# Evaluation of a Batch Task Migration Approach in Distributed Global Rescheduling

Vinicius Freitas
Universidade Federal de Santa Catarina
Florianópolis, Brazil
vinicius.mct.freitas@gmail.com

Alexandre de L. Santana
and Márcio Castro
Universidade Federal de Santa Catarina
Florianópolis, Brazil
alexandre.limas.santana@gmail.com
marcio.castro@ufsc.br

Laércio Lima Pilla
INRIA
Grenoble, France
laercio.lima@inria.fr

***Abstract*—Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.**

## I. Introduction

- Motivation: Why load balancing
- Context: Scenarios in which global rescheduling applies
- Problem: Scaling iterative applications: algorithm limitations
- Justification: Scalable load balancing, distributed rescheduling
- Related work: Hierarchical (limited scaling), Distributed (excessive comm), Diffusive (iterative), Refinement Based (centralized)
- Proposed solution: Batch task migration to mitigate communication costs
- Paper contributions
- Results brief
- Paper structure

## II. Related Work

Global rescheduling is a well studied theme in high performance computing [1], [2], [3], [4], [5], [6], [7]. 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, they lack scalability and new approaches must be pursued as systems grow.

Refinement based solutions in centralized load balancing are very effective in dealing with low imbalance or high migration costs [8]. Since these strategies use a limited migration heuristic, they are able to efficiently balance load with a low overhead. However, since refine strategies limit a total number of migrations, they cannot deal with very unbalanced systems.

Citar balanceadores centralizados é mesmo necessário? Não parecem faltar referencias de distribuídos para colocar algo distante como relacionado

Different approaches have been used to scale scheduling strategies. Hierarchical algorithms will work differently in different levels, exploring parallelism and delivering better performance [3], [4], [9]. Strategies in this domain have been able to ensure scalability so far, but are still limited by a centralized beginning and data dependency.

Also in the hierarchical domain, multilevel graph partitioning has been used to efficiently reschedule tasks [5]. The hypergraph abstraction accurately represents communication, which provides more precision for load balance [10], [11]. However, the cost of this representation is very high for a first task mapping process, being more recommended for repartitioning.

Work stealing is one of the most broadly used techniques for balancing load in parallel systems [12], [13]. The completely distributed essence of this techniques make them very effective scheduling solutions for parallel and distributed irregular applications.

Diffusive load balancing is a completely distributed solution that may benefit iterative applications [6]. These schedulers irradiate work from overloaded PEs to their neighbors, in an effort to achieve load balance in an iterative fashion. Very unbalanced systems suffer with this kind of approach, since the scheduler may take too long to reach a solution.

*Grapevine* is a completely distributed refinement-based scheduling solution. It uses probabilistic transfer of load and epidemic communication protocols to achieve scalability [2]. We intend to use what was created in *Grapevine* and reduce its communication, using the *Task Packing* approach.

Add related work on BinLPT Marcio

Add related work to MigBSP [14] Pilla

| Symbol | Meaning | Definition |
|---|---|---|
| $\rightarrow$ | Remote procedure call | Section III |
| $\Rightarrow$ | Reduction process | Section III |
| $s$ | Compute load in a batch of tasks | Section III-A |
| $T$ | Set of tasks | Equations 2, 3 |
| $load$ | Compute load of the local PE | |
| $load_{set}(T)$ | Load of a set ($T$) | Equation 2 |
| $load_{avg}$ | Average system load of a PE | Section III-A |
| $l$ | Short for $load$ | Equation 1 |
| $v$ | Load threshold in % | Equation 1 |
| $ceil(l,v)$ | Upper bound of $l$ with threshold $v$ | Equation 1 |
| $LT$ | 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 |
| $rand(S)$ | Random element of $S$ | Section III-B |
| $Send(T) \rightarrow G$ | Send a set $T$ to target $G$ | Algorithm 2 |
| $M$ | Mapping of tasks | Equation 3 |
| $P$ | Global set of PEs | Section III-C |
| $Id_l$ | Local PE identifier | Algorithm 3 |
| $\|A\|$ | Number of elements in a set $A$ | |
| $Gossip$ | Start of information propagation | Section III-C |
| $TaskMap$ | Call RTS to start migrations | Algorithm 3 |
| $pack$ | Set of leaving tasks | Section III-C |

TABLE I
IMPORTANT SYMBOLS FOR ALGORITHMS

## 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 it's 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.

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*.

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 for the *PackDrop* strategy 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.

$$ceil(l,v) = (1+v) \times l \tag{1}$$

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

With this information, each PE will take the task with the lower load within its pool, and pack it (lines $3-6$). Then, if the sum of all tasks in the pack is greater then the expected batch size ($s$), the batch is assembled and the strategy starts creating another one (lines $7-10$). The process is repeated while $load$ is greater than the upper bound (line 2).

**Algorithm 1:** Batch Assembly

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

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.

### 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* [15]), 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.

Each subset $b \subset LT$ (as assigned in Algorithm 1) 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).

When *Sends* receive a negative response, Algorithm 2 can be refed with the failed attempts and initiate another round of sends, until every member of $LT$ is migrated.

### C. PackDrop Algorithm

The PackDrop strategy is presented in Algorithm 3. For the sake of 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).

**Algorithm 2:** Batch Sending

**Input:** $LT$, $G$
**Output:** $BG$

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

It 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 map of tasks contains only tasks that are assigned to the current PE.

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

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 Creation process (line 8), further explained in Algorithm 1. Meanwhile, *underloaded* PEs will initiate a *Gossip Protocol* [15] in order to inform other elements they are willing to receive work (line 11). *Gossip* is a well known epidemic algorithm used to spread information on a system, providing fast convergence and near-global awareness of what was shared.

After the information propagation, 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, a PE 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).

*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.

**Algorithm 3:** PackDrop

**Input:** $M$, $load$, $P$, $Id_l$
**Output:** $M'$

```
1  M' ← ∅
2  load_avg ← (AveragePeLoadReduction(load) ⇒ P)
3  ttc ← (TotalTaskCountReduction(|M|) ⇒ P)
4  ats ← load_avg / ttc                    //Average task size
5  v ← 0.05                                //5% precision on balance
6  s ← ats × (2 − |P| / ttc)               //Pack load
7  if load > ceil(load_avg, v) then
8  |   packs ← BatchCreation(ps, T(M), load, v)
9  else
10 |   packs ← ∅
11 |   G ← (Gossip → P)                    //Targets for migration
12 end
13 − − −SynchronizationBarrier − − −
14 //Requests are processed as they are received back
     R ← BatchSend(packs, G)
15 TaskMap(R, M, Id_l)
```

## IV. Analysis of the Algorithm

This section presents an analysis of the Parallel Algorithm 3 (PackDrop). The complexity of the information propagation (*Gossip*) has being evaluated as $O(log_f n)$ [2], where $f$ is the fanout for the algorithm and $n$ is the number of PEs in the system. Here we use $f = 2$, 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. $C(f)$ is refered here as the total cost for a given function $f$. 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:

$$max(C(p_c m), C(c_c log_f n)) \tag{5}$$

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 $O(log_f n)$.

Finally, line 14 will have a complexity equal to the largest number of packs migrated by an overloaded PE. Let $ps$ be the

mean number of tasks inside of a pack, 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:

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

This shows that determining a good $s$ value is crucial to achieve the best performance with this algorithm. Higher values of $s$ will lower communication complexity, but may lead to an imprecise scheduling. In our Implementation, we chose a moderate value of $s$, 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]. `Charm++` is a parallel programming model that provides a *load balancing framework* based on migration of its parallel, message-driven objects, the *chares* [16]. *Chares* are mapped as *tasks* onto PEs and the `Charm++` runtime system (RTS) provides the load information needed for the *PackDrop* 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 (that implements a PUP framework [17]), may use global rescheduling strategies implemented for this RTS. This means that 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 communication topologies, such as ring, meshes and randomized patterns. *LB Test* is known to have a low migration cost, with light *tasks*, and most of its load bound to computation, instead of communication.

The second is a molecular dynamics (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 [2]. 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.

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

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

### B. Other schedulers

*PackDrop* was compared strategies available on the `Charm++` benchmark suite. More specifically, strategies that may be selected by `Charm++`'s workload-aware *Metabalancer* [8]. 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 imbalance isn't too high.
- *Greedy* creates two heaps, one for *tasks* (max-min) and one for PEs (min-max). Then, it assigns tasks to PEs, putting the most work-heavy tasks on 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.

## VI. Performance Evaluation

Two platforms were for performance evaluation of our novel *PackDrop* scheduler: i) A tightly coupled *Supercomputer* 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* with 4 PEs per node, using a Gigabit Ethernet interconnection. Details of both platforms are available on Table II.

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

| Node Info. | Supercomputer | Cluster |
|---|---|---|
| CPUs | $2 \times 12$ | 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` |

Ahead we'll present the metrics used to compare our new global rescheduling strategy with *Greedy*, *Refine* and *Distributed*. Then, we'll discuss results obtained in both platforms descripted in Table II and the scalability of our proposed solution. All raw data of our results, as well as parsing scripts for analysis are publicly available[3].

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

TABLE III
LB TEST MEAN APPLICATION TIME ON THE CLUSTER EXECUTION.

| Scheduler | Time (Ring) | Time (2D) | Time (3D) |
|---|---|---|---|
| Distributed | 47.49298s | 48.64839s | 49.05481s |
| Greedy | 46.54101s | 49.56052s | 51.06850s |
| Dummy | 52.43016s | 53.17254s | 53.94068s |
| PackDrop | 46.81598s | 47.37120s | 47.97426s |
| Refine | 45.49095s | 46.29277s | 47.21924s |

*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, application time.

*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.

### A. Evaluation on Cluster

All experiments executed on cluster were compiled with `Charm++` using the `--with-production` option, combined with the specifications detailed on Table II. 32 homogeneous compute nodes were used, with a total of 128 PEs. In addition to previously mentioned schedulers, *Dummy* was added as the representative of a situation with no remap of tasks, it only aggregates the information a centralized strategy needs (its cost is the base for every centralized strategy).

*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 and Table III. 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 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.

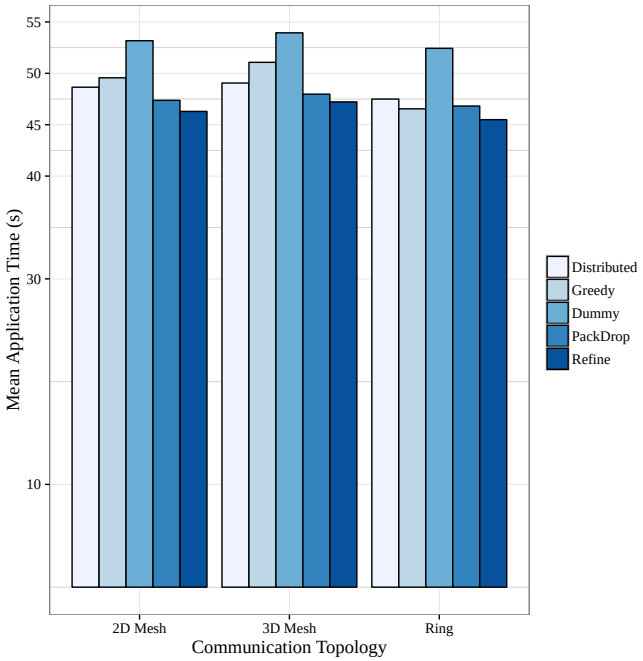


Fig. 1. LB Test cluster execution results.

*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 IV
LEANMD MEAN APPLICATION TIME ON THE CLUSTER EXECUTION.

| Scheduler | Time (Short RP) | Time (Long RP) | Mean LB Time |
|---|---|---|---|
| Distributed | 69.35606s | 68.36055s | 167.0444ms |
| PackDrop | 55.98428s | 55.51554s | 143.1028ms |
| Refine | 59.35696s | 55.89895s | 539.8364ms |

Each configuration of LeanMD was executed 10 times, making a total of 5000 steps per configuration and are depicted in Table IV and in Figures 2, 3. 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.

Figure 3 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 agregation costs and dealing with the high amounts of application data in a centralized fashion. *PackDrop* displays its effectiviness in rescheduling time, outperforming

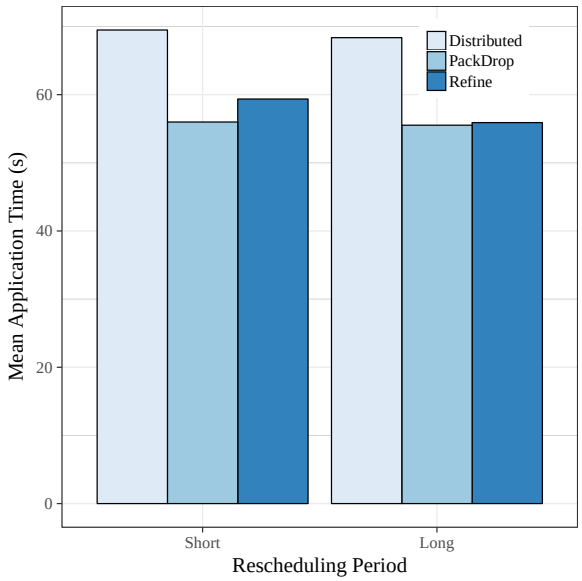


Fig. 2. LeanMD cluster execution results.

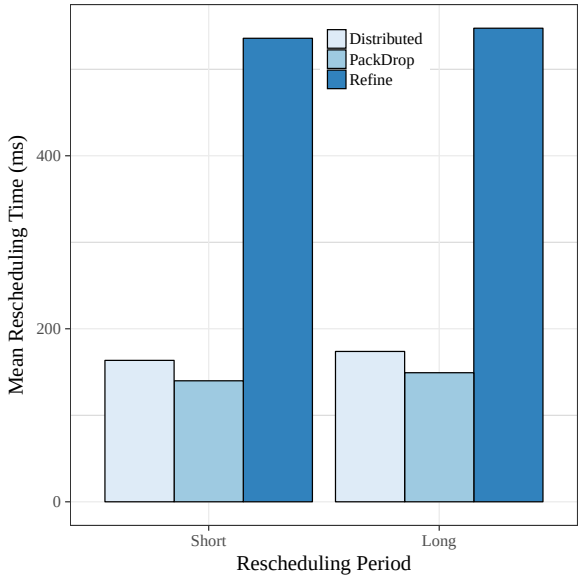


Fig. 3. LeanMD cluster execution results.

both compared strategies, and resulting in an overall better application time.

### B. Evaluation on Supercomputer

All experiments executed on supercomputer were compiled with `Charm++` using the `--with-production` option, combined with the specifications detailed on Table II. Different numbers of homogeneous $2 \times 12$ 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.

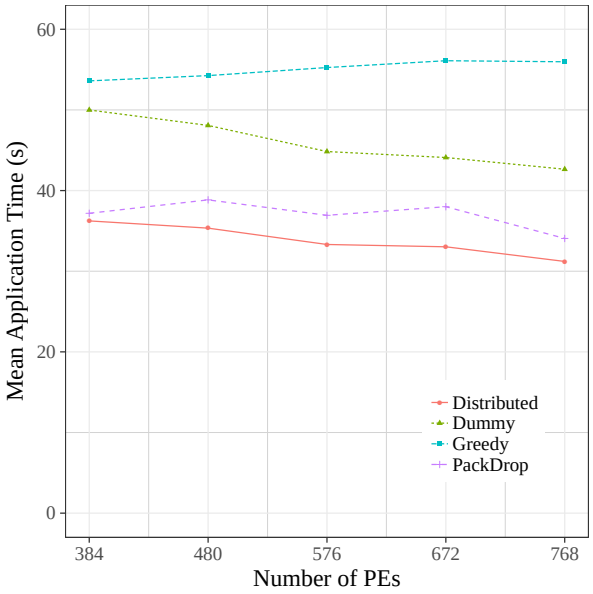


Fig. 4. LeanMD supercomputer AppTime execution results.

Results of mean application time are displayed in Figure 4 and mean rescheduling time in Figure 5. *Refine* was excluded from this evaluation since this LeanMD input presents more data than Refine is able to process in a reasonable time.

*Distributed* benefits from this platform due to the Infiniband low latency communication, which reflects on improved total application times, as seen in Figure 4. Our novel ap-

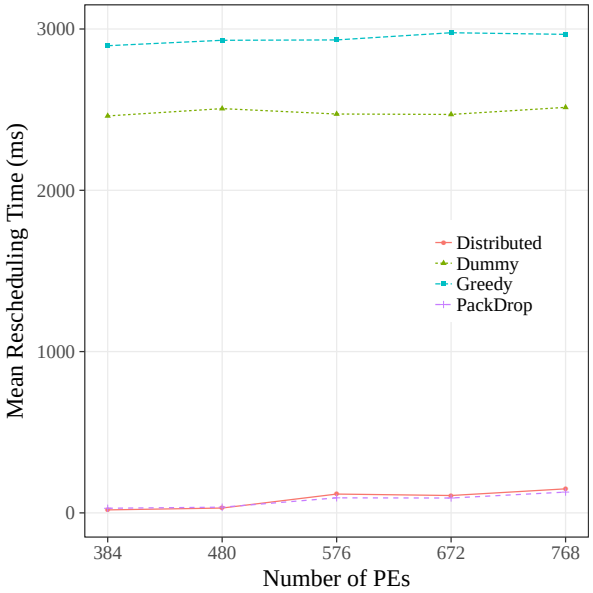


Fig. 5. LeanMD supercomputer SchedTime execution results.

proach, *PackDrop*, followed it closely and we can see that its rescheduling time in larger systems outperforms *Distributed*, displayed in Figure 5.

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, 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.

### C. 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-A1 shows the efficiency of our strategy. 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. Our strategy was only outperformed by *Refine* in all test cases, which is expected, since the total load dealt with is considerably small ($\sim$ 19K tasks) and migrations cheap, which benefits centralized schedulers.

Sections VI-A2 and VI-B1 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 work we have presented the *Batch Task Migration* approach for distributed global rescheduling. It benefits from task communication locality, migrating multiple work units from one source to one destination. This increases communication efficiency in comparison to other workload-aware strategies.

Our approach also mitigates communication costs during algorithm execution time. We do this by transmiting information on multiple migrations at a time. Thanks to this, our novel scheduler (presented in Section III-C) has an increased performance in high communication overhead platforms, discussed in Section VI-A.

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-A).

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, statiscally, 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 *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 proggraming model is something to be considered, prioritizing migration of the edges, instead of random parts of the stencil.

Further work will also be developed in order to increase performance in heterogenous clusters. These may have heterogenous processing capacities and network capabilities, which enhaces complexity of load balancing significantly. In this given scenario, enhancing rescheduling decision processes may be crucial to ensure gains in application performance.

add stencil basis and communication citation

## Acknowledgment

@Laércio, adicionar agradecimentos no contexto do projeto de pesquisa ao CNPq?

Adicionar agradecimentos para ambas as bolsas de mestrado

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[10] 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.

[11] A. Bhatele and L. V. Kale, "Heuristic-based techniques for mapping irregular communication graphs to mesh topologies," in *2011 IEEE International Conference on High Performance Computing and Communications*, Sept 2011, pp. 765–771.

[12] 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

[13] 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.

[14] R. da Rosa Righi, R. de Quadros Gomes, V. F. Rodrigues, C. A. da Costa, A. M. Alberti, L. L. Pilla, and P. O. A. Navaux, "Migpf: Towards on self-organizing process rescheduling of bulk-synchronous parallel applications," *Future Generation Computer Systems*, vol. 78, pp. 272 – 286, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16301145

[15] 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.

[16] 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.

[17] 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.