs per node and a Gigabit Ethernet interconnection network.
- **Platform 2:** A subset of a larger computational cluster called *Santos Dumont* from the Brazilian National Laboratory for Scientific Computing (LNCC) with 24 PEs per node and an Infiniband interconnection network.

All applications were compiled with `gcc` with the following flags: `-std=c++11 -O3`. Details of both platforms are available on Table 3.

In the next sections we present the metrics used to compare *PackDrop* with *Greedy*, *Refine*, *Distributed* and *Dummy*. Then, we discuss the results obtained in both platforms presented in Table 3 and the scalability of *PackDrop*. All raw data of our results, as well as parsing scripts for analysis are publicly available[3].

6.1 Metrics

Application time (makespan) is one of the most relevant metrics to evaluate load balancers. Since migrations may induce high overheads and could impact communication costs, a bad load balancing strategy may present fast decision times, while increasing load imbalance considerably. Thus, the overall execution time of the application will also be increased. This is a powerful metric to measure both load balancing precision and the overall impact of the strategy on parallel applications.

Load balancer decision time, on the other hand, is an indicator of its scalability. Some centralized schedulers, such as *Greedy*, work very well on local machines, with a reasonable data input. However, when executing on distributed memory environments, the scalability of centralized strategies is limited due to their high decision time in these scenarios. Throughout this section, load balancer decision time will also be referred to as *rescheduling time*.

[3]Available at: `https://github.com/eclufsc/packdrop-data-analysis`

Table 3: Overview of the experimental platforms.

Characteristics	Platform 1	Platform 2
# of nodes	32	16 – 32
# of CPUs/node	4 (UMA)	2×12 (NUMA)
CPU Model	Intel Xeon X3440	Intel Xeon E5-2695v2
CPU Freq.	2.53GHz	2.4GHz
RAM/node	16GB	64GB
Network	Gigabit Ethernet	Infiniband FDR
OS	Ubuntu 14.04	RedHat Linux 6.4
GCC version	5.4.0	5.3.1
Charm++ version	6.8.1	6.8.1
Communication	UDP	MPI 3.1.0

6.2 Evaluation on Platform 1

All experiments executed on Platform 1 were compiled with `Charm++` using the `--with-production` option, combined with the specifications detailed on Table 3. Overall, 32 homogeneous compute nodes were used, with a total of 128 PEs.

6.2.1 Evaluation with Synthetic Load

LB Test experiments had a total of $18,990$ *tasks*, executed over 150 iterations, performing load balance every 40 iterations. *Task* loads varied from 30ms to $9,000$ms, which provides reasonable load imbalance, causing global rescheduling to be useful in this case. Ring, 2D mesh and 3D mesh communication topologies were used to provide different levels of migration impact and communication costs.

Table 4: Average application time for *LB Test* on Platform 1.

Scheduler	Network Topologies		
	Ring	Mesh2D	Mesh3D
Distributed	47.493s	48.648s	49.055s
Greedy	46.541s	49.560s	51.068s
Dummy	52.430s	53.172s	53.941s
PackDrop	46.816s	47.371s	47.974s
Refine	45.491s	46.293s	47.219s

Table 5: Average application time for *LeanMD* on Platform 1.

Scheduler	Rescheduling Period		Rescheduling Time
	Short	Long	
Distributed	69.356s	68.360s	167.044ms
PackDrop	55.984s	55.516s	143.103ms
Refine	59.357s	55.899s	539.836ms

Each configuration of the benchmark was executed 15 times, with results depicted in Table 4. Observed application times present a maximum 2% standard deviation from the mean. Results for *Greedy* show how different communication topologies affect the scheduling performance. Since *Greedy* migrates many *tasks*, the more they communicate, the more migrations impact the application time.

The increased in communication cost can be verified among all scheduling strategies, but in none as much as in *Greedy*. Our novel approach (*PackDrop*) has outperformed the other decentralized strategy (*Distributed*) in the *LB Test* case in this scale. However, since the platform is not large enough to present all of the potential gains of decentralized strategies, *Refine* still outperformed any other scheduler in this benchmark. Nevertheless, this indicates a good scalability potential for *PackDrop*, specially in a cluster with high communication overhead due to its Gigabit Ethernet connection.

6.2.2 Evaluation with Molecular Dynamics

LeanMD experiments generated a $9 \times 9 \times 9$ space, with a total of $27,702$ *tasks*. Each execution ran 500 iterations, with a first rescheduling step at the 10th iteration. Rescheduling periods (RP) of every 30 (short) and every 60 (long) iterations were used, providing different impacts of rescheduling on the application. *Greedy* and *Dummy* were excluded from this evaluation due to their high cost in an application such as *LeanMD*.

Each configuration of *LeanMD* was executed 10 times, making a total of $5,000$ iterations per configuration and are depicted in Table 5. Observed application times presented a standard deviation from the mean lower than 2% for all results presented.

Results show a better overall performance of *PackDrop*, outperforming the other strategies in the two scenarios chosen. Since our strategy migrates groups of tasks, it preserves locality of tasks after migration. Because of that, it was able to outperform *Distributed*.

The *Rescheduling Time*, presented in Table 5, shows the time taken by the periodical rescheduling (LB), task migration and the first iteration after the LB call. It shows the increased cost of *Refine*, which is due to both information aggregation costs and dealing with the high amounts of application data in a centralized fashion. *PackDrop* displays its effectiveness in rescheduling time, outperforming the other strategies and resulting in an overall better application time.

6.3 Evaluation on Platform 2

All experiments executed on Platform 2 were compiled with `Charm++` using the `--with-production` option, combined with the specifications detailed on Table 3. Different numbers of homogeneous 2×12 PEs compute nodes (2 NUMA-nodes with 12 cores each) were used to evaluate the scalability of *PackDrop*. We ranged from 16 (384 PEs) to 32 (768 PEs) unique nodes in our evaluation. In this platform, we focused on the *LeanMD* application, since *LB Test*'s synthetic behavior is well represented in Section 6.2.1.

6.3.1 Evaluation with Molecular Dynamics

LeanMD experiments generated a $10 \times 15 \times 10$ space, with a total of 171K *tasks*. Each execution ran 100 iterations, with a first rescheduling step at the 9th iteration. Rescheduling was performed every 30 iterations and each configuration of *LeanMD* was executed 10 times, making a total of $1,000$ iterations per configuration.

Results of mean application and rescheduling time are displayed in Figure 1 (to the left and right, respectively), both in a *log* scale to better exhibit the differences between *Distributed* and *PackDrop*. In this

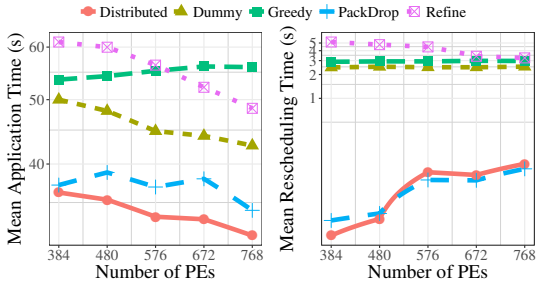

Figure 1: *LeanMD* execution results on Platform 2, displayed in logarithmic scale.

evaluation, *Refine* had a very low efficiency, being 1s slower than *Greedy* in the 384 PEs scenario. It was able to perform better as we increased the number of PEs, but was ultimately unable to compete with distributed approaches.

Distributed benefits from this platform due to Infiniband's low latency communication costs, which reflects on improved total application times, as seen in Figure 1. *PackDrop* followed it closely and we can see that its rescheduling time in larger systems outperforms *Distributed*.

The rescheduling and application time results of *LeanMD* in this platform highlight the importance of using scalable approaches to balance system load, as well as using available parallelism in execution environments. This is specially visible in *Greedy* results on Figure 1, where the application performance was decreased after the global rescheduling process. Increased migration costs and higher *hop* counts in communication, consequences of load balancing, heavily impacted *LeanMD* in this case.

6.4 Performance Evaluation Overview

Most scientific applications today seek strong scaling, increasing their computational platforms to solve problems faster. Our results showed that, to achieve such an objective, an application must implement efficient load balancing strategies. We presented *PackDrop* as a solution for scalable rescheduling of work in distributed memory systems.

Section 6.2.1 showed that *PackDrop* is able to effectively balance the load. Results highlight the importance of load balancing even in synthetic loads. The *LB Test* benchmark used has very low migration and communication overhead, and most of its work is done locally, which is optimal for rescheduling evaluation of raw computational workload. Moreover, *LB Test* is known for having a very low migration cost and simple tasks, which enhances the effectiveness of centralized approaches such as *Refine*. These results also portray the addition of communication overheads in different topologies (2, 4, and 6 communication edges for Ring, Mesh2D, and Mesh3D, respectively). As communication affects the application time more, migrations impact the total application time more, as we can see in the *Greedy* results.

In Sections 6.2.2 and 6.3.1, we evaluated *PackDrop* in *LeanMD* (better described in Section 5.1). This represents "a real world-like" scenario, in which applications may have dynamic communication patterns and high migration overhead. Results presented here highlight the overhead of centralized rescheduling approaches when joined with large-scale applications (171K tasks) and big environments, which increases work and information aggregation costs, respectively.

Distributed outperformed our approach in Platform 2, due to its more refined take on load balancing and high-speed network interconnection. However, the results show that *PackDrop* and its locality friendly batching of tasks for migration guarantees better performance in Platform 1, which portrays a Gigabit Ethernet interconnection. Finally, *PackDrop* was able to efficiently scale applications among all observed platforms, and had a faster rescheduling time than *Distributed* in most of the observed cases.

7 Conclusion

In this paper we have presented the *Batch Task Migration* approach for distributed global rescheduling. It intends to preserve task communication locality, migrating multiple work units from a source to the same destination, in order to balance system load. This preserves communication efficiency, while other workload-aware strategies perform rescheduling without considering task locality.

Our approach also mitigates communication costs during algorithm execution time. We guarantee this by transmitting information about multiple migrations at a time, in *batches*. Thanks to this, our novel scheduler (presented in Section 3.3) has an increased performance in high communication overhead platforms, discussed in Section 6.2.

We have evaluated our strategy in two different execution environments. The first was a high communica-

tion cost, 4 cores/node cluster, executing over 32 cores. In this scenario, *PackDrop* had a rescheduling speedup of up to 3.75 and 1.15 when compared to centralized and distributed approaches, respectively (Section 6.2).

The second scenario was a highly coupled cluster with low communication overhead, with 24 cores/node. We executed our experiments varying platform size from 16 to 32 nodes. In this scenario, rescheduling time of *PackDrop* and *Distributed* were very similar, although both had a time up to 3 orders of magnitude faster than any centralized approach. This reinforces the relevance of work in the distributed scheduling domain, and approaches such as our *Batch Task Migration*.

7.1 Future Work

Future work on this theme includes the use of *Batch Task Migration* in the communication-aware domain. Since our approach already has locality-based benefits, combining this with communication pattern information may incur on even greater performance increase in applications [16, 25]. We believe a novel strategy focused on the *Stencil* programming model is something to be considered, prioritizing migration of edges among PEs, instead of random parts of the stencil [26].

Further work will also be developed in order to increase performance in heterogeneous clusters. These may have heterogeneous processing capacities and network capabilities, which enhances complexity of load balancing significantly [20, 27]. In this given scenario, enhancing rescheduling decision processes may be crucial to ensure gains in application performance. Finally, we also intend to evaluate the impact of fine-tuning the *ps* factor in different computing platforms.

ACKNOWLEDGEMENTS

The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper (see `http://sdumont.lncc.br`).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

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

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

[3] E. L. Padoin, V. Martínez, P. O. Navaux, and J.-F. Méhaut, "Using power demand and residual load imbalance in the load balancing to save energy of parallel systems," *Procedia Computer Science*, vol. 108, 2017.

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

[5] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards *et al.*, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. Boston, USA: IEEE, 2013.

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

[7] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *Proceedings of International Conference on Cluster Computing (CLUSTER)*. New Orleans, USA: IEEE, 2009.

[8] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé, "Periodic hierarchical load balancing for

large supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 25, no. 4, 2011.

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

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

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

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

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

[14] E. Jeannot, G. Mercier, and F. Tessier, "Topology and affinity aware hierarchical and distributed load-balancing in charm++," in *Proceedings of International Workshop on Communication Optimizations in HPC (COMHPC)*. Salt Lake City, USA: ACM, 2016.

[15] J. Benson, T. Estrada, A. L. Rosenberg, and M. Taufer, "Scheduling matters: Area-oriented heuristic for resource management," in *Proceedings of International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Los Angeles, USA: IEEE, 2016.

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

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

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

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

[20] T. Beri, S. Bansal, and S. Kumar, "Prosteal: A proactive work stealer for bulk synchronous tasks distributed on a cluster of heterogeneous machines with multiple accelerators," in *Proceedings of International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. Hyderab, India: IEEE, 2015.

[21] P. H. Penna, M. Castro, P. D. M. Plentz, H. C. Freitas, F. Broquedis, and J.-F. Méhaut, "BinLPT: A workload-aware parallel loop scheduler for large-scale multicore platforms," in *Proceedings of Brazilian Symposium on High Performance Computing (WSCAD)*. Campinas, Brazil: SBC, 2017.

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

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

[24] H. Menon, "Adaptive load balancing for HPC applications," Ph.D. dissertation, Department of

Computer Science, University of Illinois at Urbana-Champaign, 2016.

[25] T. Hoefler, E. Jeannot, and G. Mercier, *An Overview of Topology Mapping Algorithms and Techniques in High-Performance Computing*. John Wiley & Sons, Inc., 2014.

[26] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with ops," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 4, 2018.

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