Received: 25 January 2023 | Revised: 25 April 2023 | Accepted: 22 May 2023

DOI: 10.1002/cpe.7828

SPECIAL ISSUE PAPER

WILEY

From reactive to proactive load balancing for task-based parallel applications in distributed memory machines

Minh Thanh Chung[1] | Josef Weidendorfer[2] | Karl Fürlinger[1] | Dieter Kranzlmüller[1,2]

[1]MNM-Team, Ludwig-Maximilians-Universität München, Munich, Germany

[2]Leibniz Supercomputing Centre (LRZ), Garching, Germany

Correspondence
Minh Thanh Chung, MNM-Team, Ludwig-Maximilians-Universitaet (LMU), Oettingenstr. 67, 80538, Munich, Germany.
Email: minh.thanh.chung@ifi.lmu.de

Summary

Load balancing is often a challenge in task-parallel applications. The balancing problems are divided into static and dynamic. "Static" means that we have some prior knowledge about load information and perform balancing before execution, while "dynamic" must rely on partial information of the execution status to balance the load at runtime. Conventionally, work stealing is a practical approach used in almost all shared memory systems. In distributed memory systems, the communication overhead can make stealing tasks too late. To improve, people have proposed a reactive approach to relax communication in balancing load. The approach leaves one dedicated thread per process to monitor the queue status and offload tasks reactively from a slow to a fast process. However, reactive decisions might be mistaken in high imbalance cases. First, this article proposes a performance model to analyze reactive balancing behaviors and understand the bound leading to incorrect decisions. Second, we introduce a proactive approach to improve further balancing tasks at runtime. The approach exploits task-based programming models with a dedicated thread as well, namely *Tcomm*. Nevertheless, the main idea is to force *Tcomm* not only to monitor load; it will characterize tasks and train load prediction models by online learning. "Proactive" indicates offloading tasks before each execution phase proactively with an appropriate number of tasks at once to a potential victim (denoted by an underloaded/fast process). The experimental results confirm speedup improvements from 1.5× to 3.4× in important use cases compared to the previous solutions. Furthermore, this approach can support co-scheduling tasks across multiple applications.

KEYWORDS

distributed memory, dynamic load balancing, machine learning, MPI+OpenMP, online prediction, task-based parallel models

1 | INTRODUCTION

Load balancing refers to scheduling problems in general.[1] The problem depends on our given contexts. Following the hierarchical classification by Casavant et al.,[2] scheduling can be categorized into "local" versus "global," then "static" versus "dynamic" if we look at the scope and time to take scheduling actions. Load balancing is often regarded as global scheduling. In the case of static, the information about load, tasks, or systems is somehow estimated before execution.[3] In contrast, "dynamic" indicates an imbalance at runtime in which the execution occurs more abnormally than

expected. For our context, the distribution of tasks is given at the beginning on distributed memory machines (so-called nodes).[4] Unexpectedly, the performance slowdown might happen on some processors at runtime.[5] Therefore, some processes execute tasks slower than others, leading to a new imbalance. To deal with the problem, tasks have to be moved around different machines.

One standard solution is work stealing.[6] When a process is idle, it broadcasts the status and asks for stealing tasks. Then, tasks can be migrated if the idle process gets a stealing agreement. The idea has been applied in almost all programming models and has shown benefits in shared memory systems.[7] With distributed memory, work stealing can be limited by migration and communication overhead, for example, latency, transmission time, or unstable bandwidth in practice. There are many efforts to improve the overhead by high-performance network technologies such as InfiniBand.[8] People have introduced RDMA-based migration frameworks over InfiniBand,[9] or implementations of RDMA-based MPI.[10] In another direction, people have proposed reactive load balancing, in which we attempt to monitor load continuously and migrate tasks in advance.[11] The idea is deployed on task-based programming models, where we need a dedicated thread running along with the main execution. The dedicated thread is so-called the communication thread (*Tcomm* as mentioned) for overlapping communication and computation. Meanwhile, *Tcomm* monitors load and notices imbalance among processes/MPI ranks*, it will offload† reactively tasks from a slow rank to another fast rank if available. Modern task-based parallel programming models are concerned because they allow users to abstract computation as tasks with fine-grained parallelism. Besides, the paradigm can employ better hybrid schemes of multithreading + multiprocessing like MPI+X;[12-14] where modern computing architectures support multiple CPU sockets, a socket has multiple cores, and each core hosts a single thread. For our context, task is defined as a code region with its data, and *Tcomm* is deployed using hybrid MPI+X. Reactive solutions use the dedicated thread to monitor the queue status on each process, exchange this information, and make decisions beforehand by offloading tasks.[15] The slow process has a queue size over average, while the fast process is under average. The queue status is checked repeatedly in periods. When the imbalance condition is met, tasks can be offloaded in advance. This is why we can reduce the impact of migration overhead better than work stealing.[11,16] However, reactive operations might be settled wrong in the cases of high imbalance because the most current status only reflects as well as conjectures an unbalanced situation for a short period. We still lack information about load and which process is a potential victim to offload tasks.

The first contribution of this article is formulating a performance model to analyze reactive balancing behaviors. Following that, we can estimate an upper bound of how many tasks should be offloaded at once under the constraints of imbalance level and delay time in task migration. The second contribution is a new proactive approach to balancing the load. The main idea is still to exploit task-based parallel models, but we force the dedicated thread to be busier. Instead of only monitoring execution, it characterizes task features, learns the load value, and predicts the execution time of each process‡. We then adapt this information to guide balancing strategies at runtime. In general, we aim at a scheme of one approach toward more balancing strategies. The approach is called "*proactive*" because if we have load knowledge, we can offload tasks more proactively. Most use cases are parallel iterative applications, where these programs have distinct multiple execution phases. We leave several first iterations to learn behaviors and predict the load of tasks, then apply proactive task-offloading strategies afterward. The experiments are performed on micro-benchmarks and a realistic use case of adaptive mesh refinement simulation (named Samoa[2][17]). The results confirmed the benefits in high imbalance cases with the speed up from 1.5× to 3.4×. Furthermore, our solution has opened a new co-scheduling scheme for balancing tasks across multiple applications.

The rest of the article begins with related work in Section 2. Section 3 describes some terminologies and how we define the problem in terms of task-based parallel. In Section 4, we introduce our proposed performance model to analyze the upper bound of task offloading under the constraint of imbalance level, delay time, and data movement. Following that is the motivation for this work. Section 5 addresses in detail how we design the proactive approach and highlights the idea toward the mentioned scheme of one approach for more balancing strategies. The implementation and experimental results are shown in Sections 6 and 7. Finally, we give an outlook of future work as well as conclusion in Section 8.

1.1 | Extended version of conference paper

This presented work is an extended version of a conference paper published in PPAM22.[18] The conference version introduced our proactive balancing approach for the primary use case, iterative parallel applications. This journal version focuses on how we get to the idea. In detail, we investigate a performance model for dynamic load balancing, leading to a proactive scheme and in-depth analysis.

2 | RELATED WORK

Such a classical problem, distributed load balancing has been studied by analytical and simulation methods.[19] Assume we have prior knowledge about load and the system performance is stable; the most popular studies were in terms of static cost models[20] and partitioning algorithms.[21,22] Nevertheless, this article focuses on issues after the work has been already partitioned. Even though pre-partitioning algorithms are expected with a balance, unexpected factors such as wrong cost model, and performance variability at system[5] can lead to a new imbalance challenge that we are

concerned about in this work. Tuncer et al. have studied an online diagnosis approach for performance variation in HPC systems.[23] Also, Zhao et al. attempt to learn the behavior of computing systems with fluctuated processing speeds for scheduling multi-server jobs.[24]

In terms of scheduling and load balancing without prior knowledge, the most relevant solution is work stealing.[6,7] Among involved processes, the idle one will share the status with other processes and try to steal tasks or work if getting accepted. Work stealing is popular, efficient in shared memory and almost all modern task parallel platforms as well as libraries.[25-28] In distributed memory systems, work stealing is tricky. To keep the stealing ideas still working, researchers attempted to reduce the overhead by exploiting high-speed network technologies known as InfiniBand[10] or RDMA with PGAS programming models.[29-31] Dinan et al. have designed a scalable model for work-stealing using PGAS by the Aggregate Remote Memory Copy Interface (ARMCI),[32] focusing on techniques to reduce locking on the critical path and contention of splitting work.[33] Larkins et al. have introduced an alternative of one-sided RDMA communication called Portals interface[34] to accelerate work stealing in distributed memory.[35] Regarding the approach for reducing migration overhead, Lifflander et al. introduced a hierarchical technique that applies the persistence principle to distribute the load of task-based applications.[36] Menon et al. proposed using partial information about the global system state to improve stealing decisions as well as balance the load by randomized work-stealing.[4] Also, Freitas et al. analyzed workload information to combine with distributed scheduling algorithms.[37] The authors reduced migration overhead by packing similar tasks to minimize messages.

In another direction, reactive solutions are proposed with the idea of migrating tasks reactively in advance. Instead of waiting for a process queue empty, the reactive approach relies on monitoring the queue status to offload tasks from an overloaded process to underloaded targets[§].[11,15] The following idea is task replication that aims at tackling unexpected performance variability.[16] However, this is difficult to know how many tasks should be offloaded at once and which processes are truly underloaded/fast in a short period. Without prior load knowledge, replication strategies need to fix the target process for replicas, such as left/right neighbor ranks. The decision is not easy and may cost higher due to not knowing how many tasks should be replicated. To get knowledge about task execution time, people have investigated load prediction both offline and online. The purpose is to predict load values based on historical or profiled data using machine learning. Almost all studies have been introduced in terms of cloud,[38] or cluster management[39] using historic logs or traces[40,41] from profilers, e.g., TAU,[42] Extrae.[43] Li et al. introduced an online prediction model to optimize task scheduling as a master-worker model in R language.[44] The master-worker model is well-centralized, but it is irrelevant to our case. The paper context is a given distribution of tasks with a new imbalance at runtime caused by performance slowdown. Therefore, offline prediction is insufficient.

3 | TASK-BASED PARALLEL RUNTIMES AND PROBLEM DEFINITION

Load balancing problem mainly depends on the context and constraint. This section gives an overview of task-based parallel runtimes. Then, we define the problem in detail with several terminologies used in the following sections. HPC clusters come to the memory layout divided into shared and distributed memory. Shared memory allows everyone to access the data, which refers to a single machine/node, while distributed memory has distinct spaces that require communication and data transfer. Following that, parallel programming models are developed when we aim for memory layout and performance. For instance, we have single-core, illustrated in Figure 1A, supported by almost all languages. Then multi-core (Figure 1B) gets popular with many supported libraries such as pthreads,[45] OpenMP.[26] Distributed memory, shown in Figure 1C, is the design of current HPC clusters, and MPI is the most popular programming model for communication and data transfer over the network/bus. To be extended, CPU sockets are equipped with accelerators/GPUs; hence, heterogeneous programming paradigms have been developed to support task offloading from the host to the device, for example, CUDA,[46] OpenACC.[47] In general, these models come around threads and process communication. Accordingly, the combination between thread and process for improving data movement is reaching more prevalent, called hybrid models

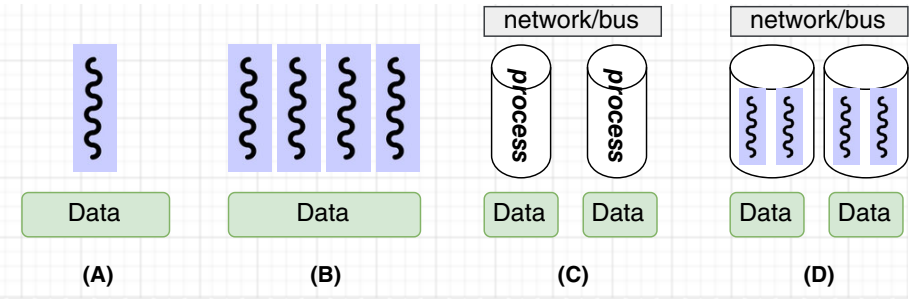

FIGURE 1 From memory layout to task-based parallel programming models. (A) Single-core. (B) Multi-core. (C) Distributed memory. (D) Hybrid.

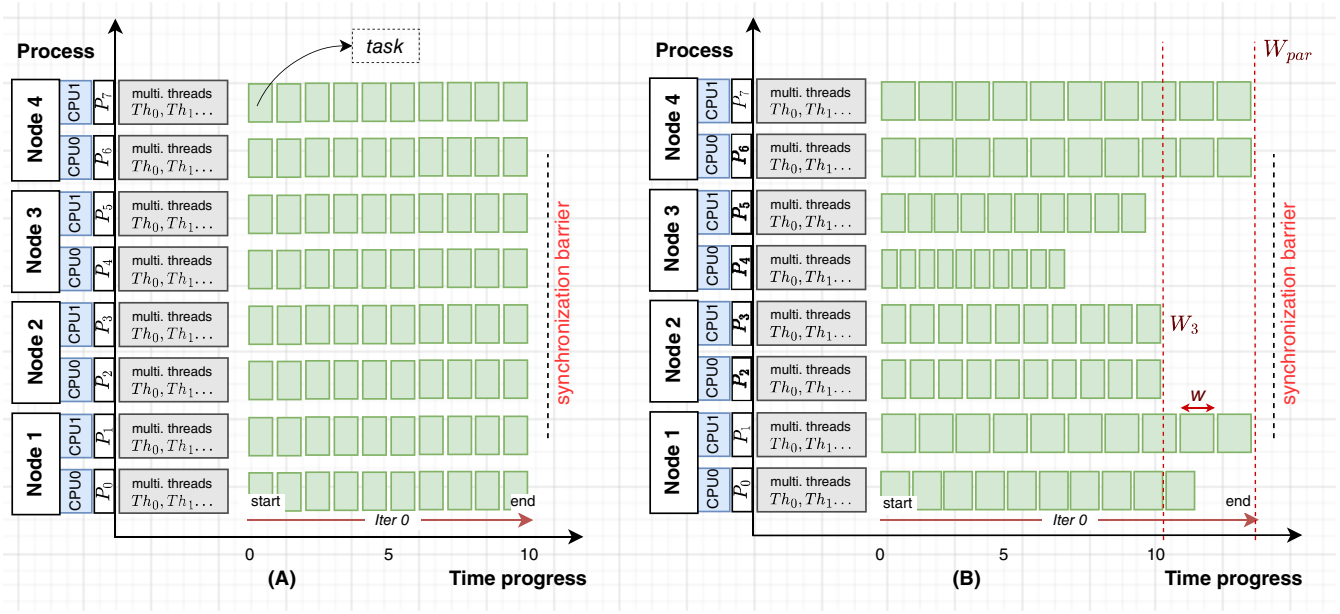

FIGURE 2 An illustration between (A) balanced and (B) unbalanced load in distributed memory.

(Figure 1D) like MPI+X. Task-based programming models are based on this combination to offer programmers an easier way to program in parallel. Recent task-based parallel runtimes allow users to split computation into tasks. A task is defined by its code and data. Thus, programmers can express fine-grained parallelism without too much overhead. Many programming languages support task-based parallelism without external dependencies.[48]

To illustrate the problem, Figure 2A,B reveal the case of balanced and unbalanced load. Each example demonstrates running on four separate compute nodes linked together by interconnection. Each node has two CPU sockets (denoted by *CPU0*, *CPU1*); a socket represents a multicore architecture. Unlike pure MPI programming, users can launch a hybrid model with one primary process and multiple threads inside to execute the program. In Figure 2, the example simply creates one multi-threaded process (so-called MPI rank and denoted by *multi. threads*) per socket. We assume two threads (Th_0, Th_1) per rank in this case. For instance, Node 1 has *CPU0*, *CPU1* corresponding to Rank P_0, P_1 and the two threads Th_0, Th_1 on each execute tasks. With iterative applications in HPC, the program is split into multiple execution phases synchronized by a barrier. Each phase has many tasks, and a given task distribution is performed before running. In both examples, the green boxes indicate task execution, where execution time is considered as the load value. Generally, we address some terminologies to define the problem as follows.

- The problem has T tasks distributed across P processes in total. Depending on system specification and NUMA architecture, the number of processes per node can differ. As mentioned, we exploit a hybrid programming model with task-based parallelism. The model allows multiple threads in a process, and a node can reduce overhead by managing fewer processes. We indicate the number of threads on each process by *nthreads*, where *nthreads* can affect the throughput of executing tasks.
- With a given distribution of tasks, each process will hold a subset T_i of tasks ($\forall\ i\ \in P$). For example, Process 0 is assigned T_0 tasks after the distribution.
- Each task has a wallclock execution time (w) denoting the load value. A task is executed by an execution thread (so-called worker/process) until termination.
- Hence, the total load in a process is calculated by all its assigned tasks and denoted by L, for example, L_i is the total load value of Process i calculated by $\sum_{j \in T_i} w_j$.
- However, T_i tasks in Process i are performed in parallel by *nthreads*. Therefore, the completion time of a process is then estimated by a wallclock execution time (W). For instance, Process 3 finishes at W_3 as Figure 2B shows.
- A process with the maximum W value will be the bottleneck process and define the parallel wallclock execution W_{par}, which is considered as the application time (makespan or program completion time C_{max}). We can see W and W_{par} illustrated in Figure 2B.
- The load values rely on how a task is executed. Assuming the performance model is unstable, different processes might differ from the execution speed. A significant difference will lead to a new imbalance at runtime. We define S_P as the execution speed model of a process.

To evaluate how much is unbalanced, we use the ratio between maximum and average load values. Assuming that the local load in a single process always keeps balanced by multiple threads sharing a subset of tasks, we calculate the imbalance ratio by L among processes or by W as Equation (1) shows.

$$R_{\text{imb}} = \frac{L_{\max}}{L_{\text{avg}}} - 1$$
$$\texttt{or } R_{\text{imb}} = \frac{W_{\max}}{W_{\text{avg}}} - 1 \tag{1}$$

Where "`max`" indicates the maximum value, and "`avg`" indicates the average value. $L_{\max} = \max_{i \in P}(L_i)$, $W_{\max} = \max_{i \in P}(W_i)$ for the max values and $L_{\text{avg}} = (\sum_{i \in P} L_i)/P$, $W_{\text{avg}} = (\sum_{i \in P} W_i)/P$ for the average values. The problem definition is emphasized by reducing the imbalance ratio as well as the completion time ($C_{\max}$) of task-parallel applications. The input before execution is only about general configuration, such as the total number of tasks T, involved processes P, execution threads *nthreads* (workers) per process, and the given distribution of tasks at the beginning, which means a subset of assigned tasks per process is known, T_i ($\forall\, i \in P$). The other information, such as load or runtime, is unknown before execution. The main use case is iterative execution, where the program is distinct by multiple phases of execution, and a global synchronization is performed for updating computation steps over each iteration. The next section will analyze the two most related solutions, that is, work stealing versus reactive load balancing. Following that, we introduce a performance model to estimate the upper bound of how many tasks could be migrated with the bottleneck of migration and communication overhead in distributed memory.

4 | PERFORMANCE MODELING AND MOTIVATION

Unexpected performance variation can affect the given partitioning algorithms because of causing incorrect cost models. That is a reason leading to a new imbalance at runtime. We define S_P as the execution speed to see how much impact the general performance from S_P. For each process, S_{P_i} represents the execution speed, for example, S_{P_1}, S_{P_2} for Process 1, 2, and so on. First, we show the impact on performance if the values of S_P are slowed down in this section. Second, we propose a model to analyze further the related balancing solutions.

As Figure 2 shows in the previous section, both examples include eight processes (indexed from 0 to 7). Assume that there are 2 execution threads per process, and each process is assigned 20 tasks at the beginning, but the figure shows only 10 tasks in a row; where each thread is supposed to get fairly 10 tasks from the queue. If the performance model is expected, the total load will be balanced like Figure 2A. Otherwise, the unbalanced case happens like Figure 2B, where S_{P_0}, S_{P_1}, S_{P_6}, S_{P_7} are slowdown, and S_{P_4} is faster. In this case, the imbalance ratio of the presented iteration is $R_{\text{imb}} \approx 0.24$. In particular, we assume the load information in that iteration (named `Iter 0`) at Case (B) shown in Equation (2).

$$\begin{aligned}
&\text{Node 1 : 20 tasks/process, } P_0(W_0^0 = 11.25, L_0^0 = 22.5) \mid P_1(W_1^0 = 13.75, L_1^0 = 27.5),\\
&\text{Node 2 : 20 tasks/process, } P_2(W_2^0 = 10.00, L_2^0 = 20.0) \mid P_3(W_3^0 = 10.00, L_3^0 = 20.0),\\
&\text{Node 3 : 20 tasks/process, } P_4(W_4^0 = 07.00, L_4^0 = 14.0) \mid P_5(W_5^0 = 09.50, L_5^0 = 19.0),\\
&\text{Node 4 : 20 tasks/process, } P_6(W_6^0 = 13.75, L_6^0 = 27.5) \mid P_7(W_7^0 = 13.75, L_7^0 = 27.5),\\
&\Rightarrow L_{\max}^0 = 27.5, L_{\text{avg}}^0 = 22.25,\\
&\Rightarrow R_{\text{imb}}^0 = \frac{L_{\max}^0}{L_{\text{avg}}^0} - 1 \approx 0.24,
\end{aligned} \tag{2}$$

where, the superscripts of "0" is to mark Iteration 0 because we can have many iterations during runtime. If we calculate the speed ratios compared to the balanced case, S_{P1} is slower than the balanced one $\approx 1.3\times$. When we change the slowdown ratio higher, Figure 3 shows how it impacts the imbalance ratios in different scales. In the first heatmap, we calculate R_{imb} values based on the direction of slowdown scales ($2\times$, $3\times$, …) and the number of slowdown processes (e.g., *num.p.*1 denotes one of 8 processes is slowdown) such as Figure 3 (left). Regarding slowdown scales, the values are ranged from $2\times$ to $9\times$ times. Generally, we can see that the worst cases of imbalance are with one or two slowdown processes. A larger scale results in a higher imbalance ratio. The number of slowdown processes indicates the number of processes/ranks being slower than others in the total of P processes. The second heatmap is the standard deviation between the total load values of all involved processes (Figure 3 (right)).

In distributed memory, task migration is practical if the imbalance happens among separate machines because we cannot increase the number of processes and cores or share the execution threads from one node to another. Therefore, work stealing is a simple and effective solution, but migration overhead might get costly. In terms of interconnect communication in HPC,[49] we summarize some risk factors when moving tasks around, including:

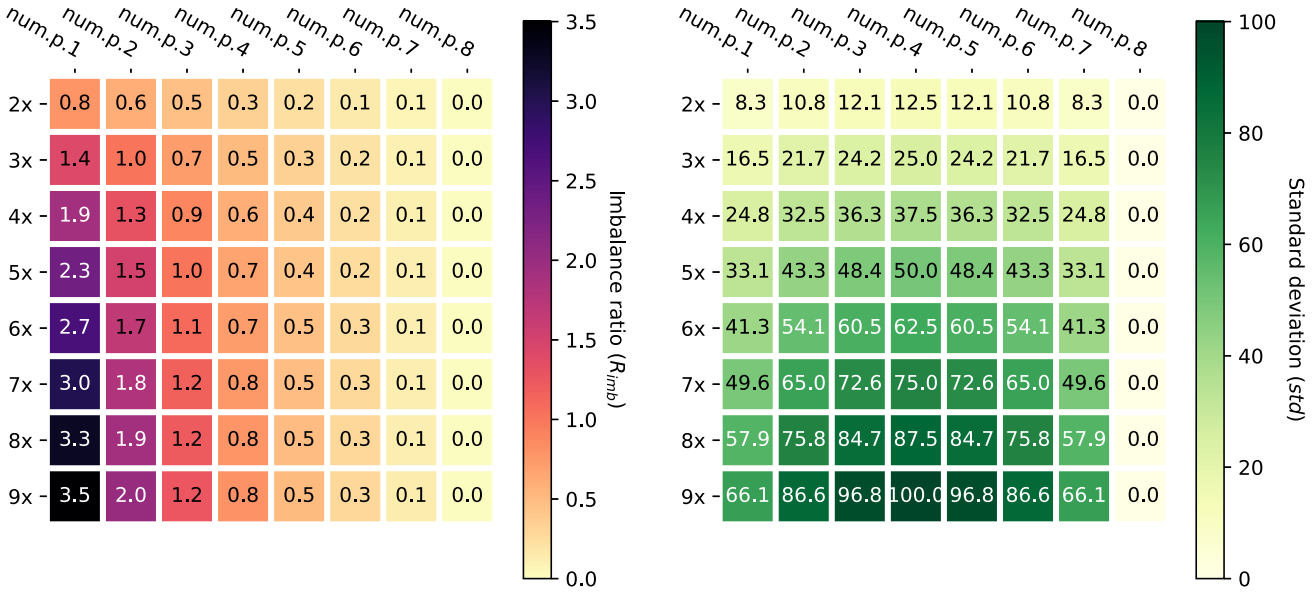

FIGURE 3 An estimation showing how performance slowdown affects imbalance ratio.

- Bandwidth (B): the maximum transmission performance of a network in a certain time. The measuring unit is often megabits, megabytes, or gigabytes per second (Mbps, MBps, GBps).
- Latency (λ): the communication delay time between sending and receiving the head of a message. People often measure the latency in milliseconds or seconds.
- Delay or transmission time (d): is the required time for transferring a whole message between two nodes in a cluster. Particularly, delay time depends on the size of a message (s), and it can be computed as $d(s) = \lambda + \frac{s}{B}$ in the cases of no conflicts. The present work also uses s accounting for the data size of a task. With a constant latency (λ) and a default B value, the delay of migrating a task at a certain time can fluctuate more or less. We are concerned about this factor as a bottleneck. Chiasson et al. have introduced a theoretical model to analyze the effect of delay on load-balancing algorithms.[50]

Alongside communication factors, we summarize the main operations of work stealing and reactive balancing approaches as the most related solutions in Figure 4. Following that, we show how these parameters affect balancing performance.

4.1 | Work stealing

The main idea is demonstrated in Figure 4A. At the time t_k, the queue of P_4 is empty; it then shares that status with the others (shown as operation (1), and we denote it by *Ops*.1). After one of the overloaded processes agrees on the stealing request, tasks can be stolen (shown as operation (2) denoted by *Ops*.2). In detail, we consider the overhead for *Ops*.1 small because of only sharing a small message, but for *Ops*.2 it can take longer depending on the data size of tasks and how many tasks are stolen at once. Besides, another issue is too late when stealing is decided to take action. Therefore, work stealing is limited in distributed memory by the *Ops*.1 and *Ops*.2 overhead.

4.2 | Reactive load balancing

Such an improvement, the main idea of reactive balancing is shown in Figure 4B; instead of waiting until one of the queues is empty, we can reactively offload tasks beforehand. Exploiting the benefit of multicore architectures and task-based parallel programming models, one core is off to dedicate a communication thread (*Tcomm*). It is dedicated only to monitoring the queue status and migrating tasks. As we can see in Figure 4B, *Tcomm* is shown on each process, and the triangles indicate reactive balancing operations. *Tcomm* runs asynchronously with the other execution threads. This scheme can migrate tasks in advance, relying on the monitored information. Because of reactive task migration, we do call "*offloading*" tasks instead

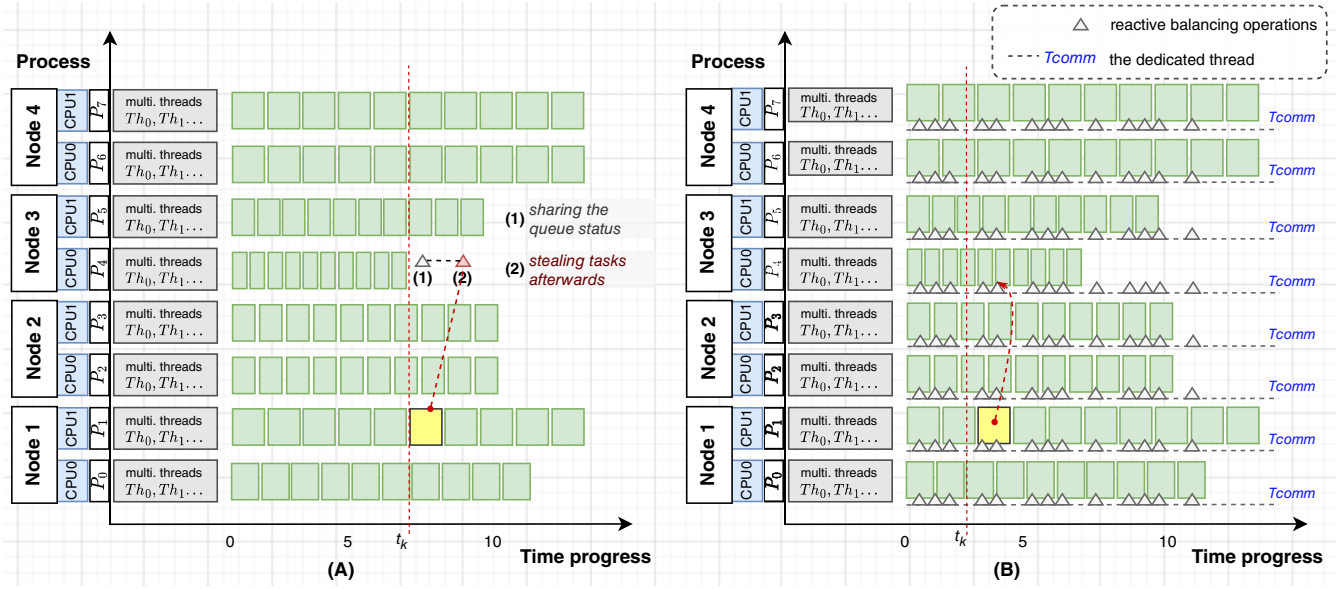

FIGURE 4 An illustration about (A) work stealing versus (B) reactive load balancing behavior.

of "*stealing*," and reactive action is taken from the overloaded/slow processes.[16] As shown in Figure 4B, the decision time of offloading tasks (t_k) is earlier than work stealing in Figure 4A. A detailed example of how tasks are offloaded reactively can be found in Appendix A.

In summary, work stealing or reactive balancing might face some bottlenecks in time to make decisions. Without prior load knowledge, stealing might be too late at runtime, and reactive approach might be wrong with speculative actions.

1. The most current status of execution reflects only a short period of balance or imbalance. Reactive actions at a time also imply that the prediction of imbalance is correct over a short period. Therefore, it is difficult to ensure how many tasks should be offloaded at once or which process is a true potential victim[¶].
2. Concerning transmission time in offloading tasks, this delay can bottleneck reactive decisions. If the delay is large enough, offloading many tasks at a time is not feasible, and offloading fewer tasks is also not good.
3. Topology information still needs to be concerned in the reactive scheme. We can quickly learn which process could be a good candidate for offloading tasks.

To further investigate the bottlenecks, we go from the average bound of task-migration throughput and delay time to a discrete-time model for modeling the reactive balancing operations. Remarkably, we analyze whether communication overhead is only the challenge or if there is something else. We consider delay time an influential factor on the system side because it cannot be controlled manually.

4.3 | Average bound

Theoretically, the ideal balance is an average load for each process, $L_{\text{avg}} = \frac{\sum_{i \in P} L_i}{P}$. With an imbalance situation, we can estimate the sum of overloaded values as well as underloaded values (like Equation 3), where $\sum L_{\text{overloaded}} = \sum L_{\text{underloaded}}$ and

$$\begin{cases} \sum L_{\text{overloaded}} & = \sum_{i \in P} L_i | L_i > L_{\text{avg}}, \\ \sum L_{\text{underloaded}} & = \sum_{i \in P} L_i | L_i < L_{\text{avg}}. \end{cases} \tag{3}$$

Assuming that K is a possible amount of tasks for offloading, then we estimate how K is bounded on average. Each task has a data size s, and the total data size for transfer is $S_{\text{transfer}} = \sum_{i \in K} s_i$. If we call d a delay (transmission time) to offload a task, the total delay is $D = \sum_{i \in K} d_i$. In detail,

- The delay for one task: $d_i = \lambda + \frac{s_i}{B_i}$.
- The total delay for K tasks: $D = \sum_{i \in K} (\lambda + \frac{s_i}{B_i})$.

Considering the average values of s and B in calculation, the total average delay should not exceed the sum of overloaded values divided by the number of underloaded processes. Where, $P_{\text{underloaded}}$ and $P_{\text{overloaded}}$ are named the number of underloaded and overloaded processes. For short, $P_{\text{underloaded}}$ is concerned as M task-offloading channels. The bound of K can be limited by the total overloaded load value sharing across M channels (shown in Equation 4). We consider this as a gap for filling the underloaded load.

$$\begin{cases} \frac{\sum L_{\text{overloaded}}}{M} & \geq K \times (\lambda + \frac{\bar{s}}{\bar{B}}), \\ \Leftrightarrow K & \leq \frac{\sum L_{\text{overloaded}}}{(\lambda + \frac{\bar{s}}{\bar{B}}) \times M}, \\ \Leftrightarrow K & \leq \frac{\sum L_{\text{overloaded}}}{\bar{d} \times M}. \end{cases} \tag{4}$$

4.4 | Does delay challenge load balancing the most in distributed memory?

From Equation (4), we estimate the average bound of K as the number of offloaded tasks under a specific imbalance constraint. This model shows: the period for offloading K tasks between two processes (one offloading and one receiving) should not exceed the average load that we need to exchange between them. K tasks are moved with a delay calculated by λ, the average of task data size and bandwidth ($\bar{s}, \bar{B}$). Thereby, the K values are bounded by the fraction of $\sum L_{overloaded}$ and $\bar{d}$ over the number of underloaded processes ($P_{\text{underloaded}}$ or M for short).

We estimate K by varying the task data sizes, $\bar{s}$. Regarding λ and $\bar{B}$, we use the measured values from real systems, which are three different HPC clusters, namely CoolMUC2[#], SuperMUC-NG[‖], and BEAST[**] at Leibniz Supercomputing Centre. These systems are also used to perform the experiments in Section 7. CoolMUC2 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, the compute nodes are equipped with a higher interconnect bandwidth, HDR 200 Gb/s InfiniBand. Figure 5A shows the latency and bandwidth values performed by OSU Benchmark,[51] where `coolmuc2` is CoolMUC-2, `sng` is SuperMUC-NG, and `beast` indicates BEAST system. The benchmark on each runs with two nodes, and the basic communication interface is MPI point-to-point. The first y-axis shows bandwidth in MB/s; the second y-axis is latency in μs. On the x-axis, we use message sizes from 128 bytes to 256 MB. Such the experiment, BEAST has the best communication performance, while CoolMUC2 is the worst. Hence, K is calculated by using the above latencies and bandwidths. There are three cases of imbalance, where the number of involved processes is kept the same, $P = 8$. Each case corresponds to a slowdown scale and the number of slowdown processes, following the example in Figure 3.

- Case 1 (Figure 5B): slowdown scale is set 5×, the number of slow processes is 2, and $R_{\text{imb}} = 1.5$. This case shows that the number of overloaded processes is larger than the number of underloaded processes, $P_{\text{overloaded}} < P_{\text{underloaded}}$.
- Case 2 (Figure 5C): slowdown scale is set 6×, the number of slow processes is 4, and $R_{\text{imb}} = 0.7$. This case shows $P_{\text{overloaded}} = P_{\text{underloaded}}$.
- Case 3 (Figure 5D): slowdown scale is set 7×, the number of slow processes is 6, and $R_{\text{imb}} = 0.3$. This case shows $P_{\text{overloaded}} > P_{\text{underloaded}}$.

On the x-axis, we set the task size from 400 KB to around 80 MB. The y-axis shows the K values, representing the maximum number of tasks we can offload under the constraint of the total overloaded value ($\sum L_{\text{overloaded}}$). As we can see, K still reaches thousands if tasks are continuously offloaded with the size of ≈ 80 MB. In distributed memory systems, communication overhead is challenging; however, our calculation shows that it is not the most influential factor. Because if we attempt to offload tasks continuously in a row, the migration throughput here is still available on these HPC clusters. Therefore, the bottlenecks must be some other costs from balancing behaviors. To understand the other influential factors, the next paragraph shows further analysis.

4.5 | Discrete time model

As mentioned above, the number of offloaded tasks directly depends on data size, latency, and bandwidth that affect the delay time of task migration. However, the imbalance level and the number of overloaded or underload processes can indirectly affect the bound of migrated task amount because it regulates the period for task offloading. To better understand the balancing operations, we propose a discrete-time model based on queue status to analyze and simulate reactive operations. Figure 6 details reactive balancing operations. The total load value of each process includes the wallclock execution time of local tasks (green boxes) and remote tasks (yellow boxes). "Local" means the original tasks assigned to a process before execution, and "remote" indicates the number of tasks received from the others. Along with the main execution threads, all *Tcomm*s are done when the whole application is finished. Inside *Tcomm*s, we consider three main operations that consume time: monitoring the queue status (T_{monitor}),

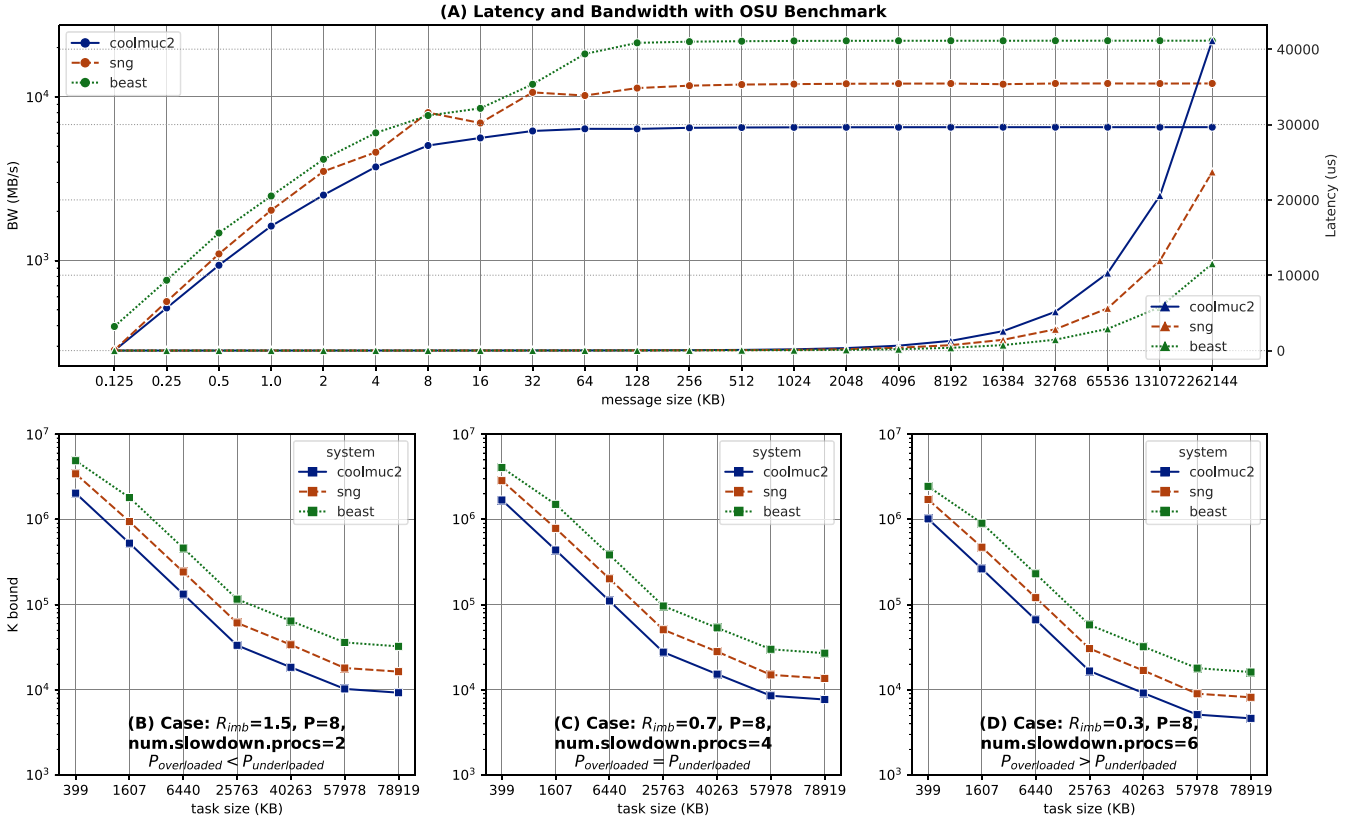

FIGURE 5 An example of the K upper bound with three different imbalance scenarios, where the latency (λ) and bandwidth values (B) are measured from three HPC clusters by OSU benchmark.

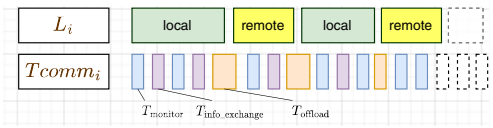

FIGURE 6 Reactive load balance operations for performance modeling.

exchanging the status ($T_{info\_exchange}$), and offloading tasks ($T_{offload}$). Generally, the total load value of each process is denoted by L_i and estimated by Equation (5).

$$L_i = \sum_{j \in T_i'} w_j^{local} + \sum_{k=0}^{K} w_k^{remote}, \tag{5}$$

where, L_i indicates the total load of Process i, and the sub-components include:

- $\sum_{j \in T_i'} w_j^{local}$ denote the total load of local tasks. Because of task offloading, the number of local tasks can be changed. Thus, we use T' to indicate an updated set of local tasks.
- Similarly, $\sum_{k=0}^{K} w_k^{remote}$ addresses the total load of remote tasks. Remote tasks are the tasks received from the offloading processes. K indicates the total number of remote tasks in Process i.

To model L_i as well as estimate $\sum_{j \in T_i'} w_j^{local}$, $\sum_{j \in T_i'} w_j^{local}$, we need to go over the behaviors via discrete time steps called Δt. The status of each process will be formulated by the decrease of queues ($Q_i(t)$). We model the progress of communication threads as a time clock because they run asynchronously with the main execution threads, and a time step is defined by Δt. Given we have P processes in total, the number of processes per

node depends on the configuration of NUMA domains and computing architectures. For example, there might be two or four processes (so-called MPI ranks) per node. To balance the load, each $\mathcal{T}comm_i$ will monitor the queue status (Q_i) at a time t, then exchange that information around to check the imbalance condition.

$$\begin{cases} Q_i(t+\Delta t) = Q_i(t) - \mu_i(t, t+\Delta t) - \sum_{i\neq j} O_{ij}(t) + \sum_{j\neq i} O_{ji}(t-d_{ji}) \\ C_i(t+\Delta t) = \frac{Q_{\max}-Q_{\min}}{Q_{\max}}, C_i(t) > \text{const}(R_{\text{imb}}) \\ M_i(t+\Delta t) = m_i(t, t+\Delta t) + b_i(t-d') \\ \mathcal{T}comm(t+\Delta t) = \mathcal{T}comm(t) + \Delta(t). \end{cases} \tag{6}$$

Equation (6) shows the model, where the decrease of $Q_i(t+\Delta t)$ is associated with the operations of $\mathcal{T}comm$ as shown in Figure 7. The detailed variables and their values are addressed as follows.

- $Q_i(t+\Delta t)$: indicates the queue status of Process i at time $t+\Delta t$ which implies the changes of Q_i in the interval $(t, \Delta t]$.
- $\mu_i(t, t+\Delta t)$: represents the task execution rate in the period Δt, for example, there could be 2, 3, … tasks finished during a time move $(t, \Delta t]$.
- $\sum_{i\neq j} O_{ij}(t)$: is the number of offloaded tasks from P_i (Process i) to P_j at time t, while $\sum_{j\neq i} O_{ji}(t-d_{ji})$ is the number of remote tasks that P_i received from P_j. This is why we need the d_{ji} term to show delay time by task offloading.
- The status of Q_i depends on the main execution threads and $\mathcal{T}comm$. Through the model, we define $M_i(t+\Delta t)$ and $C_i(t+\Delta t)$ based on the behavior of $\mathcal{T}comm$. $C_i(t+\Delta t)$ shows at which time steps the imbalance condition is met to offload tasks reactively. The C_i values impact the number of offloaded tasks in Q_i.
- $M_i(t+\Delta t)$ denotes the overhead for monitoring load ($m_i(t, t+\Delta t)$) and distributing the status information $b_i(t-d')$, from P_i to the others.
- $\mathcal{T}comm(t+\Delta t)$: is used to count time steps as a time clock in this model.

As mentioned, the model terms are illustrated in Figure 7. There are two processes, P_i and P_j. On the y-axis, they are highlighted with the progress bar of total load (L_i, L_j) and the corresponding $\mathcal{T}comms$. The horizontal L bars show how the load value and queue information increase/decrease. The bar of $\mathcal{T}comm$ shows the operations when the load is monitored (driven by the variable $m_i(t+\Delta t)$ as `blue` boxes), status information is exchanged (driven by $b_i(t-d')$ as `purple` boxes), and tasks are offloaded (driven by checking $C_i(t+\Delta t)$ as `yellow` boxes). If we map these actions onto a simulation model, we can emulate the reactive balancing behaviors and estimate the bound of efficiency.

In detail, Figure 8 shows the simulator. We have `Task Model` for defining task (`ID`), execution time (w), and data size (s). `Simulator Engine` manages `Queuing module` associated with the decrease function of queue length ($Q_i(t+\Delta t)$). `Task execution module` controls the execution speed of each process which can be adjusted for performance variability. `Clocking module` indicates the timer for simulation; by default, we can

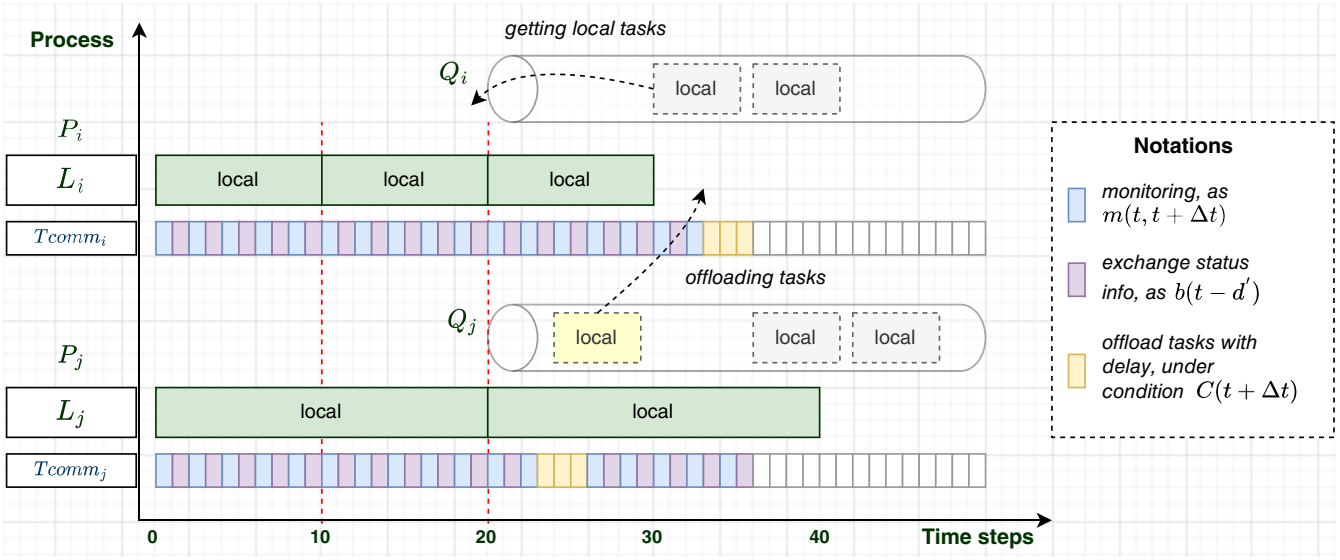

FIGURE 7 An anatomy of reactive load balancing events over discrete time steps.

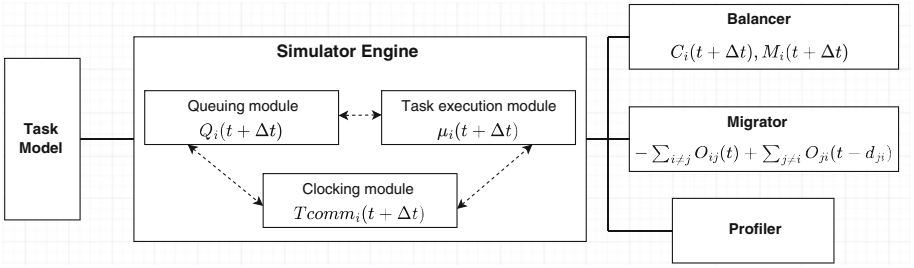

FIGURE 8 A design diagram of reactive load balancing simulator.

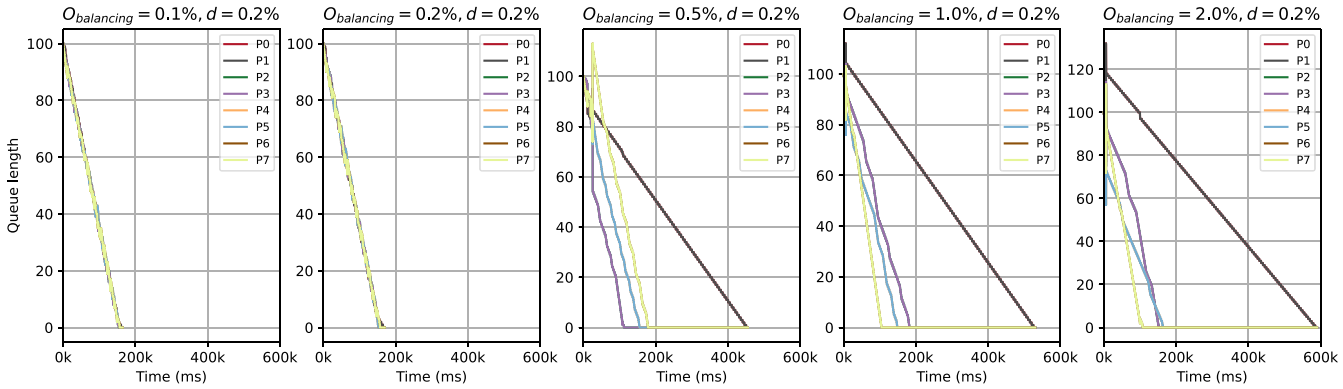

FIGURE 9 An experiment with reactive load balancing simulator under varied balancing overheads.

set a ratio between task runtime with seconds, and the clock counter is in milliseconds such as 1 : 1000. Alongside the simulator engine, there are `Balancer`, `Migrator`, and `Profiler`. `Balancer` represents balancing operations before tasks are decided to offload. `Migrator` controls task offloading with delay time, and we can vary its value to observe the effect. `Profiler` is to help visualize and profile task execution during simulation. We show different model components to emulate reactive behaviors as practically as possible. Nevertheless, this article only concentrates on the effect between the overhead of balancing and delay time over task migration. Therefore, we show an experiment with our simulator based on two simplified parameters as follows.

- Balancing overhead ($O_{\text{balancing}}$): the values can be varied such as 1, 2, 5, … , accounting for 0.1%, 0.2%, 0.5% compared to task runtime. This means that assume a task runtime is set 1 s (1000 ms) as time unit, then $O_{\text{balancing}}$ occupied 1, 2, 5 ms corresponding to 0.1%, 0.2%, 0.5%. Besides, $O_{\text{balancing}}$ represents T_{monitor} and $T_{\text{info\_exchange}}$ shown in Figure 6.
- Delay time (d): is similarly varied such as 1, 2, 5, … , accounting for 0.1%, 0.2%, 0.5%. This parameter represents T_{offload} as shown in Figure 6 or 7 for in detail.

To be consistent, we perform a simulation experiment with 8 processes or ranks, each initially keeping 100 tasks as the given distribution. Two of 8 ranks are configured slowdown than normal, such a bad case of imbalance. The normal ranks are set 1 tasks/second, which means a task runtime is 1 s. For intending to see the impact of $O_{\text{balancing}}$ on reactive balancing, we keep d stable and vary the value of $O_{\text{balancing}}$ in a range [0.1%, 0.2%, 0.5%, 1.0%, 2.0%] accounting for [1, 2, 5, 10, 20 ms]. Figure 9 shows the experiment results with keeping $d = 2$ ms (0.2%). The y-axis denotes the queue length accounting for the number of remaining tasks in a queue before it is converged into 0 through execution. The x-axis shows the execution time progress in milliseconds. We simulate the imbalance case of $R_{\text{imb}} = 1.5$ in Figure 5B, with eight processes in detail, the same distribution of tasks, two processes are slowdown (P_0 and P_1). The queue length of each process is presented by a separate line. When $O_{\text{balancing}}$ is varied, the difference in convergence speed (to 0) is based on reactive balancing behaviors. A faster convergence indicates a better performance. With $O_{\text{balancing}} = 0.1\%$ and 0.2%, it is not difficult to get balanced. However, $\geq 0.5\%$ makes reactive decisions worst. Therefore, along with task migration delay the overhead of balancing operations is sensitive and might critically impact our balancing decisions at runtime.

In addition, our simulator can be extended to analyze other parameters related to the proposed model, such as the number of tasks, involved processes, slowdown scales and so forth. However, we expect future work to investigate these in detail for a mathematical function showing their

relationship. This article only emphasizes the impact between $O_{balancing}$ and d that motivate us to a proactive balancing approach shown in the next section. All in all, the above model summarizes some challenges in reactive balancing as follows.

1. Time to take offloading actions is a main factor causing a late decision or mistake, where delay is only a part. For example, we have to take other operations before deciding to offload tasks, for example, monitoring queue status, exchanging it with other processes, searching for a good victim, and then migrating the task.
2. It's uncertain to decide how many tasks to offload at once. A large number can lead to risk in the end, and a less number also makes us perform many operations.
3. The tasks can be offloaded across nodes; therefore, choosing a good victim to send tasks also support the balancing efficiency. However, getting along with the system topology information at runtime is challenging.

5 | A PROACTIVE APPROACH AND DIFFERENT BALANCING STRATEGIES

The main idea is based on two questions: How can we provide an adaptive knowledge of load at runtime? Then based on that knowledge, how can we offload tasks proactively to balance the load? "Proactive" aims at reducing the number of reactive operations, for example, repeatedly monitoring and sharing the queue status. Besides, we can also offload tasks earlier to relax migration delay. Technically, our approach still exploits the dedicated thread, *Tcomm*, but we force it to do more work, that is, characterizing tasks and learning load online. After that, we use load prediction to guide task offloading. This section introduces a scheme for the approach design in practice. The following subsections will address developing the approach toward different balancing strategies.

5.1 | A scheme for proactive approach

We show a scheme design in Figure 10A. A process is deployed with two components: execution threads and a dedicated thread (*Tcomm*). For iterative applications, the program can run with thousands of execution phases depending on computation algorithms and problem scales. People can exploit task-based parallel models to abstract tasks as fine-grained computation units. At the end of each iteration, we synchronize all processes by a barrier which we can see in many HPC applications and bulk synchronous parallel models. Therefore, the figure shows *Iteration 0* as the first iteration and the following is the next coming. In this scheme, the number of execution threads is ≥1, depending on multicore architectures. *Tcomm* is separate from the execution threads and can invoke different actions during execution. Thereby, we modularize *Tcomm* actions as the callback functions; in detail, Figure 10A illustrates them as boxes with extended arrows. The user applications in our contest are considered as black boxes. Accordingly, the proactive scheme can perform in the background with the following events.

1. Characterization: indicates characterizing task features and system information, including input arguments, data size, code region, core frequencies, related performance counters, or topology information.

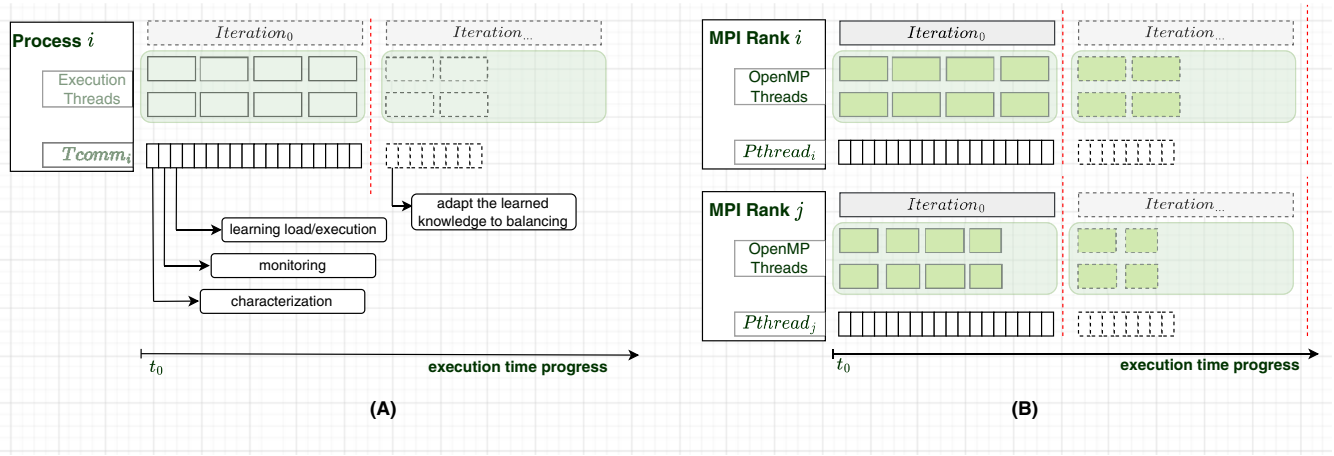

FIGURE 10 Design and implementation for proactive balancing approach. (A) A scheme design for proactive load balancing using one dedicated thread. (B) A reference implementation in practice with hybrid MPI+OpenMP.

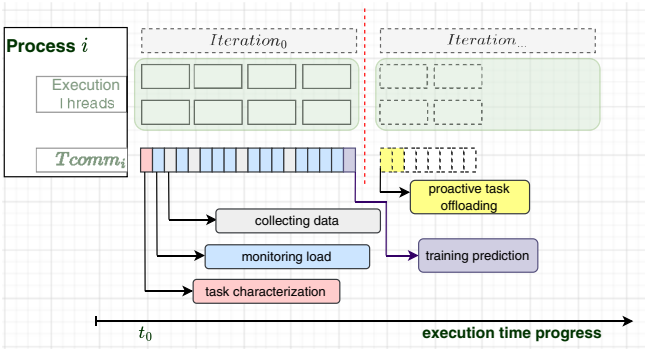

FIGURE 11 ML-based task offloading strategy for dynamic load balancing with *Tcomm*.

2. Monitoring: also measures the queue status and additional values, including task runtime and total load after each iteration.
3. Learning load/execution: performs statistic or trains prediction models with the data from (1) and (2) to predict the load values on-the-fly.
4. Adapting to balancing: aims at transferring the prediction knowledge from (3) into balancing strategies as an input.

We modularize the events in (1)–(4) as callback functions so that users can define or adjust the inputs/outputs to be suitable for domain-specific applications as well as system architectures. From the knowledge based on load information, we can generate better strategies to decide how many tasks should be offloaded at once and which processes are potential victims. Importantly, the prediction and time to adapt it for balancing depend on a specific strategy. We further show our ML-based task offloading strategy in the next subsection and propose another strategy called feedback task offloading as an extension for future work.

Regarding a reference implementation in practice based on the design in Figure 10A, Figure 10B shows the hybrid MPI+OpenMP model. Process *i* can be presented as MPI Rank *i* spawning multiple OpenMP threads for task execution; the last thread is *Pthread* denoting *Tcomm* and pinned to the last core. Also, the figure illustrates two MPI ranks with OpenMP threads and one dedicated *Pthread* on each. Our implementation (named Chameleon) refers to this design, which will be described in the next section.

5.2 | ML-based task offloading

The strategy is called ML-based task offloading, where machine learning (ML) is applied to predict the load of tasks. Then, we use the predicted load information to generate a proactive task-offloading algorithm. We leave the first iterations for characterizing tasks, monitoring load values, and learning to train a prediction model. Before starting a new iteration, we load the prediction and attempt to offload tasks proactively. This strategy provides benefits about a prognostication which process is potential, and how many tasks should be offloaded at once. Besides, the advantage is to offload tasks earlier than the reactive approach. Figure 11 shows the working events of $Tcomm_i$ in this strategy. $Tcomm_i$ is deployed to do

- *task characterization*: the properties of task definition, including input arguments, data size, … This callback event can be invoked at the creating-tasks phase. Users can determine which task features are configured before running the application.
- *monitoring load*: denotes the module for checking load values. Depending on how a prediction model is built, we can track the load value of every single task or the total load value of a process.
- *collecting data*: implies we collect dataset for training machine learning model. Inputs (*IN*) are the influence-characterized information, and outputs (*OUT*) are the predicted load values.
- *training prediction*: calls the module of training prediction models. Each process trains an ML model on its side to predict the load.
- *proactive task offloading*: implies loading the ready prediction model to predict the total load of each process. Then we exchange this information once and perform task offloading early.

In general, the ML-based strategy depends on domain-specific applications. Therefore, we cannot hardcode the configuration for all cases. Users should be able to redefine what should be characterized in their applications. Two related questions and examples below emphasize that this strategy can work flexibly in practice. They are also parts of our experiments shown in Section 5.2.1.

5.2.1 | Where is dataset from?

This means the inputs for training prediction models. The inputs (*IN*) can be collected from two sides: application (IN_{app}) and system (IN_{sys}), where IN_{app} is task-related features (arguments, data sizes) and IN_{sys} is related to processor frequencies or relevant performance counters. The output (*OUT*) emphasizes the load values that might be considered as the wallclock execution time of a task or the total load of a process. During execution, *IN* and *OUT* can be normalized from the characterized information, and transformed into an available dataset. Therefore, we designed the modules shown in Figure 11 as a user-defined tool (or a plug-in) of the main library.[52]

5.2.2 | When is a prediction model trained?

Iterative applications have many iterations (execution phases) relying on computation algorithms. When *Tcomm* generates a prediction model, we need a small setup initially, for example, which features are IN_{app}, IN_{sys}, *OUT*, and which machine learning model is trained. We can simplify them as configuration parameters or users can tune these parameters before execution. Thereby, the follow-up sub-questions can be discussed, for example,

- 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 a specific case, it is difficult to confirm how much data are generally adequate. Therefore, an external user-defined tool is relevant and needs some hints from users. Second, the hypothesis is a correlation between application and system attributes that can map to a prediction target over iterations. Also, the repetition of iterative applications facilitates machine learning to learn their execution behavior. Third, learning models can be adaptive by re-training in the scope of performance variability. However, the article has not addressed how many levels of variability make the model ineffective; this can be extended in future work. Besides, a generative model for load prediction revolves around task and system features might be new directions and open more opportunities in online load prediction and scheduling. Paul et al.[53] have proposed a solution in 2022 for the generative models of HPC applications. In particular, this work uses I/O trace information and proposes two models called feature generator and trace generator to support training generative models. One limitation is only concerned with POSIX I/O traces. Future work is promised by investigating MPI-IO and STDIO.

5.2.3 | Evaluation of online load prediction with MxM and Sam(oa)^2

To be more detailed, we describe the input and output parameters of the online prediction models for two examples shown in Table 1. They are synthetic matrix multiplication (MxM) and Sam(oa)^2. In MxM, the matrix size arguments of a task mainly impact its execution time. Thereby, we configure the training inputs being the matrix sizes and core frequency queried before a task is executed. For Sam(oa)^2, it uses the concept of grid sections where each section is processed by a single thread.[17] A traversed section is an independent computation unit that is defined as a task. Following the canonical approach of cutting the grid into parts of a 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 load of each task (w), where L_i^I denotes the total load value of Rank i after Iteration I. To get w, we can divide L by the number of assigned tasks per rank. Furthermore, our observation shows that the correlation between the current iteration and the previous iterations can predict L_i^I. For example, suppose Rank 0 has finished Iteration I, and we take the total load values of the four previous iterations, $I-4$, $I-3$, $I-2$, $I-1$. Assuming that the current finished iteration is 9, our dataset can be generated as Equation (7) shows. Where the left

TABLE 1 The input–output features for training 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	∅	Next L_i

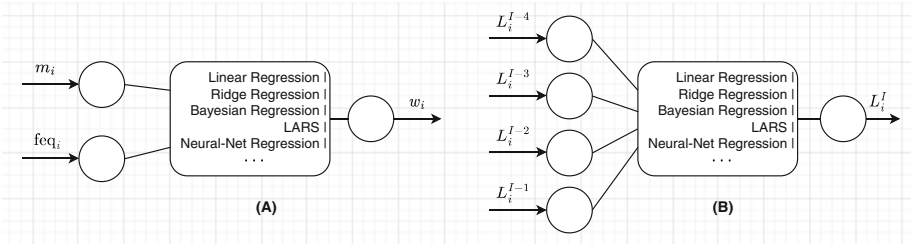

FIGURE 12 Machine learning models for MxM and Sam(oa)^2. (A) ML-regression models for predicting w in MxM. (B) ML-regression models for predicting L in Sam(oa)^2.

TABLE 2 The loss evaluation of online load prediction using different ML-regression models.

	Matrix multiplication (MxM)			
Loss	**MAE**	**MSE**	**RMSE**	**R2score**
Linear regression	0.23947	0.12241	0.34001	0.68828
Ridge regression	0.23916	0.12242	0.33997	0.68828
Bayesian regression	0.26755	0.14175	0.36823	0.68828
LARS	0.26745	0.14175	0.36823	0.68828
	Sam(oa)^2			
Linear regression	0.00179	0.00001	0.00251	0.96883
Ridge regression	0.01027	0.00011	0.01052	0.96812
Bayesian regression	0.00166	0.00001	0.00279	0.96125
LARS	0.04791	0.00248	0.04980	0.96231

part of the arrow is training inputs, and the right part is training labels. Other ranks also use this format to generate the dataset for the case of Sam(oa)^2.

$$\begin{cases} \dots \\ L_0^3, L_0^4, L_0^5, L_0^6 \rightarrow L_0^7 \\ L_0^4, L_0^5, L_0^6, L_0^7 \rightarrow L_0^8 \\ L_0^5, L_0^6, L_0^7, L_0^8 \rightarrow L_0^9 \end{cases} \tag{7}$$

Following the examples of MxM and Sam(oa)^2, we can see that the setup for training the prediction models depends on how users define the input/output features. Input, output, and the chosen models are flexible with the application domains. We evaluate the accuracy of load prediction models through the experiments of MxM and Sam(oa)^2 on CoolMUC2, one of the HPC clusters mentioned in Section 4, Paragraph 4.0.0.5.

Table 2 shows the loss evaluation for predicting load values in MxM and Sam(oa)^2. The average loss values are calculated with MAE, MSE, RMSE, and R2 scores.[54] Such a scheme with the dedicated *Tcomm*, we can try with different machine learning models. The results emphasize that we have tried four different regression models: linear regression, Ridge, Bayesian, and LARS (least angle regression (Stagewise/laSso)). LARS is a stage-wise homotopy-based algorithm for L1-regularized linear regression (LASSO) and L1+L2-regularized linear regression (Elastic Net).[55] Figure 12 illustrates clearer the model view corresponding to our examples and experiments. In (A), it shows MxM with the inputs are matrix sizes (m_i), CPU core frequencies (feq_i), and the output is load per task (w_i). In (B), the inputs are the total load values of the previous iterations, and the output is the following for Sam(oa)^2. The middle of both diagrams is our training algorithms, such as linear regression, Ridge, Bayesian and so forth.

Besides, Figure 13 shows the prediction results of total load for Sam(oa)^2. We configure Sam(oa)^2 with 100 time steps to simulate the oscillating lake scenario. Sam(oa)^2 has several configuration parameters that can be found at Reference 17, for example, the number of grid sections, grid size and so forth. This article uses a default setup to reproduce the experiments. As mentioned, the training input features are the finished iterations,

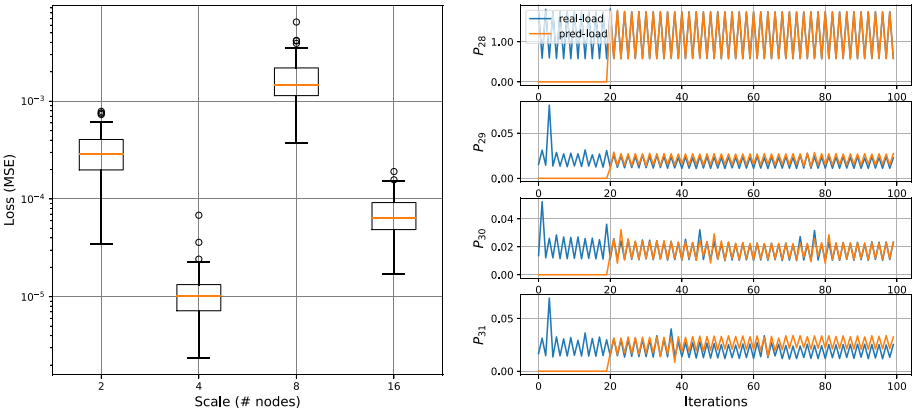

FIGURE 13 An evaluation of online load prediction for Sam(oa)^2 in simulating the oscillating lake scenario.

where we use the 20 first iterations (from 0 to 19), and the current iteration index is 20. Figure 13 on the left shows the evaluation by MSE loss between real and predicted values as the boxplot. Figure 13 on the right highlights the comparison between real and predicted load values from P_{28} to P_{31}. Our configuration is run on 16 nodes, two MPI ranks per node, where the x-axis points to the scale of machines, and the y-axis is the loss value. The comparison is from Iteration 20 to 99 because we collect data in Iterations 0 to 19 to generate the training dataset. These results reveal the feasibility of adapting our prediction scheme to balance load at runtime.

5.2.4 | How can proactive task offloading work?

After the prediction results are ready (for example, at Iteration 20 with Sam(oa)^2 or Iteration 1 with MxM), they will be the input for balancing algorithms. The predicted values give us an estimation of the total load per process as well as the load value per task. This information is used to guide task offloading before a new iteration starts. Specifically, the output will be: the victims (underloaded processes) for offloading tasks and the number of offloaded tasks at once. As shown in Algorithm 1, the input is simplified as the arrays of L and N, where each has P elements accounting for the total load value of each process, and the number of assigned tasks before running the applications. In the first step, array L is sorted and stored into $\hat{L}$. After that, the average load is calculated by $\sum_{i=0}^{P-1} \frac{L[i]}{P}$. We create a new array R to record the remote load values in case tasks are migrated from one original process to another. Similarly, `TABLE` is an array to record the number of local and remote tasks that might be changed step-by-step during task migration.

In detail, the outer loop goes forward to each victim ($\hat{L}[i] < L_{\text{avg}}$). The underloaded value between Rank i and L_{avg} is then calculated, named Δ_{under}, which means Rank i needs a load of Δ_{under} to be balanced. The inner loop goes backward each offloader (overloaded rank, $\hat{L}[j] > L_{\text{avg}}$). The overloaded load ($\Delta_{\text{over}}$) between Rank j and L_{avg} is then calculated. We need the load per task to compute the number of tasks for offloading (w). In MxM, we directly predict the load per task because the matrix size mainly affects its runtime. In Sam(oa)^2, we predict the total load per rank (W); therefore, w of each task can be estimated by dividing W by the total number of assigned tasks. We name the estimated load per task $\hat{w}$, such as at Line 11. After that, the number of offloaded tasks (N_{off}) and the total offloaded load (L_{off}) are calculated. In principle, the N_{off} value can be calculated basically through the division of Δ_{under} by w of the current overloaded process (P_j). However, we should consider how much faster in execution speed is between P_i and P_j when all tasks are uniform but the execution speed is highly different. Or in the case of a high delay time for task migration, the number of offloading tasks can be adjusted by a scale of N_{off} after calculating.

The following values of Δ_{under}, $\hat{L}$, N, R, `TABLE` will be updated at the corresponding indices. At Line 19, the absolute value (denoted by ABS) 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 (the next overloaded process). Regarding complexity, if we have P ranks in total, where Q is the number of victims, $P - Q$ will be offloaders; then the algorithm takes $O(Q(P - Q))$. As mentioned, our implementation is described in more detail at[††].

Further consider offloading strategies

For offloading tasks, we try to use two migration strategies after the proactive task offload algorithm suggests the number of tasks and the potential victim for migration. They are called round-robin and packed-tasks offloading, shown in Figure 14. Round-robin sends task by task, for example, Algorithm 1 says that P_0 needs to offload three tasks to P_1 and five tasks to P_2. It will send the first task to P_1, the second one to P_2, and repeat the progress until all tasks are sent. In contrast, packed-tasks offloading encodes the three tasks for P_1 as a package and send it once before proceeding P_2.

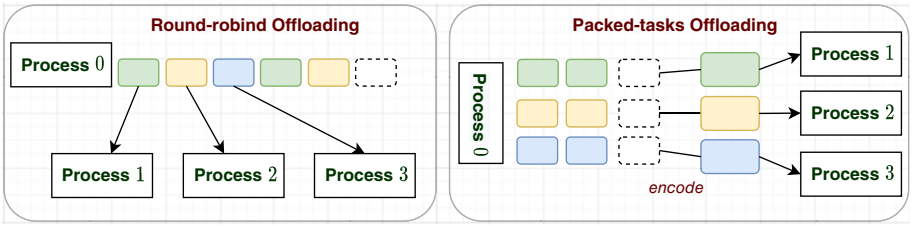

FIGURE 14 Two offloading strategies for task migration after getting the number of tasks and victims.

Algorithm 1. Proactive task offloading algorithm

Input: Array L, N, where each has P elements representing the number of processes; $L[i]$ is the predicted load, $N[i]$ is the number of assigned tasks on Rank i.

Output:

New Array $\hat{L} \leftarrow$ Sort L by the load values
$L_{\text{avg}} \leftarrow \sum_{i=0}^{P-1} \frac{L[i]}{P}$
New Array R; `TABLE`
▷ R has P elements denoting the total load of remote tasks per rank, `TABLE` has $P \times P$ elements which record the number of local and remote tasks
for $i \leftarrow 0$ to $(P-1)$ **do**
  if $\hat{L}[i] < L_{\text{avg}}$ **then**
    $\Delta_{\text{under}} \leftarrow L_{\text{avg}} - \hat{L}[i]$ ▷ the load value under average
    for $j \leftarrow (P-1)$ to 0 **do**
      if $\hat{L}[j] > L_{\text{avg}}$ **then**
        $\Delta_{\text{over}} \leftarrow \hat{L}[j] - L_{\text{avg}}$ ▷ the load value over average
        $\hat{w} \leftarrow$ Estimate the load per task and ASSERT $\Delta_{\text{over}} \geq \hat{w}$
        if $\Delta_{\text{over}} \geq \Delta_{\text{under}}$ **then**
          $N_{\text{Off}}, L_{\text{Off}} \leftarrow$ Calculate the number of tasks to offload and the total load of remote tasks by $\hat{w}, \Delta_{\text{under}}$
        else
          $N_{\text{Off}}, L_{\text{Off}} \leftarrow$ Calculate the number of tasks to offload and the total load of remote tasks by $\hat{w}, \Delta_{\text{over}}$
        end if
        Update $\Delta_{\text{under}}, \hat{L}$ at the index i and j based on $N_{\text{Off}}, L_{\text{Off}}$
        Update $N[j], R[j]$; `TABLE` at the index $(i,j), (j,i), (j,j)$
        BREAK when ABS$(\Delta_{\text{under}}, L_{\text{avg}}) < \hat{w}$
      end if
    end for
  end if
end for

5.3 | Further strategies with feedback task offloading

Our proactive approach implies more balancing strategies that we can drive for task offloading. One potential candidate is called feedback task offloading. The main idea is still to keep operations of reactive load balancing. However, after each execution phase, we review the progress and use statistics to give feedback for balancing the next iterations proactively. This solution expects a probability model for assigning priorities to process victims before tasks are offloaded at runtime. The idea can be addressed as follows.

- Reactive load balancing performs operations based on reaction to the current status of queues among processes. It is more about random and speculative in selecting the victims for offloading tasks. This might lead to wrong decisions.
- Therefore, feedback aims at a probability function for driving victim selection over making decisions on task offloading.

We refer to this strategy as an extension in future work. Nonetheless, the detailed diagram illustrating our idea is shown in Appendix B

6 | IMPLEMENTATION

For reference implementation, this section introduces Chameleon,[11] a task-based parallel framework for applications in distributed memory. We will then show the implementation of our proactive balancing scheme in Section 6.2. It is designed as a plug-in tool upon the main library.

6.1 | Chameleon: A task-based programming framework

Chameleon is a framework supporting task-based programming models in both shared and distributed memory. The implementation is referred as an extended library in C++. Particularly, parallel applications, which follow a bulk-synchronous paradigm, can be supported by Chameleon with overlapping computation and communication phases. The framework uses hybrid MPI+OpenMP, where compute-bound tasks express the computation units. In terms of simplifying tasks in a program, we can define independent tasks or packages without side effects, for example, access to global variables by more than one task. In detail, there are two ways to expose tasks and their data environment. Such a load-balancing target, tasks in Chameleon are migratable.

1. We use a pragma-based approach to extend OpenMP for supporting migratable tasks (*#pragma omp target construct*). This helps Chameleon perform distributed tasking. The map clause is used to identify the input and output data of tasks. Technically, the implementation is portable for adding a custom libomptarget plug-in by the Clang compiler.
2. In arbitrary C/C++, or Fortran compilers, Chameleon provides a manual API to create and add tasks with the given information of their input and output arguments.

```
1
2  void user_defined_task(double *A, int size); // migratable task entry function
3
4  int main()
5  {
6     ... // MPI initialization, allocations ...
7     void* lit_size = *(void**)(&size); // pointer literal representing value of size
8
9     #pragma omp parallel
10    {
11       #pragma omp for nowait
12       for(int i=0; i<num_tasks; i++) {
13          double *A = matrices_a[i];
14
15 #if USE_OPENMP_TARGET_CONSTRUCT
16          #pragma omp target map(tofrom: A[0:size*size])
17          user_defined_task(A, size);
18 #else
19          map_data_entry_t* args = new map_data_entry_t[2];
20          args[0] = chameleon_map_data_entry_create(A, size*size*sizeof(double), MAPTYPE_INOUT);
21          args[1] = chameleon_map_data_entry_create(lit_size, sizeof(void*), MAPTYPE_IN | MAPTYPE_LITERAL);
22
23          cham_migratable_task_t *cur_task = chameleon_create_task((void *)&compute_task, 2, args);
24          chameleon_add_task(cur_task);
25 #endif
26       }
27
28       // trigger parallel execution phase
29       chameleon_distributed_taskwait();
30    }
31    ... // MPI finalization, clean up
32 }
```

Listing 1: An example shows how to generate tasks in application with Chameleon[11].

Listing 1 shows a code snippet detailing how to create tasks in Chameleon. This highlights that the application is considered as a black box; Chameleon takes care of what a task is defined, then schedules them for parallel execution. If an imbalance happens, tasks can be migrated around to balance the load.

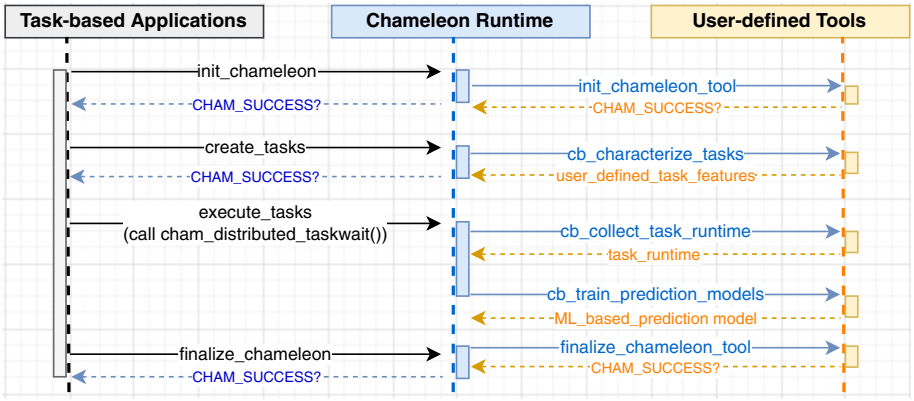

FIGURE 15 Simplified call sequence in the proactive scheme between user application, task-based runtime, and plug-in tool.

6.2 | Plug-in tool for proactive load balancing

The key design of Chameleon is to overlap communication, queue status monitoring, and load balancing during the main execution progress. Therefore, using a dedicated core in each MPI rank for a communication thread asynchronously (defined by *Tcomm*) is essential. This thread will repeatedly monitor load (e.g., computation speed per queue), exchanging the information. If the condition of imbalance ratio at a time is met, task migration decisions will be made. Chameleon is responsible for the parallel task execution phase; whenever the phase has finished, its control is given back to the application side.

Typically, the application programmer can override the internals via defining callbacks of the Chameleon tools interface (like OpenMP Tool interface[56]). Figure 15 shows a simplified call sequence between the application, task-based runtime (Chameleon library), and the callback tool. As we can see, tasks with their arguments are defined by users on the application side. Chameleon's runtime manages task execution in parallel and load-balancing operations. The other side is the Chameleon tool which we design as a plug-in for user-defined tools supporting the proactive scheme. The plug-in tool's interface is to interfere with the functions inside Chameleon driven as callback events. For example, application in the figure is abstracted as a black box, the main functions of task creation and execution are managed by Chameleon's runtime, for example, `cham_distributed_taskwait()` for running tasks. The user-defined plug-in tool is triggered during execution. Along with `create_tasks()`, `cham_distributed_taskwait()`, the callback events can be invoked from the tool, depending on what users pre-defined. Therefore, training machine learning models can work during execution. As mentioned, the tool is possible to adjust by users. For example, we adjust task features that we want to characterize or even change machine learning models, for example, linear regression, Bayesian, or neural network regression.

7 | EXPERIMENTS

As mentioned in Section 4, Paragraph 4.0.0.5, all experiments are performed on three clusters at Leibniz Supercomputing Centre, CoolMUC2, SuperMUC-NG, and BEAST. Technically, the three clusters have different interconnect architectures. While CoolMUC2 has an older interconnection called FDR14 Infiniband, SuperMUC-NG and BEAST feature new technologies, Intel OmniPath and HDR 200 Gb/s InfiniBand. To see the benefits of our proactive approaches, we compare them with the mentioned methods in Table 3. In detail,

- `baseline`: means no load balancing; the application itself has its default pre-partitioning algorithm for task distribution.
- `random_ws`: indicates randomized work stealing.
- `react_off`: is reactive task offloading as described in Section 4.
- `react_rep`: is a variant of react_off. Instead of task offloading, we replicate the tasks reactively with the dedicated thread (*Tcomm*).
- `react_off_rep`: is another variant of react_off when we combine both reactive task offloading and task replication.
- `proact_off1`: denotes our proactive approach with the first task offloading strategy, namely round-robin.
- `proact_off2`: denotes our proactive approach with the second task offloading strategy, namely packed-tasks.

Finally, we perform the experiments with a synthetic micro-benchmark (matrix-multiplication) and a real use case named Sam(oa)[2]. In this article, we are more interested in the proactive approach of ML-based task offloading; therefore, feedback task offloading is considered as future work.

7.1 | Artificial benchmark

For MxM, the experiment is easy to reproduce with tasks defined by MxM compute kernel, where the tasks are independent and have uniform load. Due to permission on the clusters at Leibniz Supercomputing Centre, we cannot adjust the core frequency to emulate performance variability, and the frequency is configured at a fixed level. Therefore, to be reproducible, we create different imbalance scenarios by varying the number of MxM tasks per rank. In detail, we generate four cases from no imbalance to high imbalance ratios (Imb.0 - Imb.3). Compared to the `baseline` and other methods, we use `proact_off1` and `proact_off2`. They have applied the same proactive scheme for predicting and balancing the load but different migration strategies. All compared methods are addressed in Table 3. In Figure 16, the smaller imbalance ratio is the better. It indicates that the W_{par} and waiting-time values between ranks are low. For reactive solutions, `react_off` and `react_off_rep` are competitive. However, the case of Imb.3 shows the ratio of ≈ 1.7 with `random_ws`, 1.5 - 1.1 with `react_off` and `react_off_rep` on CoolMUC2. `proact_off1` and `proact_off2` reduce this under 0.6. On the BEAST system, communication overhead is mitigated by higher bandwidth interconnection, showing that the reactive methods are still robust. Corresponding to the *Imb.* values, the second row of charts highlights the speedup values calculated by the

TABLE 3 The overview of compared load balancing methods.

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

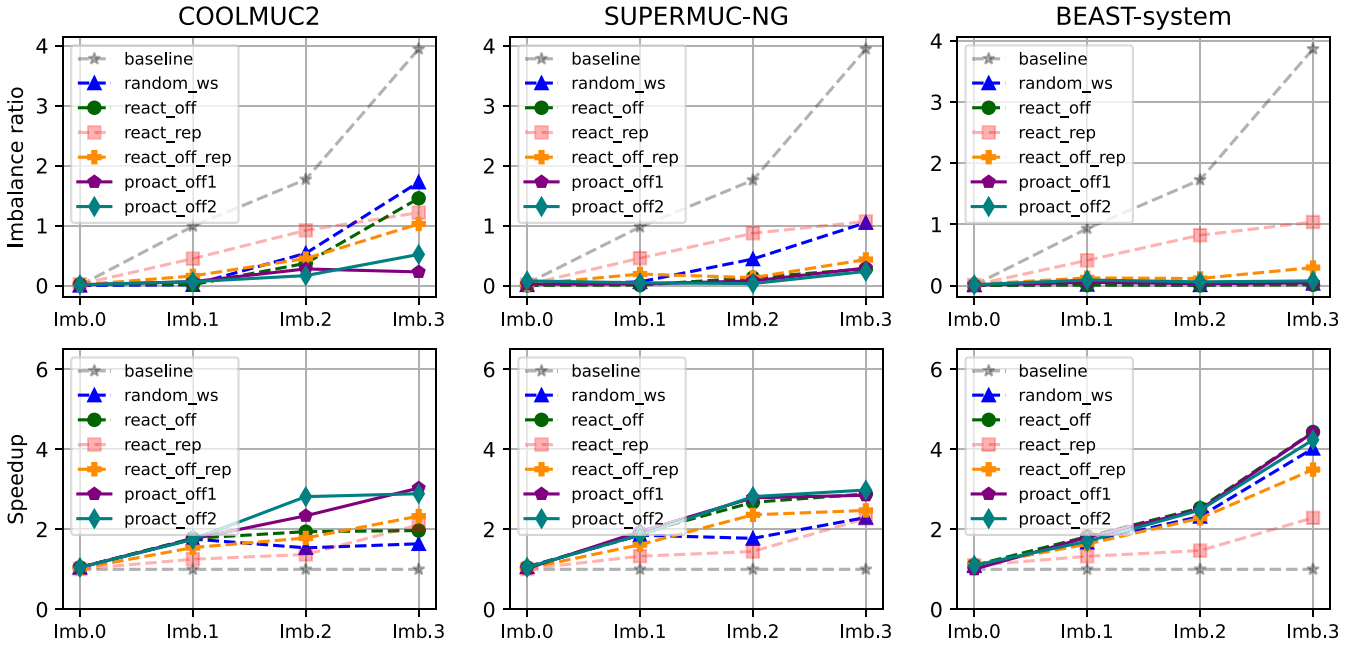

FIGURE 16 The comparison of MxM test cases with 8 ranks in total, 2 ranks per node.

TABLE 4 Case 2: Imbalance ratio (`imb.2`).

Average	COOLMUC2	SUPERMUC-NG	BEAST-system
∑ (#migrated tasks) over ranks	965.20	1017.60	1179.60
#balancing calculation calls	47965.85	10755.78	202010.57
Cost/balancing calculation (μs)	0.313387	0.314263	0.126097

TABLE 5 Case 3: Imbalance ratio (`imb.3`).

Average	COOLMUC2	SUPERMUC-NG	BEAST-system
∑ (#migrated tasks) over ranks	592.00	566.80	709.40
#balancing calculation calls	10534.00	3201.38	37528.60
Cost/balancing calculation (μs)	0.403371	0.309864	0.129224

execution time of each method over the baseline. The proactive approaches gain approximately 1.5× speedup compared to `react_off`, and 1.8× to `random_ws` on CoolMUC2. For SuperMUC-NG, they get 1.6× compared to `random_ws`, but only ≈ 1.1× to `react_off`. The reactive approaches on SuperMUC-NG and BEAST still work well in this case.

To explain the reactive approaches, which are still robust on BEAST, we present the detailed information in Tables 4 and 5. We collect the profiled data summarizing the balancing overhead of reactive balancing methods based on an average statistic. The two tables show the profiled data in Case `imb.2` and Case `imb.3`. The first value is the average sum of the number of migrated tasks among processes (∑ (#migrated tasks)), which gives an overview of how many tasks are moved in a case. A large or small number of this value does not mainly indicate good or bad migration because there could be some wrong task migration at an incorrect reactive decision. However, this could show the availability and capability of each system in migrating tasks. The second value is the average number of reactive balancing calculation calls (#balancing calculation calls), which denotes the operation for checking queue status and calculating imbalance conditions on *Tcomm*. A larger number shows *Tcomm* works more or *Tcomm* is more active in checking imbalance status. The last value is an average cost per balancing calculation call in μs. These three values are calculated by average first over the number of iterations, then second over the number of processes. We can see that on BEAST-system, the balancing cost is ≈ 2 × −3× lower than SuperMUC-NG and CoolMUC2, and the number of migrated tasks is also higher. This can emphasize that BEAST-system has benefited from the migration throughput as well as the lower cost of balancing operations. *Tcomm* is more reactive, for example, in the case of wrong task migration, then it can migrate tasks back around. Furthermore, the interconnection on BEAST-system is more stable because this system is used as a testbed. On SuperMUC-NG and CoolMUC2, a job running can be affected by other jobs using the same interconnection in the same rack.

7.2 | Realistic use cases

In this experiment, Sam(oa)^2 is also set up with 100 time steps for simulation. The simulation scenario is oscillating lake, where tasks are the grid traverse kernels. The load of tasks on a process is dynamically changed due to the adaptive mesh refinement algorithm in Sam(oa)^2,[17] for example, wetting, drying, the limiter being applied to several cells of the grid or so-called mesh. Basically, we could say that the numeric algorithm merely causes the imbalance. We vary the number of ranks on each system, where two ranks per node, and each rank uses full cores of a CPU socket, for example, 14 threads per rank on CoolMUC2. For different communication and migration overheads, the tests can show scalability and adaptation in various methods. In Figure 17, reactive or proactive methods obtain higher performance than the `baseline`. Compared to `react_off`, speculative replication (`react_rep`) usually comes to some cost. However, their combination `react_off_rep` could help in the cases from 16 ranks on CoolMUC2 and BEAST. The replication strategy is difficult to deal with in 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 (≥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 longer to proceed 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. Besides, our expected accuracy of the prediction models is not too high, which means an acceptable result to estimate the difference of load among processes. Therefore, this

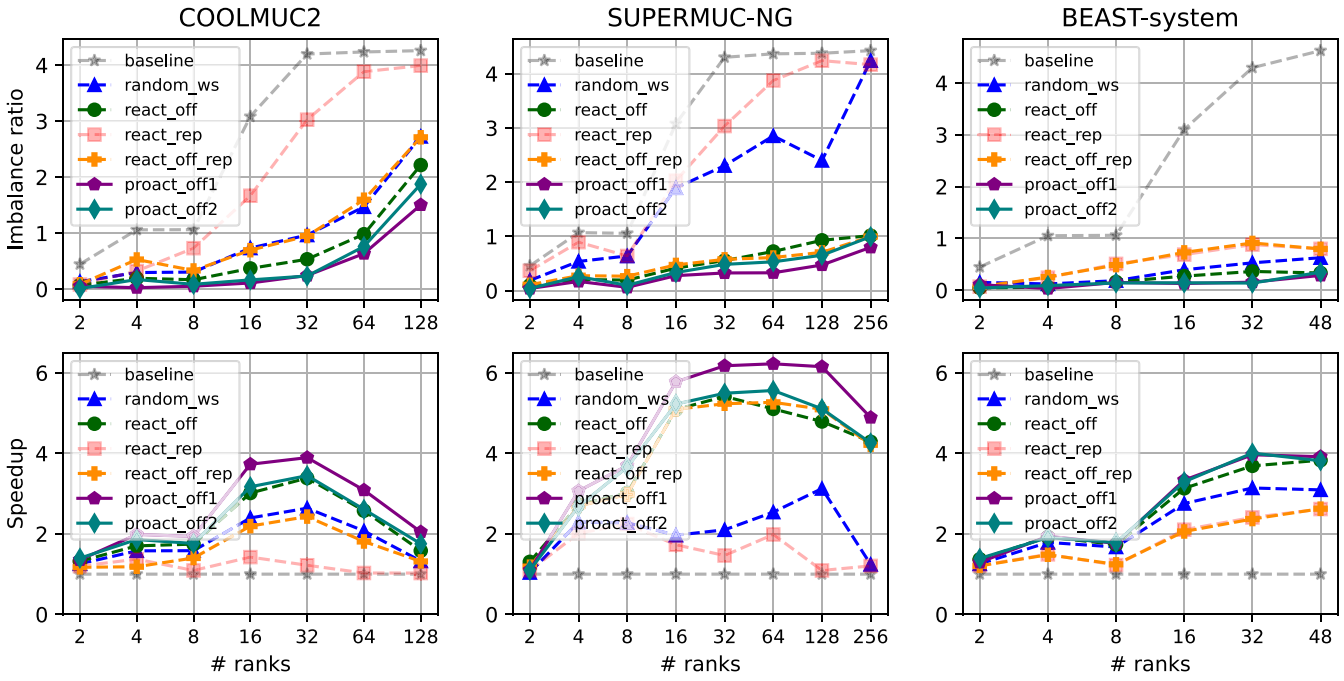

FIGURE 17 The comparison of imbalance ratios and speedup in various methods by the use case of oscillating lake simulation.

scheme is feasible to generate a reasonable runtime cost model. To be extended, we can combine reactive and proactive approaches to improve each other.

8 | CONCLUSION

We have analyzed the challenges of dynamic load balancing problems in distributed memory systems. Work stealing is a conventional approach, but stealing tasks can be too late due to migration overhead. An improvement is reactive load balancing based on offloading tasks beforehand. The idea leaves one core for a dedicated thread to monitor each process's queue status repeatedly. Then, tasks can be offloaded from a slow process to a faster process earlier. However, these reactive operations can be mistaken because we lack load information at runtime, and the most current status of queues at a time cannot correctly reflect how many tasks should be migrated at once as well as which process is a potential victim.

In detail, the article proposed a performance model to simulate the reactive approach. Besides, it helps to estimate the upper bound about how many tasks can be offloaded under the constraints of imbalance level, task size, and delay time. Furthermore, we introduced a new proactive approach for balancing tasks. The solution combines online load prediction and proactive task offloading at runtime. One-core-off is still employed, but we force it busier by task characterization, collecting data, and training machine learning models. After the model is ready, it is loaded to predict the load values in the next coming iterations of computing tasks. The predicted load values will input into a task-offloading algorithm. We proposed a fully distributed algorithm that utilizes prediction results to guide task offloading. Our implementation is deployed in a task-based library called Chameleon, and we perform the experiments on three different clusters. The results confirm the benefits in important use cases. Besides, the paper's solution could work as a plugin on top of a task-based framework or library. For a long-term vision, we can extend this work as a conceivable scheme to co-schedule tasks across multiple applications in future parallel systems.

ACKNOWLEDGMENTS

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. The authors gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre (www.lrz.de). Performance results have been obtained on Linux Cluster (CoolMUC–2), SuperMUC–NG, and systems in the test environment BEAST (Bavarian Energy Architecture & Software Testbed) at the Leibniz Supercomputing Centre. Additionally, we sincerely thank DAAD (German Academic Exchange Service) for their support in facilitating the doctoral research of Minh Thanh Chung. Open Access funding enabled and organized by Projekt DEAL.

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interests.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in chameleon-hpc at https://github.com/chameleon-hpc/chameleon-apps/tree/master/tools/tool_load_prediction.

ENDNOTES

*Rank and process might be used interchangeably in this article.
†"Offload" and "migrate" might be used interchangeably. Besides, using "offload" instead of "steal" is to emphasize that tasks are offloaded from a slow process to a fast process reactively.
‡Load is considered as execution time in this work.
§Underloaded targets/processes indicate the victims with an under-average load.
¶In work stealing, a victim is an overloaded process who is stolen tasks. In reactive balancing, we call an underloaded/fast process victim because tasks are offloaded from the overloaded/slow process to a fast one reactively.
#https://doku.lrz.de/display/PUBLIC/CoolMUC-2
‖https://doku.lrz.de/display/PUBLIC/SuperMUC-NG
**https://www.lrz.de/presse/ereignisse/2020-11-06_BEAST/
††https://github.com/chameleon-hpc/chameleon-apps/tree/master/tools/tool_load_prediction

ORCID

Minh Thanh Chung https://orcid.org/0000-0001-6119-3852
Josef Weidendorfer https://orcid.org/0000-0001-7159-1432
Karl Fürlinger https://orcid.org/0000-0003-0398-4087
Dieter Kranzlmüller https://orcid.org/0000-0002-8319-0123

REFERENCES

1. Chengzhong X, Lau FCM. Chapter 1, Section 1.2: The load balancing problem. *Load Balancing in Parallel Computers: Theory and Practice*. Springer; 1997. doi:10.1007/b102252
2. Casavant T, Kuhl J. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans Softw Eng*. 1988;14(2):141-154. doi:10.1109/32.4634
3. Pınar A, Aykanat C. Fast optimal load balancing algorithms for 1D partitioning. *J Parallel Distrib Comput*. 2004;64(8):974-996. doi:10.1016/j.jpdc.2004.05.003
4. Menon H, Kalé L. A distributed dynamic load balancer for iterative applications. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE; 2013, 11:1. doi:10.1145/2503210.2503284
5. Skinner D, Kramer W. Understanding the causes of performance variability in HPC workloads. *IEEE International 2005 Proceedings of the IEEE Workload Characterization Symposium*. IEEE; 2005:137-149. doi:10.1109/IISWC.2005.1526010
6. Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. *J ACM*. 1999;46(5):720-748. doi:10.1145/324133.324234
7. Yang J, He Q. Scheduling parallel computations by work stealing: A survey. *Int J Parallel Program*. 2018;46(2):173-197. doi:10.1007/s10766-016-0484-8
8. Shanley T. *InfiniBand Network Architecture*. Addison-Wesley Professional; 2003.
9. Ouyang X, Marcarelli S, Rajachandrasekar R, Panda DK. RDMA-based job migration framework for MPI over InfiniBand. *2010 IEEE International Conference on Cluster Computing*. IEEE; 2010:116-125. doi:10.1109/CLUSTER.2010.20
10. Liu J, Wu J, Kini SP, Wyckoff P, Panda DK. High performance RDMA-based MPI implementation over InfiniBand. *Proceedings of the 17th Annual International Conference on Supercomputing*. ACM; 2003:295-304. doi:10.1145/782814.782855
11. Klinkenberg J, Samfass P, Bader M, Terboven C, Müller MS. Chameleon: reactive load balancing for hybrid MPI+OpenMP task-parallel applications. *J Parallel Distrib Comput*. 2020;138:55-64. doi:10.1016/j.jpdc.2019.12.005
12. Stark DT, Barrett RF, Grant RE, Olivier SL, Pedretti KT, Vaughan CT. Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications. *2014 Workshop on Exascale MPI at Supercomputing Conference*. IEEE; 2014:9-19. doi:10.1109/ExaMPI.2014.6
13. Si M, Peña AJ, Balaji P, Takagi M, Ishikawa Y. MT-MPI: multithreaded MPI for many-core environments. *Proceedings of the 28th ACM International Conference on Supercomputing*. ACM; 2014:125-134. doi:10.1145/2597652.2597658
14. Zambre R, Chandramowlishwaran A. Lessons learned on MPI+threads communication. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM; 2022:1-16. doi:10.5555/3571885.3571987
15. Samfass P, Klinkenberg J, Bader M. Hybrid MPI+OpenMP reactive work stealing in distributed memory in the PDE framework Sam(oa)^2. *IEEE International Conference on Cluster Computing*. IEEE; 2018:337-347. doi:10.1109/CLUSTER.2018.00051
16. Samfass P, Klinkenberg J, Chung MT, Bader M. Predictive, reactive and replication-based load balancing of tasks in chameleon and Sam(Oa)2. *Proceedings of the Platform for Advanced Scientific Computing Conference*. ACM; 2021:1-10. doi:10.1145/3468267.3470574
17. 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*. 2016;43(3):19. doi:10.1145/2947668
18. Chung MT, Weidendorfer J, Fürlinger K, Kranzlmüller D. Proactive task offloading for load balancing in iterative applications. In: Wyrzykowski R, Dongarra J, Deelman E, Karczewski K, eds. *Proceedings of the 14th International Conference on Parallel Processing and Applied Mathematics*. Springer; 2023:263-275.

19. Luling R, Monien B, Ramme F. Load balancing in large networks: a comparative study. *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*. IEEE; 1991:686-689. doi:10.1109/SPDP.1991.218196
20. Tantawi AN, Towsley D. Optimal static load balancing in distributed computer systems. *J ACM*. 1985;32(2):445-465. doi:10.1145/3149.3156
21. Karypis G, Kumar V. A coarse-grain parallel formulation of multilevel K-way graph partitioning algorithm. Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing; 1997.
22. Catalyurek UV, Boman EG, others. Hypergraph-based dynamic load balancing for adaptive scientific computations. *International Parallel and Distributed Processing Symposium*. IEEE; 2007:1-11. doi:10.1109/IPDPS.2007.370258
23. Tuncer O, Ates E, Zhang Y, et al. Online diagnosis of performance variation in HPC systems using machine learning. *IEEE Trans Parallel Distrib Syst*. 2019;30(4):883-896. doi:10.1109/TPDS.2018.2870403
24. Zhao H, Deng S, Chen F, Yin J, Dustdar S, Zomaya AY. Learning to schedule multi-server jobs with fluctuated processing speeds. *IEEE Trans Parallel Distrib Syst*. 2023;34(1):234-245. doi:10.1109/TPDS.2022.3215947
25. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: an efficient multithreaded runtime system. *J Parallel Distrib Comput*. 1996;37(1):55-69. doi:10.1006/jpdc.1996.0107
26. Olivier SL, Porterfield AK, Wheeler KB, Spiegel M, Prins JF. OpenMP task scheduling strategies for multicore NUMA systems. *Int J High Perform Comput Appl*. 2012;26(2):110-124. doi:10.1177/1094342011434065
27. Iordan AC, Jahre M, Natvig L. Tuning the victim selection policy of Intel TBB. *J Syst Archit*. 2015;61(10):584-591. doi:10.1016/j.sysarc.2015.07.004
28. Lin CX, Huang TW, Wong MDF. An efficient work-stealing scheduler for task dependency graph. *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE; 2020:64-71. doi:10.1109/ICPADS51040.2020.00018
29. Charles P, Grothoff C, Saraswat V, et al. X10: an object-oriented approach to non-uniform cluster computing. *ACS SIGPLAN Not*. 2005;40(10):519-538. doi:10.1145/1103845.1094852
30. Chamberlain B, Callahan D, Zima H. Parallel programmability and the chapel language. *Int J High Perform Comput Appl*. 2007;21(3):291-312. doi:10.1177/1094342007078442
31. Brock B, Buluç A, Yelick K. BCL: a cross-platform distributed data structures library. *Proceedings of the 48th International Conference on Parallel Processing*. ACM; 2019:102. doi:10.1145/3337821.3337912
32. Nieplocha J, Carpenter B. ARMCI: a portable remote memory copy library for distributed array libraries and compiler run-time systems. *Parallel and Distributed Processing*. Springer; 2006:533-546. doi:10.1007/BFb0097937
33. Dinan J, Larkins DB, Sadayappan P, Krishnamoorthy S, Nieplocha J. Scalable work stealing. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE; 2009:1-11. doi:10.1145/1654059.1654113
34. Barrett B, Brightwell RB, Grant R, et al. The Portals 4.3 network programming interface; 2022. doi:10.2172/1875218
35. Larkins DB, Snyder J, Dinan J. Accelerated work stealing. *Proceedings of the 48th International Conference on Parallel Processing*. ACM; 2019:1-10. doi:10.1145/3337821.3337878
36. Lifflander J, Krishnamoorthy S, Kale LV. Work stealing and persistence-based load balancers for iterative overdecomposed applications. *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. ACM; 2012:137-148. doi:10.1145/2287076.2287103
37. Freitas V, Pilla LL, Santana AL, Castro M, Cohen J. PackStealLB: a scalable distributed load balancer based on work stealing and workload discretization. *J Parallel Distrib Comput*. 2021;150:34-45. doi:10.1016/j.jpdc.2020.12.005
38. Amiri M, Mohammad-Khanli L. Survey on prediction models of applications for resources provisioning in cloud. *J Netw Comput Appl*. 2017;82:93-113. doi:10.1016/j.jnca.2017.01.016
39. Delimitrou C, Kozyrakis C. Quasar: resource-efficient and QoS-aware cluster management. *SIGPLAN Not*. 2014;49(4):127-144. doi:10.1145/2644865.2541941
40. Carrington L, Laurenzano M, Snavely A, Campbell RL, Davis LP. How well can simple metrics represent the performance of HPC applications? *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE; 2015. doi:10.1109/SC.2005.33
41. Sharkawi S, DeSota D, Panda R, et al. Performance projection of HPC applications using SPEC CFP2006 benchmarks. *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE; 2009:1-12. doi:10.1109/IPDPS.2009.5161057
42. Shende S, Malony AD, Cuny J, Beckman P, Karmesin S, Lindlan K. Portable profiling and tracing for parallel, scientific applications using C++. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM; 1998:134-145. doi:10.1145/281035.281049
43. Munera A, Royuela S, Llort G, Mercadal E, Wartel F, Quiñones E. Experiences on the characterization of parallel applications in embedded systems with extrae/paraver. *The 49th International Conference on Parallel Processing—ICPP*. ACM; 2020:1-11. doi:10.1145/3404397.3404440
44. Li J, Ma X, Singh K, Schulz M, Supinski BR, McKee SA. Machine learning based online performance prediction for runtime parallelization and task scheduling. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE; 2009:89-100. doi:10.1109/ISPASS.2009.4919641
45. Butenhof DR. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc.; 1997. doi:10.5555/263953
46. Nickolls J. GPU parallel computing architecture and CUDA programming model. *2007 IEEE Hot Chips 19 Symposium (HCS)*. IEEE; 2007:1-12. doi:10.1109/HOTCHIPS.2007.7482491
47. Wienke S, Springer P, Terboven C, Da M. OpenACC-first experiences with real-world applications. *Euro-Par 2012 Parallel Processing*. Springer; 2012:859-870. doi:10.1007/978-3-642-32820-6_85
48. Thoman P, Dichev K, Heller T, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *J Supercomput*. 2018;74(4):1422-1434. doi:10.1007/s11227-018-2238-4
49. Kurose J, Ross K. *Computer Networking: A Top-Down Approach*. Pearson; 2021.
50. Chiasson J, Tang Z, Ghanem J, et al. The effect of time delays on the stability of load balancing algorithms for parallel computations. *IEEE Trans Control Syst Technol*. 2005;13(6):932-942. doi:10.1109/TCST.2005.854339
51. Liu J, Chandrasekaran B, Yu W, et al. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*. 2004;24(1):42-51. doi:10.1109/MM.2004.1268994
52. Chung MT, Kranzlmüller D. User-defined tools for characterizing task-parallel applications and predicting load imbalance. *2021 15th International Conference on Advanced Computing and Applications (ACOMP)*. IEEE; 2021:98-105. doi:10.1109/ACOMP53746.2021.00020

53. Paul AK, Choi JY, Karimi AM, Wang F. Machine learning assisted HPC workload trace generation for leadership scale storage systems. *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM; 2022:199-212. doi:10.1145/3502181.3531457
54. Naser M, Alavi AH. Error metrics and performance fitness indicators for artificial intelligence and machine learning in engineering and sciences. *Archit Struct Constr*. 2021. doi:10.1007/s44150-021-00015-8
55. Bonaccorso G. *Machine Learning Algorithms*. Packt Publishing. 2017. https://packt.link/free-ebook/9781785889622
56. Eichenberger AE, Mellor-Crummey J, Schulz M, et al. OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell AP, Chapman BM, Müller MS, eds. *OpenMP in the Era of Low Power Devices and Accelerators (IWOMP 2013)*. Springer; 2013:171-185. doi:10.1007/978-3-642-40698-0_13

How to cite this article: Thanh Chung M, Weidendorfer J, Fürlinger K, Kranzlmüller D. From reactive to proactive load balancing for task-based parallel applications in distributed memory machines. *Concurrency Computat Pract Exper*. 2023;35(24):e7828. doi: 10.1002/cpe.7828

APPENDIX A. REACTIVE LOAD BALANCING AND TASK OFFLOADING STRATEGIES

As mentioned and illustrated in Figure 4B, reactive task offloading takes actions based on the most current length of a queue. This information is tracked repeatedly, and offloading decisions are performed when we meet an imbalance condition, for example, $R_{\text{imb}} \geq 0.05$ (5%). The queue length is the number of remaining tasks at a time point. For instance, the queue length of each process in Figure 4B at time t_k (named $Q_i, \forall i \in P$) is: $Q_0 = 16$, $Q_1 = 16, Q_2 = 14, Q_3 = 14, Q_4 = 12, Q_5 = 14, Q_6 = 16, Q_7 = 16$.

Considering these numbers as the representative load values at t_k, we have $L_{\text{max}} = 16$, $L_{\text{avg}} = 14.75$; then the imbalance ratio is $R_{\text{imb}} \approx 0.1$. Therefore, a task in P_1 is offloaded to P_4 reactively right after t_k (denoted by the yellow box). Due to plotting space, we illustrate only 10 tasks in a row. Assuming that we have two threads per process for task execution, the illustrated green boxes imply doubling the number of tasks, for example, the finished two tasks at t_k means four executed tasks in total. In principle, this idea can overlap computation and communication and migrate tasks earlier in distributed memory. However, it is difficult to consistently decide when tasks should be offloaded, how many, and which processes are potential victims. An improved version can be developed as the idea of feedback task offloading, which is mentioned in the next section (Appendix B).

APPENDIX B. FEEDBACK TASK OFFLOADING

This appendix continues to describe another balancing strategy based on our proactive approach. The idea is called feedback task offloading, which is mentioned in Section 5.3. Figure B1 shows the design. We keep reactive task offloading operations in the first iteration. $Tcomm_i$ repeatedly monitors each process's queue status, exchanges this status among processes, and offloads tasks reactively if the imbalance ratio meets. After the first iteration, all task-offloading data will be recorded. We summarize statistics about:

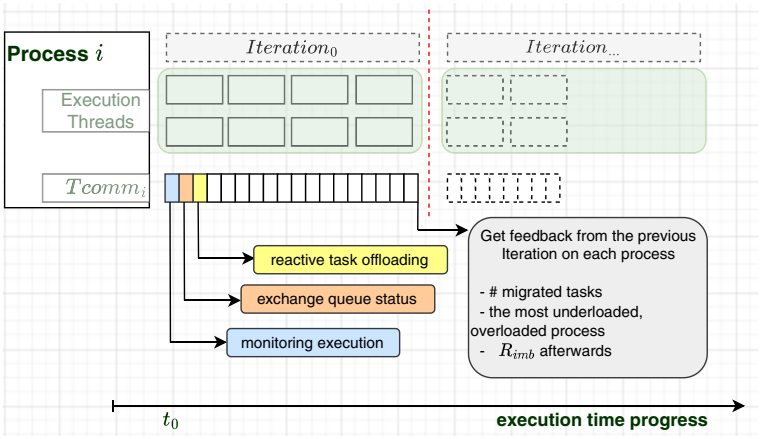

FIGURE B1 Feedback task offloading strategy.

- The number of offloaded and received tasks in a process.
- The total load values by executing local and remote tasks.
- The R_{imb} ratio and speed up afterwards.

In the next iterations, the statistic is used to adjust the strategy for assigning priorities when we choose a task-offloading victim. Feedback is based on the statistical result of migrated tasks in the previous iterations to improve task offloading more proactive in the next iterations. The article does not focus in-depth on this strategy; however, we argue that strong statistical information can turn forward reactive balancing operations into more accurate and proactive.