A Batch Task Migration Approach for Decentralized Global Rescheduling

Vinicius Freitas*, Alexandre de L. Santana*, Márcio Castro*, Laércio L. Pilla*†
* Universidade Federal de Santa Catarina (UFSC), Florianópolis, Brazil
† Institut National de Recherche en Informatique (INRIA), Grenoble, France
Email: {vinicius.mctf,alexandre.santana}@posgrad.ufsc.br, marcio.castro@ufsc.br, laercio.lima@inria.fr

***Abstract*—Effectively mapping tasks on parallel systems is crucial to assure performance on demanding applications. Throughout the years, global rescheduling of work has been a viable solution to mitigate the problem of load imbalance. As platforms and applications grow, load imbalance becomes a priority issue, and the previous algorithms are not able to deal with it as efficiently. In this paper we propose a novel approach and algorithm to improve decentralized global rescheduling, ultimately reducing communication costs and conserving task locality. We evaluated our approach in two different parallel platforms, using both synthetic load and a molecular dynamics (MD) benchmark, obtaining speedup of up to 3.75 and 1.15 on rescheduling time, when compared to centralized and distributed approaches, respectively; and up to 1.34 and 1.22 on MD application time, when compared to no rescheduling in both our two platforms.**

***Keywords*—High Performance Computing, Global Rescheduling, Load Balancing, Performance Evaluation, Parallel Algorithms**

I. Introduction

Parallel machines are at their best when work 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. Uneven distribution of work and high communication overheads are the main villains when devoloping parallel applications [1], [2]. Concern towards these problems increases as system grow in size, consuming more power and resources to solve problems.

Applications such as simulations of molecular dynamics (MD) and hydrodynamics suffer from load imbalance due to their intrinsic dynamic and iterative nature [3], [4]. Although rescheduling algorithms have been able to greatly improve applications [5], new approaches are needed to guarantee their performance as parallel systems grow. Since mapping work to processing elements (PEs) is a NP-Hard problem [6], the increase in application data and platform size makes centralized approaches to rescheduling inefficient. This creates a need for scalable, decentralized, rescheduling approaches, avoiding both excess of data to process and network contention [7].

The two main paths to achieve scalable global rescheduling in the iterative application domain are Hierarchical and Distributed approaches. Hierarchical load balancing explores parallelism using different approaches for fine-grain and coarse-grain steps [8]. These can scale, but are usually tied to the same limitations as the Centralized, as data is still aggregated in master nodes. On the other hand, Distributed load balancing seeks to achieve scalability in a decentralized fashion. These scale better, but have limited system information and may incur in high amounts of communication.

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

In this paper, we present the following contributions:

1) A *Batch Task Migration* approach for distributed rescheduling algorithms, which presents high scalability and task affinity potential.
2) A novel distributed rescheduling algorithm, using our *Batch Task Migration* approach, *PackDrop*.
3) An implementation of our algorithm, as well as a performance evaluation of this implementation.

The remainder of this paper is divided as follows: Section II presents recent work in dynamic rescheduling of scientific applications. Section III presents our novel approach and the developed algorithms. Section IV is a complexity analysis of *PackDrop*, our distributed algorithm. Section V displays implementation details, execution environments and benchmarks used in this paper. Section VI displays our performance evaluation and discussed experiments. And Section VII presents the conclusion of this work and our plans for future research.

II. Related Work

Global rescheduling is a well studied problem in high performance computing [1], [11]. Redistributing work inside the 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 present how different approaches seek balancing load 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 and new approaches must be pursued as systems grow. Different

approaches have been used to scale scheduling strategies, being *Hierarchical* and *Distributed* the most widespread ones. In this work we propose a new approach for the Distributed rescheduling domain.

Hierarchical algorithms will work differently in different granularity levels, exploring parallelism and delivering better performance [8], [12]. These strategies are able to acquire as much static system information as the centralized techniques, while taking advantage of system parallelism. Some hierarchical strategies have used topology-aware approaches to increase mapping affinity [12], [13], while others rely on a hypergraph representation to precisely describe application communication [14]. However, these approaches still tend to create communication bottlenecks and may take too long to aggregate the necessary information to perform the rescheduling itself. As parallel systems grow, the cost of having all this system information increases, and tend become inefficient.

Work stealing is one of the most broadly used distributed techniques for balancing load in parallel systems [15], [16]. 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 interfer with iterative application execution [17].

Also in the distributed domain, diffusive techniques have been used to irradiate work in an iterative fashion among PEs [10]. Although such an approach is interesting since it may not impact much communication costs, it may also have a high convergence time, easily becoming inefficient in very imbalanced scenarios. Differing this approach, refinement-based distributed techniques are able to provide fast and efficient rescheduling unaware of much system information to do so [9]. The main disadvantage of these techniques is the lack of affinity in migrated tasks, diminishing task locality, and thus, increasing their total communication load.

In the loop scheduling domain, a *Bin Packing* oriented approach has been able to exploit iteration affinity by adaptively partitioning loops [18]. Due to its greedy approach, this strategy can efficiently distribute work among chunks before scheduling, increasing the overall application performance. We intend to utilize a similar approach to preserve affinity and diminish communication overheads in a Distributed rescheduler. Our novel completely decentralized approach intends to take profit from both distributed and affinity oriented strategies alike.

III. 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, ultimately 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.

TABLE I
List of symbols, variables and functions.

Symbol	Meaning	Definition
v	Load threshold in %	Equation 1
$ub(load, v)$	Upper bound of l 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	Equations 2, 3
M	Mapping of tasks	Equation 3
$\rightarrow$	Remote procedure call	Section III
$\Rightarrow$	Reduction process	Section III
$load_{avg}$	Average system load of a PE	Section III-A
s	Compute load in a batch of tasks	Section III-A
$rand(S)$	Random element of S	Section III-B
P	Global set of PEs	Section III-C
$Gossip$	Start of information propagation	Section III-C
$pack$	Set of leaving tasks	Section III-C
LT	Set of tasks leaving a PE	Algorithm 1
L	Set of tasks, subset of LT	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 RTS to start migrations	Algorithm 3

The main benefits we expect with our approach are:

1) *Reducing* unnecessary communication;
2) *Exploring locality* of tasks mapped to the same PE, since migrations are done in groups;
3) An *accelerated decision making process*, consequence of the first;
4) And a diminished total application runtime.

First we will explain the *Batch Assembly* (Algorithm 1) and the *Batch Sending* (Algorithm 2) processes of this strategy. Then the complete strategy will be presented in Algorithm 3, *PackDrop*. All presented algorithms execute individually on each PE and communicate via message passing.

For simplicity, in all algorithms presented here: i) the symbol: "$\rightarrow$" will represent a remote procedure call; ii) and the symbol: "$\Rightarrow$" will represent the start of a reduction process. Other symbols used in the algorithms may be found in Table I.

A. Batch Assembly

The *Batch Assembly* process 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 set of leaving tasks (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.

$$ub(load, v) = (1 + v) \times load \tag{1}$$

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

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 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 while the set load is greater than the upper bound (line 2).

Any unfinished *LT*s should be sent even if those are not complete. This is done to avoid having an overloaded PE that can still migrate tasks. 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.

Algorithm 1: Batch Assembly

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

1 $L \leftarrow \varnothing,\ LT \leftarrow \varnothing$
2 **while** $load_{set}(T) > ub(load_{avg}, v)$ **do**
3 $\quad t \leftarrow a \in T \mid a$ is the lower bound of T
4 $\quad T \leftarrow T \setminus \{t\}$
5 $\quad L \leftarrow L \cup \{t\}$
6 $\quad$ **if** $load_{set}(L) > s$ **then**
7 $\quad\quad LT \leftarrow LT \cup L$
8 $\quad\quad L \leftarrow \varnothing$
9 $\quad$ **end**
10 **end**
11 $LT \leftarrow LT \cup L$

B. Batch Sending

The *Batch Sending* process is presented in Algorithm 2. The algorithm will use the *LT*s, produced by *Batch Assembly*, and the set of *Targets*, produced by an information propagation step (*Gossip* [19]), 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 $b \subset LT$ (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).

Algorithm 2: Batch Sending

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

1 $BG \leftarrow \varnothing$
2 **foreach** $b \subset LT$ **do**
3 $\quad g \leftarrow rand(G)$
4 $\quad Send(b) \rightarrow g$
5 $\quad BG \leftarrow BG \cup \{(b,\ g)\}$
6 **end**

When *Sends* receive a negative response, Algorithm 2 may initiate another round of sends with the failed attempts so every member of *LT* is migrated.

C. PackDrop Algorithm

The *PackDrop* strategy is presented in Algorithm 3. For simplicity, in the explanation of this algorithm *packs* will be a short for "set of leaving tasks" (seen in Sections III-A and III-B).

PackDrop will run individually on each PE, in a distributed fashion. Using a current local mapping of tasks to PEs (M), local load ($load$), a local PE identification (Id_l) and knowing all PEs within the system (P), to produce a new mapping (M'). The mapping of tasks is defined by Equation 3 as a set of pairs ($task$,PE), describing the location of tasks. A local mapping of tasks contains only tasks that are assigned to the current PE.

$$M:\ T \rightarrow P \tag{3}$$

Algorithm 3: PackDrop

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 $s \leftarrow ats \times (2 - \frac{|P|}{ttc})$ `// Pack load`
7 **if** $load > ub(load_{avg}, v)$ **then**
8 $\quad packs \leftarrow BatchCreation(ps, T(M), load, v)$
9 **else**
10 $\quad packs \leftarrow \varnothing$
11 $\quad G \leftarrow (Gossip \rightarrow P)$ `// 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 $TaskMap(R, M, M', Id_l)$

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: 0.05, in order to limit the imbalance at 5% (on line 5), and 2, in order to regulate the size of packs (on line 6).

After setup PEs are divided between two different workflows (line 7). At this time, *overloaded* PEs will start the Batch Assembly procedure (line 8), previously explained in Algorithm 1. Meanwhile, *underloaded* PEs will initiate a *Gossip Protocol* [19] 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 remap (line 13). At this point, the

remapping process will begin. PEs will send their packs using Algorithm 2, *Batch Send*, asynchronously (line 14). After a pack is sent, PEs will accept or reject it based on their current load, this is done via *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 *Batch Send*, in order to achieve load balance, as specified in Section III-B. Once the PEs know their new mappings, tasks are migrated and the strategy is finished, requesting the confirmed migrations to the RTS (line 15). The *TaskMap* function will take care of informing on the new mapping (M') all tasks received via *Send* and removed via *BatchSend*.

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

IV. Analysis of the Algorithm

This section presents an analysis of the Parallel Algorithm 3 (PackDrop). Symbols presented in this section are available on Tables I and II. The complexity of the information propagation (*Gossip*) has being evaluated as $O(log_{fout} n)$ [9], where $fout$ is the fanout for the algorithm and n is the number of PEs in the system. Here we use $fout = 2$, in order to avoid network congestion.

TABLE II
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 total cost of a given fuction A
m	Number of tasks in the system
ps	The average number of tasks in LTs
m_l	$max(\|T\|)$ in an overloaded PE

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 refered here as the total cost for a given workload A. Unmentioned lines are assumed to have non-varying cost, and thus will not interfere in the asymptotic analysis.

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 max among both:

$$max(C(BatchCreation), C(Gossip)) \tag{4}$$

We also know that the worst case for *BatchCreation* (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, $O(m)$.

This takes lines 8 and 11 cost to:

$$max((p_c \times m),\ (c_c \times log\ n)) \tag{5}$$

and since c_c is several orders of magnitude bigger than p_c, we could assume $C(BatchCreation) \in C(Gossip)$, 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 a 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 in Algorithm 3, and as such it is the most interesting one to optimize. Our final asymptotic complexity will be:

$$C(PackDrop) = O(m_l/ps) + O(log\ n) \tag{6}$$

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 a moderate value of ps, of around 2 average tasks, varying to smaller values according to system load. This guarantees a precise scheduling, but still gives margin to have several small tasks migrating at once, saving communication costs.

V. 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* [20], [21]. *Chares* are mapped as *tasks* to PEs and the `Charm++` runtime system (RTS) provides the load information needed for our rescheduling strategy.

The `Charm++` RTS allows for the desired asynchronous behavior of *PackDrop*. It also provides 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 our strategy and compare it to other load balancers available for this RTS.

A. Benchmarks

We experimented our strategy with 2 benchmarks that are publicly available for `Charm++`. The first is a synthetic benchmark called *LB Test*. It simulates work with a variety of

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

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 is a MD mini-app called *LeanMD*[2]. This mini-app simulates the behavior of atoms and it mimics the force computation done in the state-or-the-art MD application NAMD, winner of the Golden Bell Award [9]. 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.

B. Other schedulers

PackDrop was compared to strategies available on the `Charm++` load balancing suite. More specifically, strategies that may be selected by `Charm++`'s workload-aware *Metabalancer* [22]. Their behavior is briefly presented ahead:

- *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.
- *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 performs as well as centralized ones in smaller scenarios.
- *Dummy*, a centralized load balancer that does not remap tasks, only gather system information other centralized approaches also use. This is the representative of a scenario without load balancing.

VI. Performance Evaluation

Two platforms were for performance evaluation of our novel *PackDrop* scheduler: i) A tightly coupled *Supercomputer*[3] with 24 (in 2 separate NUMA nodes) PEs per node, using an Infiniband interconnection supported by Intel's Parallel Studio XE implementation of `MPI` (v2017.4). ii) A smaller *Cluster*[4] with 4 PEs per node, using a Gigabit Ethernet interconnection. Details of both platforms are available on Table III.

Ahead we present the metrics used to compare our new global rescheduling strategy with *Greedy*, *Refine*, *Distributed* and *Dummy*. Then, we discuss results obtained in both platforms presented in Table III and the scalability of our proposed solution. All raw data of our results, as well as parsing scripts for analysis are publicly available[5].

TABLE III
Platform Information of each Node from Supercomputer and Cluster evaluations.

Node Info.	Supercomputer	Cluster
CPUs	2×12	4
Intel Xeon	E5-2695v2	X3440
CPU Freq.	2.4GHz	2.53GHz
RAM	64GB	16GB
Network	Infiniband FDR	Gigabit Ethernet
OS	RedHat Linux 6.4	Ubuntu 14.04
`GCC`	5.3.1	5.4.0
`Charm++`	6.8.1	6.8.1
`MPI`	3.1.0	-
`GCC` Flags	`-std=c++11 -O3`	`-std=c++11 -O3`

A. Metrics

Application time is one of the most relevant metrics to evaluate load balancers in `Charm++`. Since migrations may induce high overhead and impact communication costs, a bad algorithm may finish fast, but increase imbalance, and thus, increased application time. This is a powerful metric to measure both load balancing precision and the overall impact application impact of a strategy.

Load balancer decision time, although not the most relevant for the application itself, the decision time 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, having a longer decision time. Throughout this section, load balancer decision time will also be referred to as *rescheduling time*.

B. Evaluation on Cluster

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

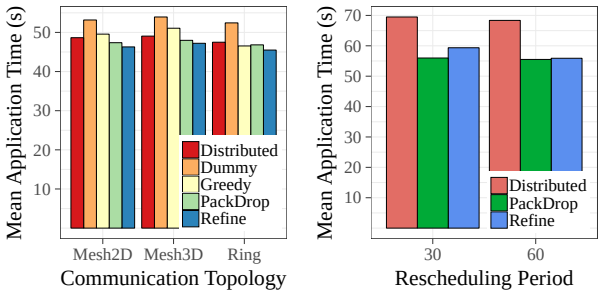
1) Evaluation with Synthetic Load: LB Test experiments had a total of 18990 *tasks*, executed over 150 iterations, performing load balance every 40 iterations. *Task* loads vary from 30ms to 9000ms, which provides reasonable imbalance of load, causing global rescheduling to be useful in this case. Ring, 2D mesh and 3D mesh communication topologies were used to provide different levels of migration impact and communication cost.

Each configuration of the benchmark was executed 15 times, with results depicted on Figure 1(a) and Table IV. 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.

[2]Available at: `http://charmplusplus.org/miniApps`
[3]`CPU` nodes in *Santos Dumont* (LNCC-BR).
[4]Nancy's *Graphene* cluster in *Grid'5000*.
[5]Available at: `https://github.com/eclufsc/packdrop-data-analysis`.

TABLE IV
AVERAGE LB TEST APPLICATION TIME ON THE CLUSTER EXECUTION.

Scheduler	Time (Ring)	Time (2D)	Time (3D)
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



(a) LB Test (b) LeanMD

Fig. 1. Cluster Execution Results for both applications.

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 descentralized 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*, with a reduced migration count approach, still outperforms any other scheduler in this benchmark. Nevertheless, this indicates a good scalability pontetial, specially in a cluster with high communication overhead, due to its Gigabit Ethernet connection.

2) Evaluation with Molecular Dynamics: LeanMD experiments generated a $9 \times 9 \times 9$ space, with a total of 27702 *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*.

TABLE V
LEANMD MEAN APPLICATION TIME ON THE CLUSTER EXECUTION.

Scheduler	Time (Short RP)	Time (Long RP)	Mean LB Time
Distributed	69.356s	68.360s	167.044ms
PackDrop	55.984s	55.516s	143.103ms
Refine	59.357s	55.899s	539.836ms

Each configuration of LeanMD was executed 10 times, making a total of 5000 steps per configuration and are depicted in Table V and in Figure 1(b). Observed application times presented a standard deviation from the mean lower than 2% for all results presented.

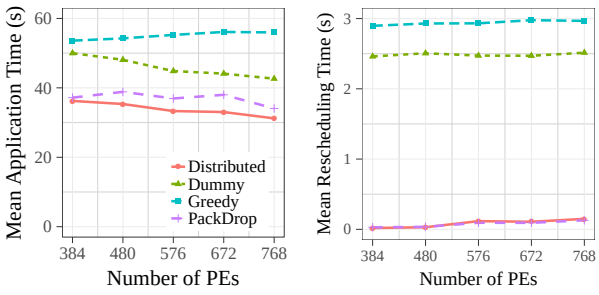
Results show a better overall performance of *PackDrop*, outperforming both compared strategies in the two scenarios chosen. Since our strategy migrates groups of tasks, it improves locality of tasks after migration, outperforming *Distributed* due to this.

The *Mean LB Time*, presented in Table V, 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 both compared strategies, and resulting in an overall better application time.

C. Evaluation on Supercomputer

All experiments executed on supercomputer were compiled with `Charm++` using the `--with-production` option, combined with the specifications detailed on Table III. Different numbers of homogeneous 2×12 PEs compute nodes (2 NUMA-nodes with 12 cores each) were used to evaluate our strategy's scalability. We ranged from 16 (384 PEs) to 32 (768 PEs) unique nodes in our evaluation.

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 1000 steps per configuration.



(a) Application Time (s) (b) Rescheduling Time (s)

Fig. 2. *LeanMD* supercomputer execution results.

Results of mean application time are displayed in Figure 2(a) and mean rescheduling time in Figure 2(b). *Refine* was excluded from this evaluation since this LeanMD input presents more data than Refine is able to process in a reasonable time, being in average 1000ms slower than *Greedy* in the 384 PEs test case.

Distributed benefits from this platform due to the Infiniband low latency communication, which reflects on improved total application times, as seen in Figure 2(a). Our novel approach, *PackDrop*, followed it closely and we can see that its rescheduling time in larger systems outperforms *Distributed*, displayed in Figure 2(b).

The rescheduling and application time results of LeanMD in this platform highlight the importance of using scalable approaches to load balancing, as well as using available parallism in execution environments. This is specially visible in *Greedy* results on Figure 1(a), 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.

D. Performance Evaluation Overview

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

Section VI-B1 shows our strategy's ability of balancing 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 peers for Ring, 2D, and 3D Mesh, respectively). As communication affects the application time more, migrations impact the total application time more, as we can see in the *Greedy* results.

Sections VI-B2 and VI-C1 display evaluation of a MD mini-app, LeanMD (better described in Section V-A). 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 the Supercomputer platform, due to its more refined approach for 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 the Cluster platform, which portrays a Gigabit Ethernet interconnection. *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.

VII. 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 III-C) has an increased performance in high communication overhead platforms, discussed in Section VI-B.

We have evaluated our strategy in two different execution environments. The first was a high communication cost, 4 cores/node cluster, executing over 32 cores. In this scenario, *PackDrop* had a rescheduling speedup of up to 3.77 and 1.16 when compared to centralized and distributed approaches, respectively (Section VI-B).

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 statistically equal, 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*.

A. 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. 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 [23].

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. In this given scenario, enhancing rescheduling decision processes may be crucial to ensure gains in application performance.

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. URL: `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 *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 197–206.

[2] T. Hoefler, E. Jeannot, and G. Mercier, *An Overview of Topology Mapping Algorithms and Techniques in High-Performance Computing*. John Wiley & Sons, Inc., 2014, pp. 73–94.

[3] 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 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[4] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[5] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations," Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep., February 2009.

[6] J. Hartmanis, "Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson)," *Siam Review*, vol. 24, no. 1, p. 90, 1982.

[7] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–8.

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

[9] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 15:1–15:11.

[10] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions on parallel and distributed systems*, vol. 4, no. 9, pp. 979–993, 1993.

[11] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for charm++ applications on large supercomputers," in *2010 39th International Conference on Parallel Processing Workshops*, Sept 2010, pp. 436–444.

[12] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. Navaux, F. Broquedis, J.-F. Mehaut, and L. V. Kale, "A hierarchical approach for load balancing on parallel multi-core systems," in *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE, 2012, pp. 118–127.

[13] 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 *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec 2012, pp. 236–243.

[14] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, best Algorithms Paper Award.

[15] J. Yang and Q. He, "Scheduling parallel computations by work stealing: A survey," *International Journal of Parallel Programming*, vol. 46, no. 2, pp. 173–197, 2018. [Online]. Available: https://doi.org/10.1007/s10766-016-0484-8

[16] V. Janjic and K. Hammond, "How to be a successful thief," in *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 114–125.

[17] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 137–148.

[18] 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 *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*. Campinas, Brazil: SBC, 2017, pp. 220–231.

[19] 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 the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 1–12.

[20] 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, pp. 30–37, 2016.

[21] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[22] H. Menon, "Adaptive load balancing for hpc applications," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2016.

[23] 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*, vol. 29, no. 4, pp. 873–886, April 2018.