Proactive Task Offloading for Load Balancing in Iterative Applications

Minh Thanh Chung[1](✉), Josef Weidendorfer[2], Karl Fürlinger[1], and Dieter Kranzlmüller[1,2]

[1] MNM-Team, Ludwig-Maximilians-Universitaet (LMU), Munich, Germany
{minh.thanh.chung,karl.fuerlinger,kranzlmueller}@ifi.lmu.de
[2] Leibniz Supercomputing Centre (LRZ), Garching, Germany
{josef.weidendorfer,kranzlmueller}@lrz.de

Abstract. Load imbalance is often a challenge for applications in parallel systems. Static cost models and pre-partitioning algorithms distribute the load at the beginning. Nevertheless, dynamic changes during execution or inaccurate cost indicators may lead to imbalance at runtime. Reactive work-stealing strategies can help monitor the execution and perform task migration to balance the load. However, the benefits depend on migration overhead and assumption about future execution.

Our proactive approach further improves existing solutions by applying machine learning to online load prediction. Following that, we propose a fully distributed algorithm for adapting the prediction result to guide task offloading. The experiments are performed with an artificial test case and a realistic application named Sam(oa)^2 on three systems with different communication overhead. Our results confirm improvements for important use cases compared to previous solutions. Furthermore, this approach can support co-scheduling tasks across multiple applications.

Keywords: HPC · Task-based Parallel Models · MPI+OpenMP · Machine Learning · Online Prediction · Dynamic Load Balancing

1 Introduction

Load balancing refers to the distribution of tasks over a set of computing resources in parallel systems. We simplify load as execution time, where the load difference between processes results in imbalance. A process is an abstract entity performing its tasks on a processor. For example, the imbalance can happen when a process waits for the others in bulk-synchronous parallel programs. The primary use case in our paper is represented by iterative applications such as adaptive mesh refinement (AMR) solving partial differential equations (PDEs) [22]. Traditional methods distribute the load at the beginning by using cost indicators. However, an unexpected performance slowdown can lead to a new imbalance. Therefore, dynamic load balancing strategies are more practical to help, such as work-stealing [9]. Work-stealing principally waits until the queue of underloaded processes is empty, then overloaded processes will steal tasks within

© The Author(s) 2023

an agreement. In contrast, the reactive approach monitors execution repeatedly to estimate the load status, and offloads[1] tasks if the imbalance ratio reaches a given condition [13]. The monitored information is the most recent number of waiting tasks on each queue that implicitly represents computing speed per process. Following that, the imbalance ratio is estimated; tasks at an overloaded process can be reactively offloaded to a corresponding underloaded process [23]. Without prior load information, this idea safely fixes a consistent number of offloaded tasks once. Nevertheless, a very high imbalance case is the challenge that can limit reactive load balancing.

We propose a proactive approach for offloading tasks to improve the performance further. The scheme is based on task characterization and online load prediction. Instead of monitoring only queue information, we characterize task features and execution time on-the-fly. Then, we apply this data to train an adaptive prediction model. The prediction knowledge is learned from dynamic change during execution. After that, our proactive algorithm will use the prediction result to guide task offloading. The idea is implemented in a task-based programming framework for shared and distributed memory called Chameleon [13]. We evaluate this work with an artificial benchmark (matrix multiplication) and an adaptive mesh refinement (AMR) named Sam(oa)^2 [18]. Sam(oa)^2 is a hybrid framework PDE systems on dynamically adaptive tree-structured triangular meshes. Variations in computation cost per element are caused by the limiting procedure, space-time predictor, and numerical inundation treatment at coastlines [21]. Our example and implementation can be found in more detail at (See footnote 5). The main contributions are:

– We discuss what limits the existing reactive approaches and define a proactive solution based on load prediction.
– Our approach shows when it is possible to apply machine learning on-the-fly to predict task execution time.
– Then, a fully distributed algorithm for offloading task is proposed to improve load balancing further.

Finally, the rest of paper begins with related work in Sect. 2. Section 3 describes the terminologies of task-based load balancing and problem motivation. Online prediction scheme and proactive algorithm for offloading tasks are addressed in Sect. 4. Finally, Sect. 5 reveals the evaluation and Sect. 6 highlights conclusion with future work.

2 Related Work

Assuming that system performance is stable, load balancing has been studied in terms of static cost models and partitioning algorithms [12] [4]. The balance is achieved by accurately mapping tasks to processors. Our paper focuses on issues after the work has been already partitioned. As mentioned, performance slowdown is a reason for imbalance during execution [27]. There are three classes of

[1] "*Offload*" and "*migrate*" are used interchangeably to denote the migration of tasks.

dynamic load balancing algorithms, centralized [5], distributed, and hierarchical [7]. Work stealing is a traditional approach employed in shared memory systems [2]. For distributed memory, work-stealing is risky because of communication overhead. Researchers attempted to improve communication by using RDMA in PGAS programming models [9,15]. Lifflander et al. introduced a hierarchical technique that applies the persistence principle to refine the load of task-based applications [17]. Focus on scientific applications where computational tasks tend to be persistent, Menon et al. proposed using partial information about the global system state to balance load by randomized work-stealing [19]. To improve stealing decisions, Freitas et al. analyzed workload information to combine with distributed scheduling algorithms [10]. The authors reduced migration overhead by packing similar tasks to minimize messages. Instead of enhancing migration, reactive solutions rely on monitoring execution speed to offload tasks from an overloaded process to underloaded targets[2] [13,23]. The following idea is replication that aims at tackling unexpected performance variability [24]. However, this is difficult to know exactly how many tasks should be offloaded or which processes are truly underloaded in high imbalance cases. Without prior load knowledge, replication strategies need to fix the target process for replicas, such as neighbor ranks. The decision is not easy to make and may get high cost. Using machine learning-based prediction to guide task scheduling is not new. However, the difference comes from the problem feature and applied context. Almost all studies have been proposed in terms of cloud [1] or cluster management [8] using historic logs or traces [3,25] in profilers, i.e., TAU [26], Extrae [20]. Li et al. introduced an online prediction model to optimize task scheduling as a master-worker model in R language [16]. Our context is a given distribution of tasks, and the imbalance is caused by online performance slowdown. Therefore, offline prediction from historical data is insufficient.

3 Preliminaries and Motivation

The many-task runtimes have been studied in shared memory architectures [28]. A task is defined by an entry function and its data (e.g., input arguments). An iterative application has a decomposition into distinct parallel phases of executing tasks. Barriers synchronize each parallel execution phase (so-called time step in numerical simulation). Figure 1(A) illustrates an execution phase, where x-axis represents the time progress, y-axis lists four processes (MPI ranks[3] from 0 to 3), and the green boxes indicate tasks. Each rank has 16 tasks, running by two threads per rank. In general, we define n_t independent tasks per phase, where $T = \{0, ..., n_t - 1\}$ denotes a set of tasks. One task has an associated execution wallclock time ($w \geq 0$) and runs on a specific core until termination. All tasks in T are distributed on n_p processes, where $P = \{0, ..., n_p - 1\}$ denotes a set of processes. The real value of w depends on task's input, CPU frequency,

[2] Underloaded targets/processes indicate victims with an under-average load.
[3] Process/rank refers interchangeably to an entity where tasks are assigned.

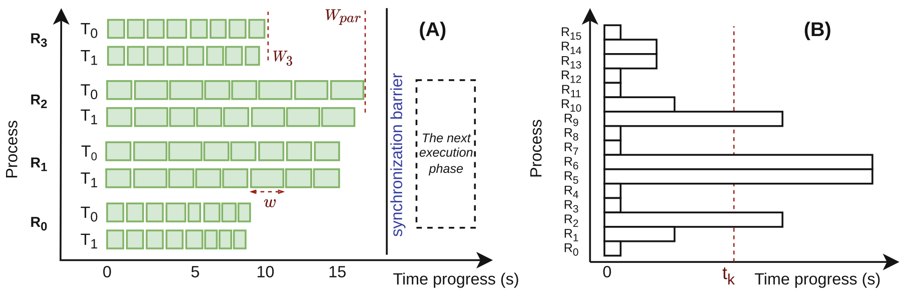

Fig. 1. The illustration of (A) an iterative task-based execution with 4 ranks, 2 threads per rank, and (B) a real load imbalance case with Sam(oa)^2.

or memory bandwidth. Therefore, it can only be measured at runtime. Below, we address some definitions and illustrate their symbols in Fig. 1(A).

- W_i: denotes the wallclock execution time of Rank i. Besides, L_i is a total load of Rank i being the sum of load values of all tasks assigned to Rank i.
- W_{par}: indicates the longest wallclock execution time (the so-called parallel wallclock execution time), where $W_{par} = \max_{\forall i \in P} W_i$.

Thereby, the maximum wallclock execution time (W_{max}) is considered as W_{par}, $W_{min} = \min_{\forall i \in P} W_i$, and the average value is $W_{avg} = avg_{\forall i \in P} W_i$. Load balancing strategies need to minimize the W_{par} value. To evaluate the balance, we use a ratio of the maximum and average W values called R_{imb} in Eq. 1, where $R_{imb} \geq 0$ and a high R_{imb} means a high imbalance.

$$R_{imb} = \frac{W_{max}}{W_{avg}} - 1 \tag{1}$$

In work-stealing, underloaded ranks exchange information with overloaded ranks when the task queues are empty, and tasks can be stolen if reaching an agreement. However, this might be too late in distributed memory because of communication overhead. In contrast, the reactive balancing approach uses a dedicated thread[4]. Based on the most current status, tasks are offloaded by speculative balancing operations early instead of waiting for empty queues [23]. This approach has two strategies: reactive task offloading [14] and reactive task replication [24]. Without prior knowledge, the balancing operation of reactive decisions must be safe at runtime about the number of offloaded tasks and potential victims. In the cases of high imbalance ratio, such as Fig. 1(B) shows, the uncertainty of balancing decision at a time t_k can affect the overall efficiency after execution. This leads to motivation for this work such the following points:

(1) For permanently task offloading, how can we know the appropriate number of tasks to offload?

[4] In hybrid MPI+OpenMP, we can spawn multiple threads per rank. One thread can be dedicated to repeatedly monitoring execution speed and communication.

(2) For victim selection from phase to phase, how can we know the potential victims to offload tasks proactively?
(3) For a long-term vision, it is necessary to learn the variability of communication overhead along with given topology information at runtime.

4 Online Load Prediction and Proactive Task Offloading

4.1 Online Load Prediction

This work exploits a task-based framework of hybrid MPI+OpenMP and a dedicated thread to perform online prediction by machine learning regression model. The results are then adapted to balance load before a new iteration begins.

Where is dataset from? The inputs (IN) are from two sides: application (IN_{app}) and system (IN_{sys}), where IN_{app} is task-related features and IN_{sys} is related to processor frequencies or performance counters. The output is defined by OUT, which can be the wallclock execution time of a task or the total load of a rank in the next execution phases. IN and OUT are normalized from the characterized information at runtime, being used to create a training dataset. Due to domain-specific applications, users should pre-define influence characteristics or parameters. Therefore, we design this scheme as a user-defined tool outside the main library [6].

When is a prediction model trained? Iterative applications can have many execution phases (iterations) relying on computation scenarios. In hybrid MPI+OpenMP model, our dedicated thread runs asynchronously with other threads, which will characterize and collect runtime data in the first iterations on each rank. We simplify in-out features as configuration parameters in the tool. Users can flexibly tune the parameters before running applications. This issue also raises some related questions below.

– Which input features and how much data are effective?
– Why is machine learning needed?
– In which ways do the learned parameters change during runtime?

First, in-out features are based on observing application characteristics. Depending on each use case, it is difficult to confirm how much data are generally adequate. Therefore, an external user-defined tool is relevant for this issue. Second, the hypothesis is a correlation between application and system characteristics that can map to a prediction target over iterations. Also, the repetition of iterative applications facilitates machine learning to learn the behavior. Third, learning models can be adaptive by re-training in the scope of performance variability. However, how many levels of variability make the model ineffective has not been addressed in the paper; this will be extended in future work.

For our experiments, we describe the input and output parameters of online prediction in Table 1. There are two use cases: synthetic matrix multiplication (denoted by MxM) and Sam(oa)^2. In MxM, the matrix size argument of a task mainly impacts its execution time. Thereby, we configure the training inputs

Table 1. The input-output features for training the prediction models.

No.	App.	Task	IN_{app}	IN_{sys}	OUT
1	MxM	MxM kernel	matrix sizes	core freq (Hz)	load/task (w)
2	Sam(oa)^2	grid traversal	previous L_i	$\emptyset$	next L_i

being matrix sizes and core frequency. For Sam(oa)^2, it uses the concept of grid sections where each section is processed by a single thread [18]. A traversed section is an independent computation unit which is defined as a task. Following the canonical approach of cutting the grid into parts of uniform load, tasks per rank are uniform and a set of tasks on different ranks might not have the same load. By characterizing Sam(oa)^2, we predict the total load of a rank in an iteration (L_i^I) instead of the wall clock time of each task (w), where L denotes the total load value of Rank i in Iteration I. To estimate w, we can divide L by the number of assigned tasks per rank. Furthermore, our observation shows that L_i^I can be predicted by the correlation between the current iteration and the previous iterations. For example, suppose Rank 0 has finished Iteration I, and we take the total load values of four previous iterations. In that case, our training features will be the load values from Iteration $I-4$ to $I-1$, such as the following samples $I = 8, 9$.

$$\begin{gathered} \dots \\ L_0^4, L_0^5, L_0^6, L_0^7 \rightarrow L_0^8 \\ L_0^5, L_0^6, L_0^7, L_0^8 \rightarrow L_0^9 \end{gathered} \tag{2}$$

Concretely, the left part of the arrow is training inputs, and the right part is training labels. Other ranks also use this format for generating their dataset.

4.2 Proactive Algorithm and Offloading Strategies

As Algorithm 1 shows, our proactive algorithm uses the prediction results as inputs, where Array L contains the total predicted load, Array N denotes the given number of tasks per rank. The number of ranks (n_p mentioned in Sect. 3) is the size of L, N. First, L is sorted by the load values and stored in a new array $\hat{L}$. Second, L_{avg} indicates the average load, which is considered an optimal balanced value. To estimate how many tasks should be offloaded, Algorithm 1 uses Array R to record the total load of offloaded tasks (so-called remote tasks). Also, Array TB is used to track the number of local tasks (remaining tasks in a local rank) and remote tasks. TB is a tracking table with the same number of rows and columns ($= n_p$), where its diagonal represents the local task number, and the others indicate the remote task number. For example, if the value of $TB[i, j] > 0$ ($i \neq j$), Rank i should offload $TB[i, j]$ tasks to Rank j.

In detail, the outer loop goes forward each victim ($\hat{L}[i] < L_{avg}$). The underloaded value between Rank i and L_{avg} is then calculated, named δ_{under}, which means that Rank i needs a load of δ_{under} to be balanced. The inner loop goes backward each offloader ($\hat{L}[j] > L_{avg}$). The overloaded load (δ_{over}) between

Algorithm 1: Proactive Task Offloading

```
Input : Array $L$, $N$, where each has $n_p$ elements; $L[i]$ is the predicted load, $N[i]$ is the
        number of assigned tasks on Rank $i$.
1  New Array $\hat{L} \leftarrow$ Sort $L$ by the load values
2  $L_{avg} \leftarrow \sum_{i=0}^{n_p-1} \frac{L[i]}{n_p}$
3  New Array $R$; $TB$    /* $R$ has $n_p$ elements denoting the total load of remote tasks per
     rank, $TB$ has $n_p \times n_p$ elements which record the number of local and remote tasks */
4  for $i \leftarrow 0$ to $n_p - 1$ do
5      if $\hat{L}[i] < L_{avg}$ then
6          $\delta_{under} \leftarrow L_{avg} - \hat{L}[i]$          /* the load value under average */
7          for $j \leftarrow n_p - 1$ to $0$ do
8              if $\hat{L}[j] > L_{avg}$ then
9                  $\delta_{over} \leftarrow \hat{L}[j] - L_{avg}$          /* the load value over average */
10                 $\hat{w} \leftarrow$ Estimate the load per task and assert $\delta_{over} \geq \hat{w}$
11                 if $\delta_{over} \geq \delta_{under}$ then
12                     $N_{\mathbf{off}}, L_{\mathbf{off}} \leftarrow$ Calculate the number of tasks to offload and the total
                         load of remote tasks by $\hat{w}, \delta_{under}$
13                 else
14                     $N_{\mathbf{off}}, L_{\mathbf{off}} \leftarrow$ Calculate the number of tasks to offload and the total
                         load of remote tasks by $\hat{w}, \delta_{over}$
15                 end if
16                 Update $\delta_{under}$, $\hat{L}$ at the index $i$ and $j$ based on $N_{\mathbf{off}}$, $L_{\mathbf{off}}$
17                 Update $N[j]$, $R[j]$; $TB$ at the index $(i, j)$, $(j, i)$, $(j, j)$
18                 $Break$ if abs $(\delta_{under}, L_{avg}) < \hat{w}$
19             end if
20         end for
21     end if
22 end for
23 return $TB$
```

Rank j and L_{avg} is then calculated and distributed around. To compute the number of tasks for offloading, we need to know the load per task (w) except in the cases we predict w directly, i.e., in MxM. Otherwise, the load per task can be estimated by the total predicted load over the number of assigned tasks per rank, named $\hat{w}$ at line 10. Afterward, the number of offloaded tasks (N_{off}) and the total offloaded load (L_{off}) are calculated. The following values of δ_{under}, $\hat{L}$, N, R, TB will be updated at the corresponding indices. In line 18, the absolute value between δ_{under} and L_{avg} is compared with $\hat{w}$ to check whether or not the current offloader has enough tasks to fill up a load of δ_{under}. If not, we will go through another offloader. Regarding complexity, if we have n_p ranks in total, where K is the number of victims, $n_p - K$ will be offloaders; then the algorithm takes $O(K(n_p - K))$. As mentioned, our implementation is described in more detail at[5]. For offloading tasks, we use two strategies: round-robin and packed-tasks offloading. Round-robin sends task by task, e.g., Algorithm 1 says that R_0 needs to offload 3 tasks to R_1 and 5 tasks to R_2. It will send the 1st task to R_1, the 2nd one to R_2, and repeat the progress until all tasks are sent. In contrast, packed-tasks offloading encodes the three tasks for R_1 as a package and send it once before proceeding R_2.

[5] https://github.com/chameleon-hpc/chameleon-apps/tree/master/tools/tool_load_prediction.

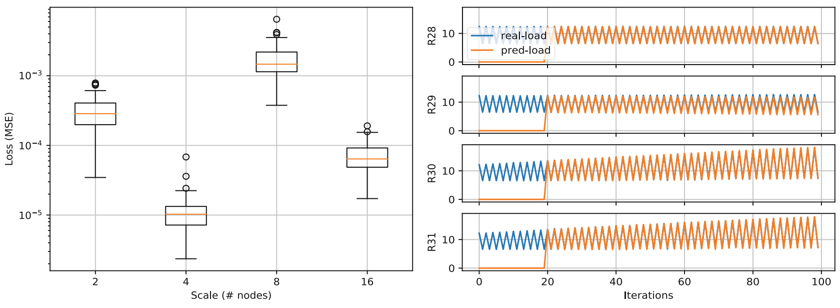

Fig. 2. An evalution of online load prediction for Sam(oa)^2 in simulating the oscillating lake scenario.

Table 2. The overview of compared load balancing methods.

No.	Method	Description
1	baseline	Applications run with default task pre-partition.
2	random_ws	Randomized work-stealing.
3	react_mig	With Chameleon, only reactive migration.
4	react_rep	With Chameleon, only a-priori speculative replication.
5	react_mig_rep	With Chameleon, both reactive migration and replication.
6	proact_off1	With Chameleon, proactive task offloading, round-robind.
7	proact_off2	With Chameleon, proactive task offloading, packed-tasks

5 Evaluation

5.1 Environment and Online Prediction Evaluation

All tests are run on three clusters with different communication infrastructures at Leibniz Supercomputing Centre, CoolMUC2[6], SuperMUC-NG[7] and BEAST[8]. The CoolMUC2 system has 28-way Haswell-based nodes and FDR14 Infiniband interconnect. SuperMUC-NG features Intel Skylake compute nodes with 48 cores per dual-socket, using Intel OmniPath interconnection. In BEAST-system, we use AMD Rome EPYC 7742 nodes with a higher interconnect bandwidth, HDR 200Gb/s InfiniBand.

The first evaluation shows the results of load prediction with Sam(oa)^2. We run 100 time-steps to simulate oscillating lake scenario. Sam(oa)^2 has several configuration parameters that can be found at [18], such as the number of grid sections, grid size, etc. This paper use a default configuration to reproduce the experiments. As mentioned in Subsect. 4.1, the training input features are the total load of the first finished iterations (the dataset from the first 20 iterations). To evaluate accuracy, we use MSE loss [11] between real and predicted values as

[6] https://doku.lrz.de/display/PUBLIC/CoolMUC-2.
[7] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG.
[8] https://www.lrz.de/presse/ereignisse/2020-11-06_BEAST/.

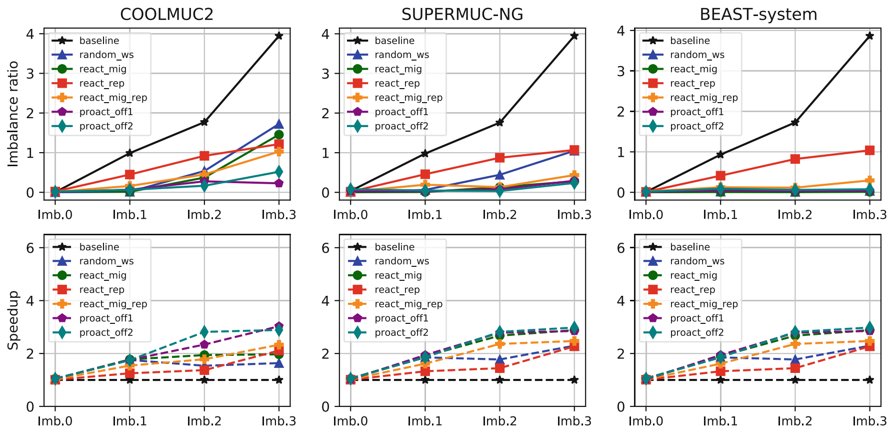

Fig. 3. The comparison of MxM testcases with 8 ranks in total, 2 ranks per node.

the boxplot in Fig. 2 (left). It shows feasibility when using this prediction scheme for load balancing, where x-axis points to the scale of machines, and y-axis is the loss values. Besides, Fig. 2 (right) highlights the comparision between real and predicted load from R_{28} to R_{31} in 16 nodes from Iteration 20 to 99, because we collect data in Iteration 0–19 to generate the training dataset.

5.2 Artificial Imbalance Benchmark

We use the synthetic MxM test cases to ease reproducibility, where tasks are independent and uniform load. The number of tasks per rank is varied to cause different imbalance scenarios. In detail, we generate 4 cases from no imbalance to a high imbalance ratio (Imb.0 - Imb.3). Compared to the baseline and other methods, we name the proposed methods *proact_off1* and *proact_off2* that apply the same prediction scheme and proactive algorithm but different offloading strategies. All compared methods are addressed in Table 2. In Fig. 3, the smaller ratio is the better. It indicates that the W_{par} and waiting-time values between ranks are low. For reactive solutions, *react_mig* and *react_rep_mig* are competitive. However, the case of Imb.3 shows the ratio of ≈ 1.7 with *random_ws*, 1.5–1.1 with *react_mig* and *react_mig_rep* on CoolMUC2. $proact\_off1$ and $proact\_off2$ reduce this under 0.6. On SuperMUC-NG and the BEAST system, the communication overhead is mitigated by higher bandwidth interconnection, showing that the reactive methods are still useful. Corresponding to the *Imb.* values, the second row of charts highlights the speedup values calculated by execution time of each method over the baseline.

5.3 Realistic PDE Use Case with Sam(oa)^2

In this experiment, we vary the number of ranks on each system, where two ranks per node and each rank uses full cores of a CPU socket, e.g., 14 threads per rank on CoolMUC2. For different communication overheads, the tests can show

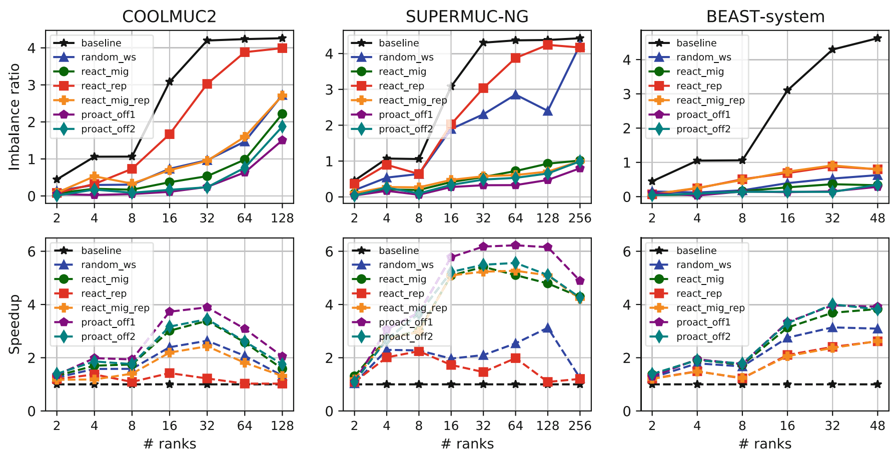

Fig. 4. The comparison of imbalance ratios and speedup in various methods by the usecase of oscillating lake simulation.

scalability and adaptation in various methods. In Fig. 4, reactive or proactive methods obtain higher performance than the baseline. Compared to *react_mig*, speculative replication (*react_rep*) usually comes to some cost. However, their combination *react_mig_rep* could help in the cases from 16 ranks on CoolMUC2 and BEAST. The replication strategy is difficult to deal with the imbalance case of consecutive underloaded ranks. In contrast, our proactive approach uses online prediction to provide information about potential victims. As we can see, $proact\_off1$ and $proact\_off2$ can improve load balancing in the high imbalance cases ($\geq$ 8 ranks). In two offloading strategies, $proact\_off2$ has some delay for encoding a set of tasks when the data is large. Therefore, if an overloaded rank has multiple victims, the second victim must wait long for proceeding the first one. Without any objection, the proactive algorithm must depend on the accuracy of prediction models. However, the features characterized by an online scheme at runtime can reflect the execution behavior flexibly. Therefore, it is feasible to generate a reasonable runtime cost model. Furthermore, we can combine reactive and proactive approaches to improve each other.

6 Conclusion

We have introduced a proactive approach for task-based load balancing in distributed memory systems, which mainly supports the use cases of iterative applications. This approach is enabled by combining online load prediction and proactive task offloading. We proposed a fully distributed algorithm that utilizes prediction results to guide task offloading. The paper shows that existing reactive approaches can be limited in high imbalance use cases by lacking load information to select victims and wisely decide the number of offloaded tasks. Our proactive approach can provide prediction knowledge to make better decisions, e.g., potential victims and how many tasks should be offloaded. We implemented this approach in a task-based parallel library and evaluated it with synthetic and real use cases. The results confirm the benefits in important use cases on three different systems. For a long-term vision, this work can be considered as a potential scheme to co-schedule tasks across multiple applications in future parallel systems. Our solution could work as a plugin on top of a task-based programming framework for load balancing improvement.

Acknowledgment. The authors would like to thank the Chameleon (http://www.chameleon-hpc.org/) and MNM team (http://www.mnm-team.org/) for their support and feedback. Part of the performance results have been obtained on systems in the test environment BEAST (Bavarian Energy Architecture & Software Testbed) at the Leibniz Supercomputing Centre.

References

1. Amiri, M., et al.: Survey on prediction models of applications for resources provisioning in cloud. J. Netw. Comput. Appl. **82**, 93–113 (2017). https://doi.org/10.1016/j.jnca.2017.01.016
2. Blumofe, R.D., Joerg, C.F., et al.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. **30**(8), 207–216 (1995). https://doi.org/10.1145/209937.209958
3. Carrington, L.C., Laurenzano, M., et al.: How well can simple metrics represent the performance of HPC applications? In: Proceedings of the ACM/IEEE Conference on Supercomputing (2015). https://doi.org/10.1109/SC.2005.33
4. Catalyurek, U.V., Boman, E.G., et al.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: International Parallel and Distributed Processing Symposium, pp. 1–11 (2007). https://doi.org/10.1109/IPDPS.2007.370258
5. Chow, Y.C., et al.: Models for dynamic load balancing in a heterogeneous multiple processor system. IEEE Trans. Comput. C-**28**(5), 354–361 (1979)
6. Chung, M.T., Kranzlmüller, D.: User-defined tools for characterizing task-parallel applications and predicting load imbalance. In: 15th International Conference on Advanced Computing and Applications (ACOMP), pp. 98–105 (2021). https://doi.org/10.1109/ACOMP53746.2021.00020
7. Corradi, A., Leonardi, L., Zambonelli, F.: Diffusive load-balancing policies for dynamic applications. IEEE Concurrency **7**(1), 22–31 (1999). https://doi.org/10.1109/4434.749133
8. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and GOS-aware cluster management. SIGPLAN Not. **49**(4), 127–144 (2014). https://doi.org/10.1145/2644865.2541941
9. Dinan, J., Larkins, D.B., et al.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (2009). https://doi.org/10.1145/1654059.1654113
10. Freitas, V., Pilla, L.L., et al.: Packsteallb: a scalable distributed load balancer based on work stealing and workload discretization. J. Parallel Distrib. Comput. **150**, 34–45 (2021)
11. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016). http://www.deeplearningbook.org

12. Karypis, G., Kumar, V.: A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In: PPSC (1997)
13. Klinkenberg, J., Samfass, P., et al.: Chameleon: reactive load balancing for hybrid MPI+OpenMP task-parallel applications. J. Parallel Distrib. Comput. **138**, 55–64 (2020). https://doi.org/10.1016/j.jpdc.2019.12.005
14. Klinkenberg, J., Samfass, P., et al.: Reactive task migration for hybrid MPI+OpenMP applications. In: Parallel Processing and Applied Mathematics, pp. 59–71 (2020). https://doi.org/10.1007/978-3-030-43222-5_6
15. Larkins, D.B., Snyder, J., Dinan, J.: Accelerated work stealing. In: Proceedings of the 48th International Conference on Parallel Processing (2019)
16. Li, J., Ma, X., et al.: Machine learning based online performance prediction for runtime parallelization and task scheduling. In: IEEE International Symposium on Performance Analysis of Systems and Software, pp. 89–100 (2009)
17. Lifflander, J., et al.: 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, pp. 137–148 (2012)
18. Meister, O., Rahnema, K., Bader, M.: Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells. ACM Trans. Math. Softw. (TOMS) **43**(3), 1–27 (2016)
19. Menon, H., Kalé, L.: A distributed dynamic load balancer for iterative applications. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2013). https://doi.org/10.1145/2503210.2503284
20. Munera, A., Royuela, S., et al.: Experiences on the characterization of parallel applications in embedded systems with Extrae/Paraver. In: 49th International Conference on Parallel Processing (2020)
21. Rannabauer, L., Dumbser, M., Bader, M.: ADER-DG with a-posteriori finite-volume limiting to simulate tsunamis in a parallel adaptive mesh refinement framework. Comput. Fluids **173**, 299–306 (2018)
22. Renardy, M., Rogers, R.C.: An introduction to partial differential equations, vol. 13. Springer, New York (2006). https://doi.org/10.1007/b97427
23. Samfass, P., Klinkenberg, J., Bader, M.: Hybrid MPI+OpenMP reactive work stealing in distributed memory in the PDE framework Sam(oa)^2. In: IEEE International Conference on Cluster Computing, pp. 337–347 (2018)
24. Samfass, P., Klinkenberg, J., et al.: Predictive, reactive and replication-based load balancing of tasks in chameleon and Sam(oa)^2. In: Proceedings of the Platform for Advanced Scientific Computing Conference (2021)
25. Sharkawi, S., Desota, D., et al.: Performance projection of HPC applications using SPEC CFP2006 benchmarks. In: International Symposium on Parallel & Distributed Processing, pp. 1–12 (2009)
26. Shende, S., Malony, A.D., et al.: Portable profiling and tracing for parallel, scientific applications using C++. In: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134–145
27. Skinner, D., Kramer, W.: Understanding the causes of performance variability in HPC workloads. In: Proceedings of the IEEE Workload Characterization Symposium, pp. 137–149 (2005). https://doi.org/10.1109/IISWC.2005.1526010
28. Thoman, P., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput. **74**(4), 1422–1434 (2018). https://doi.org/10.1007/s11227-018-2238-4

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.