Evaluation of a Bin Packing Approach for 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

With the arrival of Petascale and highly parallel computing platforms, scientific simulations, such as molecular dynamics, have been able to achieve great advance in their respective fields [1], [2]. However, load imbalance is a recurrent problem in scientific applications, specially in those with an intrinsic dynamic nature [3]. Global rescheduling strategies try to solve this issue, redistributing work amongst Processing Elements (PEs), seeking a more homogeneous state of the system load.

The cost of dynamically rescheduling an application is bound to two main steps: i) a decision step, in which the scheduler remaps tasks to PEs; ii) a migration step, in which the data associated to the tasks is migrated [4]. In a small scale system, the higher cost is associated with migration costs, since communication is limited (less PEs to communicate). As systems scale, the communication costs associated with the decision step may considerably grow. Thus, applications may present significant performance losses when rescheduling strategies do not take into account the communication costs [5].

Centralized global rescheduling strategies are very efficient when aggregating data and dealing with it does not create unexpectedly high migration or decision time overheads. However, as researchers want to execute scientific applications to solve larger problems in highly parallel machines, the overhead of centralized rescheduling may be higher than the effects of load imbalance. This creates a need for scalable schedulers, able to present close to homogeneous distribution of load with a low overhead.

Check new version

Distributed rescheduling strategies use the local information available on PEs to redistribute work without global information. This way PEs take decisions based on a best local scenario, making most of these algorithms *Greedy*. This is advantageous, since scheduling work amongst PEs perfectly is an NP-Hard problem [6].

As Exascale approaches, computing platforms become able of higher levels of scalability. Schedulers have to work with ever increasing amounts of information in these platforms, which demands higher performance and efficiency on load balancing. Even though few distributed strategies exist [5], [7], it is still a field to be explored since each platform and application may benefit from different strategies.

In this paper, we propose and present an evaluation of a *Bin Packing* approach for refinement-based distributed global rescheduling called *Task Packing*. Our novel strategy is based on previous distributed approaches and attempts to minimize the communication during the remapping process. Migrating groups of work-units instead one-by-one exchanges should mitigate most network-related issues, such as network congestion and jitter.

Our results show that ...

Add paper structure.

II. Related Work

Global rescheduling is a well studied theme in high performance computing [3], [5], [8], [9], [10], [7], [11]. 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 [12]. 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.

Different approaches have been used to scale scheduling strategies. Hierarchical algorithms will work differently in different levels, exploring parallelism and delivering better performance [8], [9], [13]. 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 [10]. The hypergraph abstraction accurately represents communication, which provides more precision for load balance [14], [15]. 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 [16], [17]. 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 [7]. 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 [5]. We intend to use what was created in *Grapevine* and reduce its communication, using the *Task Packing* approach.

Add related work on BinLPT

III. The Task Packing Approach

Global rescheduling algorithms must be effective in order to improve scheduling, but should also have low overhead in order to avoid reducing its benefits. To ensure a quick and informed remapping of tasks, we present the *Task Packing Approach*. Sharing global information (e.g. processor affinity, estimated computational load, expected communication patterns, etc) allows a PE to create groups of the best tasks to leave a given processor and take the migration decision with one message, instead of several.

In this work we present the *PackDrop* strategy, a distributed refinement-based technique implementing our new approach. We expect that: i) reducing unnecessary communication; ii) exploring locality of tasks mapped to the same PE, since migrations are done in groups; thus iii) resulting in an accelerated decision making process, which should bring a better overall runtime for applications.

First we will explain the *Pack Creation* (Algorithm 1) and the *Pack 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.

A. Pack Creation

The *Pack Creation* process for the *PackDrop* strategy is presented in Algorithm 1. It uses an estimated pack size (ps), a set of local tasks (T), the current PE $load$ and a threshold for PE loads ($thrs$), to create a set of *Packs*. The threshold is used to calculate an upper bound (ub) of the average system load ($\overline{load}$), using Equation 1.

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

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

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 pack size (ps), the pack is closed and the strategy start filling another one (lines $7-10$). The process is repeated while ub is greater than $load$ (line 2).

Algorithm 1: Pack Creation

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

B. Pack Sending

The *Pack Sending* process is presented in Algorithm 2. The algorithm will use the set of *Packs*, produced by *Pack Creation*, and the set of *Targets*, produced by an information propagation step (*Gossip* [18]), in order to schedule packs on remote PEs. This will produce a set with expected *Pack*/*Target* pairs, which should be confirmed by the remote target.

Basically, while the local PE still has available *Packs* to send (line 2), it will choose a random *Target* for it (line 4) and invoke a remote *Send* procedure on its *target* (line 5). The selected *pack* will be removed from the *Packs* set (line 6) and paired up with its *target* on the R set (line 7), waiting for confirmation. This process is repeated until all elements in *Packs* have attempted a *Send*.

C. The PackDrop Algorithm

The PackDrop strategy is presented in Algorithm 3. It will run individually on each PE, in a distributed fashion. Using a current local mapping of tasks to PEs (M), local load (l) and

Algorithm 2: Pack Sending

Input: $Packs$, $Targets$
Output: R

```
1  R ← ∅
2  while Packs ≠ ∅ do
3  |   pack ← p | p ∈ Packs
4  |   target ← rand(Targets)
5  |   Send(pack) → target
6  |   Packs ← Packs \ {pack}
7  |   R ← R ∪ {(pack, target)}
8  end
```

Algorithm 3: PackDrop

Input: M, l, P
Output: M'

```
1  M' ← ∅
2  al ← (AveragePeLoadReduction(l) ⇒ P)
3  tc ← (TotalTaskCountReduction(|M|) ⇒ P)
4  //Average task size     //5% precision on balance
     ats ← al/tc            thrs ← 0.05
5  ub ← ceil(al, thrs)      //Upper migration threshold
6  ps ← ats × (2 − |P|/tc)          //Pack load
7  if l > ub then
8  |   packs ← PackCreation(ps, T(M), l, thrs)
9  else
10 |   packs ← ∅
11 |   T ← (Gossip → P)      //Targets for migration
12 end
13 − − −SynchronizationBarrier− − −
14 //Requests are processed as they are received back
     R ← PackSend(packs, T)
15 TaskMap(R, M, MyId)
```

knowing all PEs in 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.

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

In this implementation we used two constants: 1.05, in order to limit the imbalance at 5% (on line 5), and 2, in order to regulate the size of packs (on line 6).

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).

Then (line 7) PEs are divided between two different workflows. At this time, *overloaded* PEs will start the Pack Creation process (line 8), further explained in Algorithm 1. Meanwhile, *underloaded* PEs will start a *Gossip Protocol* [18] 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 almost-global awareness of what was shared.

Then, 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, *Pack 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 *Pack Send*, in order to achieve load balance. Once the PEs know their new mappings, tasks are migrated and the strategy is finished (line 15).

PackDrop intends to remap tasks to PEs in a distributed, workload-aware fashion. This approach is the basis for new bin packing 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). The complexity of the information propagation (*Gossip*) has being evaluated as $O(log_f n)$ [5], 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. $T(f)$ 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:

$$max(T(PackCreation), T(Gossip)) \tag{4}$$

We also know that the worst case for *PackCreation* (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:

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

and since c_c is several orders of magnitude bigger than p_c, we could assume $T(PackCreation) \in T(Gossip)$, 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 Task Packing would have a cost of $c_c \times m_l$, while our approach will divide this complexity by ps. Great

gains come with optimizations of this step, since it is the most expensive in Algorithm 3. Our final asymptotic complexity will be:

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

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 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]. `Charm++` is a parallel programming model that provides a *load balancing framework* based on migration of its parallel, message-driven objects, the *chares* [19]. *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 [20]), 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 [5]. 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/OMMITED-FOR-BLIND-REVIEW`

[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 *Meta-balancer* [12]. 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 I.

TABLE I
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`

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 I and the scalability of our proposed solution. All raw data of our results, as well as parsing scripts for analysis are publicly available[3].

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

[3]Available at: `https://github.com/OMMITED-FOR-BLIND-REVIEW`.

TABLE II
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

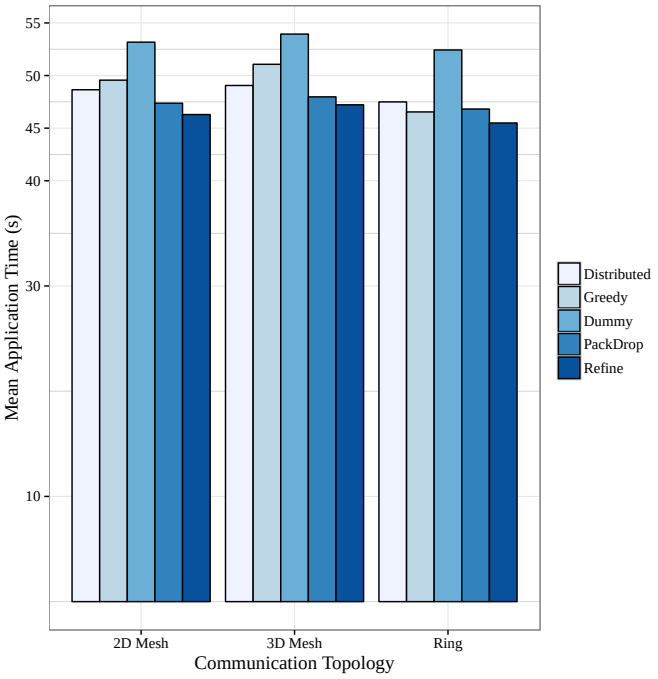

Fig. 1. LB Test cluster execution results.

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 I. 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).

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 II. 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.

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 III
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

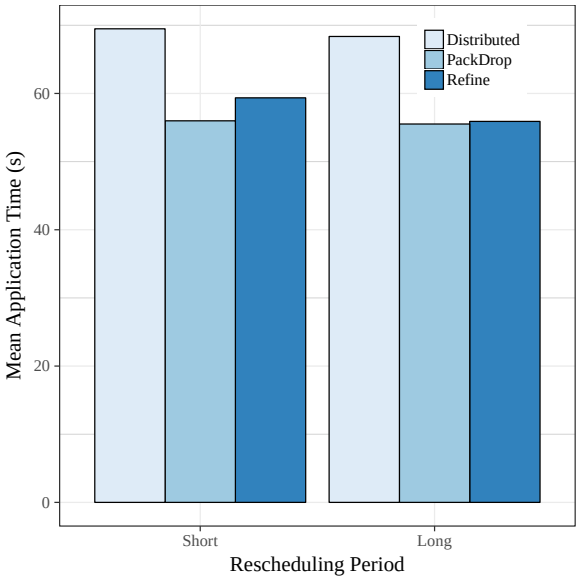

Fig. 2. LeanMD cluster execution results.

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

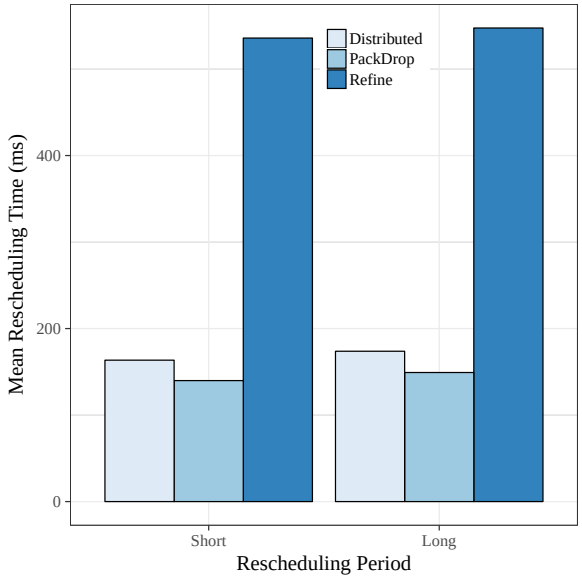

Fig. 3. LeanMD cluster execution results.

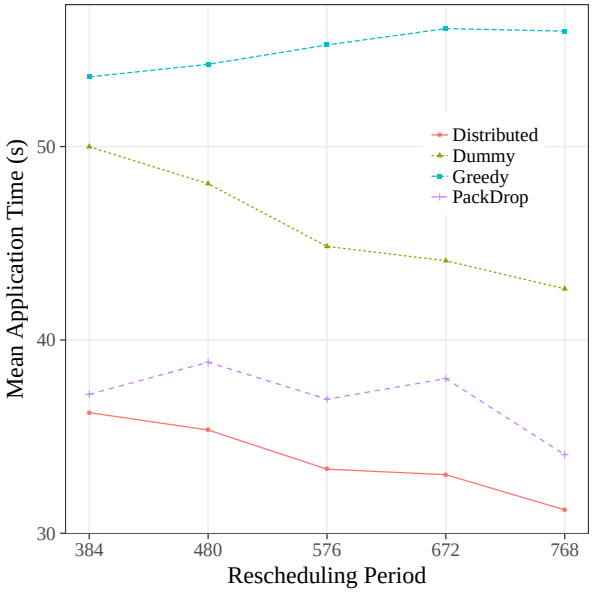

Fig. 4. LeanMD supercomputer AppTime execution results.

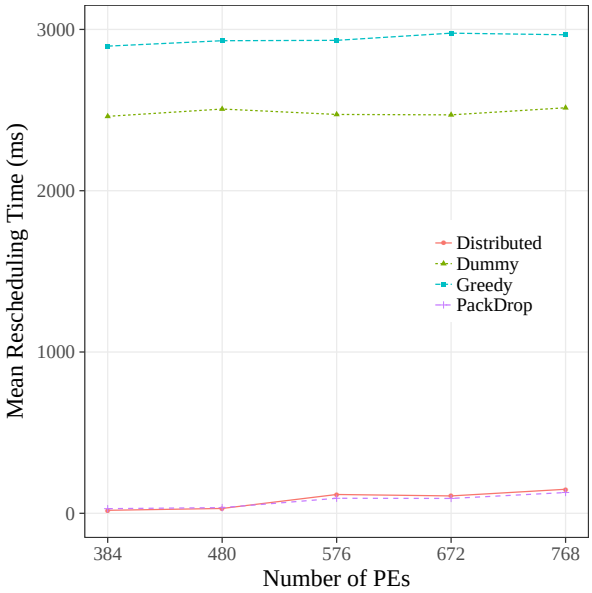

Fig. 5. LeanMD supercomputer SchedTime execution results.

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 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 I. 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.

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.

C. Scalability

VII. Conclusion

A. Future Work

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] 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.

[2] 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.

[3] 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.

[4] L. L. Pilla, "Topology-aware load balancing for performance portability over parallel high performance systems," Ph.D. dissertation, Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, 4 2014.

[5] 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.

[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] 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.

[8] 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.

[9] 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.

[10] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] 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.

[18] 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.

[19] 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.

[20] 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.