INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Dissertation

an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

A Proactive Approach of Co-scheduling Tasks for Dynamic Load Balancing in Parallel Applications

eingereicht von

Minh Chung

am: 07. Jan 2024

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Dissertation

an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

A Proactive Approach of Co-scheduling Tasks for Dynamic Load Balancing in Parallel Applications

eingereicht von

Minh Chung

am: 07. Jan 2024

Erstberichterstatter:	Prof. Dr. Dieter Kranzlmüller
Zweiberichterstatter:	Prof. Dr. Raymond Namyst

Tag der mündlichen Prüfung: 07. January 2024

Eidesstattliche Versicherung
(Siehe Promotionsordnung vom 12.07.11, §8, Abs.2, Pkt.5)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Chung, Minh
Name, Vorname

München, den 07.01.2024
Ort, Datum

(Unterschrift des Doktoranden)

Acknowledgments

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi tincidunt vehicula neque, vel faucibus elit. Vestibulum volutpat id libero a mattis. Maecenas a est lorem. Sed risus neque, mollis sit amet placerat quis, lacinia eget ipsum. Sed enim arcu, tristique non nisi ac, tristique faucibus velit. Ut erat nisi, tincidunt quis vehicula at, malesuada eget tortor. Etiam iaculis ex nisl, sit amet convallis justo ultrices nec. In hac habitasse platea dictumst. Aenean posuere vitae magna facilisis dapibus. Praesent velit enim, eleifend sit amet feugiat nec, facilisis et neque. Quisque non ex metus. Duis accumsan, turpis a volutpat elementum, est neque malesuada odio, non dignissim nulla ipsum vel lorem. Cras efficitur dui vitae urna tincidunt, in dapibus mi molestie. Duis sit amet porta neque.

Abstract

High performance computing (HPC) plays a key role in scientific and research, solving intricate computational problems. HPC relies on parallel computing, involving the use of parallel compute clusters for advanced computations. Thus, computational problems have to be represented as task parallel applications on these clusters. However, load balancing is often a challenge for task parallel applications, particularly when dealing with imbalances caused by performance slowdown. This dynamic nature makes traditional approaches like pre-partitioning or static load balancing less effective.

A useful approach to deal with performance slowdown at runtime is "work stealing", where an idle process can take tasks from others. It involves the migration of tasks and data among nodes, but communication overhead in parallel compute clusters can delay stealing operations. An improvement is "reactive load balancing", which proposes task offloading from a slow process to a fast one earlier than stealing. This reactive approach has limitations when task offloading decisions are driven by speculation. Furthermore, lack of load information is also a reason, impeding these decisions.

Our work contributes a performance model to analyze the limits of reactive load balancing. We propose a new proactive approach to enhance performance even further. The main concept revolves around real-time load information acquisition, enabling a more proactive load balancing strategy. In detail, we leverage the execution scheme of task-based programming models, where one thread within a process is dedicated to task characterization, online load prediction, and proactive task offloading based on prediction outputs. We obtain load prediction knowledge instead of missing prior knowledge during execution. We can make more accurate estimations regarding the number of tasks and potential processes for task migration. Within our approach, we introduce two balancing methods: feedback task offloading and machine learning-based task offloading. Besides that, we propose an extension that facilitates co-scheduling tasks across multiple applications.

To evaluate the efficacy of our approach, we employ both simulations and experimental tests. The test cases include synthetic microbenchmarks and a realistic application involving adaptive mesh refinement. Our results confirm benefits in high imbalance scenarios, showcasing a speedup of approximately 3.5× when compared to randomized work stealing, and 1.7× when compared to reactive load balancing. In a broader vision, our approach opens a new scheme for co-scheduling tasks across multiple applications, enhancing not only the performance and resource utilization of a single application but multiple ones.

Zusammenfassung

Hochleistungsrechnen (HPC) spielt eine Schlüsselrolle in der Wissenschaft und Forschung und löst komplexe Berechnungsprobleme. HPC setzt auf das parallele Rechnen und nutzt parallele Rechencluster für komplexe Berechnungen. Daher müssen Berechnungsprobleme als taskbasierte Applikationen auf diesen Clustern dargestellt werden. Die Lastenverteilung ist jedoch oft eine Herausforderung für taskbasierte Applikationen, insbesondere im Umgang mit Ungleichgewichten aufgrund von Leistungsverlangsamungen. Diese dynamische Natur macht traditionelle Ansätze wie die Vorverteilung oder die statische Lastenverteilung weniger effektiv.

Ein nützlicher Ansatz für Leistungsverlangsamungen zur Laufzeit ist das "Work-stealing", bei dem ein inaktiver Prozess Aufgaben von anderen übernehmen kann. Dies beinhaltet die Migration von Aufgaben und Daten zwischen Knoten, jedoch kann die Kommunikationsüberlastung in parallelen Rechenclustern die Diebstahlvorgänge verzögern. Eine Verbesserung ist das "Reactive-load-balancing", das die Verlagerung von Aufgaben von einem langsamen Prozess zu einem schnelleren Prozess früher als das Stehlen vorschlägt. Dieser reaktive Ansatz hat Einschränkungen, wenn Entscheidungen zur Aufgabenverlagerung auf Spekulationen basieren. Darüber hinaus ist der Mangel an Lastinformation ebenfalls ein Grund, der diese Entscheidungen behindert.

Unsere Arbeit trägt ein Leistungsmodell bei, um die Grenzen des reaktiven Lastenausgleichs zu analysieren. Wir schlagen einen neuen proaktiven Ansatz vor, um die Leistung noch weiter zu steigern. Das Hauptkonzept dreht sich um die Echtzeit-Erfassung von Lastinformationen, die eine proaktivere Lastenausgleichsstrategie ermöglicht. Im Detail nutzen wir das Ausführungsschema taskbasierter Programmiermodelle, bei dem ein Thread innerhalb eines Prozesses der Aufgabencharakterisierung, der Online-Lastvorhersage und der proaktiven Verlagerung von Aufgaben basierend auf Vorhersageergebnissen gewidmet ist. Daher erhalten wir Lastvorhersagewissen anstelle von fehlendem Vorwissen während der Ausführung. Wir können genauere Schätzungen zur Anzahl der Aufgaben und potenziellen Prozessen für die Verlagerung von Aufgaben machen. In unserem Ansatz stellen wir zwei Ausgleichsmethoden vor: die Feedback-Aufgabenverlagerung und die auf maschinellem Lernen basierende Aufgabenverlagerung. Außerdem schlagen wir eine Erweiterung vor, die die Koordination von Aufgaben über mehrere Applikationen hinweg erleichtert.

Um die Wirksamkeit unseres Ansatzes zu bewerten, verwenden wir Simulationen und experimentelle Tests. Die Testfälle umfassen synthetische Mikrobenchmarks und eine realistische Applikation mit adaptiver Netzverfeinerung. Unsere Ergebnisse bestätigen Vorteile in Szenarien mit hohem Ungleichgewicht und zeigen eine Beschleunigung von etwa 3.5× im Vergleich zum zufälligen Work-stealing und 1.7× zum reaktiven Lastenausgleich. In einer umfassenderen Sichtweise eröffnet unser Ansatz ein neues Schema zur Koordination von Aufgaben über mehrere Applikationen hinweg, um nicht nur die Leistung und Ressourcennutzung einer einzelnen Applikation, sondern auch mehrerer zu verbessern.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Overview

High performance computing (HPC) has gained an important role in solving complex problems and accelerating scientific research. HPC implies using supercomputers to perform computationally intensive operations [SAB18]. Compared to a general-purpose computer, a supercomputer indicates a machine with high-performance level. The architecture of high-performance computers is designed towards parallel computing that can quickly process such large amounts of data and perform advanced computations.

Parallel computing is a computational approach in which problems are split into smaller pieces and solved in parallel. Defining parallelism is difficult because it can appear at different levels, e.g., bit-level, instruction-level, data-level, and task-parallelism level [Eij10]. Over the past two decades, these terms are frequently used to denote parallel computers, which are machines with more than one processor. A processor in computer science is called central processing unit (CPU) that performs operations on a data source. One way to characterize parallel computers is based on the hardware support level, such as multi-core processors [BDM09] where a processor may consist of multiple cores and each core is a processing element. Another way is Flynn's characterization [Fly72] based on data flow and control flow, which is known as the following four types:

- SISD: Single Instruction Single Data
- SIMD: Single Instruction Multiple Data
- MISD: Multiple Instruction Single Data
- MIMD: Multiple Instruction Multiple Data

Alongside multiple processors working together, efficient access to memory is a critical consideration. For this reason, we can characterize parallel computers through different types of memory access. The main distinction is between shared memory and distributed memory [JNW08]. Shared memory enables all processors to access the same memory pool, while in distributed memory, each processor has its own physical memory and address space.

Building upon the concept of parallel computers, a supercomputer today refers to a parallel compute cluster, also considered as a common example for distributed memory machines/systems. We define a cluster as a set of parallel computers (called compute nodes) that work together and effectively function as a single system. For example, the SuperMUC-NG supercomputer at the Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities [Bav20] has more than 6400 nodes with 311040 cores and 719 TB memory in total. All nodes are connected to each other via high-speed network, where each node has its private memory, the same hardware, and the same operating system. However, in certain configurations, there may be variations in hardware

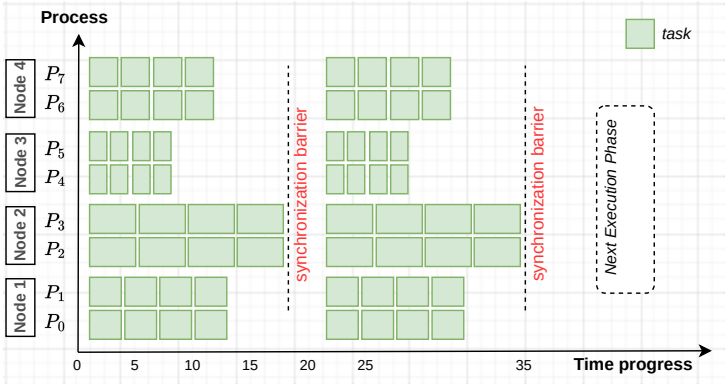


Figure 1.1.: An illustration of iterative execution and load imbalance in distributed memory systems with 4 compute nodes, 2 processes per node.

or operating systems for specific purposes. The scale of clusters may be small as a two-node system or large with many nodes as a supercomputer like SuperMUC-NG. Regardless of the scale, the interconnection among nodes has to be across a network [LBC91]. When multiple nodes exchange data, a message-passing technique is needed; and in most cases the open library standard, Message Passing Interface (MPI) [Gro+96], is used. Commonly, there are different parallel programming models, e.g., shared memory programming models, distributed memory or message passing programming models, data parallel models, and hybrid models. However, it is important to note that almost all applications should be broken into discrete pieces or smaller tasks. Then, the tasks can be executed simultaneously. We consider such applications as task or task-based parallel applications [Tho+18]. On one side, executing tasks in parallel should gain speedup in completion time. On the other side, challenges during execution are communication overhead, synchronization, and load balancing.

This thesis is concerned with load balancing in distributed memory systems. The goal is to treat each processor with an equal share of the total load [Cyb89]. Load balancing is important because a load imbalance might affect the completion time of parallel applications. For example, we assume a given distribution, where each processor is assigned a number of tasks before execution. A processor executing tasks refers to a process that is defined as a logical instance at operating system level. During execution, when all processes are subject to a synchronization barrier but some of them are slower than others, the overall performance will be determined by load imbalance.

Regarding the definition of load, load in this thesis refers to the amount of time that a process executes tasks. Occasionally, we might use this amount of time, which a process is active to execute one or more tasks, as the execution time of tasks. Therefore, the values of load and execution time can be understood interchangeably. We target task-based parallel applications with iterative execution. A task is often defined by a compute function, where each task points to a code region and data. "Iterative" indicates applications with multiple execution phases that can be repeated over time steps based on user configuration. "Iterative" widely refers to bulk synchronous parallel (BSP) models [Val90]. Computation in BSP is divided into a sequence of execution phases as shown in Figure 1.1, which illustrates an iterative execution on 4 nodes, 2 processes per node. The x-axis is the time progress of task execution, where green boxes represent tasks and their length denotes the load value or the execution time. The y-axis lists compute nodes and processes executing tasks in each phase. A phase finishes when all processes are done. At the end of a phase, a barrier is used to ensure synchronization for the next phase.

In consideration of dealing with imbalance at runtime, load balancing can be classified as "static" and "dynamic" [XL97].

- "Static" means using the estimated execution time of tasks per process to balance the load before running applications. Thereby, an accurate cost model for either optimal task assignment or task partitioning algorithms is needed.

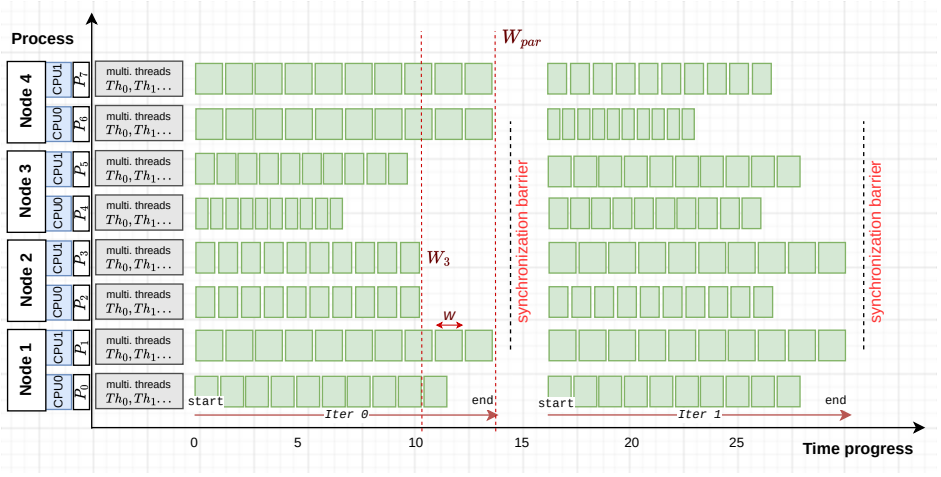


Figure 1.2.: An illustration refers to a real iterative execution with load imbalance in task parallel application, running on 4 compute nodes, 2 processes per node, and multiple threads per process.

- "Dynamic" means scheduling tasks at runtime without prior knowledge about the load values. Application behavior and system performance variability might change these values at runtime.

Specifically, our work are concerned with dynamic load balancing approaches. There are two reference schemes of dynamic load balancing approaches: master-worker [Ria+11] [Chr+05] and work stealing [BL99] [Din+09].

- Master-worker denotes a scheme, in which the master monitors and distributes the load to all workers. The downside of master-worker is that it is difficult to scale the number of compute nodes up because the master node can be overloaded by an increasing number of worker nodes. Instead of nodes, master process and worker process can be used as alternatives.
- Work stealing denotes that we allow an idle process to steal work[1] from the busy ones without prior knowledge. Work stealing can be particularly effective, but its efficiency can be limited by communication overhead in distributed memory systems. Communication overhead can cause delays when processes exchange information and steal tasks.

In general, dynamic load balancing depends on a typical context. Our target is to balance the load of a given distribution of tasks over processes running on distributed memory systems. We explore various approaches through the lens of work stealing. The subsequent section outlines our research problem formulation and motivation.

1.2. Research Problem and Motivation

We illustrate the load imbalance context in Figure 1.2 to explain the problem formulation intuitively. This figure reveals a similar case shown in Figure 1.1, load imbalance at runtime on 4 compute nodes. Again with the x, y coordinates, the x-dimension shows the time progress, the y-dimension shows the involved processes within compute nodes but in more detail with the threads executing tasks on each process. We intend to highlight this illustration as a realistic execution and refer to a hybrid parallel programming model in practice, multiprocessing + multithreading. A modern processor today is mostly multicore architecture; hence, a process mapped to a processor can create multiple threads pinned to multiple corresponding cores within that processor.

Suppose we use MPI to leverage communication between processes, then the hybrid model is known as MPI+X [Rab+06]. In Figure 1.2, each process, e.g., P_0, P_1, indicates an MPI process (also called

[1] "*Work*" and "*task*" might be used interchangeably.

MPI rank). One MPI process can spawn multiple threads to execute tasks denoted by `multi.threads` (Th_0, Th_1, and so on). The term "multiple threads" refers to many threads, where a thread is the segment of a process. While processes are mostly isolated, threads share memory and data. Therefore, we assume tasks are automatically scheduled and fairly executed within a process. An imbalance at runtime comes from different processes across different compute nodes, not within a process. Before every execution phase, each process is assigned a given number of tasks, and this assignment normally follows the partitioning algorithm of an application called a given task distribution on processes. In Figure 1.2, we highlight the first execution phase (denoted by Iteration 0 or `Iter 0` for short), where tasks belonging to process P_1, P_6, and P_7 are executed slower than the others, causing the load imbalance.

For comprehensive analysis, we formulate the problem as follows. Given T tasks in total, the tasks are indexed from $\{0, ..., (T-1)\}$ and distributed among P processes before execution. When T is mentioned as a set, we can understand it as $\{0, ..., (T-1)\}$; when T is mentioned as an integer number, it is the total number of tasks. Each process is denoted by P_i, where i in the range of $\{0, ..., (P-1)\}$. Similar to T, P can be mentioned as the number of processes if it is an integer number or the set $\{0, ..., (P-1)\}$. A scheduled task is atomic and runs on a specific thread until termination. Tasks, load values, and evaluation metrics are summarized in detail below.

- T_i: a set of assigned tasks for process P_i, e.g., T_1 is the set of assigned tasks belonging to process P_1. When T_i is mentioned as an integer number, it indicates the number of tasks assigned to process P_i.
- w: the load value of a task in general. The value of w is also called a wallclock execution time and $w > 0$. When we refer to a load of a particular task in a set T_i of process P_i, w can be attached with an index j, which means task j belongs to the set T_i. Overall, T_i is a subset of T, and all tasks in T_i belong to the set T. Therefore, a particular value w_j must be > 0 and $j \in \{0, ..., (T-1)\}$. The standard deviation between the values of w determines the type of tasks, uniform or non-uniform. Uniform tasks indicate the tasks with approximate load values, while non-uniform tasks indicate the tasks with different load values resulting in a high standard deviation of the load values of tasks. In Figure 1.2, the length of green boxes highlights the length of w. For this illustrated case, we can say that the load values of tasks in a process are uniform, but different tasks in different processes are non-uniform.
- $L_i, \forall i \in P$: the total load of process P_i, where $L_i = \sum_{j \in T_i} w_j$.
- $W_i, \forall i \in P$: the wallclock execution time of process P_i. Unlike L_i, W_i indicates the completion time of process P_i when all tasks are done. The execution of all tasks here indicates that they are executed in parallel by multiple threads in process P_i. Both L_i and W_i at the end will reflect the imbalance ratio. Figure 1.2 marks W_3 as an example to indicate W_i.
- W_{par}: the parallel wallclock execution time known as the completion time, makespan, or application time, $W_{par} = max_{i \in P}(W_i)$. For example, in Figure 1.2 W_{par} is determined by process P_6, P_7 at `Iter 0`.
- R_{imb}: imbalance ratio, $R_{imb} = \frac{W_{max}}{W_{avg}} - 1 = \frac{W_{par}}{W_{avg}} - 1$, where W_{max} is the maximum wallclock execution time ($max_{i \in P}(W_i) = W_{par}$), and W_{avg} is the average value ($avg_{i \in P}(W_i)$). Alternatively, we can also calculate R_{imb} via the total load values such as $R_{imb} = \frac{L_{max}}{L_{avg}} - 1$.

If $R_{imb} \approx 0$, there is no imbalance. If R_{imb} is greater than or equal to a certain threshold, R_{imb} is determined imbalance at runtime. For example, users define a threshold 0.2, and $R_{imb} \geq 0.2$ is determined as an imbalance. In this case, W_{par} is seen as unoptimized. Our work targets reducing the values of R_{imb} and W_{par}.

In our context, task migration is the only way to perform dynamic load balancing because tasks are already distributed among compute nodes. At runtime, we cannot share or add more computing resources in distributed memory systems. Otherwise, the technology must be changed to enable sharing resources across separate machines.

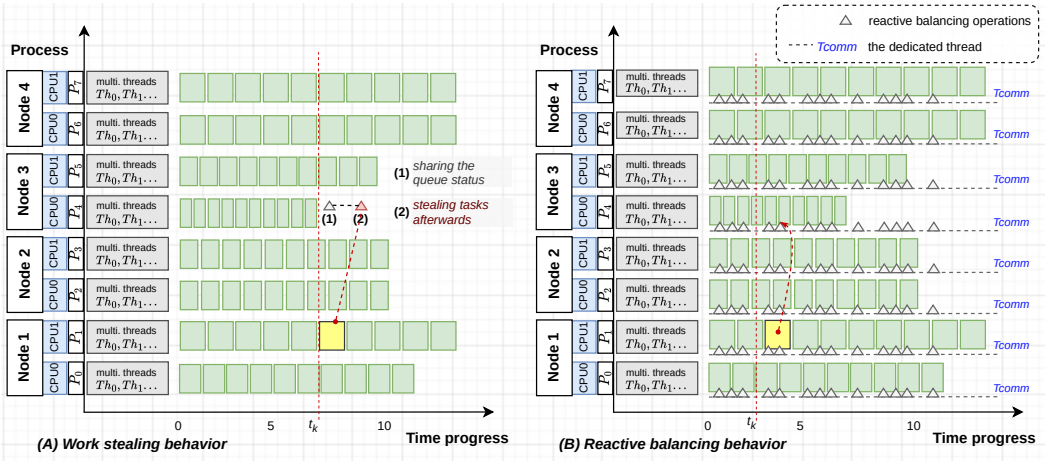


Figure 1.3.: Behaviors of work stealing and reactive load balancing.

As mentioned in Section 1.1, the previous approaches are mostly based on two schemes of dynamic load balancing: master-worker and work stealing. When considered in principle, all of these approaches comprise four components to their operations [XL97]. The four components can be explicit or implicit, including:

- Load measurement: indicates monitoring, checking either the load status or the execution speed.
- Information exchange: indicates the operations of status exchange among processes. This operation helps to synchronize the execution status to calculate imbalance ratio.
- Initiation rule: implies when to initiate a balancing operation. It can be initialized by overloaded processes or underloaded processes or periodically at runtime.
- Load balancing operations: implies the actions used to balance the load, e.g., task migration, thread/process migration. There might be different rules [XL97] for making decisions on these actions.

A master-worker scheme works as a centralized scheme, where the master is a central point, and workers wait to receive commands from the master. It is easy to manage load and control the balance of load, but it is challenging to scale up the number of distributed memory machines.

Work stealing is more relevant in our imbalance context. However, if we take communication overhead in distributed memory machines into account, the behavior of work stealing can be limited by the components of "information exchange" and "load balancing operations". The reasons are that tasks have to be migrated, and the behaviors of work stealing can make it too late to steal tasks. In a detailed inspection, these behaviors can refer to different variants. They are classified as follows.

- Passive: indicates work stealing in distributed memory systems, which has several proposed approaches for HPC clusters [RLP11] and also in-depth analysis of stealing-tasks behavior through performance modeling [Gas+21]. The algorithms and behaviors of work stealing in distributed memory differ from shared memory because the global operations, such as synchronization primitives, caching, or coherence protocols, often result in high overhead and latency. Another point is a contention that may occur when multiple thieves try to steal tasks from the same victim process [PTA13]. Therefore, in most algorithms for distributed memory, the decision of stealing tasks is taken when an idle process exists. Basically, the idle process sends out a steal request when its local queue is empty. Depending on a specific algorithm, the busy processes may also be in charge of responding to steal requests before a task is stolen. Thereby, we consider the behavior of processes performing work stealing in distributed memory systems as a passive action.

For illustration, Figure 1.3 (A) shows work stealing applied to the imbalance case illustrated in Figure 1.2. The x-axis again represents time progress during execution, and the y-axis highlights the labels of compute nodes, processes with their executing-tasks threads. At time t_k, process P_4 is idle, then it needs to mainly perform: (1) sharing/exchanging the idle status with other processes, (2) stealing tasks afterward if one of the overloaded processes responds to the request. When an agreement is made, process P_4 steals a task from process P_1. The stolen task is denoted by the yellow box. As we can see, these behaviors can be too late if communication overhead is dominant, leading to missed requests, an unexpected number of stolen tasks, or even some overloaded processes that are not handled fairly.

- Active: indicates the behaviors as seen in the state-of-the-art approaches. Instead of waiting until a process is idle, we attempt to send tasks actively from one process to another beforehand, if the imbalance is speculatively observed or predicted. Depending on how much information we have in a specific context, we can divide "active" into two variants:
  - Reactive: emphasizes responding to situations after certain conditions are satisfied. When an imbalance is detected periodically, this reacts with a decision of migrating or not migrating tasks. In particular, prior knowledge about load values is not needed. This reaction is only based on the most current status of execution and short-period speculation. After that, we perform these reactive operations repeatedly and continuously. Therefore, task migration here does not mean stealing tasks but offloading tasks[2] and we refer to reactive task offloading [Kli+20b].
  - Proactive: emphasizes taking actions in advance based on anticipating/predicting the future. This means we know more details about the imbalance situation, e.g., how much imbalance and load value are. Therefore, proactive task offloading means offloading tasks earlier and more controllable than reactive task offloading.

Several reactive load balancing approaches have been proposed [Kli+20a] [Sam+21]. These approaches continuously monitor the execution status instead of waiting for an empty queue. The monitored status information helps calculate imbalance ratio (R_{imb}) based on the queue length. Then, we can determine which process executes tasks slow and which process is fast. Following that, we can offload tasks from a slow process to a fast process reactively.

"Slow" and "fast" imply overloaded and underloaded processes. For example, Figure 1.3 (B) shows a reference implementation with MPI+OpenMP [RHJ09]. One thread per process is dedicated to be a communication thread (called $Tcomm$). $Tcomm$ is used to overlap communication and computation. For reactive load balancing, $Tcomm$ on each process monitors the execution speed, shares this information with other processes, and calculates the imbalance ratio. These operations are continuous and repeated. If an imbalance is detected over a given condition, we can offload tasks promptly from a slow process to a fast one. For example, at time t_k in Figure 1.3 (B), the imbalance is detected, and the reactive decision is made earlier than work stealing shown in Figure 1.3 (A). In this case, assuming that process P_1 is slow and process P_4 is fast as a good victim for offloading tasks. Victims in work stealing indicate the overloaded processes whose tasks are stolen. In reactive approaches, victims indicate the underloaded (fast) processes where tasks are offloaded to. The downside of reactive load balancing can be wrong speculation because we rely only on the most current status of execution speed. Periodically, reactive task offloading can be decided incorrectly under high imbalance cases. Besides, performance can be affected at the end when tasks are offloaded with an inappropriate number or wrong victims.

The limitations of work stealing and reactive load balancing motivate us to investigate a new proactive approach. In principle, our idea tries to tackle dynamic load balancing with the following constraints:

- Performance slowdown at runtime causes imbalance, which is unknown before running applications. This refers to the inherent variation in supercomputer architectures with the design of today multicore CPUs. Acun et al. use compute-intensive kernels and applications to analyze

[2] "Offload" and "migrate" might be used interchangeably in this thesis

the variation among processors in different supercomputers [AMK16]. They observe that there is an execution time difference of up to 16% among processors under enabling Turbo Boost. The intrinsic differences in the chips' power efficiency are the culprit behind the frequency variation. A similar work is from Weisbach et al. [Wei+18]; they also show six orders of magnitude difference in relative variation among CPU cores across different compute platforms such as Intel x86, Cavium ARM64, Fujitsu SPARC64, IBM Power.

- Communication overhead is a bottleneck for exchanging execution status and migrating tasks. The communication overhead in distributed memory systems comes from:
  - Latency (λ): time between sending and receiving the head of a message, where λ does not depend on the size of a message.
  - Delay time (d): time for transmitting an entire message between two nodes, where d depends on the size of a message.
- At runtime, we lack load information as well as system performance information to estimate better when, where, and how many tasks should be migrated at once.

1.3. Research Questions

Based on the motivation above, this thesis focuses on the following two main research questions (RQs).

- **RQ1: How can we model the behavior of reactive load balancing to understand its limits in distributed memory systems?**
- **RQ2: How can we proactively balance the load of task parallel applications at runtime?**

RQ1 indicates a performance model to analyze the limit of reactive load balancing as well as work stealing. Due to the overhead of balancing operations or task migration, they might reach a limit if actions are late and insufficient. The answer of RQ1 supports in-depth analysis and leads to a new approach mentioned in RQ2.

RQ2 introduces a novel proactive load balancing approach, where "proactive" means that task migration will be anticipated with prediction information. We use prediction information to estimate the number of tasks and potential victims for migrating tasks. Unlike work stealing or reactive load balancing, our approach is controlled proactively. Tasks can be migrated earlier when we have knowledge about load values.

To answer RQ2 comprehensively, we address two sub-questions (SQs):

- SQ1: How can we predict the load of tasks at runtime to support proactive load balancing?
- SQ2: How can we proactively co-schedule tasks to balance the load?

In order to perform offloading tasks proactively, we need to know the load values of tasks and how much load is different between processes, indicating the difference between overloaded and underloaded values. SQ1 and SQ2 support each other to answer RQ2, detailing how proactive load balancing works. Typically, SQ1 asks for load prediction, which is necessary to calculate how much load is different. We can then determine which processes are potential and available for offloading tasks. SQ2 implies methods to guide task offloading, where "co-schedule tasks" implies not only how to migrate tasks among processes in an application to balance the load, but also migrate tasks across multiple applications to improve overall performance. The output of SQ1, predicted load values, is the input of SQ2. Importantly, the output of SQ2 comes to how many tasks should be migrated from which process to which process. In this thesis, "approach" implies a broader strategy and perspective that drives load balancing, while "method" refers to a structured procedure that is more specific and often includes steps to guide task offloading for load balancing.

1.4. Methodology and Contributions

The methodology of this work is derived from experimental research. First, we conduct experiments on micro-benchmarks and real applications with different imbalance levels. Each experiment is deployed in distributed memory systems with and without load balancing at runtime. Second, we profile the execution to analyze the behavior of load balancing. Specifically, we deploy both work stealing and reactive load balancing. Upon thoroughly analyzing their runtime behaviors, we determine that overall performance is still limited, particularly in high imbalance cases. Notably, the decision taking actions in work stealing and reactive load balancing might be insufficient occasionally.

To explore a cause-and-effect relationship, we create a vector space of influence parameters. Then, we formulate the problem to support building a performance model for reactive load balancing. Drawing upon the performance model, a simulator is developed for further testing and proving the hypothesis. This methodology motivates us to investigate a new proactive load balancing approach.

Concretely, our work contributes a performance model to analyze the efficiency bound of reactive load balancing and work stealing. The model is represented as function F, where its inputs are the given information, including the distribution of T tasks on P processes, the load of tasks (w), and the constraints under communication such as latency, delay time when tasks are migrated. In P processes, the inputs indicate the number of overloaded and underloaded processes. Following that, the proposed model can analyze performance in case of applying work stealing or reactive load balancing. With our performance model, we create a simulator to leverage the analysis by manipulating variables and collecting quantitative data.

Apart from the abstracted model, this work contributes a new proactive approach, in which "proactive" can improve dynamic load balancing further by anticipating the future and taking actions in advance. We can offload tasks earlier and thus more proactively. We propose this toward a proactive scheme for a task-based programming model. Our approach divides the main execution of parallel applications into two streams of execution: one is threads for executing tasks and another one is a dedicated thread for proactive load balancing. The dedicated thread is named $Tcomm$, which will manage (1) task characterization, (2) online load prediction, (3) proactive task offloading. (1) and (2) provide load information to better estimating the number of tasks and potential victims for offloading tasks. (3) can be performed with different task offloading methods based on the prediction knowledge. In particular, we show two methods and one extension of co-scheduling tasks:

- Method 1: feedback task offloading
- Method 2: ML-based (Machine Learning-based) task offloading

The extension is co-scheduling tasks across multiple applications, which is considered as a new scheduling scheme in HPC clusters. For a long-term vision, we can co-schedule tasks from one application to another application during concurrent execution. Taking this into account, the benefit can be not only balancing load but also increasing compute resource utilization.

1.5. Publications

The following publications are associated with the thesis. We split the publications into two groups linked directly or indirectly to the thesis. We refer to the contributor roles taxonomy named *CRediT* at https://casrai.org/credit/ to show the author's contribution to each paper.

There are some main roles used to describe the contribution:

- ***Conceptualization***: indicates the contribution of ideas, formulation, or evolution of overarching research goals.
- ***Data curation***: points to the management activities around data, e.g., annotate data, scrub data, maintain research data, etc.

- ***Formal analysis***: means using statistical, mathematical, computational, or other formal techniques to analyze the data.
- ***Investigation***: is to conduct a research and investigation process, explicitly performing the experiments or data collection.
- ***Methodology***: is the contribution of creating or designing methodology, model.
- ***Validation***: indicates the verification as a part of the activity or the overall reproducibility of results/experiments.
- ***Visualization***: is the preparation, creation, and presentation of the published work, e.g., visualization, data presentation.
- ***Writing – original draft***: is specifically to write the initial draft (including substantive translation). This is also the contribution of creating and preparing the published work.
- ***Writing – review & editing***: contribute to the published work by critical review, commentary, or revision (including pre- or post-publication stages).

(A) Publications directly associated with the dissertation:

"From Reactive to Proactive Load Balancing for Task-based Parallel Applications in Distributed Memory Machines." Journal of Concurrency and Computation: Practice and Experience (CCPE), 2023, https://doi.org/10.1002/cpe.7828.

- *Authors*: Minh Thanh Chung, Josef Weidendorfer, Karl Fürlinger, and Dieter Kranzlmüller
- *Paper's contribution/Abstract*: First, this paper proposes a performance model to analyze reactive balancing behaviors and understand the bound leading to incorrect decisions. Second, we introduce a proactive approach to further improve balancing tasks at runtime. The approach also exploits task-based programming models with a dedicated thread (named *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 with an approriate number at once to a potential victim (denoted by an underloaded/fast process). The experimental results confirm speedup improvements from 1.5× to 3.5× in important use cases compared to the previous solutions. Furthermore, this approach can support co-scheduling tasks across multiple applications.
- *Relation to thesis*: This paper is directly related to RQ1 and SQ2 in this thesis. We show how to perform task characterization and generate load prediction at runtime. Then, this information is applied to a proactive algorithm which guides task offloading to balance the load.
- *Author's contribution*: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft.

"Proactive Task Offloading for Load Balancing in Iterative Applications." The 14th International Conference on Parallel Processing and Applied Mathematics (PPAM22), 2022, https://doi.org/10.1007/978-3-031-30442-2_20.

- *Authors*: Minh Thanh Chung, Josef Weidendorfer, Karl Fürlinger, and Dieter Kranzlmüller
- *Paper's contribution/Abstract*: Load imbalance is often a challenge for applications in parallel systems. Static cost models and pre-partitioning algorithms distribute the load at the beginning. However, during execution, dynamic changes or inaccurate cost indicators may lead to imbalance at runtime. Reactive work-stealing strategies can help monitor the execution and perform task migration to balance the load. The benefits depend on the migration overhead and assumption about future execution. Our proactive approach further improves existing solutions by applying machine learning to online load prediction. We propose a fully distributed algorithm for adapting the prediction result to guide task offloading. The experiments are performed with an artificial test case and a realistic application named Sam(oa)^2 on three systems with different communication latencies. The results confirm that improvements can be achieved for important use cases compared to previous methods. Furthermore, this approach can support co-scheduling tasks across multiple applications.

- *Relation to thesis*: This paper is directly related to RQ2. We show how to perform online task characterization and generate the load prediction at runtime. Then, this information is applied to a proactive algorithm which guides task offloading to balance the load.
- *Author's contribution*: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft.

"User-defined Tools for Characterizing Task-Parallel Applications and Predicting Load Imbalance." In 2021 15th International Conference on Advanced Computing and Applications (ACOMP21), pp.98-105. IEEE, 2021, https://doi.org/10.1109/ACOMP53746.2021.00020.

- *Authors*: Minh Thanh Chung and Dieter Kranzlmüller
- *Paper's contribution/Abstract*: Parallel applications can be built portably on heterogeneous shared or distributed memory systems using task-based programming models. The decomposition into tasks allows to address load imbalance in parallel programs more easily, if knowledge about the application's characteristics can be obtained. In our paper, we introduce an approach for characterizing tasks at runtime using callback functions and a dedicated thread per rank for tool isolation with minimal perturbation of the target application's execution. With the characterization, prediction of execution time can be achieved using a machine learning approach. This serves as input for repartitioning or migrating the tasks such that the overall execution time can be improved. The results of our experiments using microbenchmarks and realistic applications confirm the benefits of our solution, in which the predicted information can adapt dynamic load balancing to large-scale use-cases.
- *Relation to thesis*: This paper is directly related to SQ1. We propose a practical scheme of how online task characterization and load prediction work. The results confirm that our design is lightweight and feasible to adapt this information to load balancing algorithms.
- *Author's contribution*: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, and Writing – original draft.

"Predictive, reactive and replication-based load balancing of tasks in Chameleon and Sam(oa)2." In Proceedings of the Platform for Advanced Scientific Computing Conference (PASC21), pp.1-10, 2021, https://doi.org/10.1145/3468267.3470574.

- *Authors*: Philipp Samfass, Jannis Klinkenberg, Minh Thanh Chung, and Michael Bader.
- *Paper's contribution/Abstract*: Increasingly complex hardware architectures as well as numerical algorithms make balancing load in parallel numerical software for adaptive mesh refinement an inherently difficult task, especially if variability of system components and unpredictability of execution time comes into play. Yet, traditional *predictive* load balancing strategies are largely based on cost models that aim to predict the execution time of computational tasks. To address this fundamental weakness, we present a novel *reactive* load balancing approach in distributed memory for MPI+OpenMP parallel applications that is based on keeping tasks speculatively replicated on multiple MPI processes. Replicated tasks are scheduled fully reactively without the need of a predictive cost model. Task cancellation mechanisms help to keep the overhead of replication minimal by avoiding redundant computation of replicated tasks. We implemented our approach in the Chameleon library for reactive load balancing. Our experiments in the parallel dynamic adaptive mesh refinement software sam(oa)^2 demonstrate performance improvements in the presence of wrong cost models and artificially introduced noise to simulate imbalances coming from hardware variability.
- *Relation to thesis*: The relevance of this paper is indirect in that we figure out the limit of current reactive approaches and open a hypothesis of proactive approach.
- *Author's contribution*: Data curation, Formal analysis, Investigation, Validation, Visualization, and Writing - review & editing

"Scheduling across multiple applications using task-based programming models." In 2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM), IEEE, pp.1-8. In conjunction with the International Conference for High Performance

Computing, Networking, Storage and Analysis (SC20), November 2020, https://doi.org/10.1109/IPDRM51949.2020.00005.

- *Authors*: Minh Thanh Chung, Josef Weidendorfer, Philipp Samfass, Karl Fürlinger and Dieter Kranzlmüller
- *Paper's contribution/Abstract*: Task-based programming models have shown their potential for efficiency and scalability in parallel and distributed systems. With such a model, a parallel application is broken down into a graph of tasks, which are subsequently scheduled for execution. Recently, implementations of task-based models have addressed distributed memory and heterogeneous systems with accelerators. However, the problem of scheduling tasks as well as allocating resources at runtime is still a challenge. In this paper, we propose coordinated and cooperative task scheduling across multiple applications. The main idea is to exploit the application's idle time e.g. from imbalance to serve tasks from another application. The experiments use Chameleon, a task-based framework for reactive tasking in distributed memory systems. In various example scenarios, we show improvements in CPU utilization of 5%-15% by coordinated scheduling.
- *Relation to thesis*: This paper is directly related to SQ2 in this thesis. We show an idea and a methodology of co-scheduling tasks across multiple applications instead of balancing only the load in a single application. The paper reveals our analysis and how the scheme works in practice. For the evaluation, our results confirm a benefit of 5%-15% improvement compared to the baseline.
- *Author's contribution*: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, and Writing - original draft.

(B) Outside the scope of this dissertation:

"A Profiling-based Approach to Cache Partitioning of Program Data." In the 23rd International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'22), 2022, https://doi.org/10.1007/978-3-031-29927-8_35.

- *Authors*: Sergej Breiter, Josef Weidendorfer, Minh Thanh Chung, and Karl Fürlinger
- *Abstract*: Cache efficiency is important to avoid unnecessary data transfers and to keep processors active. Cache partitioning, a technique to virtually divide a cache into multiple partitions, has become available in recent hardware. Cache partitioning can improve efficiency by isolating data with high temporal locality to avoid its early eviction before reuse. However, deciding on the partitioning is challenging, because it depends on the locality of reference. To facilitate the decision-making, we propose a profiling-based approach that measures locality, providing knowledge for cache partitioning without requiring manual code analysis. We present a profiling tool and confirm its benefits through experiments on Fujitsu's A64FX processor, which supports the cache partitioning mechanism called *sector cache*. Our results show ways to optimize program codes to improve cache efficiency.
- *Author's contribution*: Formal analysis, Validation, Visualization, and Writing - review & editing.

"From Transcripts to Insights for Recommending the Curriculum to University Students." SN Computer Science Journal (1), No.6, pp.1-14, 2020, https://doi.org/10.1007/s42979-020-00332-7.

- *Authors*: Thong Le Mai, Minh Thanh Chung, Van Thanh Le, and Nam Thoai
- *Abstract*: Student data plays an important role in evaluating the effectiveness of educational programs in the universities. All data is aggregated to calculate the education criteria by year, region, or organization. Remarkably, recent researches showed the data impacts when making exploration to predict student performance objectives. Many methods in terms of data mining were proposed to be suitable to extract useful information in regards to data characteristics. However, the reconciliation between applied methods and data characteristics still exists some challenges. Our paper will demonstrate the analysis of this relationship for a specific dataset in practice. The paper describes a distributed framework based on Spark for

extracting information from raw data. Then, we integrate machine learning techniques to train the prediction model. The experiments results are analyzed through different scenarios to show the harmony between the influencing factors and applied techniques.

- *Author's contribution*: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, and Writing - original draft.

1.6. Thesis Outline

The structure of this thesis is shown in Figure 1.4. The presented order emphasizes seven parts, where each indicates one chapter. Chapter 1 highlights the overview of our topic and the problem formulation, leading to the motivation and research questions. Additionally, it also introduces the methodology and contribution. Chapter 2 is followed by From Work Stealing to Reactive Load Balancing. We aim to clarify the preliminaries, terminologies, and go through:

1. Parallel Programming Models: implies the instruments we use to study dynamic load balancing.
2. Task-based Parallel Applications: alludes to the object we use to study dynamic load balancing.
3. Related Work: shows the taxonomy of dynamic load balancing and the state-of-the-art approaches in distributed memory systems.

Chapter 3 shows our performance model to analyze the bound of reactive load balancing approaches. The inputs are generalized as a given distribution of T tasks on P processes, the load value of tasks (w), imbalance ratio (R_{imb}), the overhead of task migration (delay time denoted by d) and balancing operation (denoted by $O_{\text{balancing}}$). The model focuses on possibilities when the existing approaches are bounded under the constraints of imbalance level and overhead.

Chapter 4 introduces our new proactive load balancing approach. The main idea revolves around how we can predict the tasks' load values at runtime and better guide balancing by proactive task offloading. When we have more knowledge about load values, we can better estimate the number of tasks and potential processes. This approach facilitates two proactive task offloading methods: feedback task offloading and ML-based task offloading. Building upon proactive task offloading, we further introduce an extension of co-scheduling tasks across multiple applications. Our implementation offers a proactive scheme exploiting task-based programming models, where the main threads (execution threads) execute tasks, a dedicated thread ($Tcomm$) performs load prediction and proactive task offloading. For load prediction, $Tcomm$ performs characterizing tasks, training prediction models, and offloading tasks.

For evaluation, both simulator and reference implementation are used as proof of concepts in Chapter 5. We describe the experiments in Chapter 6, where we use synthetic microbenchmarks and a realistic use case of adaptive mesh refinement. Finally, we summarize the conclusions, discuss identified problems, and provide an outlook for future work in Chapter 7.

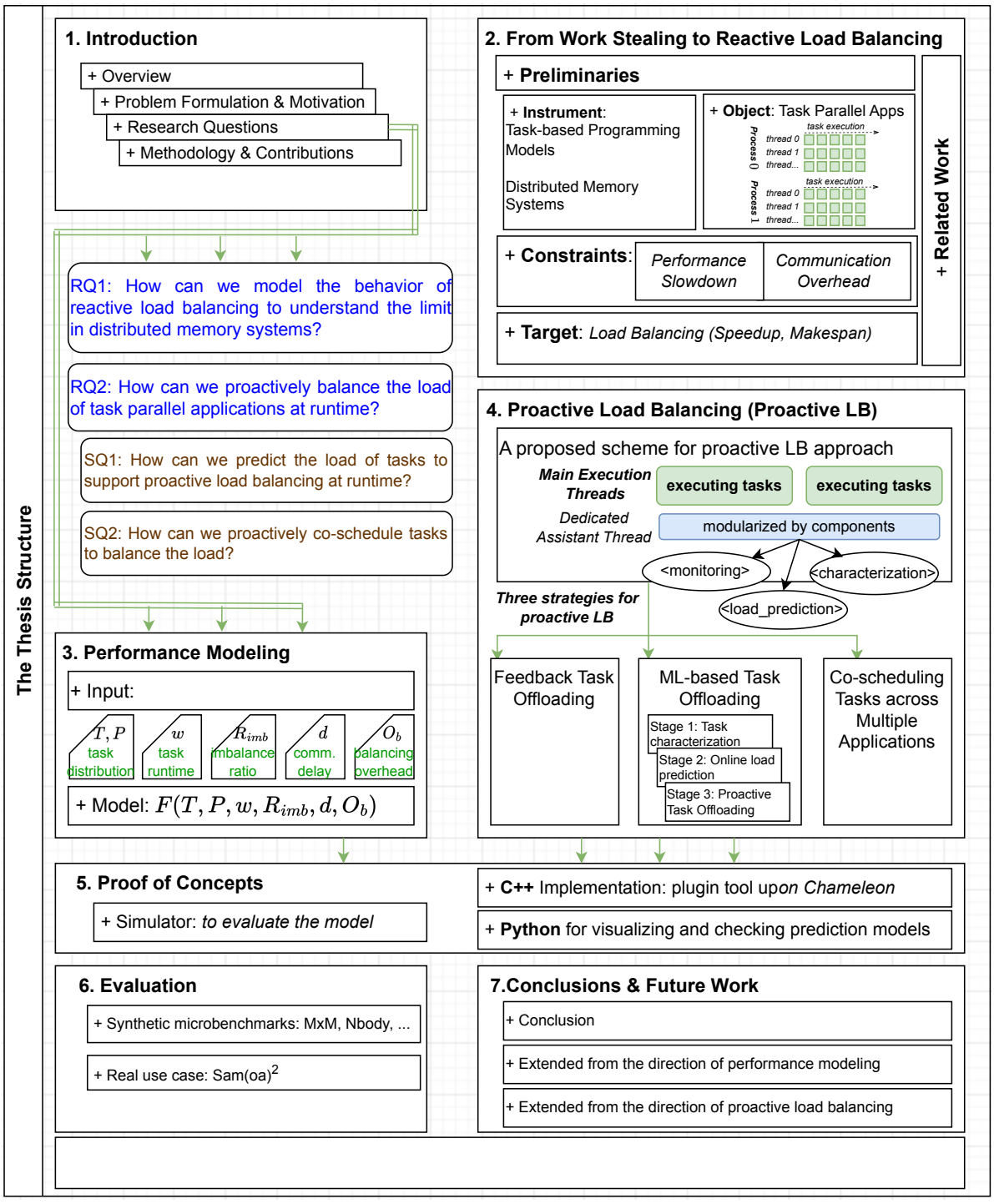


Figure 1.4.: The structure of this thesis.

2. From Work Stealing to Reactive Load Balancing

Based on the overview in Chapter 1, Chapter 2 provides in more detail terminologies, task-based programming models and runtimes, applications and state-of-the-art related to dynamic load balancing in distributed memory systems. The content of Chapter 2 is summarized in Figure 2.1 as an inverted pyramid. First, the top view of parallel computing is shown in Section 2.1 and we describe in more detail the concepts as well as terminologies associated with the architecture of distributed memory systems. Second, we go in-depth on how programming models are designed for shared and distributed memory systems in Section 2.2. Hereby, task-based parallel programming models are emphasized as an instrument to research dynamic load balancing. Following that, task-based parallel runtimes and applications are addressed as the objects introduced in Section 2.3. Finally, the related work from work stealing to reactive load balancing is analyzed in Section 2.4.

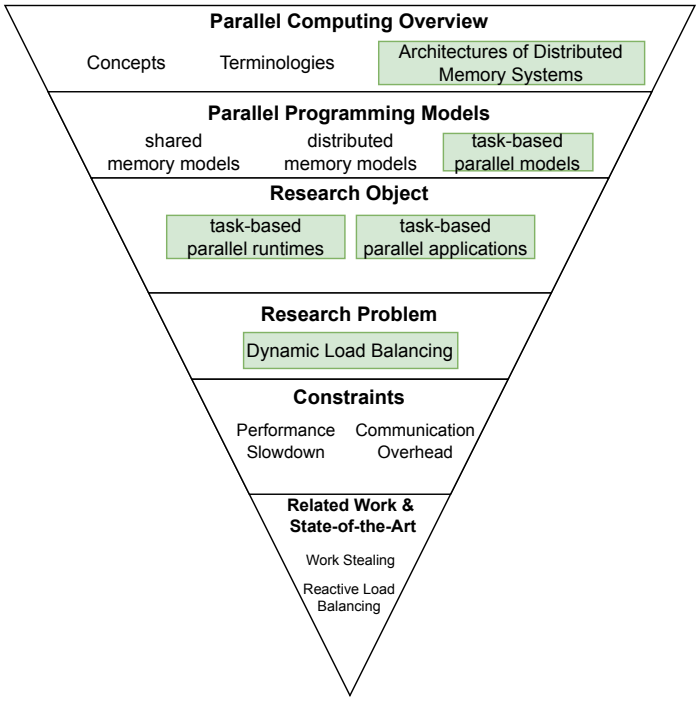


Figure 2.1.: The overview of Chapter 2

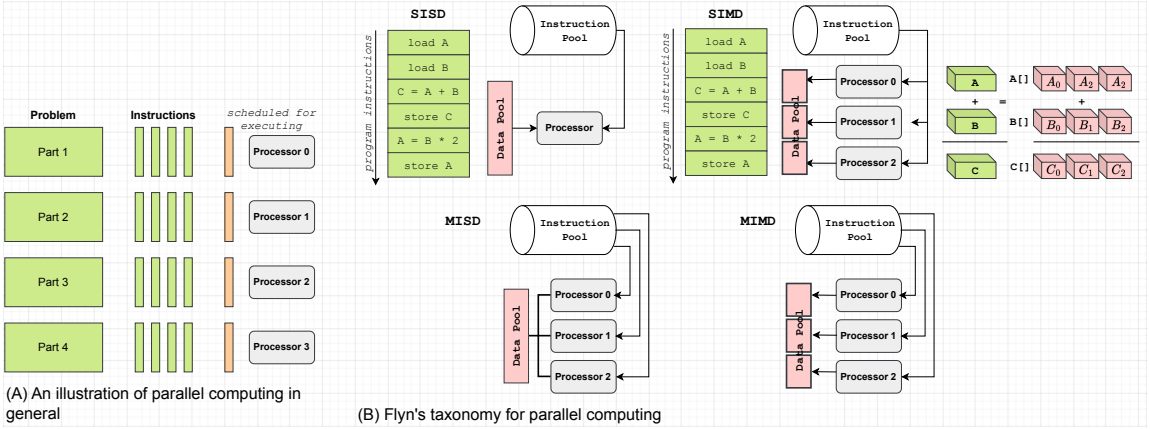


Figure 2.2.: A generic view of parallel computing [LLN].

2.1. Preliminaries

The majority of applications are written as serial or sequential computations. However, suppose our applications have complex computations and are able to be broken into smaller tasks running simultaneously. In that case, the concept of parallel computing can be applied to reduce overall execution time. Parallel computing denotes the use of multiple computing resources to execute a computational problem simultaneously. This leads to the motivation of parallelism in both software and hardware. Software refers to parallel applications, programming models, or support libraries. Hardware refers to parallel computing technology, such as multicore architectures or compute clusters.

Parallelism: can happen in different forms, e.g., bit-level parallelism, instruction-level parallelism, data parallelism, and task parallelism [Kum+94] [LLN]. Simply put, we assume that a program includes a series of instructions. Sequentially, the program executes one instruction after another, where only one processor is used, and one instruction is executed at once. In parallel, the program can be split into smaller parts running on different processors, where each part now includes a subset of instructions. At the same time, the instructions of different parts can be scheduled to run on different processors. Figure 2.2 (A) illustrates an example of parallel computing with four processors. From left to right, we can see the problem divided into four parts, shown as *Part* 1, *Part* 2, *Part* 3, and *Part* 4. Each part includes instructions that are scheduled for executing simultaneously on four processors (indexed from *Processor* 0 to *Processor* 3).

Regarding classification, Flynn's taxonomy [Kum+94] is widely used. There are four classes shown in Figure 2.2 (B), including SISD, SIMD, MISD, and MIMD. Each class is depicted next to its corresponding acronym. The main distinction is based on **Instruction Stream** and **Data Stream**, corresponding to the view of von Neumann Computer Architecture [EL99]. In detail, each class is characterized as follows.

- SISD: Single Instruction Single Data (illustrated in Figure 2.2 (B) under the acronym, SISD). Only one instruction is performed, and one data item is manipulated at a time. One arrow stands for an instruction loaded from the instruction pool, and another stands for data manipulation.
- SIMD: Single Instruction Multiple Data (illustrated in Figure 2.2 (B) under the acronym, SIMD). This class of parallel computing has multiple processors, where each processor runs the same instruction on its own data.
- MISD: Multiple Instruction Single Data (illustrated in Figure 2.2 (B) next to the acronym, MISD).
- MIMD: Multiple Instruction Multiple Data (illustrated in Figure 2.2 (B) next to the acronym, MIMD), showing that multiple processors execute multiple instructions and operate multiple data items simultaneously.

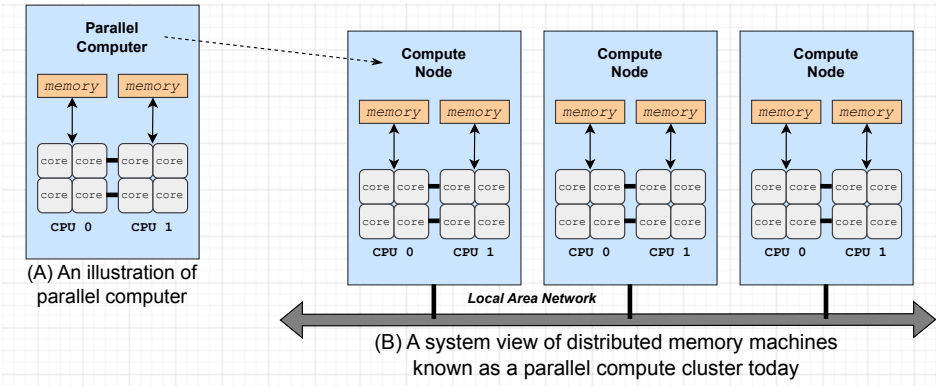


Figure 2.3.: An example of parallel computers and distributed memory systems know as a parallel compute cluster.

Expanding from MIMD, some researches divide it into the programming models called Single Program Multiple Data streams (SPMD) [Dar01] and Multiple Programs Multiple Data streams (MPMD) [FZJ].

Parallel computers: refers to a computer or a machine with more than one processor (CPU). Sometimes, people imply a CPU as a CPU socket, the physical interface on a motherboard where a CPU is located. In modern computing architectures, a CPU contains multiple cores called single processing units, and each core can execute instructions independently. Each CPU can have its individual memory, where the total memory in a parallel computer is the sum of all individual memories. Figure 2.3 (A) reveals an example of a parallel computer with two CPUs, each with its own memory and containing four cores.

Memory access in parallel computers is a complex topic. We can also use the types of memory access to characterize the type of parallel computers. There are mainly two types, shared memory and distributed memory. Shared memory can be noted with all processors accessing the same memory, or a so-called shared address space. Distributed memory can be seen that each CPU has a memory and each refers to a variable on its own memory.

From the view of hardware manufactures, the memory architectures of parallel computers are known as

- Uniform Memory Access (UMA): all memory locations are accessible to all processors. The access time can be assumed to be identical. A limitation of UMA is the maximum number of processors that can be difficult to expand.
- Non-Uniform Memory Access (NUMA): each process has a separate memory. This memory architecture can deal with the limit of UMA but leads to a situation where a processor or the cores of a processor access its own memory (local memory) fast and access the other processors' memory (remote memory) slower. Distributed memory implies that a processor cannot directly see another processor's memory. From the programmer's view, there can be a reference to physically and logically distributed memory, which means whether memory is distributed or just appears distributed [Eij10]. NUMA can be seen as logically and physically distributed, although the distributed nature of NUMA is not apparent to the programmer. Today, most modern computers are made with multicore processors and NUMA architectures.

In general, the goal of using parallel computing is to get higher computation performance and access more memory. However, as scientific problems scale up, the complexity of solving them can take a long time to execute on a single computer. Therefore, the idea of parallel compute clusters was built and widely used.

Parallel compute clusters: refers to the architecture used in most supercomputers today. Parallel compute clusters are examples of distributed memory systems. A cluster consists of many computers; in particular, we imply many parallel computers, as explained above. The computers in a cluster are so-called compute nodes. Figure 2.3 (B) demonstrates a parallel compute cluster. All compute nodes can operate as a unified system. Each node connects to another via a local area network (the so-called interconnection network) [BB99]. The exchange of data or messages between different processors across different nodes is physically distributed memory, where communication overhead might affect the performance of parallel applications. In this thesis, distributed memory indicates the view of memory access in parallel compute clusters.

Terminologies: summarize and emphasize several terminologies based on the explanation above. The following might repeat some notations as well as terms used throughout the thesis.

- Cluster: denotes a parallel compute cluster, and is also called an HPC cluster, HPC system, or supercomputer.
- Node: indicates a compute node in a cluster. Inside a compute node, we have
  - Processor or CPU: indicates central processing unit. In computer hardware, a CPU is placed in a CPU socket, which contains mechanical components providing mechanical and electrical connections between a microprocessor and a printed circuit board (PCB). Sometimes, a mentioned CPU socket can also be understood as referring to a CPU.
  - Core: is single processing unit in a processor. There can be multiple cores inside a processor. Different cores in a node can belong to the same processor or different processors. At the operating system (OS) level, there are
    - Process: is an instance of a program. When a process is created and executed, it is defined with its own resources, e.g., memory, file descriptors.
    - Thread: is an entity within a process. A process can spawn multiple threads inside. While processes are isolated, threads share memory and address space.

    Depending on context switching and mapping algorithm, processes or threads can be mapped dynamically to cores. For performance purposes, a process or thread is often pinned to a specific core in HPC, reducing context switching time. Furthermore, a technology introduced by manufacturer Intel is hyperthreading (Hyper-Threading Technology as HTT or HT) that a single core can run multiple threads. In this case, we call them logical cores in a physical core.
  - Memory: this thesis refers to NUMA architecture in a node (NUMA domain). Each CPU has an individual memory, often called a NUMA node. However, we avoid using NUMA node to ensure no conflict with "node" in compute node. We consider the view of memory access in a single node as shared memory. Therefore, it is important to note that
    - Shared memory: indicates memory access in a single node.
    - Distributed memory: indicates memory access across nodes.
- Communication: implies data or information exchange among processes. These processes can be inside a node or across nodes. To evaluate communication speed, bandwidth is defined as the maximum transmission performance of a network. Generally, bandwidth (B) today is measured by megabits, megabytes, or gigabytes per second (Mbps, MBps, or GBps). In HPC networks, communication overhead refers to
  - Latency (λ): the delay time between sending and receiving the header of a message. λ can be measured in seconds or milliseconds.
  - Transmission time or Delay time (d): the time for transmitting an entire message between two nodes. Delay (d) depends on the size of a message. In the case of no conflict, $d = \lambda + \frac{s}{B}$, where s is the message size.

From the application perspective, users can implement parallel applications using different programming models. The next section introduces general and widespread parallel programming models

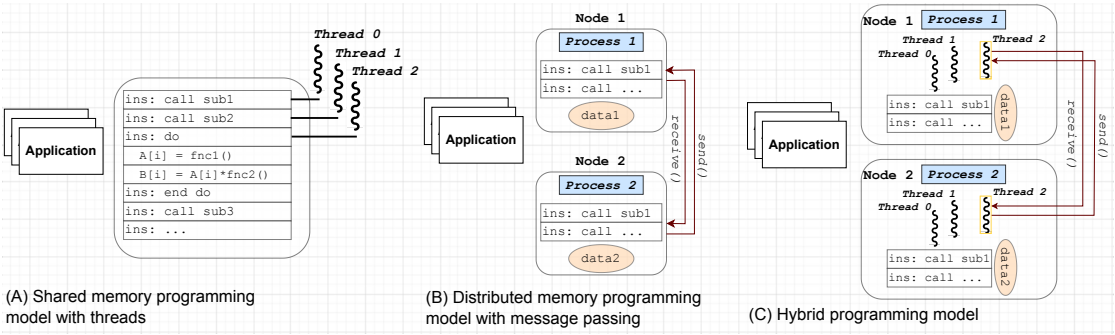


Figure 2.4.: An overview of shared memory and distributed memory programming models along with their hybrid model.

in common use. Following that, we emphasize our focus on task-based parallel programming models and task-based parallel applications to deal with dynamic load balancing.

2.2. Task-based Parallel Programming Models

A programming model forms an abstraction layer that connects user applications and parallel computer architectures, leveraging a programming-support library/runtime. Before the advent of task-based parallel programming models, shared and distributed memory models have been used mainly. However, there is no clear boundary of these models. The model design relies on the development of computing and memory architectures, such as multicore, manycore, NUMA architectures. The following introduces conventional models, then details the principle of task-based parallel models.

Shared Memory Programming Model: enables processes or threads to share a common address space. In this model, processes or threads can communicate and synchronize by reading/writing directly from/to shared memory locations. The model allows efficient data sharing. Figure 2.4 (A) illustrates a parallel program using shared memory programming with threads. Here, Thread 0, 1, 2 execute the instructions simultaneously, and the data such as array A and array B are shared. Shared memory programming can be characterized by:

- Advantage: simple and straightforward to write programs in parallel. Communication between processes/threads is faster and more efficient than other models requiring inter-process communication.
- Disadvantage: difficult to manage data locality. There are challenges in dealing with complex synchronization, race conditions. Incorrect management of shared memory locations can lead to bugs. Furthermore, the scalability might be limited in the scope of a single machine.
- Implementation: the most common library is POSIX threads (so-called Pthreads) [But97]. Besides that, OpenMP [DM98] is an industry-standard library to support shared memory programming, widely used in various operating systems and compilers.

Distributed Memory Programming Model: does not have a shared address space. Distributed memory in this thesis points to physically distributed memory systems where compute nodes connect to each other via a network. Note that memory architectures can also be called distributed in a single machine with NUMA architecture. However, it just appears as distributed memory from the programming view. We demonstrate a distributed memory programming example in Figure 2.4 (B), assuming we have two compute nodes, each creates a process to execute parallel application. The only way to exchange information across these nodes is message passing. For example, Process 2 sends a message to Process 1 and vice versa. Distributed programming models can be characterized by:

- Advantage: data and information are exchanged explicitly by sending and receiving messages. Programmers can coordinate nodes to execute tasks and share data on large problems. The

scalability of this model is well-suited for HPC and large-scale parallel applications.

- Disadvantage: communication overhead is challenging in data decomposition and combination. The problem in a distributed memory model is divided into smaller parts, which are distributed over nodes. After execution, their results and required data need to be combined, which may take time and overhead. Load balancing is also a challenge when task distribution is irrelevant to the system performance model.
- Implementation: the most popular library is based on the Message Passing Interface (MPI) standard [Gro+96]. MPI supports API and subroutines that users can define data exchange communication explicitly or implicitly. Besides, several libraries that support message passing at higher levels can also be characterized as distributed memory programming models known as Partitioned Global Address Space (PGAS) [De +15]. PGAS provides an abstraction layer for programmers. We may not need to define the communication of data exchange explicitly. In this thesis, we classify the programming models like PGAS as data-parallel programming models.

Data Parallel Programming Model: refers to Partitioned Global Address Space (PGAS) [De +15]. The idea is to achieve a higher level of abstraction from a programmers perspective. Even on shared or distributed memory, it treats address space globally. For example, the global data structure can be split up logically across tasks in distributed memory systems. Data parallel programming can be characterized by:

- Advantage: user-friendly from the programming point of view. The model shows considerable benefits on regular data structures like arrays or matrices.
- Disadvantage: in some use cases with complicated data structures, this model cannot ensure performance as well as portability, such as data locality references, the distribution of sparse and irregular data objects. These are important issues for evaluating the efficiency of data-parallel programming models.
- Implementation: several implementations such as Coarray Fortran [Mel+09], UPC [Zhe+14], X10 [Cha+05], Chapel [CCZ07].

Hybrid Programming Model: refers to combining more than one programming model. We have mentioned the hybrid model of shared memory and distributed memory programming, such as MPI+X or MPI+OpenMP [RHJ09] in Chapter 1. Shared memory programming models such as OpenMP allow users to specify shared memory locations and threads, but the scalability is limited in a single node. Distributed memory programming models such as MPI allow users to specify communication patterns and processes across nodes. A MPI process is denoted as MPI rank. In MPI-based applications, message passing can occur inside a node when we fully create the number of MPI ranks corresponding to the number of cores. Therefore, the hybrid model exploits benefits from both shared memory and distributed memory programming, such as reducing unnecessary MPI communication inside a node, overlapping communication and computation. Technically, a multi-core processor within a node only needs to create one MPI process, where this process can spawn multiple threads. One or a few threads take care of MPI communication while the others execute computation tasks. With the trend of heterogenous computing architectures today, this model is the most relevant. Heterogeneous computing architectures are then implemented as CPUs accelerated by graphics processing unit (GPU) architectures [ZO11], which are widely used in today's HPC clusters and supercomputers [TOP93]. Generally, hybrid models can be characterized by:

- Advantage: enables overlapping communication and computation and dealing with data locality. This can be applied similarly for CPU-GPU applications in distributed memory systems, where one or a few threads control communication across nodes, the other threads control computation tasks run on CPU or offloaded to GPU.
- Disadvantage: traditional parallel applications need to be changed following the hybrid model. This might lead to conflicts if support libraries or frameworks cannot control the mix of multiple processes along with multiple threads. Furthermore, the hybrid programming model such as MPI+OpenMP might also require users' effort to change conventional applications with pure MPI or pure OpenMP.

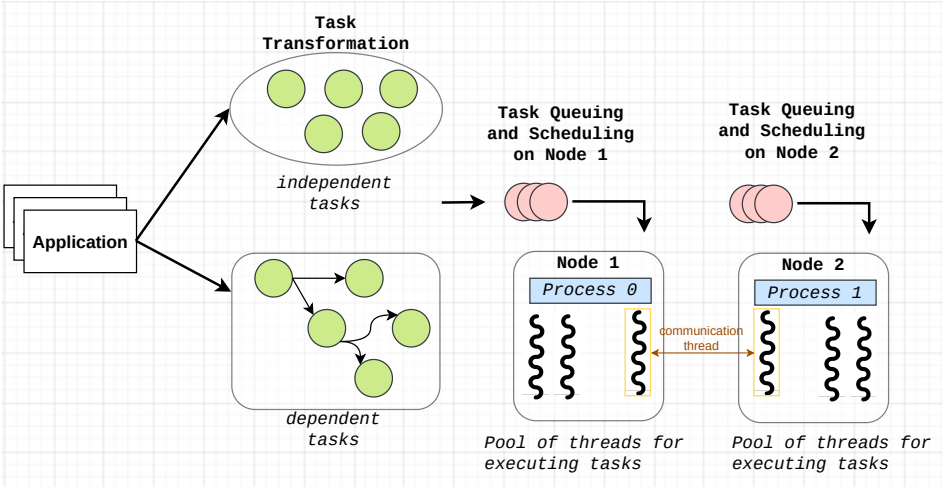


Figure 2.5.: An overview of task-based parallel programming model.

- Implementation: MPI+OpenMP as an example. Besides OpenMP, MPI can also combine with Pthreads for supporting multithreading inside a process.

Figure 2.4 (C) shows an example of a hybrid MPI+OpenMP programming model. The illustrated application is executing on two compute nodes (*Node* 1 and *Node* 2). Each node deploys an MPI process, *Process* 1 belonging to *Node* 1 and *Process* 2 belonging to *Node* 2. Each process spawns three OpenMP threads, where *Thread* 0 and *Thread* 1 execute the main computation parts, while *Thread* 2 is dedicated to performing communication.

Task-based Parallel Programming Model: is a higher abstraction model based on the hybrid programming model. With the previous programming models in general as well as the hybrid model in specific, it is difficult to manage concurrency issues effectively. Furthermore, today's perspective of low-level programming, such as better performance by tightly controlling hardware, has been changed because users have to make an effort to optimize code among computation, communication, and synchronization. For example, synchronization barriers are often declared in shared and distributed memory programming models and require the user's responsibility. This can lead to performance issues, e.g., causing much idle or waiting time, if we do not control the synchronization tightly.

Task-based parallel programming models focus on the criteria of user-friendly coding, portability, and well-expressing parallelism. The idea of task-based models is to separate the part of tasks, that need to be computed, from the part of computing resources in parallel execution. Users can reduce the burden of controlling how tasks are executed, synchronized, or parallelized. Instead, we just need to define what tasks are, their relationship, and whether they are dependent or independent. Dependent tasks can be represented as a graph of tasks, which is a so-called task graph. An interesting case study formulated as a task graph is game engine [Reg+22], which is studied to explore task scheduling by Regragui et al.

Figure 2.5 illustrates an example of task-based parallel programming. From left to right, the application is assumed to be transformed into a set of tasks. Task transformation depends on how we define a task. A task often points to a compute code entry and its data. The way of transforming tasks or porting normal applications into task-based applications is called tasking. In Figure 2.5, we assume two cases of task transformation: independent and dependent. The dependent tasks can be addressed as a graph. The next step is queuing and scheduling tasks. Assume we have two compute nodes, *Node* 1 and *Node* 2. Tasks assigned to *Node* 1 are queued and scheduled for execution on *Node* 1. Similarly, tasks assigned to *Node* 2 are queued and scheduled for execution on *Node* 2. Following this, executing tasks on each node is controlled by the task-based programming model, which is usually implemented as a library or framework. During execution, the task-based programming

model controls mapping tasks to threads for computation; thus, a task-based model is also called a task-based runtime. After computation, the application's working flow is returned to users.

In Figure 2.5, the view of computing resources on *Node* 1 and *Node* 2 is similar to the hybrid model, where each node contains a process, a process spawns a pool of threads for task execution. However, users only need to define tasks and are relaxed in managing parallelization, communication, concurrency, or synchronization. We characterize task-based models as follows.

- Advantage: user-friendly to develop parallel applications. We can simplify parallel applications with tasks and task relationships. It can be easier to control synchronization and concurrence.
- Disadvantage: porting applications from conventional to task-based models might lead to performance issues. Besides, defining tasks in some specific applications is sometimes difficult.
- Implementation: there are more and more libraries or frameworks developed to support task-based parallel programming, such as StarPU [Aug+11], OmpSS [Dur+11], HPX [Kai+14], Taskflow C++ [Hua+21], etc. They are developed in different languages like C++, Python.

The next section will show how task-based parallel runtimes are categorized and how task-based parallel applications are characterized in terms of performance issues.

2.3. Task-based Parallel Runtimes and Applications

Tasks can appear in different contexts. For example, we often use tasks to indicate a code region in shared and distributed memory programming models. However, this definition is loose and not clear to determine where is the task space and which is the task data or task input/output.

In task-based parallel programming, tasks are defined more consistently. We define a task by a code entry and its data. The data indicate input, output, or data region that a task can manipulate. When the input/output of a task is associated with other tasks, these tasks are dependent. Porting a normal application into a task-based one can help simplify the parallel execution such a well-expressed parallelism.

In terms of execution, a specific implementation of task-based programming models is considered as a library or a framework. This library attempts to separate parallel programs into a pool of tasks and a pool of computing resource units. The pool of computing resource units indicates threads within a process. These two pools imply two levels: application level and system level. From the application level, we can see task-based applications as a black box. From the system level, we can see task-based programming models as runtimes. The runtimes manage tasks and handle performance portability [ACB21]. Performance portability means:

- Users can simplify parallel applications as tasks alongside their data.
- Users can quickly port parallel applications to new and different platforms.
- Users can be relaxed in terms of task scheduling, load balancing, and overlapping communication-computation at the side of task-based runtimes.

Based on the survey of Thoman et al. [Tho+18], we summarize the main characteristics for classifying task-based parallel programming runtimes. These characteristics refer to application programming interfaces (API) in a given task-based runtime, which can be developed like a library, a language extension, or a language. The classification has four aspects: Architecture, Task System, Task Management, and Engineering. Table 2.1 summarizes this classification associated with various task-based runtimes and libraries today.

- Architecture: highlights the target systems or computer architectures. In detail, the related factors of architecture are further divided into three components.
  - Communication model: shared memory, message passing, and an abstraction model called global address space, which is created as a shared memory model for distributed memory systems. These terms are denoted by *smem*, *msg*, *gas*, respecively in Table 2.1.

APIs	Architecture			Task System		Task Management		Engineering
	Communication model	Distributed memory	Heterogeneity	Graph structure	Task partitioning	Worker management	Work mapping	
C++ STL [Kor15]	smem	✗	✗	dag	✗	**i**	**i/e**	lib.
TBB [WP08]	smem	✗	✗	tree	✗	**i**	**i**	lib.
HPX [Kai+14]	gas	i	**e**	dag	✓	**i/e**	**i/e**	lib.
Legion [Bau14]	gas	i	**e**	tree	✓	**i**	**i/e**	lib.
PaRSEC [Hoq+17]	msg	e	**e**	dag	✗	**i**	**i/e**	lib.
Chameleon [Kli+20a]	msg	e	✗	dag	✓	**e**	**i/e**	lib.
Taskflow [Hua+21]	smem	✗	**e**	dag	✓	**i/e**	**i/e**	lib.
OpenMP [Ayg+09]	smem	✗	**i**	dag	✗	**e**	**i**	ext.
Charm++ [Acu+14]	gas	**i**	**e**	dag	✓	**i**	**i/e**	ext.
OmpSs [Dur+11]	smem	✗	**i**	dag	✗	**i**	**i**	ext.
StarPU [Aug+11]	smem	**e**	**e**	dag	✓	**i**	**i/e**	ext.
Cilk Plus [Rob13]	smem	✗	✗	tree	✗	**i**	**i**	lang.
X10 [Cha+05]	gas	**i**	**i**	dag	✓	**i**	**i/e**	lang.

Table 2.1.: An overview of the programming interface characteristics in various task-based parallel programming runtimes.

  - Distributed memory system: the compatibility with distributed memory is explicit, implicit, or no support. The terms of "implicit", "explicit", "no support" are denoted by *i*, *e*, ✗, respectively.
  - Heterogeneity: indicates whether tasks can be ported to accelerators or not. The classification is also denoted by explicit, implicit, or no support with *i*, *e*, and ✗, respectively.
- Task System: shows how tasks are built and simplified with or without dependencies. The main factors are structure and task partitioning. Furthermore, we can also classify them by result handling and task cancellation.
  - Graph structure: the possibilities of representing task dependency include tree (*tree*), acyclic graph (*dag*), arbitrary graph (*graph*), or no dependency.
  - Task partitioning: indicates whether a task is atomic (*atom* or ✓) or can be divisible (*div* or ✗).
- Task Management: is related to task assignment and resource management. Moreover, this characteristic focuses on how tasks are supported for resilience as well as debugging.
  - Worker management: indicates whether the worker threads/processes (which execute tasks) are maintained "explicit" by users or "implicit" (automatically) by the environment.
  - Work mapping: describes the strategy that tasks are assigned to computing resource units. The possibilities include "explicit" mapping or "implicit" mapping.
- Engineering: shows how a task-based programming interface is used or implemented, such as a library (*lib.*), a language extension (*ext.*), or a language (*lang.*).

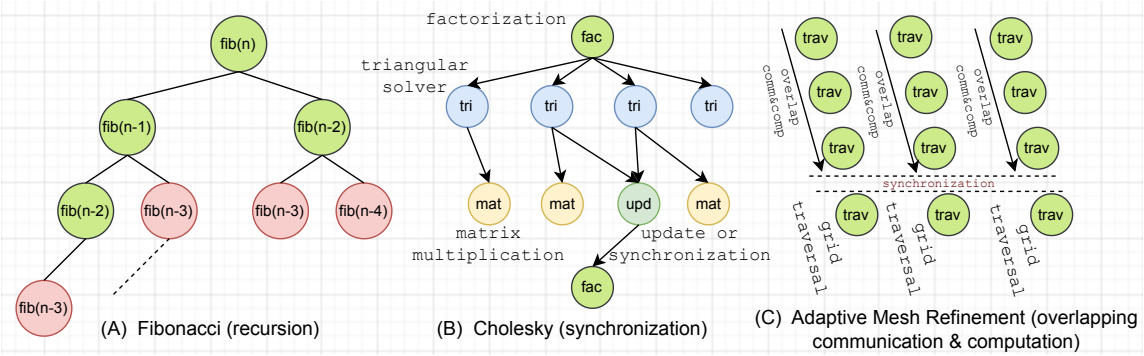


Figure 2.6.: Three examples illustrated as the three use cases of task-based parallel applications.

There are three common use cases for porting an application to a task-based parallel application. They are known as recursion, complex synchronization, overlapping computation and communication for numerical simulations. These three use cases are illustrated in Figure 2.6, including (A) Fibonacci, (B) Cholesky factorization, and (C) Adaptive Mesh Refinement.

The task-based version of Fibonacci defines tasks by its compute function (named `fib(n)`). The input data of tasks is an integer number. All tasks are generated recursively, tasks nested in another task. Using task-based parallel programming, we can facilitate the parallelism in Fibonacci with cut-off strategies as shown in Figure 2.6 (A).

For Cholesky factorization, the difficulty is complex synchronization patterns. Porting Cholesky to a task-based parallel version can improve composability and clarify synchronization patterns. As shown in Figure 2.6 (B), the program behavior is broken into separate tasks, such as computing factorization (`fac`), triangular solver (`tri`), matrix multiplication (`mat`), update (`upd`) as well as synchronization.

The rest is an example of Adaptive Mesh Refinement shown in Figure 2.6 (C). It is known as a typical approach of numerical simulations in HPC. Here, tasks are defined by the function of traversing grid/mesh cells (denoted by `trav`). These tasks are independent, and their input data depends on the constraint parameters of simulation contexts, e.g., earthquake, tsunami. Executing Adaptive Mesh Refinement comprises multiple computation phases interwoven with synchronization phases, also called iterative execution. When running in distributed memory systems, each node executes an assigned part of the given grid. Therefore, there might be communication for exchanging grid information to compute the tasks. Using task-based programming, communication and computation can easily overlap. Especially the isolation between a pool of tasks and a pool of resources helps facilitate a dedicated communication thread by exchanging information during computation. Furthermore, Prat et al. [PCN18] also show the efficiency of task-based parallelism with another example, molecular dynamics (MD) simulations. Their MD implementation is 4.7× faster than the state-of-the-art implementations (LAMMPS [lam08]).

Task-based parallel runtimes help to simplify parallel programming. It is easier to manage tasks and control resources by separate pools. This offers an advantage for solving dynamic load balancing in distributed memory systems. The following section details how load balancing is formulated in task-based parallel applications and related work.

2.4. Related Work

2.4.1. Dynamic Load Balancing

This work revolves around load balancing problems in distributed memory systems. Our target objects are task-based parallel applications, where the main use case is shown in Figure 2.6 as iterative

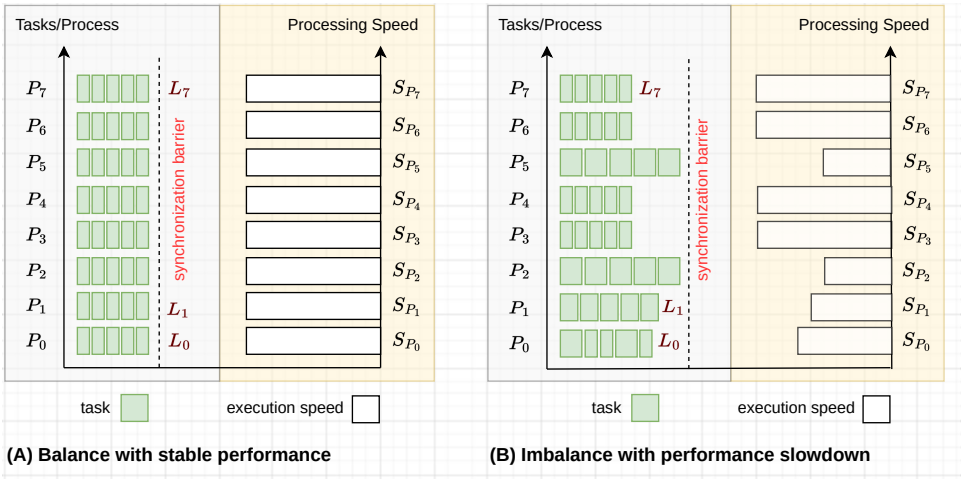


Figure 2.7.: Balance vs imbalance in the case of executing tasks on 8 processes.

execution of HPC numerical simulations.

In general, load balancing is to treat each process an equal share of the total load [Cyb89]. Almost all problem contexts and solutions are broadly categorized into "static" and "dynamic".

"Static" relies on a *priori* knowledge of load, implying that the static methods are performed before execution. People assume the given cost models are stable at runtime, which is feasible to generate optimal balancing algorithms. There are three simplified branches of static balancing methods:

- Static optimal task distribution
- Static optimal thread/process assignment
- Approximate or heuristic task scheduling

Static load balancing works optimally in ideal systems with a stable performance model. All processors execute tasks at the same speed. We can manipulate tasks, threads, or processes to balance the load. In principle, finding an optimal algorithm of load balancing for more than two processes is studied and addressed as an NP-hard problem [Bok81], [Bok88]. Under certain assumptions about the application behavior or system characteristic, there are several studies about the optimal algorithms for task distribution and thread/process assignment, such as [Bok87], [NST96]. Another direction is heuristic algorithms, which are also common because of their simplicity in implementation [Ben91] [BNG92]. The advantage of static load balancing is minimizing runtime overhead and is suitable when the execution time of tasks can be estimated before execution.

"Dynamic" or dynamic load balancing (DLB) is more suitable when load and performance models are unknown or variable at runtime. The cost models might also be varied during execution. For example, unexpected performance slowdown affects the load values of tasks.

As introduced in Section 1.2, we summarize again our imbalance context here. Assume that a parallel application is simplified as tasks by a given distribution of T tasks on P processes. Each process is assigned a subset of tasks T_i, where i indicates the process index, $i \in \{0, ..., P-1\}$. Each task has a load value, w_j where j indicates the task index, $j \in T_i$. The total load value of a process is determined by the sum of load values associated with the tasks assigned to that process, i.e., $L_i = \sum_{j \in T_i} w_j$. Compared to the balanced case, we highlight the cause of the imbalance case in Figure 2.7, where (A) shows balance with a stable performance model, while (B) shows imbalance caused by performance slowdown in processes P_0, P_1, P_2, P_5. Each figure is a double-axis chart, in which the left x-axis reveals the execution flow from left to right; the green boxes illustrate tasks. The length of the boxes determines the length of task execution time. The right x-axis demonstrates the

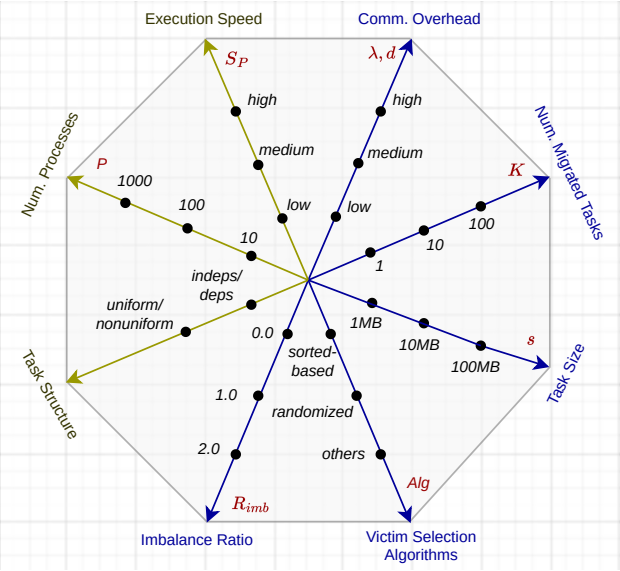


Figure 2.8.: Direct and indirect parameters impact the performance of dynamic load balancing.

execution speed of each process, where the length of horizontal bars determines a low or high speed. Figure 2.7 (B) emphasizes that performance might fluctuate during execution, and task migration is the only way to balance the load. Tasks should be migrated between a fast process and a slow process.

Regarding the evaluation metric, an imbalance is measured by a ratio calculated by maximum and average load values. In Section 1.2, this ratio is named R_{imb}. We can use the total load values like L_i to calculate R_{imb}, such as L_0, L_1, L_7 shown in Figure 2.7 (B). Here, the maximum L is L_5 (or L_2), and the average L is named $L_{avg} = \frac{\sum_{i=0}^{7} L_i}{8}$, where $P = 8$ and $R_{imb} = \frac{L_{max}}{L_{avg}} - 1 = \frac{L_2}{L_{avg}} - 1$. If each process has multiple threads to execute tasks, R_{imb} can also be calculated by the wallclock execution time per process, for example, $W_{avg} = \frac{\sum_{i=0}^{7} W_i}{8}$.

- If $R_{imb} \approx 0$, there is no imbalance.
- If $R_{imb} > 0$, there exists imbalance between the involved processes.

Corresponding to the imbalance, two groups of related parameters can directly or indirectly affect load balancing performance. Due to the isolation of computing resources between nodes in distributed memory systems, the only way to balance the load is moving tasks across processes or even across nodes. In the group of indirect parameters, we imply the parameters that affect imbalance level. In the group of direct parameters, we show the parameters that affect the performance of load balancing.

The green arrows of Figure 2.8 show the group of indirect parameters.

- **Num. Processes** (P): the number of involved processes. This value might affect process selection algorithms for task migration and number of migrated tasks.
- **Execution Speed** (S_P): the execution speed of a process. The difference in execution speeds causes imbalance. The number of slow and fast processes affects imbalance level (R_{imb}). Notably, R_{imb} can indirectly affect the number of migrated tasks.
- **Task Structure**: task system, where tasks can be dependent or independent, uniform or nonuniform load (determined by the w values of tasks).

The blue arrows of Figure 2.8 show the group of direct parameters.

- **Imbalance Ratio** (R_{imb}): imbalance level. It might challenge the applied balancing approaches because R_{imb} might be high or low. For example, when R_{imb} is very high, the number of migrated tasks can be large.

- **Num. Migrated Tasks** (K): the total number of migrated tasks. The number of migrated tasks at once (or per operation at a time) determines K. Therefore, getting an optimal value of K can be seen as a necessary condition, and getting a reasonable number of migrated tasks at once to an adequate process can be seen as a sufficient condition.
- **Task Size**: the data size of a task. This size affects delay time when we migrate tasks. Importantly, the number of migrated tasks is also affected by data size.
- **Communication Overhead** (`Comm.Overhead`): indicates latency (λ) and delay time (d) in migrating tasks. The value of d relies on the bandwidth (B) of a network and message size (s) [PD21]. Communication overhead (`Comm.Overhead`) is considered as a constraint of performing load balancing in distributed memory systems, which can delay information exchange and task migration. The total load per process is changed when tasks are migrated from one process to another. Hence, the load balancing performance is changed when `Comm.Overhead` affects task migration. In detail, the definition and calculation of λ and d are explained in Section 2.1.
- **Victim Selection Algorithms**: indicates how we can choose an adequate process for migrating tasks, for example, in the case we have multiple victims. The selection algorithms directly affect how many tasks are migrated and which processes are good candidates.

As mentioned in Chapter 1, the most closely related approach is called "work stealing" [BL99], which is common in shared memory systems. Also, we look upon shared memory systems as the group of load balancing without taking `Comm.Overhead` into account. The idea is to steal tasks from a busy process when the current process is idle. Many researchers have proposed methods for optimizing work stealing in terms of:

- Algorithms: optimize the general strategies for stealing tasks. The algorithms are divided into *nearest neighbor* (iterative) and *global* (direct). The group of *nearest neighbor* relies on successive approximations to a global optimal task distribution, and each balancing operation intends to task migration. These various methods are then categorized into *deterministic* and *stochastic*. On the one hand, *deterministic* methods proceed according to the rules of predefined distribution. On the other hand, *stochastic* methods distribute the tasks based on randomization.
  - *Deterministic* methods: diffusion [HW80] [SB77], dimension exchange [RWS88], and the gradient model [LK87].
  - *Stochastic* methods: randomized allocation and physical optimization. Randomized allocation is the simplest method in that the victims of stealing tasks are randomly selected [Adl+95]. Physical optimization is based on the analogies of parallel computers. These physical methods map a load balancing problem to a physical system and solve the problem using theoretical or experimental analysis by simulation [RSU91].
- Task Selection: refers to algorithms that focus on selecting appropriate tasks to optimize work stealing. This branch of research is often proposed when we are concerned with topology-aware for dynamic load balancing. For instance, Vikranth et al. [VWR13] introduce a task stealing strategy that dynamically analyzes the topology of the underlying hardware connections for on-chip NUMA multicore processors. Their experiments have shown an average of 1.24× better performance on NAS[1] Parallel Benchmark [Div96] compared to popular runtimes Cilk and OpenMP.
- Async. Stealing: refers to algorithms or strategies that support the operations of executing and stealing tasks asynchronously. This branch of research is often proposed to optimize work stealing as well as load balancing for multi-socket multi-core architectures [CGG14], or heterogeneous architectures [Kum+15].

Instead of stealing, another idea is explored from the level of controlling compute resources. When we look at the perspective of applications and parallel compute clusters, two levels consist of tasks and compute processing units (denoted by threads/processes). The pool of threads is malleable and

[1] A small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmark suite has been extended to include new benchmarks for unstructured adaptive meshes, parallel I/O, multi-zone applications, and computational grids.

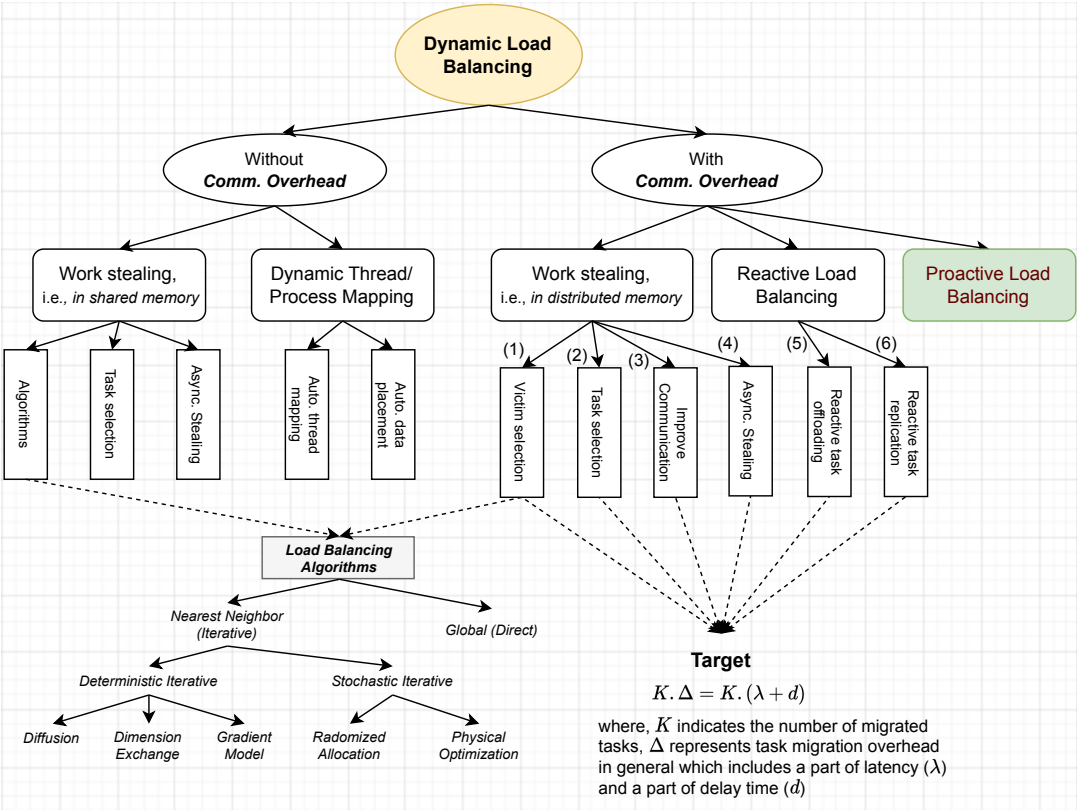


Figure 2.9.: A top-down tree of related work in dynamic load balancing.

can be used to balance the load. Therefore, threads can be dynamically controlled during execution [GCL09] [HIB10]. Other works are proposed by Papin et al. [Pap+14] [Pap+21], inspired by molecular dynamics simulations. The authors propose a load balancer as well as a scheduling tool with NUMA-aware allocations that allows cores to move virtually to balance the load by changing their computing load. Furthermore, some works are done by redistributing the computational power of cores that threads are mapped to [SMB06], [DGC05].

We give an overview of these research branches on the left side of Figure 2.9. Starting from the root "Dynamic Load Balancing", the left side turns to the branch without taking communication overhead into account (*Without* ***Comm. Overhead***). This branch divides most load balancing researches in shared memory systems into a branch of work stealing and a branch of dynamic thread/process mapping. The details of these two branches are the studies explained and mentioned above.

The next subsection introduces related works on the right side of Figure 2.9 (*With* ***Comm. Overhead***), which implies taking communication overhead into account. We go to the analysis of dynamic load balancing in distributed memory systems.

2.4.2. Work Stealing in Distributed Memory Systems

Work stealing might be limited due to communication in distributed memory systems. The latency and delay of task migration are usually determined as the bottleneck of this limit [Din+09]. Extended from Figure 1.3 (A) on Page 5, we highlight the operations of work stealing in Figure 2.10. Again with the coordinates, the x-axis shows the flow of task execution following time progress, the y-axis shows compute nodes along with two processes per node and multiple threads deployed in a process to execute tasks. In particular, the green rectangles indicate task execution, whereas the yellow one indicates a task being migrated from an original process to a remote one. This task is called a remote task. In Figure 2.10, (1) and (2) indicate two main operations of work stealing in distributed memory.

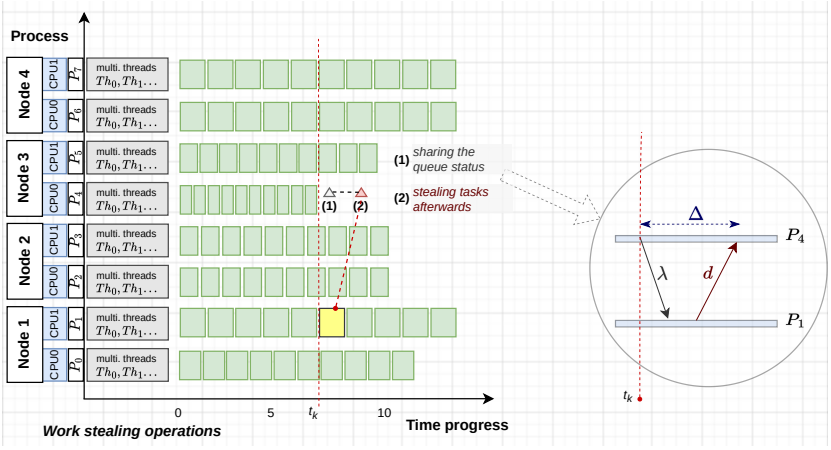


Figure 2.10.: An overview of work stealing operations in distributed memory systems.

The example is kept the same with 8 processes (or 8 MPI ranks) in total. Assume each process spawns 2 threads for executing tasks. The illustrated problem here is addressed by $P = 8$, $T = 80$ tasks in total, where we suppose each process is assigned equally 10 tasks. As a side note, revolving around load balancing also leads to similar problems, including:

- Task distribution/partitioning algorithms: aim to optimize the distribution of T tasks on P processes such that the total load values are equal. Alongside these algorithms, several constraints can be specified. For example, tasks are uniform/nonuniform, P processes are different from execution speed, and the load values w of tasks are deterministic or unknown.
- Dynamic task assignment: instead of partitioning tasks, people use a centralized scheduler. In which a master or a task manager is generated. When tasks are created, the master schedules them directly to available processes to maintain balance. This is well performed, but it is hard to conduct large-scale problems because the master can be overloaded if our problem is scaled running up on many nodes in parallel compute clusters.

Unlike the algorithms of task partitioning and task assignment, work stealing solves imbalance by moving tasks during execution. Assume the execution in Figure 2.10 starts at time t_0. Following the time progress, process P_4 is recognized as an idle process at time t_k. With work stealing, process P_4 sends its idle status around to request for stealing tasks. After receiving the status, assume process P_1 first accepts that request and process P_1 accepts process P_4 to steal its tasks. In detail, the first operations `(1)` are supposed to take a latency of λ because the status information is considered small. However, on the way of stealing tasks, for example, process P_4 steals a task from process P_1. This might take more time, and the transmission time refers to a delay of d illustrated as the second operations `(2)` in Figure 2.10, where delay d is dependent on the data size of tasks.

One or a few of these stealing operations might be negligible, but many of them might lead to large overheads. In overall, the overhead for stealing a task is shown as Δ, where Δ is rounded by $\lambda + d$ if we ignore the noise as well as the turn-around message exchange between an idle process and its victim.

The turn-around message exchange in this case includes: process P_4 receives back a confirmation of stealing acceptance from process P_1. Totally, suppose there are K successful migrated tasks, then the total overhead can be roughly calculated by $K \cdot \Delta = \sum_{i=0}^{K-1}(\lambda + d_i)$. The value of d_i indicates the delay time of migrating task i. Related to the load changed between process P_1 and process P_4, we can see that process P_4 adds more load, while process P_1 subtracts load.

Equation 2.1 gives an example of the total load in process P_4 and P_1 after execution. Within the scope of process P_4, k_4 denotes the number of tasks received on the side of process P_4, where $k_4 \in K$, and Δ_4 represents the total overhead of each stealing operation in average. Similarly, k_1 is the number of tasks that process P_1 migrates to the others, such as migrating tasks to process P_4.

$$\begin{aligned} L_4 &\approx L_4 + \sum_{k_4 \in K} w_{k_4}^{\text{remote}} + k_4 \cdot \Delta_4 \\ L_1 &\approx L_1 - \sum_{k_1 \in K} w_{k_1}^{\text{local}} \end{aligned} \tag{2.1}$$

Grounded in K, its value represents two aspects: the number of migrated tasks is appropriate or not appropriate, and the processes of migrating/receiving tasks are reasonable or not reasonable. Therefore, K is important for work stealing as well as load balancing in general. K is directly or indirectly affected by imbalance level (R_{imb}), communication overhead (λ, d), task size (s), and work stealing methods. To optimize K, several branches of interest (shown in Figure 2.9 on Page 28) are highlighted as follows:

(1) Victim Selection: similar to shared memory, work stealing in distributed memory is also improved by victim selection algorithms.

(2) Task Selection: indicates an improvement of work stealing in distributed memory by task selection algorithms.

(3) Improve Communication: indicates improving the communication bandwidth or hides the communication latency to improve work stealing in distributed memory.

(4) Async. Stealing: indicates asynchronous work stealing in distributed memory.

In branch (1) and (2), several works attempt to improve the victim and task selection algorithms of stealing tasks. Implicitly, these methods help reduce the number of failed or wrong requests. One proposed solution is from Lifflander et al. [LKK12], targeting persistence-based imbalance and centralized load balancing. The authors created a local database for each process to collect the exact duration of each task, which helped them determine good victims and estimate the appropriate number of migrated tasks.

Menon et al. [Men+12] proposed Meta-Balancer towards automating balancing by periodic statistic collection at runtime. The periodic information is then used to calculate imbalance ratio and recommend victim selection. Within victim selection algorithms, there are two criteria of interest:

- **topology-aware**: improving victim selection associated with the topology information of clusters. For example, we have multiple cores in a processor, multiple processors in a node, and multiple nodes in a cluster. The topology information shows how they connect can affect migration overhead when we select a victim or a task for work stealing.
- **optimal-K-aware**: aiming to find K with an optimal or near-optimal algorithm. The inputs require a good estimation of load values to generate a good distribution of K.

Drebes et al. [Dre+14] introduced a mechanism of topology-aware memory allocation that can help work stealing optimizing task placement over NUMA domains. This work analyzed producer-consumer relationships between tasks and promoted short-distance memory accesses during balancing operations. Considering data distribution and inter-node task migration costs is difficult for randomized work-stealing. Barghi et al. [BK18] showed an improved hierarchical scheduling policy: a task-based actor model relying on local topology information. Furthermore, Kumar et al. [Kum+16] also presented two different algorithms to eliminate failed stealing requests. The authors' idea mainly relies on application characteristics to identify potential victims and how many tasks should be migrated.

Solutions	Victim Selection		Task Selection		Reduce Communication Overhead	Resist Performance Variability
	topology-aware	optimal-K-aware	data-driven-aware	optimal-K-aware		
[LKK12]	✓	✓	✗	✗	✗	✗
[Dre+14]	✓	-	✓	✓	✗	✗
[BK18]	✓	✗	✓	✗	✗	✗
[Men+12]	✗	✓	✗	✓	✗	✗
[Kum+16]	✗	✓	✗	✓	✗	✗
[Bak+18]	✗	✓	✓	✓	✗	✗
[ZL18]	✗	✗	✓	✓	✗	✗
[Fre+21]	✗	✗	✓	✓	✗	✗
[Din+09]	✗	✗	✓	✗	✓	✗
[MK13]	✗	✓	✗	-	✓	-
[LSD19]	✗	✗	-	✗	✓	✗
[Kli+20a]	✗	✗	-	✗	✓	✓
[Sam+21]	✗	✗	-	✗	✓	✓
Our approach	✗	✓	✗	✓	✓	✓

Table 2.2.: A comparison table of the state-of-the-art related to the approach proposed in this thesis.

For task selection algorithms, several researchers investigate more about task properties, e.g., task data-driven awareness. Bak et al. [Bak+18] discussed handling persistent and transient load imbalance. The authors used a relatively infrequent periodic work assignment on cores. Then, they utilize the idle cycles of other cores to execute tasks belonging to the overloaded core. This method provides a fast strategy for data-driven awareness, but it might work only at a node level because the idle cores from other nodes cannot share tasks with the current node. Zafari et al. [ZL18] and Freitas et al. [Fre+21] proposed a technique of low overhead to improve distributed memory work-stealing. First, they use the application workload information to get task characteristics. Second, tasks are better selected to steal from a potential victim. Finally, similar tasks of the same victim are packed to reduce the message size. These works can ensure the criteria for optimizing K based on task properties. Within task selection algorithms, there are also two criteria of interest:

- **data-driven-aware**: investigating task properties, such as data size, input feature or data affinity, to decide a good strategy for task selection.
- **optimal-K-aware**: finding an optimal number of migrated tasks (K), such as estimating a fixed chunk size within K or the appropriate number of tasks stolen at once. We deal with this criterion through an adaptive strategy or auto-tuning strategy.

For an overview, we summarize these related works along with the mentioned criteria in Table 2.2. There are three marks in the table, where "✗" denotes "*not available*" ("*do not take*" into account), "✓" means "*available*" ("*take*" into account), and "−" indicates "*not mentioned*". For example, the first row of Table 2.2 shows the work from Lifflander et al. [LKK12], which presented a hierarchical persistence-based rebalancing algorithm to balance the load, mainly based on statistical data on the duration of each task. From our review, this approach improves the criterion of victim selection through "topology-aware" and "optimal-K-aware", while the others, such as "Task Selection", "Reduce Communication Overhead", and "Resist Performance Variability", are not presented. Therefore, "✓" and "✗" are marked on the corresponding columns, as shown in the first row of Table 2.2.

In branch (3) and branch (4), which are explained above on Page 30 as well as illustrated in Figure 2.9 on Page 28, we show the research criterion corresponding to the column of "Reduce Communication Overhead" in Table 2.2. Dinan et al. [Din+09] introduced the design of a scalable runtime system to support work stealing by using PGAS programming models. Their method helps reduce locking on the critical path and reduce task creation overhead. Also, this work allows early aborting to avoid waiting on stale resources. Another study attempted to hide the network latency by aggregating remote memory copy interface [NC99]. Menon et al. [MK13] proposed an algorithm that uses partial information about global system state to perform balancing, named GrapevineLB. It consists of two stages: propagating global information inspired by an epidemic algorithm [KMW32] and transferring work units using a randomized algorithm. The method from Menon et al. [MK13] implicitly hides latency with low overhead. Larkins et al. [LSD19] have shown that using a traditional

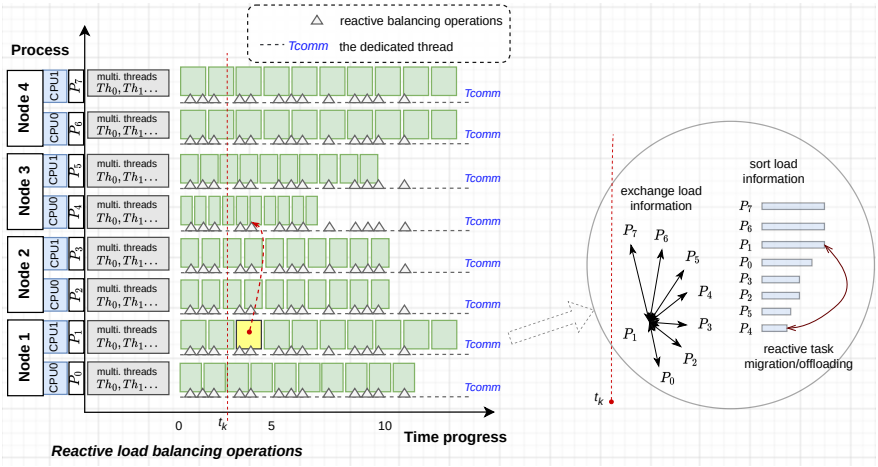


Figure 2.11.: An overview of reactive load balancing operations in distributed memory systems.

PGAS approach, work stealing requires a sequence of one-sided RDMA communications to complete a steal. Then, the conventional system must locate, identify work, steal it, and update the state. This progress still faces a high latency. Another research direction presents a distributed work-stealing system, which is amenable to acceleration using the Portals 4 [Der+15] network programming interface [LSD19]. They use the Portals 4 primitives in a novel manner that can significantly reduce the overhead of load-balancing operations.

2.4.3. Reactive Load Balancing

Instead of waiting for an empty queue to steal tasks, an approach called "reactive load balancing" is proposed by [Kli+20a]. The target is to resist performance variability or performance slowdown during executing tasks. Reactive load balancing monitors the queue status continuously on each process, then exchanges this information, checks the imbalance ratio periodically, and offloads tasks reactively.

Figure 2.11 shows the operations of reactive load balancing. At time t_k, it detects a difference between processes in queue length, exceeding an imbalance condition. For example, on the side of process P_1, which is known as an overloaded process, process P_1 sorts the process indices by the number of remaining tasks in their queue (called the queue lengths). After that, process P_1 reactively offloads tasks to a suitable process based on the sorted list, such as process P_4 in this case.

Regarding the focused criterion of reactive load balancing, we show a column of "Resist Performance Variability" in Table 2.2, which is illustrated as branch (5) and (6) in Figure 2.9 on Page 28. Implementing reactive load balancing also leads to overlapping computation and communication by a dedicated thread. Particularly, a process spawns multiple threads to execute tasks, where one thread is dedicated to performing information exchange and task migration. Therefore, instead of waiting until a process is idle before deciding to steal tasks, reactive load balancing can exploit this dedicated thread to check load imbalance and offload tasks earlier if the speculation of load imbalance is right. This reactive approach is developed with two methods:

1. Reactive task offloading
2. Reactive task replication

In corresponse to an implementation, Klinkenberg et al. [Kli+20a] introduced Chameleon, a task-based parallel framework for distributed memory systems. The framework is developed by using hybrid MPI+OpenMP. One thread within an MPI process is dedicated to repeatedly monitoring the queue length (namely $Tcomm$). When an imbalance condition is met, $Tcomm$ reactively offloads tasks from an overloaded rank (a slow process) to an underloaded rank (a fast process). In the case of high imbalance, the difference in load among ranks might be large; therefore, we might need to

offload many tasks periodically. This can lead to being late for reactive task offloading.

Following that, Samfass et al. [Sam+21] have proposed reactive task replication. Instead of only offloading tasks, the authors use replication and task-canceling techniques. Several tasks are replicated on certain ranks before execution. This work can significantly resist communication overhead and performance variability on distributed memory; however, it is difficult to ensure an optimization for K due to no prior knowledge, especially when we meet scenarios with high imbalance levels.

This work first approaches a performance model to estimate when reactive load balancing is limited. Second, we propose a new approach called a "proactive load balancing" approach for further improvement. The last row in Table 2.2 shows our approach, and the targeted criteria refer to the columns of "optimal-K-aware", "Reduce Communication Overhead", and "Resist Performance Variability".

3. Performance Modeling and Analysis

Generally, work-stealing or reactive load balancing make decisions based on the current execution status at runtime. Their decisions take balancing operation and task migration into account. Balancing operations denote checking imbalance status and exchanging information, while task migration denotes moving tasks from one process to another. These can affect the overall performance. Therefore, it is difficult to determine efficiency as well as performance of balancing decisions before running applications. We address this with a performance model.

Regarding the previous works, several performance models for work stealing have been proposed, such as [Gas+21], [Tch+10], [Hay+04]. The authors use discrete-time models to address the behavior of work stealing, mainly focusing on the constraints of communication latency when tasks are migrated. In their work, the latency is considered a constant, and all tasks have the same overhead in migration. However, we show that tasks can be different in type and data size in terms of HPC as well as task-based parallel applications. Therefore, communication latency alone cannot model balancing behavior very well in practice, and these models are not suitable in our context. To analyze how the previous models are built, we do not go into detail in this chapter because work stealing has passive behavior from the view of load balancing operations, and it is also extensively analyzed with the existing models mentioned above. Therefore, we summarize one of the most related models in Appendix A.

In constrast, we introduce a new performance model, mainly focusing on the behaviors of reactive load balancing. We assume a given imbalance context along with the overhead of balancing operations and task migration, where the given imbalance context is already mentioned in Subsection 2.4.1. The overhead is bounded by two metrics, latency (λ) and delay time (d). In detail, the input and output of our model can be simplified as follows.

- The main influence inputs include the number of involved processes, number of tasks in total, number of slowdown processes, slowdown factors of corresponding processes, and the overhead for balancing operations as well as task migration (delay time).
- The output indicates the performance of modeled and simulated cases in execution time, number of local and migrated (offloaded) tasks on each process.

The model is leveraged to design a simulator, where its output can be used to analyze the bound of dynamic load balancing approaches among different scenarios.

An overview of this chapter is as follows: Section 3.1 presents the performance model concerning reactive load balancing, followed by an introduction to the associated simulator and model evaluation in Section 3.2. Additionally, Section 3.3 highlights the idea of proactive load balancing, which is grounded in our performance model.

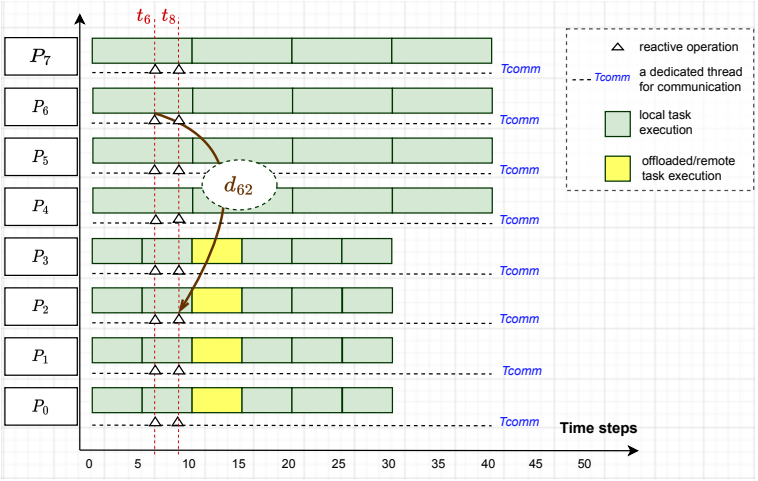


Figure 3.1.: An illustration about the analysis of reactive load balancing based on deterministic estimation.

3.1. A Proposed Model for Reactive Load Balancing in HPC

For modeling the performance of dynamic load balancing, we need to understand the behavior associated with balancing operations. For this, we rely on experiments and profiling information to build the model. To stay consistent with the example in Figure 2.11 on Page 32, we highlight the same illustration for reactive balancing behavior in Figure 3.1, but we emphasize in detail the steps when task offloading is performed.

Figure 3.1 can be considered as a profiled information of 8 processes. The y-axis shows 8 involved processes ($P = 8$), while the x-axis shows execution progress over time steps (time units) from 0 until termination. In Figure 3.1, before tasks are executed, we assume there is a given distribution of 5 tasks on each process (formulated by $T_i = 5$). The performance of each process is expected to be stable, and total load value is balanced. However, the execution speed of processes P_4, P_5, P_6, P_7 is slowed down, leading to an imbalance at runtime as Figure 3.1 reveals. With reactive load balancing, each process is deploying a dedicated thread, `Tcomm`, which is denoted by the dashed line, and the small triangles on each line illustrate reactive operations. For example, at time t_6 and time t_8, tasks from the slower processes are offloaded to the faster processes. In terms of two objective factors affecting the decisions of task offloading, we have:

- Imbalance level: points to the effect of slowdown on the imbalance ratio.
- Reasonable task migration: points to the effect of balancing approach and overhead in balancing operations & task migration on the offloading decision.

Following the slowdown affecting execution speed, we formulate the speed of corresponding processes such as speed S_{P_0}, S_{P_1}, S_{P_2}, ..., S_{P_7}. The total load value of each processes is formulated by load L_0, L_1, L_2, ..., L_7. In the case of balance, all speed values and their total load values are approximately similar. In the case of imbalance, some speed values are affected by a slowdown factor which we denote by $Slow_i$. If $Slow_i < 1.0$, it can make the execution speed of process P_i slower. For example, the expected speed of process P_i is $S_{P_i} = 1.0$ (can be simplified as 1 task/time unit). When it is slowed down by 50% ($\times 0.5$), the execution speed would be 0.5 task/time unit. Otherwise, if $Slow = 1.0$, the execution speed is not changed. For mapping slowdown to the execution time of a task, assuming a time unit is in seconds, then the execution of a task takes one second ($1s$) if the speed S_{P_i} is 1.0 task/second. If $Slow_i = 0.5$ affects the speed S_{P_i}, that task needs $2s$ to be finished.

In a specific context, when we can analyze the impact of slowdown factor on imbalance levels by keeping a given distribution of tasks and a number of involved processes, then varying slowdown

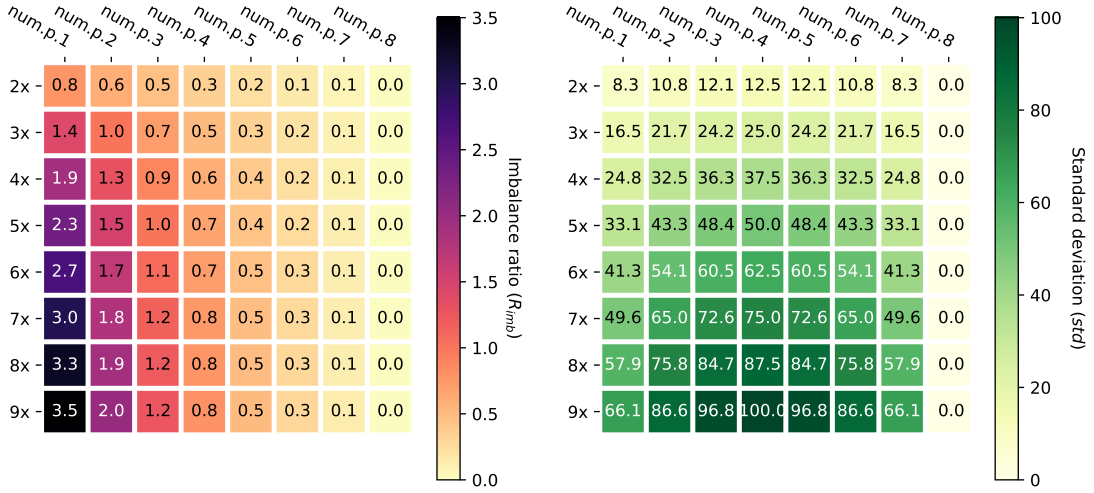


Figure 3.2.: The impact of slowdown values on load imbalance ratios.

factor as well as the number of slowdown processes. For changing the $Slow_i$ values, we can formulate its impact on speed S_{P_i} and load L_i by Equation 3.1.

$$\begin{aligned} Slow_i &\overset{impact}{\rightarrow} S_{P_i} : S_{P_i} \times Slow_i \\ Slow'_i = \frac{1}{Slow_i} &\overset{impact}{\rightarrow} L_i : L_i \times Slow'_i \end{aligned} \tag{3.1}$$

In Equation 3.1, $Slow_i$ affects directly execution speed, and the value of S_{P_i} can be decreased. $Slow'_i$ affects directly the total load value at the end; therefore, the value of L_i can be increased. Taking the case in Figure 3.1 for instance, we assume the total load values of all processes are multiplied with an array of $Slow'_i$ values. Then, the slowdown at runtime makes them unbalanced and R_{imb} is calculated along with $Slow'_i$. Note that load L_i without performance slowdown are expected equally; therefore, we can assume that load $L_0 \approx L_1 \approx ... \; L_7$ and $\approx L$. In average, the total load values and imbalance ratio can be calculated by Equation 3.2.

$$\begin{aligned} L_{max} &= \max(L_0 \times Slow'_0, ..., L_7 \times Slow'_7) \\ L_{avg} &= L \times \frac{Slow'_0 + ... + Slow'_7}{P}, \text{where } L \approx L_0 \approx ... \approx L_7) \\ R_{imb} &= \frac{L \times Slow'_{max}}{L \times \frac{Slow'_0 + ... + Slow'_7}{P}} - 1 \\ &= \frac{P \times Slow'_{max}}{\sum_{i=0}^{P-1} Slow'_i} - 1 \end{aligned} \tag{3.2}$$

The following estimation analyzes two common cases of slowdown effects.

Case 1: Slowdown remains constant during execution, and this happens on some processes, which are called slowdown processes. The slowdown values among processes can be different. However, to easily show the slowdown effect changing over its values and over the number of involved processes, Figure 3.2 highlights an estimation by two heatmaps, where one on the left shows the heatmap color of imbalance ratio (R_{imb}) and one on the right shows the color of standard deviation (std). Each figure is labeled on the horizontal axis with the number of slowdown processes (such as $num.p.1$, $num.p.2$, ..., $num.p.8$), and on the vertical axis with the factor of slowdown (such as 2×, 3×, ..., 9×). The experiments in Figure 3.2 are stayed with 8 processes. As an example, $num.p.1$ indicates that one of 8 processes is slowed down, while 2× indicates the slowdown factor in that process is

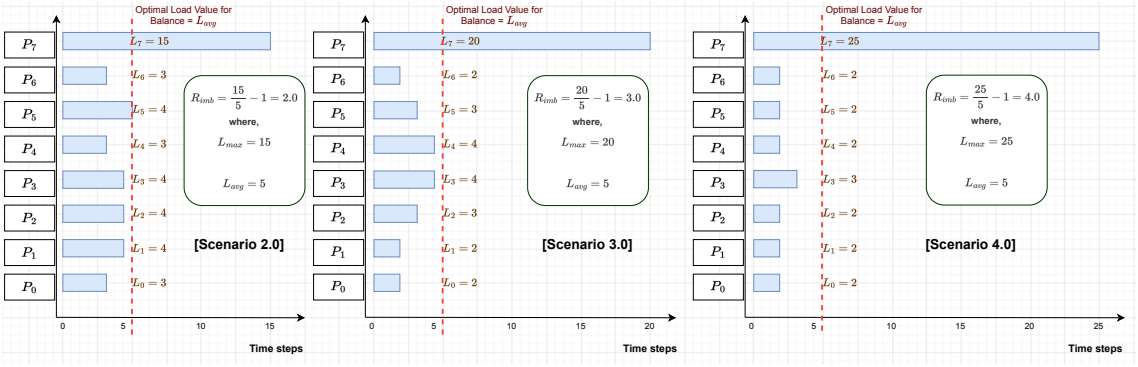


Figure 3.3.: Three imbalance scenarios when only one process is slowed down.

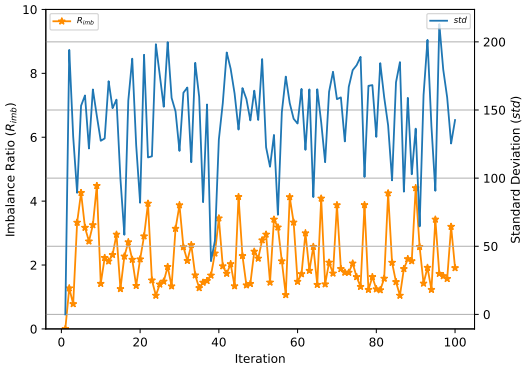


Figure 3.4.: Randomized slowdown and unpredictable imbalance ratios over 100-iterations execution.

2 times compared to the stable processes. Depending on the color of the heatmap, we determine an imbalance ratio corresponding to a standard deviation. For example, in the case of *num.p.*1 and 2×, the imbalance ratio is 0.8, corresponding to a standard deviation of 8.3. The value of standard deviation indicates load dispersion between different processes.

As can be seen, bad situations occur when slowdown cases happen only in a few processors/-cores corresponding to the mapped processes or threads. The worst in this example is when one process is significantly slowed down. Figure 3.3 highlights three imbalance scenarios with $R_{imb} = 2.0, 3.0, 4.0$ (named accordingly Scenario 2.0, Scenario 3.0, Scenario 4.0), where only process P_7 is slowed down. If tasks are migrated to balance the load, only process P_7 should offload tasks to others.

Case 2: Slowdown is unpredictable, where $Slow_i$ is varied during execution. Assume that the execution example is still stayed with 8 processes as in Figure 3.1, we perform the execution running over 100 iterations, where the slowdown values are randomized over each iteration. Figure 3.4 shows an estimation of Case 2 with both lines: imbalance ratio (R_{imb}) and standard deviation (std). The x-coordinate denotes the indices of iterations; the left and right y-coordinates denote imbalance ratio and standard deviation respectively. If the uncertainty of slowdown is high, this leads to a challenge in load prediction and dynamic load balancing. Particularly, it is difficult to propose a good strategy for migrating tasks.

In practice, Case 1 and Case 2 might be overlapped except when the fluctuation level of Case 2 is too high. Case 2 usually happens when our computing resources are over-subscribed by sharing between different programs simultaneously. Otherwise, Case 1 occurs more often in parallel compute clusters because of the inherent variation of chip design today, such as variation in power and temperature of the chips. Acun et al. [AMK16] have shown similar experiments like Case 1 caused by the variation among processors on the Turbo Boost-enabled supercomputers (an execution time

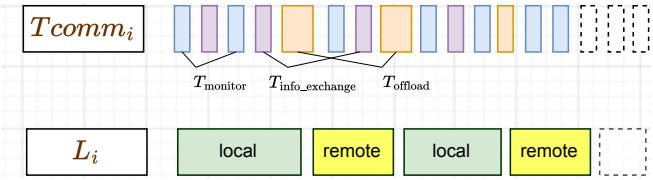


Figure 3.5.: Three main operations driven by reactive load balancing and total load value afterward.

difference of up to 16%). The other works from [Wei+18] [Tun+19] have introduced the methods and tools to characterize hardware performance variation, even reproduce performance variations for research purposes [Ate+19]. In addition, not only CPUs, similar analyzes are also performed on current GPU architectures [Yos+22].

Following a given imbalance level, the next subsection introduces how the overhead in balancing operation and task migration can affect the decision of task offloading. Typically, we show an estimation of the maximum number of tasks that need to be exchanged to balance the load in a specific situation.

3.1.1. Deterministic Estimation

Deterministic estimation aims at calculating on average for how many tasks can be offloaded. For example, we have a given imbalance ratio, a number of involved processes which we know overloaded and underloaded processes. Then, according to the overhead of task migration, we can estimate how many tasks should be migrated. The model for deterministic estimation provides an overview in advance about how task migration is limited in a specific case and in a specific system.

Before a task is migrated, there are three main operations driven by the reactive load balancing approach, including monitoring execution status, exchanging status, and offloading tasks. These operations are performed by $Tcomm$ at runtime and shown as T_{monitor}, $T_{\text{info\_exchange}}$, and T_{offload} in Figure 3.5. The line of $Tcomm_i$ illustrates balancing operations according to process P_i. Because these three operations cause the most overhead that we count for an impact on reactive load balancing; therefore, we name T_{monitor} to indicate the time of monitoring operation, $T_{\text{info\_exchange}}$ for the time of current execution status (queue status) exchange, and T_{offload} for the time of task offloading. Following that, tasks might be migrated or not migrated over time progress. We imply that the value of T_{monitor}, $T_{\text{info\_exchange}}$, T_{offload} might be varied during execution, depending on a given time and data size of migrated tasks.

In Figure 3.5, we also highlight the line of L_i to denote the final load value of the corresponding process P_i. Load L_i is a sum of the wallclock execution time values of tasks, including local and remote tasks. Local tasks (green rectangles) are associated with the load values, while remote tasks (yellow rectangles) are related to the remote load values, where remote tasks indicate the tasks received from the other processes. Overall, the operations and load values shown in Figure 3.5 are formulated in Equation 3.3.

$$\begin{cases} Tcomm_i = T_{\text{monitor}} + T_{\text{info\_exchange}} + T_{\text{offload}} \\ L_i = \sum_{j \in T_i'} w_j^{\text{local}} \pm \sum_{k=0}^{k_i} w_k^{\text{remote}} \end{cases} \tag{3.3}$$

The number of local tasks of the original process might not be the same as before execution if we perform reactive load balancing. Therefore, in $\sum_{j \in T_i'} w_j^{\text{local}}$, T_i' is the new set of local tasks belonging to process P_i, value w with "local" indicates the wallclock execution time of local tasks. In $\sum_{k=0}^{k_i} w_k^{\text{remote}}$, "remote" indicate remote tasks. The value of k_i represents the total number of tasks offloaded to process P_i. If process P_i is the one offloading tasks, the sign will be negative $(-)$ for $\sum_{k=0}^{k_i} w_k^{\text{remote}}$. With all migrated tasks, we use K to denote the total number between all involved processes. For example, with the number of P processes in total, $K = k_0 + k_1 + k_2 + ... + k_{P-1}$,

where obviously if a process does not receive remote tasks, then the corresponding k_i value is zero. The main idea of deterministic estimation is to estimate K in general.

There are two bounds estimated, average bound and maximum bound. The average bound relies on the average load of accumulative overloaded and underloaded values, while the maximum bound relies on the maximum and minimum load values of corresponding processes.

Average Bound: revolves around the difference of total load values among processes to calculate overloaded and underloaded values. We assume that the total overloaded value must fill the gap between average and underloaded values.

- Theoretically, the ideal balance is average load value ($L_{avg} = \text{average}(L_i), \forall i \in P$).
- The sum of overloaded values should be equal to the sum of underloaded values, $\sum L_{\text{overloaded}} = \sum L_{\text{underloaded}}$, where

$$\begin{cases} \sum L_{\text{overloaded}} & = \sum_{i \in P}(L_i - L_{avg}) \mid L_i > L_{avg} \\ \sum L_{\text{underloaded}} & = \sum_{i \in P}(L_{avg} - L_i) \mid L_i < L_{avg} \end{cases} \tag{3.4}$$

- To balance the gap between $\sum L_{\text{overloaded}}$ and $\sum L_{\text{underloaded}}$, suppose K tasks are migrated.

K tasks are offloaded to the number of underloaded processes denoted by M. Following M, the number of overloaded processes is $P - M$. Alternatively, we use $P_{underloaded}$ to indicate the number of underloaded processes and $P_{overloaded}$ for the number of overloaded processes, where $P_{overloaded} = P - M$. If the sum of data size of K tasks is denoted by S_{transfer} and the sum of delay time values for offloading K tasks is D, then we can estimate how much K is bounded. This bound is under the constraints: a given latency (λ) and an average bandwidth ($\overline{B}$). In detail, $S_{\text{transfer}} = \sum_{i \in K} s_i$, and $D = \sum_{i \in K} d_i$, where s_i is the data size of task i and d_i is the delay time calculated by $\lambda + \frac{s_i}{B_i}$; B_i is the bandwidth at the time task i is offloaded.

Simply put of $\overline{s}$ as the average data size for each task, the total delay time of offloading K tasks should not exceed the ratio between $\sum L_{\text{overloaded}}$ and M. This is considered as a gap for filling the underloaded load. Equation 3.5 shows the bound of K limited by this ratio, implying the total overloaded value sharing across M underloaded processes.

$$\begin{cases} K \times (\lambda + \frac{\overline{s}}{\overline{B}}) & \leq \frac{\sum L_{overloaded}}{M} \\ \Leftrightarrow K & \leq \frac{\sum L_{overloaded}}{(\lambda + \frac{\overline{s}}{\overline{B}}) \times M} \\ \Leftrightarrow K & \leq \frac{\sum L_{overloaded}}{\overline{d} \times M}, \text{where } \overline{d} \text{ is the average delay time per task migration} \end{cases} \tag{3.5}$$

Based on the estimation model in Equation 3.5, we show an illustration of how it works with the latency and bandwidth information from real HPC clusters. The example is kept the same with 8 processes. For a specific imbalance scenario, we assume there are two slowdown processes. The slowdown factor is set 5× slower than normal processes, and R_{imb} is about 1.5.

This scenario is illustrated in Figure 3.6. The chart on the left side shows 8 processes from P_0 to P_7 along with their total load values. The L_{avg} value is determined by the dashed line, *ideal balanced*. The chart on the right side highlights the total overloaded value of process P_0 and process P_1, which is fairly shared to 6 underloaded processes, $M = 6$. Hence, we can estimate how much time and how many tasks (K) are limited for migration. The scenario here represents the number of overloaded processes smaller than the number of underloaded processes, $P_{\text{overloaded}} < P_{\text{underloaded}}$.

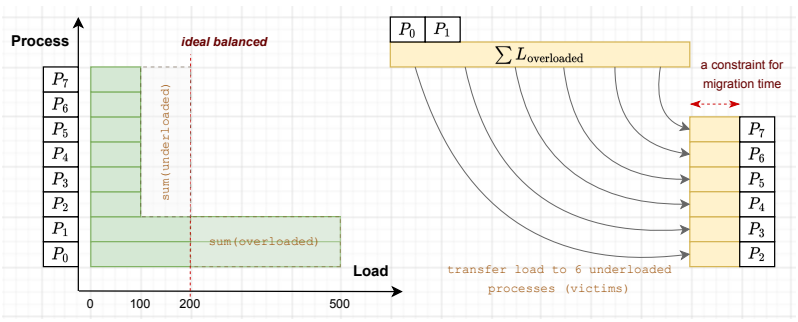


Figure 3.6.: An illustration of deterministic estimation on average.

Expanding this scenario, we emphasize two more scenarios that are summarized as follows.

- Scenario 1: is explained above and shown in Figure 3.6. The slowdown factor is 5×, the number of slowed down processes is 2, and $R_{imb} = 1.5$. This scenario shows the number of overloaded processes smaller than the number of underloaded processes, $P_{\text{overloaded}} < P_{\text{underloaded}}$.
- Scenario 2: the slowdown factor is also 5×, the number of slowed down processes is 4, $R_{imb} = 0.7$, and $P_{\text{overloaded}} = P_{\text{underloaded}}$.
- Scenario 3: the slowdown factor is set ≈ 8×, the number of slowed down processes is 6, $R_{imb} = 0.3$, and $P_{\text{overloaded}} > P_{\text{underloaded}}$.

The corresponding latency and bandwidth information is collected from three different HPC clusters. We use the average λ and $\overline{B}$ measured from CoolMUC2[1], SuperMUC-NG[2], and BEAST[3] at Leibniz Supercomputing Centre (LRZ). These systems are also used to perform experiments in the following sections. CoolMUC2 has 28-way Intel Haswell-based nodes and an 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 connected via a higher interconnect bandwidth, HDR 200Gb/s InfiniBand.

The average latency λ and bandwidth $\overline{B}$ of CoolMUC2 (`coolmuc2`), SuperMUC-NG (`sng`), and BEAST system (`beast`) are shown in Figure 3.7 (A), measured by OSU Benchmark [Pan+21]. The benchmark is run on each cluster with two nodes, and the basic communication interface is MPI point-to-point. The x-axis shows message sizes from 128 bytes to 256 MB that can account for the data size of tasks during migration. The left y-axis shows bandwidth in MB/s, while the right y-axis shows latency in μs. As a result, BEAST has the best communication performance, while CoolMUC2 is the worst due to different interconnection technologies.

In Figure 3.7 (B), (C), (D), we show the results of K estimation associated with the mentioned scenarios 1, 2, 3. Following the x-axis, we show the task sizes from 400 KB to 78919 KB (almost 80 MB), which are enough to highlight the trend of K estimation and to represent the data sizes of each task in common task-based parallel applications. The y-axis shows the K values, representing the maximum number of tasks that we can offload under the constraints of $\sum L_{\text{overloaded}}$, λ, and M. In Figure 3.7 (B), R_{imb} is 1.5, and two processes are slowed down. The total overloaded value is from process P_0 and process P_1, where we share this amount of load over 6 victim processes. We highlight that the migration time of tasks should not exceed the shared value shown as the constraint segment over migration time in Figure 3.6.

Similarly, Figure 3.7 (C) and (D) show that K is decreased when the data size of tasks increases. The value of K still reaches thousands tasks if we offload the tasks continuously, in particular with the size ≈ 80 MB. This shows that communication overhead for task migration in distributed memory

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

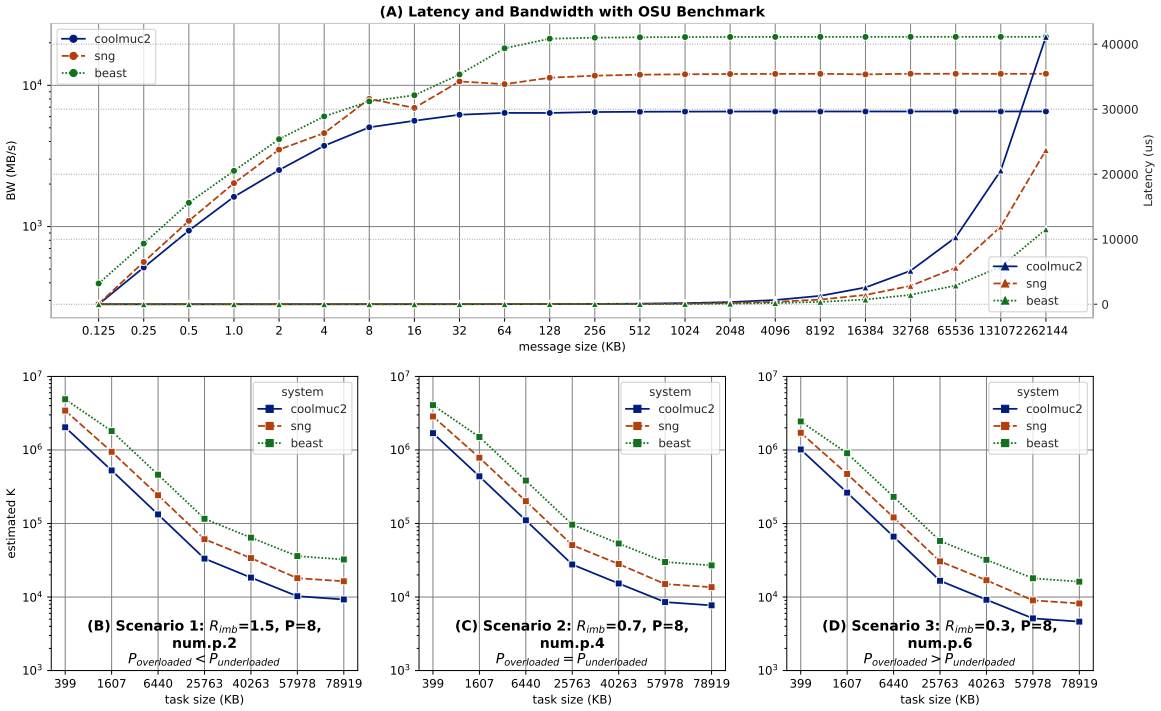


Figure 3.7.: Estimation of K offloaded tasks on average under the constraints of delay time, imbalance level, task data size, and the number of slowdown processes in task migration.

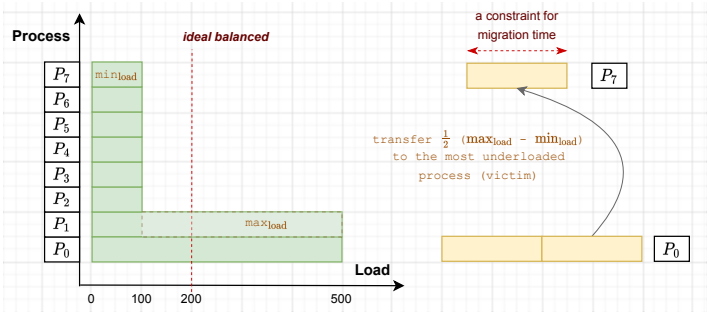


Figure 3.8.: An illustration of deterministic estimation on min-max load values.

might not be the most influential factor. There might be some impacts from the other factors in balancing operations before a task can be migrated. Besides that, K estimated in this bound relies on the average of accumulative overloaded and underloaded values. In the following paragraph, K estimation is conducted in the bound of maximum overloaded and minimum underloaded values, which might stretch the constraint segment of migration time and K may be more than the average bound.

Min-Max Bound: simply revolves around the maximum (max) and minimum (min) load values of corresponding processes. Min-max implies the longest way to offload tasks between the most overloaded and the most underloaded processes. Their difference can represent the effect of delay time in task migration.

Figure 3.8 illustrates estimating K based on the most overloaded and underloaded process. The longest way is between process P_0 and process P_7, which dominates the most time progress for offloading tasks. Then, we cut a half of the load difference to share each other. The min-max bound of estimating K is summarized as follows.

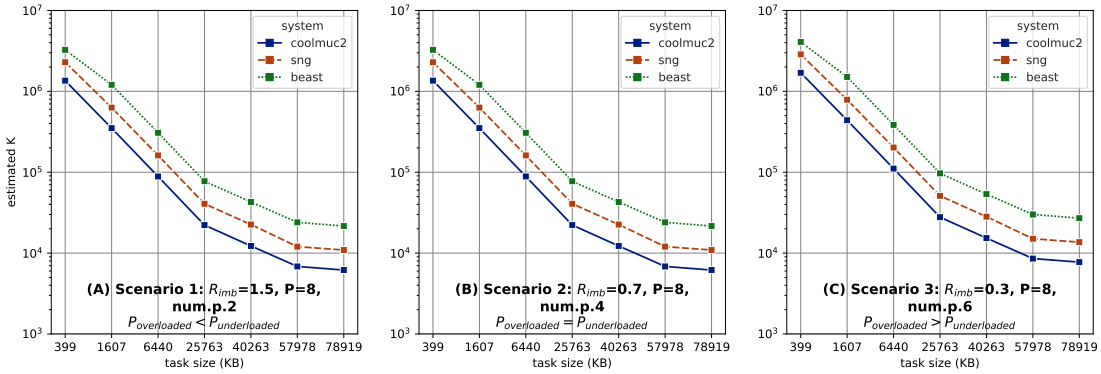


Figure 3.9.: The estimation of K offloaded tasks under the constraints of delay, imbalance level, and task size with the bound of min-max load difference.

- The difference between the min and max load values is denoted by $\Delta_{max-min} = L_{max} - L_{min}$.
- Based on $\Delta_{max-min}$, we estimate K as the number of offloaded tasks. Similarly, the assumption about data size in migrating tasks is kept the same as average bound.
- The bound of K is calculated by

$$\begin{cases} K \times (\lambda + \frac{\bar{s}}{\overline{B}}) & \leq \frac{\Delta_{max-min}}{2} \\ \Leftrightarrow K & \leq \frac{\Delta_{max-min}}{2 \times (\lambda + \frac{\bar{s}}{\overline{B}})} \\ \Leftrightarrow K & \leq \frac{\Delta_{max-min} \times \overline{B}}{2 \times (\lambda \times \overline{B} + \bar{s})} \end{cases} \tag{3.6}$$

The results of K estimation are revealed in Figure 3.9. With the given latency and delay time on three corresponding clusters, the bound of K is higher than the average bound. Likewise, K is still high if we continuously perform offloading tasks during execution.

In practice, most analyses and hypotheses suggest that the impact of communication overhead on task migration contributes to slow work stealing as well as reactive load balancing. However, we show that task migration overhead is not only the most influential factor if we perform offloading tasks continuously. These estimations motivate a more in-depth analysis between task migration overhead and balancing operation overhead. The overhead of balancing operations includes the time of monitoring load, exchanging information, and calculating imbalance ratio before we decide to offload a task. The following subsection will show how they impact task offloading decisions.

3.1.2. Discrete Time Model

The idea of discrete-time models is to break execution time progress into time steps. In our work, we aim to model the behaviors of reactive load balancing based on queue length over time steps. There are time variables and non-time variables. The values of non-time variables will jump from one value to another depending on how time moves along. The queue length of each process is denoted by $Q_i(t)$, where i means process P_i. The queue length is decreased over a certain number of time steps indicated by Δt. With task offloading, the queue of a process might receive tasks or also send tasks to others. In principle, the queue length status is changed during execution, mainly following three operations shown in Figure 3.5 on Page 39. To be intuitive, we illustrate these operations through process P_i and process P_j in Figure 3.10.

The y-axis shows each process with the label of total load values (L_i, L_j) and dedicated communication threads ($Tcomm_i$, $Tcomm_j$). The x-axis points to the execution progress over time steps, where we can see reactive balancing operations on $Tcomm$, for example, task execution, actions on

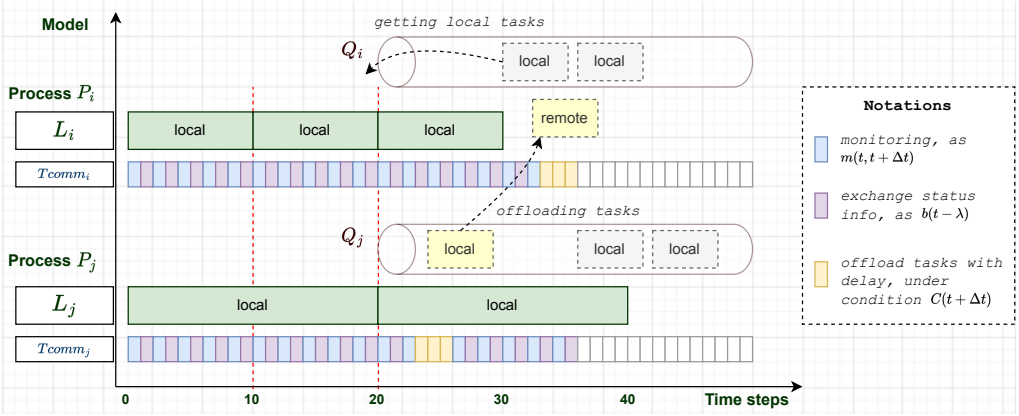


Figure 3.10.: An anatomy of reactive load balancing events over discrete time steps.

each queue such as enqueuing tasks, offloading tasks. A task offloaded from its original queue (`local`) to another is called a remote task (`remote`). Our model is presented as follows (in Equation 3.7).

$$\begin{cases} Q_i(t+\Delta t) = Q_i(t) - \mu_i(t, t+\Delta t) - \sum_{i \neq j} O_{ij}(t) + \sum_{j \neq i} O_{ji}(t - d_{ji}) \\ C_i(t+\Delta t) = \frac{Q_{max} - Q_{min}}{Q_{max}}, C_i(t++\Delta t) > \text{const}(R_{imb}) \\ M_i(t+\Delta t) = m_i(t, t+\Delta t) + b_i(t-\lambda) \\ Tc_i(t+\Delta t) = Tc_i(t) + \Delta(t) \end{cases} \tag{3.7}$$

The parameters of the model are driven by:

- $Q_i(t+\Delta t)$: indicates the queue length status varied over a number of Δt time steps. Alternatively, we can say that $t+\Delta t$ addresses the changes of queue Q_i of process P_i in the interval $(t, \Delta t]$. In response to Q_i at $t+\Delta t$, the status at the previous step (t) is affected by $\mu_i(t, t+\Delta t)$ (task execution rate), $\sum_{i \neq j} O_{ij}(t)$ (the number of offloaded tasks from process P_i to another, i.e., process P_j), and $\sum_{j \neq i} O_{ji}(t - d_{ji})$ (the number of remote tasks that process P_i received from another, i.e., process P_j).
- $\mu_i(t, t+\Delta t)$: represents how many tasks are executed in a period of Δt. For instance, there could be 2, 3, ... tasks finished.
- $\sum_{i \neq j} O_{ij}(t)$: indicates the number of offloaded tasks from process P_i to P_j, leading its sign negative. $\sum_{j \neq i} O_{ji}(t - d_{ji})$ indicates the number of remote tasks received from process P_j. Additionally, sending tasks from process P_i to process P_j is performed asynchronously by the dedicated thread, and during migration, the other threads still execute local tasks. Therefore, we do not count delay time for sending tasks. In contrast, receiving tasks at the side of process P_i from process P_j affects task execution on process P_i if the delay time is taken into account. This is why we need the term of d_{ji} in $\sum_{j \neq i} O_{ji}(t - d_{ji})$, and its sign is positive.
- $C_i(t+\Delta t)$: models when the imbalance condition is checked and met. Notably, the values of $\sum_{i \neq j} O_{ij}(t)$ and $\sum_{j \neq i} O_{ji}(t - d_{ji})$ depend on when an imbalance is detected and when reactive load balancing takes decision. Corresponding to the strategy of reactive task offloading, an imbalance condition is met when the queue length ratio is satisfied with a given constant of R_{imb}. The returned value of $C_i(t+\Delta t)$ at a time is 0 or 1, which means "balanced" or "unbalanced".
- $M_i(t+\Delta t)$: models the operations of monitoring load and exchanging load. Each process needs load information to calculate $C_i(t+\Delta t)$. The current length of queues represents the load information here. Therefore, $M_i(t+\Delta t)$ is used to model the action of monitoring and exchanging load, denoted by $m_i(t, t+\Delta t)$ and $b_i(t-\lambda)$ respectively. Unlike the data size of tasks, each action of exchanging information is modeled with latency because the size of load information is supposed to be small as sending and receiving the header of a message.

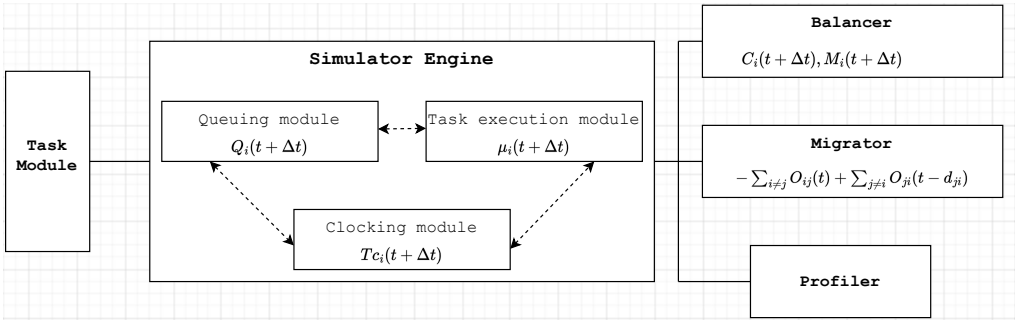


Figure 3.11.: Modularized components for a simulator design following the performance model of reactive load balancing.

- $Tc_i(t+\Delta t)$: is considered as the time clock and represents the dedicated thread, $Tcomm$. In this model, we use Tc_i to indicate clocking (counting time steps).

To illustrate how the model in Equation 3.7 works, we reuse the example illustrated in Figure 3.10. Following the operations of $Tcomm$, their notations on the right side show the corresponding parameters of the model. Different colors distinguish different operations. For example, the blue boxes illustrate monitoring load modeled by $m_i(t, t+\Delta t)$, the violet boxes illustrate load information exchange ($b_i(t-\lambda)$), and the orange ones illustrate offloading tasks under the condition of $C_i(t+\Delta t)$.

Starting at time $t = 0$, we assume 5 tasks per process. The execution speed on process P_j is slower than process P_i, $\mu_i = 2\mu_j$, leading to an imbalance.

- At $t = 10$, we have $Q_i = 4-1-0+0$, $Q_j = 4-0-0+0$, $M_i = M_j = Tc_i = Tc_j = 10$, where M_i and M_j just have the overhead of monitoring load (m_i, m_j).
- At $t = 20$, we have $Q_i = 3-1-0+0$, $Q_j = 4-1-0+0$. With the current status at $t = 20$, Process j meets the imbalance condition. Then, it exchanges this information at $t = 21$, and one task is offloaded reactively at $t = 23$.

In practice, delay time depends on the data size of tasks; therefore, offloading different tasks might have different delays. Besides, the values of latency (λ) and delay time (d) might fluctuate at runtime. For example, the offloaded task from process P_j to process P_i is started asynchronously at $t = 23$; then it arrives on the side of process P_i at $t = 33$ if we assume delay d_{ji} takes 10 time steps. Simultanously, d_{ji} might be varied in practice such as 12, 13, etc. Our discrete-time model is intended to simulate reactive load balancing in particular and dynamic load balancing behaviors in general. The following section will show how we design a simulator and evaluate it.

3.2. Model Simulation and Evaluation

From the proposed model, this section introduces how we map its parameters to design a simulator. Figure 3.11 shows five modules corresponding to five components from left to right, including `Task Module`, `Simulator Engine`, `Balancer`, `Migrator`, and `Profiler`. The role of each module is addressed as follows.

- `Task Module`: to define a task with its properties, e.g., ID, load value (or wallclock execution time, w), data size (s), local process, remote process (if a task is migrated), start time, end time, and migration time (if yes).
- `Simulator Engine`: the core of the simulator. There are three modules inside the engine, including `Queuing module`, `Task execution module`, and `Clocking module`. Clocking starts after the engine is triggered to run. It will end counting after all tasks are finished. Queuing controls task distribution and task dequeuing. In detail, each process has two types of queues, a local queue for executing local tasks and a remote queue for executing remote tasks. Task

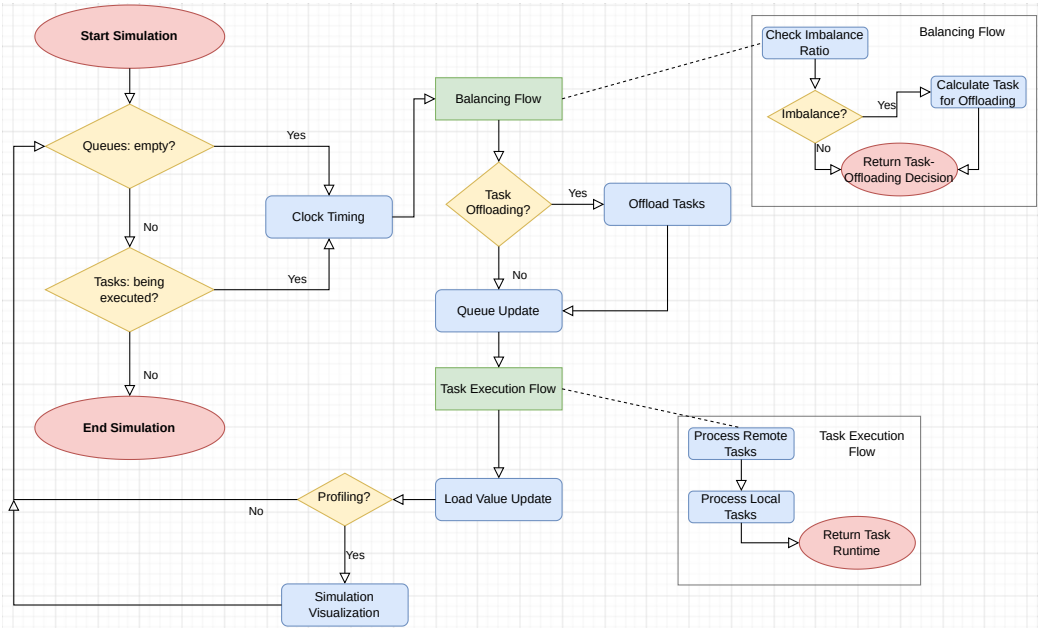


Figure 3.12.: Workflow of the simulation.

execution module manages the execution of tasks with timing, i.e., start time, execution time, and end time of a task.

- `Balancer`: the module for triggering load balancing operations. We emulate dynamic load balancing by revoking this module at every time step to make sure the condition of imbalance is checked. Therefore, if the ratio between clocking and runtime units is infinitesimal, `Balancer` can show an execution more realistic.
- `Migrator`: decides task migration based on the results of `Balancer`. This module takes care of migration time, delay, and arrival time of tasks as well as when they are executed on remote side.
- `Profiler`: performs statistics and visualization for profiling an execution by simulator.

Associated with the modularized design in Figure 3.11, we present how the workflow of the simulator runs. The actions of all modules are linked together and controlled through each time clock (or time step). First, the simulator inputs include:

- The number of tasks in total (T).
- The number of processes (P) with a given distribution of T on P processes, indicating the number of tasks on each process (T_i).
- The slowdown information, for example, the number of slowed down processes and how is the slowdown factor compared to the normal case ($Slow_i$). This value indicates execution speed (S_{P_i}) per process. Assume we have a baseline is 1.0 tasks/time unit. Plus slowdown, it could be slowed down as 0.5 or 0.25 tasks/time unit.
- Overhead information, in balancing operation and task migration, indicate latency (λ) and bandwidth (B) to calculate delay time. For generalization, we denote the overhead information by two types, balancing overhead ($O_{balancing}$) and task migration overhead (d). $O_{balancing}$ consists of monitoring and exchanging information, $m_i(t, t + \Delta t)$ and $b_i(t - \lambda)$ respectively. d represents delay time when tasks are migrated. With the data size of each task, d regulates the arrival time of a migrated task.

Figure 3.12 addresses the workflow of the simulator. The starting point opens a while loop (shown as `Start Simulatior`) with the conditions: queue length is not empty, or some tasks are being

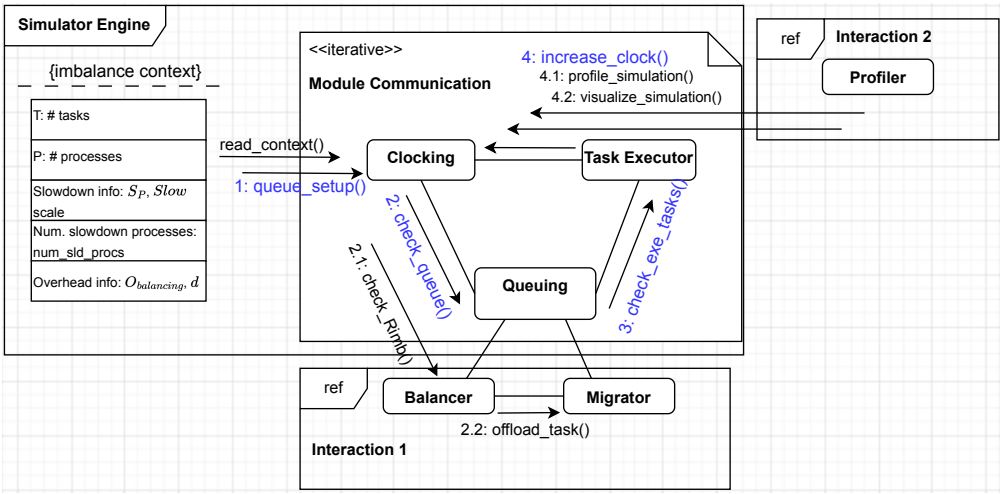


Figure 3.13.: A communication diagram of dynamic load balancing simulation.

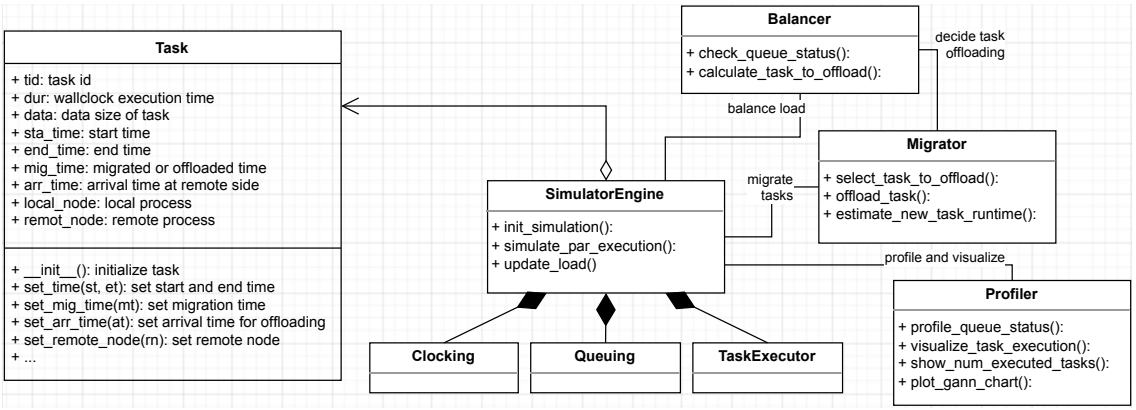


Figure 3.14.: A class diagram of dynamic load balancing simulator.

executed. If yes, the clock is ticked to simulate our scenario. “Tick” or “ticking” indicates counting time steps in the simulator. Next, we load `Balancer` denoted by a sub-flow (`Balancing Flow`). In detail, we check imbalance ratio and calculate whether tasks are offloaded or not. After that, we update all the queues if some tasks are offloaded or even if nothing is changed. The simulation engine will then load the task execution flow (`Task Execution Flow`). We prioritize remote tasks being executed first and local tasks afterward. This intends to make the simulation with the same reactive load balancing strategy from the original work proposed by Klinkenberg et al. [Kli+20a]. After `Task Execution Flow`, all task properties, e.g., end-time, runtime, are updated. Typically, these properties are forwarded to `Load Value Update` to keep tracking the total load values of each process. In general, the loop repeats and ends if there are no more tasks for execution. The following subsection will explain the implementation of our simulation design as a reference simulator.

3.2.1. Simulator Implementation

A communication diagram and a class diagram are used to present the simulator implementation. The communication diagram shown in Figure 3.11 explains how the simulator modules communicate with each other, while the class diagram shows at a more detailed level how classes and objects are defined in the implementation. Intuitively, we show the communication and class diagrams in Figure 3.13 and Figure 3.14, respectively.

The communication diagram is depicted with three main blocks: `Simulator Engine`, `Interaction 1`, and `Interaction 2`. Block `Simulator Engine` highlights the simulator input standing for the constraints of imbalance context (`imbalance context`). Then, it is followed by the sub-block - `Module Communication`, including **Clocking**, **Queuing**, and **Task Executor** that indicate the corresponding modules explained in Figure 3.11. At the bottom, block `Interaction 1` contains the modules, **Balancer** and **Migrator**. On the right corner, Block `Interaction 2` contains the module, **Profiler**.

Initially, the input is declared by the parameters, such as T, P, slowdown information, and overhead information that are mentioned in the previous sub-section. From here, the communication of all modules in block `Simulator Engine` with the other modules in block `Interaction 1` and `Interaction 2` can be described by four main actions:

1. `queue_setup()`
2. `check_queue()`
3. `check_exe_tasks()`
4. `increase_clock()`

These actions are highlighted in blue in Figure 3.13, and their order is also considered as the simulation workflow.

`queue_setup()` distributes tasks over the local queue of each process. Tasks are then queued and executed locally or remotely, managed by module **Queuing**. Users can adjust the setup of each queue to simulate a specific imbalance scenario.

`check_queue()` follows by each time step from module **Clocking**, where **Clocking** accounts for ticking or counting time steps. With time units in **Clocking**, their value can be configured by a ratio between the number of time steps and the amount of time that a task is executed (task execution time). Thereby, when task migration and balancing operation overhead are set by several time units, the configured ratio can reflect the proportion of overhead over task execution time. For example, suppose task execution time is in seconds and time steps are counted in milliseconds; the ratio will be $1 : 1000$. If the overhead of task migration is assumed to be a few milliseconds, then the simulation engine can emulate this more realistically. There are two sub-actions inside `check_queue()`, including `check_Rimb()` and `offload_task()`. If an imbalance condition is satisfied, tasks are offloaded depending on our offloading strategies. The action of task offloading is modularized as an interaction region (`Interaction 1`) where we can modify balancing algorithms and task offloading strategies.

`check_exe_tasks()` is a connection between **Queuing** and **Task Executor**. **Task Executor** takes tasks from the queues and simulates the execution of tasks in parallel. At a time step, all queues are checked. Tasks are executed slowly or quickly depending on the values of S_{P_i} and $Slow_i$ factors that we set up, where i indicates process P_i.

`increase_clock()` simply checks and increases the time step value. The interaction region (`Interaction 2`) is shown here to indicate module **Profiler**. **Profiler** supports profiling and visualizing task-by-task execution.

Building upon the communication diagram, we define several classes in response to the above-mentioned modules. Figure 3.14 shows the class diagram, where each class is illustrated with some corresponding properties and methods. In detail, we can see classes and interfaces revealed as a reference implementation, namely **Task**, **SimulatorEngine**, **Clocking**, **Queuing**, **TaskExecutor**, **Balancer**, **Migrator**, and **Profiler**.

Class **Task** needs relevant properties to manage a task, such as a unique id (`tid`), wallclock execution time (`dur`), and other attributes related to time, i.e., start time (`sta_time`), end time (`end_time`), migrated/offloaded time (`mig_time`). Behind the properties, there are relevant methods to setting or getting the corresponding values, e.g., `set_time(st, et)` to set start and end time of a task,

`set_mig_time(mt)` to set migrated time if a task is selected for offloading.

The main interface is `SimulatorEngine`, which has an aggregation link to class `Task`. Inside `SimulatorEngine`, we have three methods, including `init_simulation()`, `simulate_par_execution()`, and `update_load()`. `SimulatorEngine` exists a composition link with the other interfaces, `Clocking`, `Queuing`, and `TaskExecutor`. Besides that, the other association links in `SimulatorEngine` are connecting to the classes, `Balancer`, `Migrator`, and `Profiler`.

`Balancer` aims to check all the queue's status over time steps (by the method `check_queue_status()`), which is used to calculate imbalance ratio. If R_{imb} is satisfied a given threshold (condition), then `calculate_task_to_offload()` is invoked to decide offloading tasks.

Associated with `Migrator`, our implementation goes into the methods of `select_task_to_offload()`, `offload_task()`, `estimate_new_task_runtime()` to perform task offloading. Within this class, the properties, called `mig_time` and `arr_time`, are updated to fit the specification about balancing operation overhead and task migration overhead. Optionally, class `Profiler` is defined to support profiling and visualizing the execution behavior of the simulator.

3.2.2. Example Simulator Run

This sub-section provides an illustrative example of running the simulator. We re-use the the imbalance scenario in Figure 3.7 (B) on Page 42, Scenario 1. In particular, the simulation example is configured by applying reactive load balancing. Scenario 1 is set up 8 processes, $R_{imb} = 1.5$. There are two processes slowed down by the scale of 5×. Assuming the simulation configuration for baseline processes is an execution rate of 1.0 tasks/time unit, while the two slow processes execute tasks with a rate of 0.2 tasks/time unit. We configure time units in milliseconds and task execution in seconds. The ratio between task execution time and balancing time step is 1 : 1000. To provide a more realistic simulation, we make some assumptions about the runtime of local tasks and remote tasks as follows.

- For a remote task offloaded from a slow process to a fast one, its execution time is considered a half decrease compared to running on the original process. The reason is the constraint of performance slowdown on the original process.
- For a remote task offloaded from a fast process to a slow one, its execution time on the side of slow processes is slow as the local tasks.
- For a remote task offloaded from fast-to-fast or slow-to-slow, its execution time is assumed not to be changed.

Following the time steps and actions simulated for Scenario 1, the module `Profiler` of our simulator can record the information, such as when tasks are queued and started execution, migration, termination, to profile the behavior of reactive load balancing. Typically, Figure 3.15 depicts the profiled task execution over time progress by the simulator. The x-axis indicates time progress, while the y-axis indicates 8 processes in this simulation scenario. We configure 8000 tasks in total and 1000 tasks/process as a given distribution. Process P_0 and process P_1 are the slowdown processes. With the given scale of 5× slower than normal, one task is executed in 5 seconds. Without load balancing, the total load of process P_0 and P_1 would be 5000s. Through applying reactive load balancing, we can see that the completion time is reduced to around 2700s.

In Figure 3.15, a range of "green" slots indicates local task execution, while "orange" indicates remote task execution. Such an example simulator run, we simply configure the overhead information, consisting of:

- Balancing operation overhead (named $O_{\text{balancing}}$): 20ms occupied 2% of task execution time unit. This means a task is executed in 1s (1000ms), then balancing operations at once take 20ms. As mentioned in Subsection 3.1.2, $O_{\text{balancing}}$ accounts for the time of monitoring ($m_i(t, t + \Delta t)$) and exchaning load information ($b_i(t - \lambda)$).

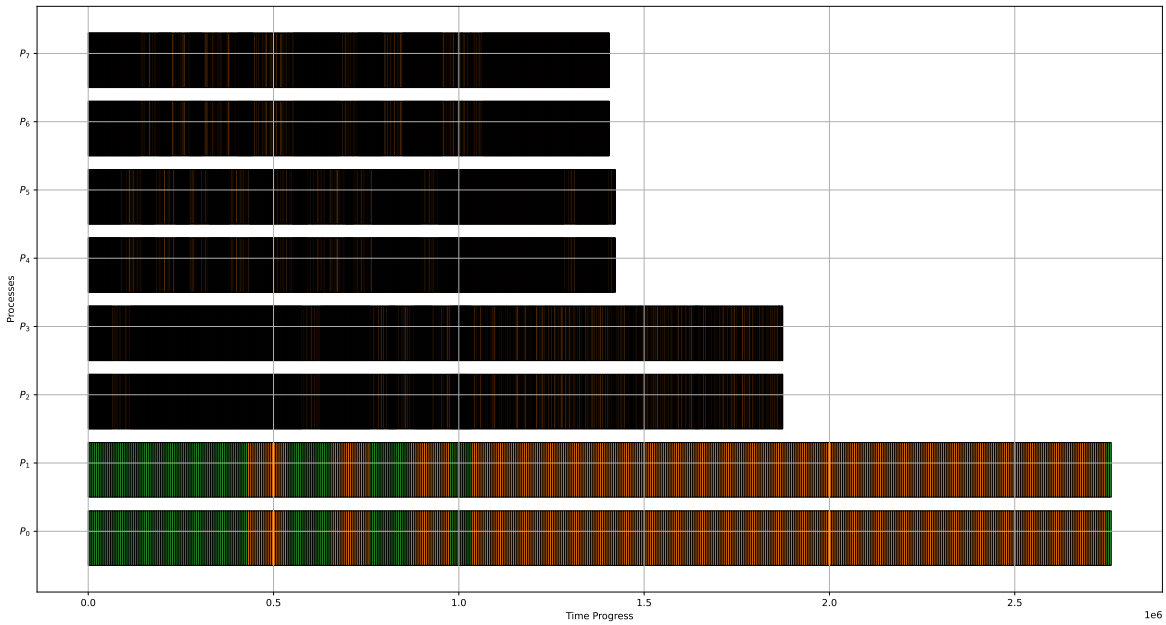


Figure 3.15.: An example simulation run with reactive load balancing in the scenario of 8 processes, 1000 tasks/process.

- Task migration overhead (represented by delay time d): 10ms occupied 1% of the task execution time unit.

As we can see in Figure 3.15, reactive task migration is sometimes incorrect because process P_0 and P_1 are known as overloaded processes but they still receive remote tasks. This also happens in a real execution with certain cases of high imbalance alongside many tasks per process and small execution time per task In Figure 3.15 with the processes P_2, ..., P_7, the area of local tasks is tight and overlaps together; therefore, we might not see clear local task execution. With process P_0 and P_1, the area of remote tasks is marked such incorrect task offloading at several time slots.

The following subsection provides further details about the evaluation of our simulator. We intend to show the impact between balancing operation overhead and task migration overhead. Notably, we address why reactive load balancing is late at runtime in distributed memory systems.

3.2.3. Simulator Evaluation

3.2.3.1. Example without load balancing

The simulation scenario in this subsection is similar to the previous example of Figure 3.15. However, we reduce the total number of tasks to reduce simulation time, where each process holds 100 tasks. The imbalance ratio is also configured $R_{imb} = 1.5$, $P = 8$ processes, and below is an input sample for configuring the simulator.

Listing 3.1: Input sample for the baseline simulation.

```
num_process: 8
num_tasks:   800
slowdown:    0.2 # the scale of slowdown
num_sld_processes: 2 # the number of slowed processes
bandwidth: 1485 MB/s # bandwidth value for calculating delay
exe_rate: 1.0 task/s # the execution rate
```

This input is used to simulate the baseline without load balancing. Technically, module `Balancer` is deactivated in this case. Therefore, there are no values for balancing operation overhead and task

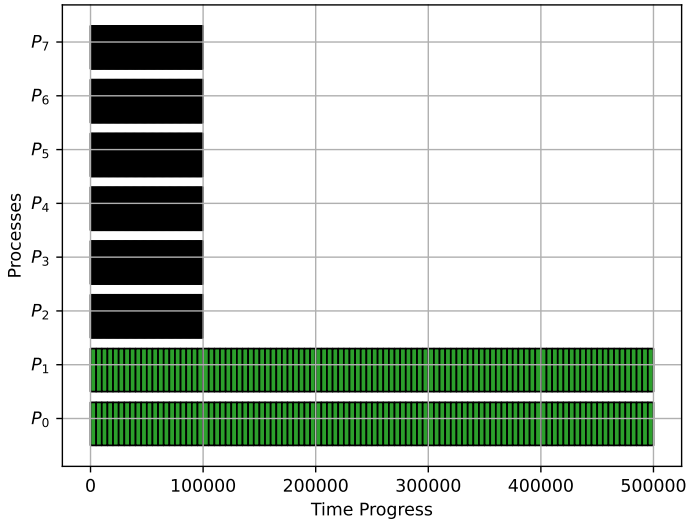


Figure 3.16.: Visualization of the baseline simulation without load balancing.

migration overhead. With time steps in milliseconds (ms), the clock is counted one by one at a time. In normal processes, the execution rate is 1.0 task/s, and the total load value of each is ideally 100s. In slowdown processes, the execution rate is 0.2, where we simply set process P_0 and P_1 as the slow processes. Executing each task in process P_0 and P_1 takes 5s. The completion time without load balancing is 500s in total. Figure 3.16 shows the simulated baseline. The x- and y-axis also point to the time progress and the load-value bars of 8 processes. Green boxes denote the executed tasks that are clearly seen at process P_0 and process P_1. The load values on other processes are smaller compared to process P_0 and P_1 due to many overlapped tasks. Hence, we might see task execution on their sides black (not clearly green as expected).

The following is a detailed statistic output from our simulator (Listing 3.2). First, the configuration information (under the block `Configuration`) is summarized. Second, under the block `Executed Tasks`, the output summarizes the number of local and remote tasks executed on each process. Then, the blocks `Total Load` and `Statistic` summarize the total load values of processes and the corresponding maximum, minimum, average load values, resulting in the imbalance ratio. Overall, the detailed output and profiled data can be directed to a `csv`-format file.

Listing 3.2: The output of the baseline simulation.

```
-----------------------------------------
Configuration
-----------------------------------------
   + num. processes:     8
   + num. tasks:       800
   + slowdown:         0.2
   + num. sld.procs:     2
   + bandwidth:      1485.0 (MB/s)
   + exe_rate:         1.0 (task/s)
-----------------------------------------

-----------------------------------------
Summary: Executed Tasks
-----------------------------------------
   + P[0]: num.local_tasks=  100, num.remote_tasks=    0
   + P[1]: num.local_tasks=  100, num.remote_tasks=    0
   + P[2]: num.local_tasks=  100, num.remote_tasks=    0
   + P[3]: num.local_tasks=  100, num.remote_tasks=    0
   + P[4]: num.local_tasks=  100, num.remote_tasks=    0
   + P[5]: num.local_tasks=  100, num.remote_tasks=    0
   + P[6]: num.local_tasks=  100, num.remote_tasks=    0
   + P[7]: num.local_tasks=  100, num.remote_tasks=    0
```

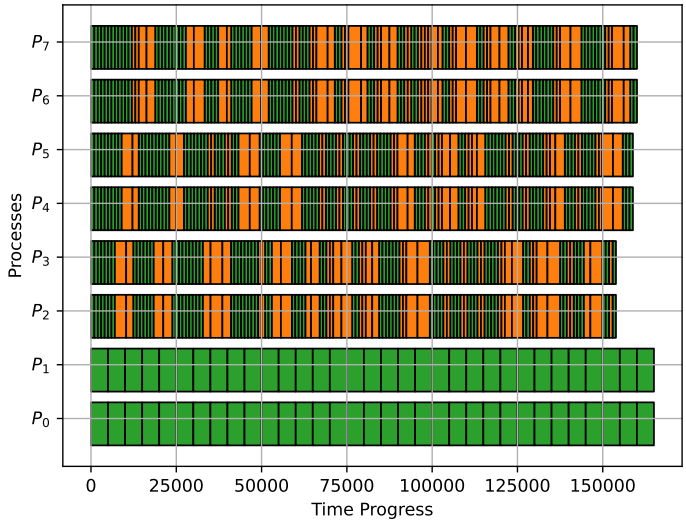


Figure 3.17.: Visualiztion of reactive load balancing case of simulation with $O_{\text{balancing}} = 1.0\text{ms}$ and $d = 1\text{ms}$.

```
-----------------------------------------

-----------------------------------------
Summary: Total Load
-----------------------------------------
   + P[0]: local_load= 500.00(s), remot_load=   0.00(s)
   + P[1]: local_load= 500.00(s), remot_load=   0.00(s)
   + P[2]: local_load= 100.00(s), remot_load=   0.00(s)
   + P[3]: local_load= 100.00(s), remot_load=   0.00(s)
   + P[4]: local_load= 100.00(s), remot_load=   0.00(s)
   + P[5]: local_load= 100.00(s), remot_load=   0.00(s)
   + P[6]: local_load= 100.00(s), remot_load=   0.00(s)
   + P[7]: local_load= 100.00(s), remot_load=   0.00(s)
-----------------------------------------

-----------------------------------------
Statistic
-----------------------------------------
max. load:   500.0
min. load:   100.0
avg. load:   200.0
R_imb:         1.5
sum. overloaded_load:    600.0
sum. underloaded_load:   600.0
-----------------------------------------
Write profiled queue data to: ./poc_visualize_baseline_case_8_procs_output.csv
-----------------------------------------
```

3.2.3.2. Example with reactive load balancing

In the case of applying reactive load balancing, we activate `Balancer` and configure the values for balancing operation overhead and task migration overhead. With a simulation trial, we initially set the balancing overhead, $O_{\text{balancing}} = 1.0\text{ms}$, and delay time, $d = 1.0\text{ms}$. Compared to the execution time unit of a task in second, we highlight that balancing overhead and delay time occupy 0.1%.

Figure 3.17 shows the visualization of the simulation with reactive load balancing, where $O_{\text{balancing}} = 1.0\text{ms}$ and $d = 1.0\text{ms}$. The x- and y-axis shows time progress and processes executing tasks, where "green" and "orange" slots indicate local can remote tasks. Overall, the completion time is reduced. The simulated operations of reactive task offloading with $O_{\text{balancing}}$ and d are acted as follows.

- Suppose at a time t_k, module `Balancer` detects an imbalance and tasks from process P_0 are then offloaded to process P_7 as an example. However, the tasks are proceeded in the queue for offloading at time $t_k + 1$ because of $O_{\text{balancing}} = 1.0$.
- Similarly, assuming tasks from process P_0 are decided to offload to process P_7 at t_k. However, these tasks arrive at process P_7 at time $t_k + 1$ because of $d = 1.0$.

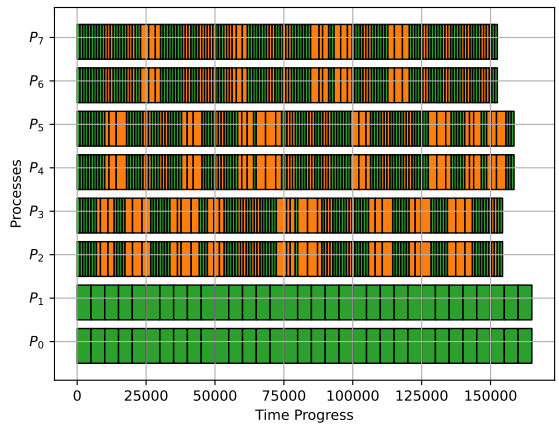
In the simulator, module `Clocking` controls the ticking procedure of time steps. We can adjust this procedure to simulate the execution of dynamic load balancing similar to running in practice. Listing 3.3 shows the output sample of this simulated scenario. The output format is equivalent to Listing 3.2, but we skip the part of `Configuration`. As a result, the summary under block `Executed Tasks` highlights the number of remote tasks from process P_2 to P_7. In response to the number of remote tasks, block `Total Load` indicates the total load values of remote tasks, proving that reactive load balancing works. In the end, the completion time is improved significantly $\approx$ 165.0ms, and R_{imb} is almost 0.0.

Listing 3.3: The simulation output of reactive load balancing with $O_{\text{balancing}} = 1.0$ms, $d = 1$ms

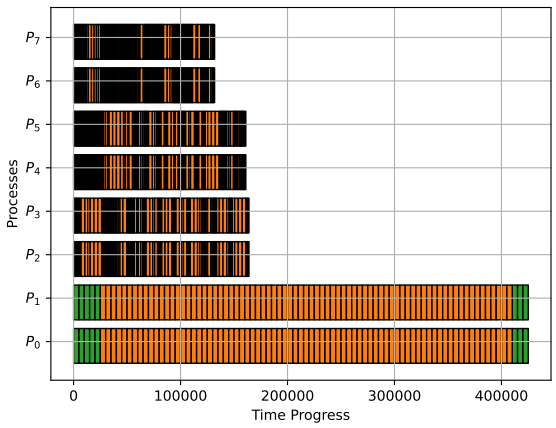
```
-----------------------------------------
Configuration
-----------------------------------------
   ...
-----------------------------------------

-----------------------------------------
Summary: Executed Tasks
-----------------------------------------
   + P[0]: num.local_tasks=   33, num.remote_tasks=    0
   + P[1]: num.local_tasks=   33, num.remote_tasks=    0
   + P[2]: num.local_tasks=   82, num.remote_tasks=   38
   + P[3]: num.local_tasks=   82, num.remote_tasks=   38
   + P[4]: num.local_tasks=   81, num.remote_tasks=   42
   + P[5]: num.local_tasks=   81, num.remote_tasks=   42
   + P[6]: num.local_tasks=   76, num.remote_tasks=   48
   + P[7]: num.local_tasks=   76, num.remote_tasks=   48
-----------------------------------------

-----------------------------------------
Summary: Total Load
-----------------------------------------
   + P[0]: local_load= 165.00(s), remot_load=   0.00(s)
   + P[1]: local_load= 165.00(s), remot_load=   0.00(s)
   + P[2]: local_load=  82.00(s), remot_load=  72.50(s)
   + P[3]: local_load=  82.00(s), remot_load=  72.50(s)
   + P[4]: local_load=  81.00(s), remot_load=  75.00(s)
   + P[5]: local_load=  81.00(s), remot_load=  75.00(s)
   + P[6]: local_load=  76.00(s), remot_load=  81.00(s)
   + P[7]: local_load=  76.00(s), remot_load=  81.00(s)
-----------------------------------------

-----------------------------------------
Statistic:
-----------------------------------------
max. load:   165.0
min. load:   154.5
avg. load:   158.1
R_imb:         0.0
sum. overloaded_load:     13.8
sum. underloaded_load:    13.8
-----------------------------------------
Write profiled queue data to: ./profiled_queues_reactlb_01_d1.csv
-----------------------------------------
```

3.2.3.3. Experiments with the simulator

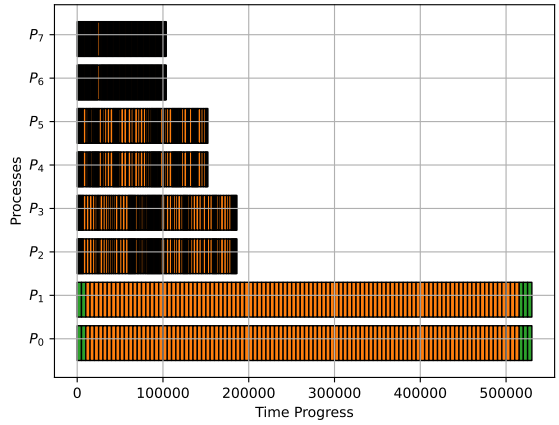
To see the effects of balancing overhead, we try to increase $O_{\text{balancing}}$ first from 2, 5, 10, to 20ms, corresponding to the occupation of 0.2%, 0.5%, 1.0%, 2.0% over task execution time unit, while $d = 2$ is



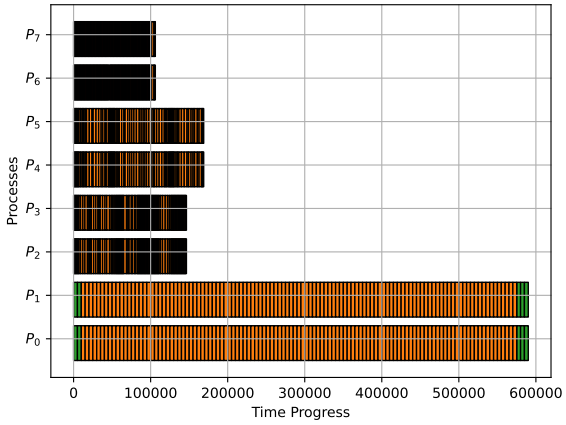
(a) Reactive LB with $O_{\text{balancing}} = 2\text{ms}$ and $d = 2\text{ms}$



(b) Reactive LB with $O_{\text{balancing}} = 5\text{ms}$ and $d = 2\text{ms}$



(c) Reactive LB with $O_{\text{balancing}} = 10\text{ms}$ and $d = 2\text{ms}$



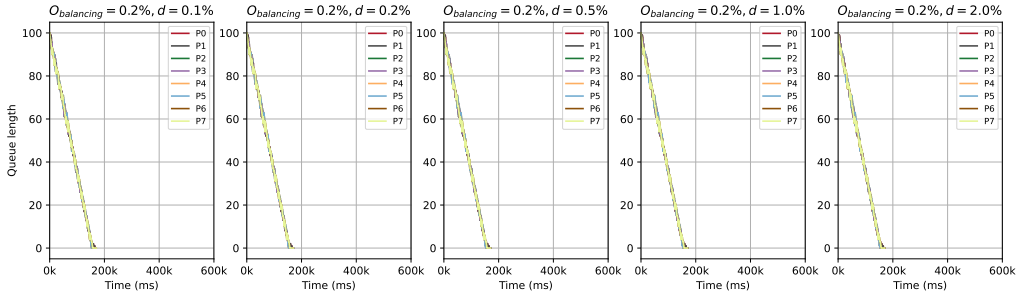
(d) Reactive LB with $O_{\text{balancing}} = 20\text{ms}$ and $d = 2\text{ms}$

Figure 3.18.: A visualization of reactive load balancing with increased balancing overhead and delay.

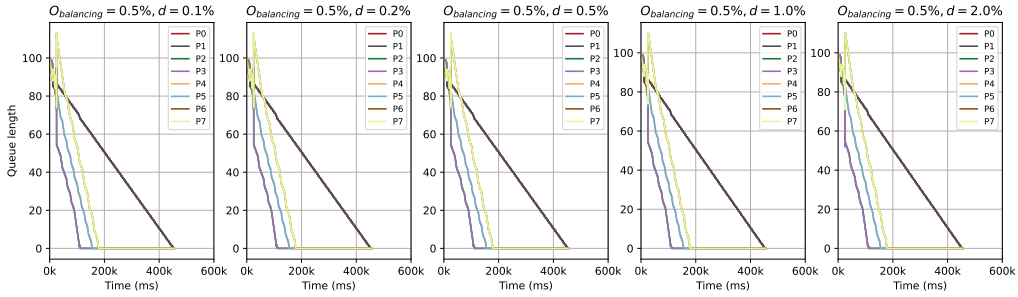
kept constantly. Figure 3.18 shows these tests in the visualization of reactive load balancing (denoted by Reactive LB). Compared to the baseline (in Figure 3.16 on Page 51) and the initial imbalance scenario (in Figure 3.17 on Page 52 with $O_{\text{balancing}} = 1\text{ms}$, $d = 1\text{ms}$), Figure 3.18 highlights the impact of balancing operation overhead. Not only delay time, load balancing operations in monitoring, exchanging load information/queue status, and calculating imbalance conditions can also significantly affect overall performance. $O_{\text{balancing}}$ is sensitive when almost all dynamic load balancing methods decide on many continuous actions and task migrations. For instance, the balancing decisions start getting worse when $O_{\text{balancing}}$ occupies 0.5% over task execution time unit.

For another evaluation, we keep $O_{\text{balancing}}$ as a constant and vary d to see the impact of task migration overhead, Figure 3.19 shows two experiments conducted, one in Figure 3.19a showing $O_{\text{balancing}} = 2$ and d is ranged from 0.1% to 2%; one in Figure 3.19b showing $O_{\text{balancing}} = 5$ and d is ranged from 0.1% to 2%. The imbalance scenario is stayed similar to the previous tests. In both figures, the x-axis indicates the progress of queues changed over time steps (in ms); the y-axis indicates the queue length as the number of remaining tasks over time steps.

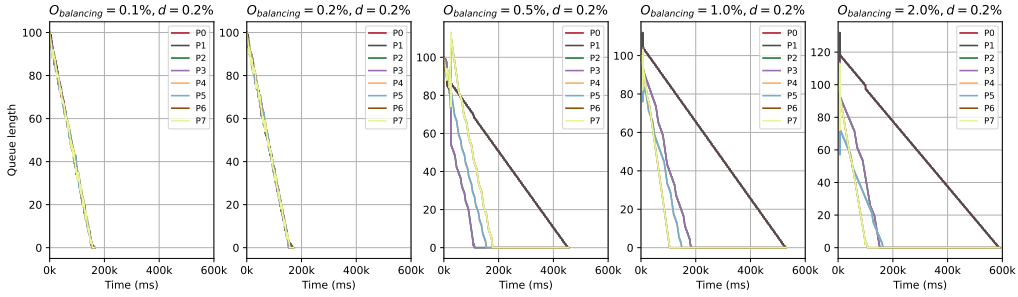
Generally, all the queues are converged into 0 at the end after the execution is finished. However, the slope of different queue lines might be different because some processes are slowed down and execute tasks slower. Note that these experiments with the simulator are applied reactive load balancing, and the final results show the load balancing performance. Typically, the slope in these figures represents how efficient the performance is. The last queue converged at shorter time steps determines a better performance. The balancing operation overhead is still trivial with $O_{\text{balancing}} = 2$. We can see that



(a) Reactive LB in that keep $O_{\text{balancing}} = 2$ constantly and vary d



(b) Reactive LB in that keep $O_{\text{balancing}} = 5$ constantly and vary d



(c) Reactive LB in that vary $O_{\text{balancing}}$ and keep $d = 2$ constantly

Figure 3.19.: Evaluation of the queue length convergence when varying task migration overhead (delay) and balancing operation overhead.

increasing d does not affect the convergence speed much. However, with $O_{\text{balancing}} = 5$, delay d here is more sensitive, and the convergence of queues gets impacted. Following this, if we compare with the experiments visualized in Figure 3.18 (on Page 54) when d is constant and $O_{\text{balancing}}$ is varied, Figure 3.19c emphasizes in detail these scenarios with the convergence of queue lengths. For a slightly change of $O_{\text{balancing}}$, the load balancing performance is significantly affected.

3.3. Towards Proactive Idea in Dynamic Load Balancing and Co-scheduling Tasks

The previous section highlights that task migration overhead in distributed memory systems is not the only primary influence factor. Instead, balancing operation overhead can also affect our performance. The balancing operations consist of monitoring queue information or status, calculating imbalance, exchanging status information, and making decisions on task migration. The overhead of each operation might be small and trivial. However, when they are performed numerously and continuously, the decisions of task migration can be significantly impacted. Eventually, the performance of load balancing is affected overall.

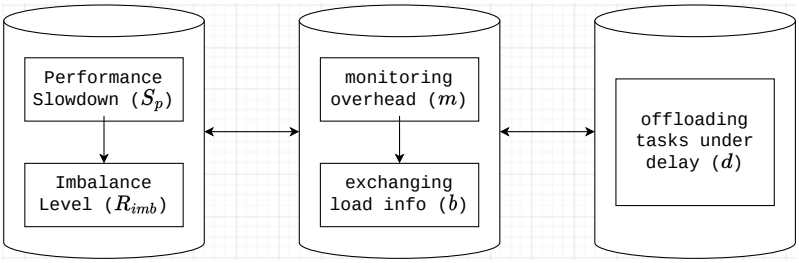


Figure 3.20.: Three pillars of factors might challenge reactive load balancing.

This subsection emphasizes three representative pillars that lead to wrong decision-making for reactive load balancing in particular and dynamic load balancing in general. Figure 3.20 shows these pillars from left to right, including:

- The first pillar implies the factors of imbalance reason (in our context, slowdown is mentioned and denoted by S_P) and imbalance level (R_{imb}).
- The second pillar is related to the overhead of load monitoring (m) and information exchanging (b).
- The third pillar is the overhead of task offloading, indicating the delay time (d) when tasks are migrated.

Concretely, the first pillar implies imbalance context related to performance slowdown, leading to high or low imbalance levels. The second pillar is overhead in the middle, revolving around balancing operations. At runtime, all dynamic load balancing methods perform seemingly random operations before a task can be migrated. The last pillar is about overhead and decision-making for task offloading as well as task migration. If the second pillar takes a small overhead, then decisions in the third pillar can be made faster. When tasks are decided to migrate, the main overhead is delay time d, which d is large or small, depending on the data size of tasks, latency, and network bandwidth. We describe the interaction between these pillars as follows.

- If R_{imb} is high, the imbalance condition might be detected in high frequency. Then, task offloading decisions are made many times.
- To know how many tasks and where to offload tasks, there are different ways to make decisions associated with the first pillar.
  - For safety, exactly one task should be moved at one point in time. However, this might lead to many monitoring, checking, and exchanging load information operations. Simultaneously, the big enough values of d can make these points challenging because it can take time to finish the exchanging operations before a task is decided for migration.
  - To reduce the number of migration procedures, several tasks can be moved at once. However, deciding how many tasks are appropriate without prior knowledge is difficult.

We can see that "reactive" load balancing implies taking actions reactively based on the most current status. This also means that we cannot plan to decide how many tasks should be migrated at a time adaptively. Besides, there is no information about which process could be a potential victim. When balancing operation and task migration produce a certain overhead, the decisions of reactive load balancing can be late and incorrect.

In general, we highlight that the challenge of dynamic load balancing is to obtain prior knowledge about load information. Assuming that we can predict load information or generate knowledge about load information at runtime, we can drive load balancing better. This thesis motivates an idea: How can we perform load balancing more proactively at runtime? "Proactive" implies that we can calculate how bad the imbalance is at a current state and how many tasks should be migrated at once. Consequently, we propose a proactive load balancing approach that enables to characterize task execution, learn load information, and predict imbalance. This is considered a better prognostication

compared to the reactive load balancing approach. From idea to practice, our proactive approach is developed by the following technical questions:

- Can we use a dedicated thread to obtain load values at runtime?
- Can we leverage the profiled information of several first iterations (or previous iterations) to generate the load knowledge?
- Can we change from reactive to more proactive task offloading between iterations?

For most of the use cases in HPC, iterative applications or simulation frameworks can benefit from our approach. Their behaviors are divided into multiple execution phases (so-called iterations). With a large simulation use case, a program can be run with many iterations, and an adaptive approach for load balancing is essential. Therefore, our proactive idea can be relevant to make load balancing more adaptive. The following chapter will show in detail how we design the idea.

4. A Proactive Approach for Dynamic Load Balancing

This chapter discusses our proactive approach for dynamic load balancing. The goal is "*more information, better load balancing*". Almost all static load balancing methods assume to have load information or related information to generate a cost model. Thereby, task migration and task distribution rely on prior knowledge to solve balancing. In dynamic load balancing, we do not rely on prior knowledge. Task migration is mainly based on the most current execution status at runtime, such as queue length and execution speed on each process. However, current status information is insufficient, implying that approaches like work stealing or reactive load balancing are seemingly based on speculation. Obviously, speculative balancing operations can be right or wrong for a given period.

4.1. Overview

In our proactive approach, we exploit influence factors related to execution status to predict and provide knowledge of task execution time. The knowledge based on load prediction helps to estimate the level of imbalance. We calculate the number of appropriate tasks and potential processes for task offloading. Benefiting from modern computer architectures and task-based programming models, the approach extends reactive load balancing by employing a dedicated thread for:

- Supporting task and system characterization instead of only monitoring queue length.
- Using the characterization information to predict load values at runtime. Instead of missing the prior knowledge before running applications, we can generate new knowledge based on predicted load values, e.g., w, L.
- Adapting the prediction knowledge to guide task offloading. We calculate the load difference, the number of tasks, and potential process candidates for better offloading tasks.

Our approach is implemented towards a proactive scheme of load balancing, which facilitates different task offloading methods. To be intuitive, Figure 4.1 shows how this scheme works. The x-coordinate again shows the time progress, while the y-coordinate shows process P_i spawning two threads for executing tasks and one dedicated thread for performing load balancing. This thread fits today's modern computing architectures with the increasing number of cores, where one core can be left off to run the dedicated thread (denoted by T_{comm}). In practice, our scheme can be deployed through hybrid MPI+OpenMP, which is mostly exploited in various task-based parallel programming models.

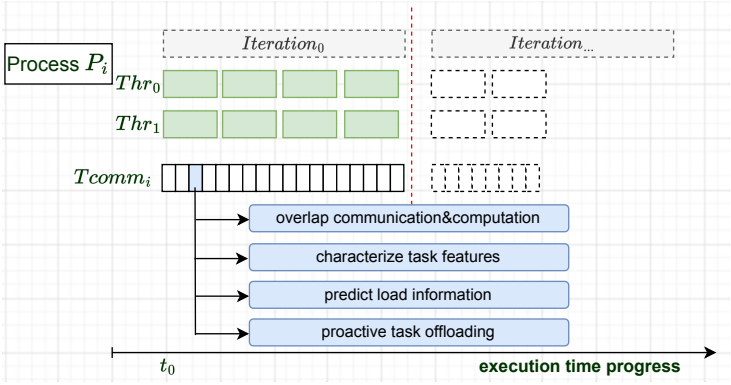


Figure 4.1.: A proactive scheme for task offloading to solve dynamic load balancing in general.

In Figure 4.1, we suppose that process P_i contains two threads, Thr_0 and Thr_1, for executing tasks and one dedicated thread, $Tcomm_i$, for communication as well as proactive load balancing. Assuming the execution is iterative, which indicates $Iteration_0$, then $Iteration_1$, and so on. Different from reactive load balancing, $Tcomm_i$ in our approach is driven to perform:

- Mainly keeping communication overlapped with computation.
- Characterizing task feature and system information along with the corresponding load values, such as w, L, core frequency, etc.
- Training a load prediction model at runtime.
- Offloading tasks proactively.

The above operations are what $Tcomm$ can perform separately from the other threads to facilitate load balancing. Furthermore, we can adapt these operations to specific application and system domains. The following sections show different task offloading methods based on our proactive balancing scheme. Specifically, we show two task offloading methods and one extension for co-scheduling tasks across multiple applications, including:

- Method 1: feedback task offloading. We introduce method 1 in Section 4.2.
- Method 2: ML-based task offloading, where "ML-based" indicates a machine learning based model for online load prediction. Method 2 is described in Section 4.3
- Extension: co-scheduling tasks across multiple applications. We address this extension in Section 4.4.

4.2. Feedback Task Offloading

The idea behind feedback is an improvement of reactive approach. Applying to iterative applications, the first iteration is kept doing with reactive load balancing. Tasks are offloaded from a slow process to a fast one during execution. After the first iteration, we use $Tcomm$ to generate a statistic on each process about the number of executed tasks in local and remote processes, the total load values of local and remote tasks. The statistic shows how good balancing in the first iteration is. Then, we use this statistic to interpolate the load difference among processes that support an estimation for which processes are overloaded and underloaded. From here, a priority function for task offloading is generated as feedback to drive proactive task offloading in the next iterations.

Figure 4.2 presents the design of $Tcomm$ to perform feedback task offloading. Again with the coordinates, the x-axis indicates execution time progress. Vertically, we illustrate process P_i, where its two main threads (Thr_0, Thr_1) are shown with executing tasks, $Tcomm_i$ with balancing operations. In the first iteration ($Iteration_0$), the operations of reactive load balancing are performed the same.

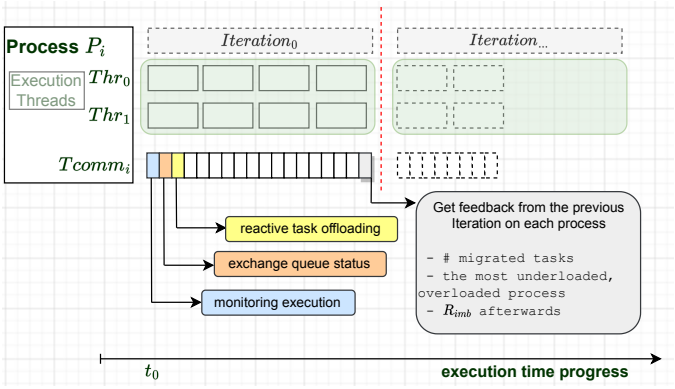


Figure 4.2.: An overview of feedback task offloading through the operations of $Tcomm$.

For instance, we can see the small rectangles with three different colors. Each rectangle represents an operation in reactive load balancing, where blue is monitoring execution (i.e., queue length), orange is exchanging the status, and yellow is offloading tasks. These operations are repeated until $Iteration_0$ is terminated. From here, the arrow at the last rectangle on $Tcomm_i$ points to a statistic for feedback.

A crucial question is what information is needed for feedback. In this method, we collect the information regarding the efficiency of the first iteration, including:

- The number of local and remote tasks executed in a process.
- The number of offloaded tasks.
- The total load values of local and remote tasks.
- The R_{imb} ratio and speed up after reactive load balancing in the first iteration.

In case the current iteration is not the first iteration (`Iteration 0`), feedback requests the iteration called the *previous* iteration. Generally, Algorithm 1 shows how the feedback information works on each process. The feedback includes recorded information about the local, remote load, number of local tasks, and number of remote tasks denoted by L^{local}`[]`, L^{remote}`[]`, N^{local}`[]`, N^{remote} `[]` respectively. The output is an array of priorities corresponding to each process. The priority gives each a reference number that says offloading tasks (if > 0) or receiving tasks (if < 0). Their values suggest a reference limit for how many tasks we should offload or receive in a process. These input/output arrays have P elements alluding to the number of involved processes.

(1) Procedure `feedback_balancing()` checks the input arrays. Then, it is ensured to run only one time every iteration by the flag $\texttt{flag}_{\text{feedback}}$, except for the first iteration of collecting data to give feedback.

(2) We summarize a statistic about how good reactive load balancing is performed in the first/previous iteration.

The total load value (L) per process is calculated by the sum of local and remote load, $L^{\text{local}} + L^{\text{remote}}$. After distributing the values L, L^{local}, L^{remote} around, each process calculates L_{max}, L_{avg} that are used to check the imbalance ratio R_{imb}. By evaluating R_{imb}, we can determine the efficiency of reactive load balancing in the first/previous iteration.

(3) The procedure interpolates information about the original load value and task execution time of each process based on average, denoted by $\hat{L}$`[i]`, $\hat{w}^{\text{local}}$`[i]`. Also, if the current process executed remote tasks, the corresponding $\hat{w}^{\text{remote}}$`[i]` is also estimated. Benefit from $\hat{L}$, we estimate information about $\hat{L}_{max}$, $\hat{L}_{avg}$ $\hat{R}_{imb}$ to check how load difference among processes if reactive load balancing is not applied. Eventually, the difference of $\hat{L}$ and $\hat{L}_{avg}$ associated with $\hat{w}^{\text{local}}$`[i]` and $\hat{w}^{\text{remote}}$`[i]`

Algorithm 1: Feedback Task Offloading

Input : Array L^{local}[], L^{remote}[], N^{local}[], N^{remote}[], each has P elements. Where,
L^{local}[i] is local load value;
L^{remote}[i] is remote load value;
N^{local}[i] is the number of local tasks; and
N^{remote}[i] is the number of remote tasks in process P_i

Output: Array $D_{\text{reference}}$[]: a reference distribution of priorities

Procedure feedback_balancing(*Array* L^{local}, L^{remote}, N^{local}, N^{remote}):

```
    /* Check the input */
1   assert(L^local, L^remote, N^local, N^remote)
2   assert(flag_feedback)

    /* Summarize statistic */
3   Array L ← L^local + L^remote
4   L_max, L_avg ← maximum and average load based on Array L
5   R_imb ← calculate imbalance ratio in overall
6   evaluate(R_imb) // evaluate the efficiency of reactive load balancing in the
     first/previous iteration

    /* Interpolate the original load values and local tasks based on average */
7   for i ← 0 to P-1 do
8       L̂[i], ŵ^local[i] ← based on L^local[i], N^local[i]
9       ŵ^remote[i] ← based on L^remote[i], N^remote[i]
10  L̂_max, L̂_avg ← maximum and average load based on Array L̂
11  R̂_imb ← calculated by L̂_max, L̂_avg
12  D_reference ← a reference distribution of priorities if R̂_imb ≥ Threshold_imb
13  Return: D_reference
```

supports generating an array of priorities, $D_{\text{reference}}$. As a result, we interpolate the number of tasks that should be offloaded/received to fill the gap between $\hat{L}$ and $\hat{L}_{avg}$. For example, process P_i has $D_{\text{reference}}[i] = 99$ that says it should be an offloader, and a reference limit is 99 tasks.

An obvious advantage in feedback task offloading is determining which process is a potential victim for offloading tasks and which is receiving tasks. Another advantage is that we can refer to a limit of how many tasks should be offloaded. These points can drive task offloading in subsequent iterations better.

4.3. ML-based Task Offloading

4.3.1. Reprequisite and Design

In feedback task offloading, we use the statistic of executed local/remote tasks after one iteration to give feedback for the next iteration. Task offloading in the upcoming iterations can be performed proactively. When system performance changes significantly or the total load values of processes

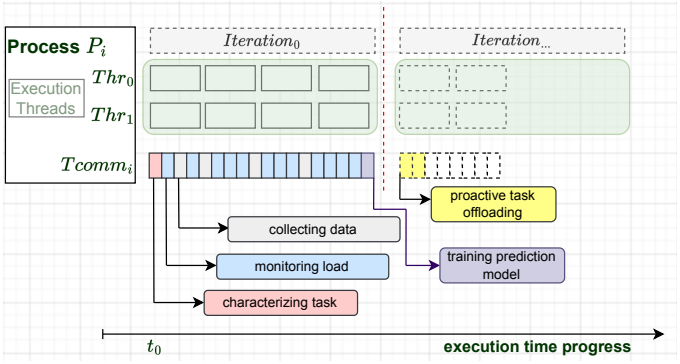


Figure 4.3.: An overview of ML-based task offloading through the corresponding operations running on $Tcomm$.

vary greatly over iterations, this method can be inefficient because the feedback after one iteration might be wrong in the next iteration. In addition, the statistical information after one iteration is finished still lacks coherence with relevant factors on the system side. Therefore, we developed another method using machine learning based on the features of both sides, application and system, for online load prediction.

We show that the ML-based idea is feasible because many use cases in HPC are numerical simulations. The program behaviors are iterative execution, where the number of tasks and their arguments are repeated over iterations. The execution phases are also repeated until termination. Therefore, one iteration passed can give useful information for the next iteration. ML-based task offloading method exploits the dedicated thread to combine three main stages:

- task characterization
- load prediction
- proactive task offloading

Figure 4.3 shows the design of this method, considered as a reference implementation scheme in practice. For keeping a consistent overview similar to the feedback task offloading method in Figure 4.2 on Page 61, the x-axis shows execution time progress, process P_i is illustrated with two threads for executing tasks (Thr_0, Thr_1), and $Tcomm_i$ is isolated to perform:

1. `characterizng task`: indicated by red rectangle. We characterize the input arguments of tasks when they are created. Besides that, system information is profiled. We profile CPU core frequency (but not limited to other performance counters that might influence the execution time of tasks, w) as a relevant factor on the system side. To define which features are profiled, we can configure them before the application is executed.
2. `monitoring load`: indicated by blue rectangle. We record the wallclock execution time (w) of each task and the total load (L) of each process after an iteration is finished.
3. `collecting data`: indicated by grey rectangle. We generate a dataset for training machine learning models based on the characterized and monitored information.
4. `training prediction model`: indicated by purple rectangle. After the dataset is generated, $Tcomm$ is triggered to train a prediction model. When the model is successfully trained, it is loaded to predict the load values of tasks/processes for the next iterations.
5. `proactive task offloading`: indicated by yellow rectangle. We employ a proactive algorithm for guiding task offloading. Tasks can be offloaded early when the next iterations get started.

Regarding the input and output layers of our prediction model, we formulate their properties as follows.

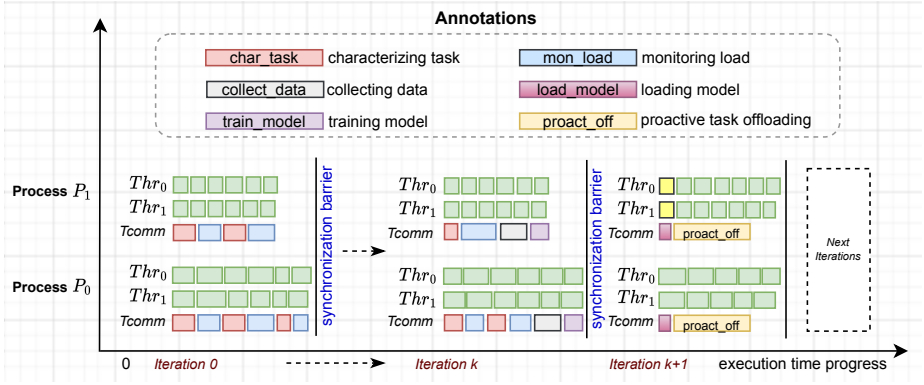


Figure 4.4.: Working flow of ML-based task offloading at runtime.

- Prediction Input: In cases of predicting w, the input must be the features that we can collect before a task is run, and these features have to reflect the values of task execution time. If the input can only be collected after the termination of tasks, it is irrelevant to apply machine learning at runtime. In cases of predicting L, one way can be using the prediction of w values to calculate L. Another can use the L values in previous iterations to train and predict L for the next iterations.
- Prediction Output: is task wallclock execution time (w) or total load value of a process after an iteration is done (L). L is estimated by the sum of w values of a process. Therefore, w and L can interpolate each other depending on the characteristics of tasks.

To be intuitive, Figure 4.4 demonstrates step-by-step the above operations. The demonstration is described with two processes, process P_0 and process P_1 following the vertical coordinate. The x-coordinate indicates execution time progress over iterations, such as *Iteration* 0, ..., *Iteration* k, *Iteration* $k+1$. Assume each process has two threads for executing tasks (so-called two *worker* threads, Thr_0 and Thr_1), and the dedicated thread denoted by $Tcomm$. The behavior is an iterative execution, where the number of iterations in a realistic HPC use case might be thousands. This implies a possibility that we can employ ML-based load prediction after a few iterations at the beginning and then apply for proactive task offloading.

In Figure 4.4, "green" boxes on the rows of threads Thr_0, Thr_1 indicate executing local tasks, while "yellow" boxes indicate remote tasks. The operations of $Tcomm$ are illustrated by the boxes with different colors, representing different operations when they are performed. Iterations are increasingly indexed by 0, ..., k, $k+1$, and so on. On the rows of $Tcomm$, the operations of characterizing tasks, monitoring load, and collecting data are performed from *Iteration* 0 to *Iteration* k. These iterations are left for preparing a dataset before training a prediction model. The number of iterations left to generate a dataset might differ depending on user applications. When the dataset is generated, we can see that the operation of training the model is triggered on each process. After training, the model is loaded to predict the load values of the next iterations. Given *Iteration* $k+1$, the model is loaded when it is just getting started. Thereby, we adapt the predicted load results to guide proactive task offloading. Corresponding to the annotation in Figure 4.4, the boxes with different colors link to the following operations.

- `char_task` (red): characterizing task. This depends on what we want to characterize. Our case focuses on the input arguments of tasks. Besides, some performance counters, which are enabled to check before a task is executed, can also be involved in `char_task`. Therefore, Figure 4.4 shows that `char_task` is triggered after tasks are created or before tasks are executed.
- `mon_load` (blue): monitoring load. We aim at recording the load values (w) after tasks are done. This operation can be triggered after executing tasks.
- `collect_data` (grey): collecting data. This indicates the monitored load values that we use to generate a dataset for training.

- `train_model` (magenta): training model. When the dataset is generated, `train_model` is called. The trained models are mainly regression types to predict load values; however, a custom model is also relevant if we can obtain a good prediction.
- `load_model` (purple): loading model. This denotes when the model is trained, it is loaded to predict the load values of the next iterations.
- `proact_off` (orange): proactive task offloading. This annotation indicates that we have prediction results, we can load them to guide proactive task offloading.

Concurrently with the other worker threads, *Tcomm* performs these operations asynchronously. Except when the main execution of worker threads stops, *Tcomm* is finished. In general, there are some requirements for ML-based task offloading, including:

- The operations invoked inside *Tcomm* should be modularized and callable. Furthermore, they are able to be pre-defined by users. The reason is that online prediction with machine learning cannot be fixed. For different use cases, users might understand their applications better.
- Task characterization (`char_task`) should be selected with influence features, depending on the application use case. This can affect the overhead of the modules such as `collect_data`, `train_model`. Compared to prediction problems in other fields, such as computer vision [Seb+06], NLP [DL18], we have to keep prediction models in load balancing context small and simple due to the scope of available input features.
- The ML-based prediction models should be lightweight to avoid overhead in training and loading the models. Regarding accuracy, we do not need a very high result because load balancing is better when relying on the difference in load between different processes.

Our use case is iterative, which is especially relevant when applying machine learning to predict execution time over iterations. Furthermore, the input features are not only from the user application side but also from the system side, which can extend to topology information, bandwidth, and memory access pattern. However, to adapt this method to different use cases, we need to answer three questions:

- Which input features and how much data are generally relevant?
- Which machine learning model is suitable?
- Which way can the learning parameters change at runtime?

With machine learning, the answer cannot be "*generalized*" because prediction and machine learning models cannot be fixed. It should be served on domain-specific applications that we can change and adapt flexibly. The following subsections will clarify our argument through two examples in practice.

4.3.2. Online Load Prediction

The main perspective of ML-based task offloading is that a users' application is considered as a black box. A task is defined by a code region and its data. The code refers to a computation kernel. Users do not need to make an effort to optimize how tasks are executed in parallel. After an execution phase is complete, the programs' flow and task results are given back to the users' control side.

During task execution, this is reasonable for predicting load values within a task-based programming model. Particularly, the dedicated thread on each process is suitable to launch a lightweight machine-learning model. Prediction with supervised learning needs input features and output labels to train. To this end, we formulate the input, output, and model using the following parameters:

- IN_{app}: input features associated with task characterization, e.g., task arguments, data size, code region. We recommend some pre-observation about the task features before choosing which features are influential.

- IN_{sys}: input features associated with system characterization, i.e., performance counters. We recommend factors that affect execution time, such as core frequency (a min-max range of frequencies), memory access patterns, and cache hit/miss when executing tasks.
- OUT: output labels. In our context, labels are simplified as the execution time of tasks (w) or total load of processes (L).
- $MODEL$: machine learning algorithms, e.g., linear regression. We use regression algorithms to predict load value because regression is suitable for prediction problems with continuous numerical values. However, other relevant algorithms are also reasonable for specific applications.

To illustrate the usage of our online prediction model, we show two examples: matrix multiplication (denoted by `MxM`) and Sam(oa)^2 [MRB16]. `MxM` is a micro-benchmark used to ease the reproducibility of almost all experiments. In `MxM`, a task is defined by a `MxM` compute kernel. The input arguments include matrix `A`, `B`, and matrix `C` is the output argument. Sam(oa)^2 is an adaptive mesh refinement framework used for oceanic HPC applications/simulations. The design of this framework is based on the numerical analysis of space-filling curves.

4.3.2.1. Example: `MxM` matrix multiplication

On the side of users, the input arguments of a task are matrices, and their sizes can impact the task execution time (w). Therefore, we configure the model inputs as matrix size, core frequency, and the model output as task execution time. The matrix size can be collected right after a task is created, and core frequency can also be quickly checked from the system call of operating system. Assuming that each process holds a number of T_i tasks (i denotes process P_i) and we have P processes in total; then, a dataset for training online load prediction can be formatted as the following.

Input			**Output**
IN_{app}	IN_{sys}		OUT
$m_0 = 256$	$\text{freq}_0 = 1185.6$	$\rightarrow$	w_0
$m_1 = 128$	$\text{freq}_1 = 800.1$	$\rightarrow$	w_1
$m_2 = 512$	$\text{freq}_2 = 900.9$	$\rightarrow$	w_2
$m_4 = 100$	$\text{freq}_4 = 1800.0$	$\rightarrow$	w_4
$m_3 = 256$	$\text{freq}_3 = 1361.8$	$\rightarrow$	w_3
...	...		...

With IN_{app}, m_j represents a matrix size corresponding to a task (task j) in a process. The notation j in this case means that suppose process P_i has a number of T_i tasks, then task j belongs to T_i (j simply points to a task index or ID). The value of m_j can be normalized as the number of elements or the volume in memory. Here, we use the number of elements, such as 256 for a matrix of 256×256 elements. Similarly, with IN_{sys}, freq_j represents the core frequency checked before a task is scheduled to execution, where j also points to a corresponding task. The value of freq_j is roughly available to characterize before executing a task. We cannot make sure 100% that freq_j can accurately reflect a task's load value. However, this system feature can relate to the runtime before and after a task is executed. Additionally, freq_j can represent how we combine application and system features to predict the load. In a process, we might have many tasks that explain the input layer indexed by 0, 1, 2, etc. With OUT, we show a corresponding execution time of a task, namely w_0, w_1, etc.

4.3.2.2. Example: Sam(oa)^2

Sam(oa)^2 is an adaptive mesh refinement framework developed for oceanic HPC applications. The use cases of Sam(oa)^2 include tsunami, earthquake, or environmental simulations. Therefore, we use

Sam(oa)^2 as a real iterative application for experiments in our work. In Sam(oa)^2, this framework utilizes a concept of grid sections where each section is processed by a single thread [MRB16]. A traversed section is an independent computation unit, which is defined as a task. Following the canonical approach of cutting the grid into parts of uniform load, tasks per process are uniform. A set of tasks belonging to different processes might not correspond to the same load. Sam(oa)^2 supports hybrid MPI+OpenMP programming models. Hence, we can port its application into a task-based parallel application, where each process assigned tasks refers to MPI ranks[1]. An MPI rank supports multiple OpenMP threads to execute tasks.

By characterizing Sam(oa)^2, we observe that predicting the w value of each task in a process is not very useful to estimate load imbalance. The tasks are uniform, and their w values (wallclock execution time) are alike in the same iteration. Notably, the total load values (L) per process are changed over iterations, and L can be predicted based on a chain of L values in the previous iterations. This example shows how we can adjust the influence features to predict the load values depending on application characterization.

For input data layer, we use the total load value of an iteration (L_i^I), where L_i^I denotes the load of process P_i in iteration I. The dimension of input features depends on how many complete iterations we want to specify. For instance, if we take 4 complete iterations and assume the current iteration as a prediction target is I, then the array of input features includes L_i^{I-4}, L_i^{I-3}, L_i^{I-2}, L_i^{I-1}, where L_i^I is the prediction output. To trace back the w values of each task from the predicted L_i^I, the estimation can be based on the average of w through L and the number of assigned tasks per process because tasks are known with uniform load following the characterization of Sam(oa)^2.

The following samples show a layout of the dataset with two processes, process P_0 and process P_1. We keep the L values of 4 complete iterations to generate a sample dataset.

Dataset on process P_0

Input					**Output**
IN_{app}					OUT
$L_0^0 = 0.029$	$L_0^1 = 0.062$	$L_0^2 = 0.040$	$L_0^3 = 0.070$	$\rightarrow$	$L_0^4 = 0.035$
$L_0^1 = 0.062$	$L_0^2 = 0.035$	$L_0^3 = 0.070$	$L_0^4 = 0.035$	$\rightarrow$	$L_0^5 = 0.062$
$L_0^2 = 0.035$	$L_0^3 = 0.070$	$L_0^4 = 0.035$	$L_0^5 = 0.062$	$\rightarrow$	$L_0^6 = 0.035$
$L_0^3 = 0.070$	$L_0^4 = 0.035$	$L_0^5 = 0.062$	$L_0^6 = 0.035$	$\rightarrow$	$L_0^7 = 0.070$
$L_0^4 = 0.035$	$L_0^5 = 0.062$	$L_0^6 = 0.035$	$L_0^7 = 0.070$	$\rightarrow$	$L_0^8 = 0.035$
$L_0^5 = 0.062$	$L_0^6 = 0.035$	$L_0^7 = 0.070$	$L_0^8 = 0.035$	$\rightarrow$	$L_0^9 = 0.062$
...	...	...	...		...

[1] "Rank" and "process" are used interchangeably in the following sections.

Dataset on process P_1

Input					**Output**
IN_{app}					OUT
$L_1^0 = 0.030$	$L_1^1 = 0.072$	$L_1^2 = 0.036$	$L_1^3 = 0.062$	$\rightarrow$	$L_1^4 = 0.036$
$L_1^1 = 0.072$	$L_1^2 = 0.036$	$L_1^3 = 0.062$	$L_1^4 = 0.036$	$\rightarrow$	$L_1^5 = 0.061$
$L_1^2 = 0.036$	$L_1^3 = 0.062$	$L_1^4 = 0.036$	$L_1^5 = 0.061$	$\rightarrow$	$L_1^6 = 0.036$
$L_1^3 = 0.062$	$L_1^4 = 0.036$	$L_1^5 = 0.061$	$L_1^6 = 0.036$	$\rightarrow$	$L_1^7 = 0.070$
$L_1^4 = 0.036$	$L_1^5 = 0.061$	$L_1^6 = 0.036$	$L_1^7 = 0.070$	$\rightarrow$	$L_1^8 = 0.035$
$L_1^5 = 0.061$	$L_1^6 = 0.036$	$L_1^7 = 0.070$	$L_1^8 = 0.035$	$\rightarrow$	$L_1^9 = 0.064$
...	...	...	...		...

In process P_0, the dataset is normalized until iteration 9. We do not need to use the profiled features from the system side in this example, such as core frequency, because the total load value in an iteration can be predicted based on the previous iterations. We exploit four previous iterations; therefore, there are four parameters in the input layer, i.e., the L values of iteration 0, 1, 2, 3. These are denoted by L_0, L_1, L_2, L_3, representing IN_{app}. The output layer, OUT, points to the L value of iteration 4 shown as the first row in the dataset. Similarly, proceeding further until iteration 9, we have a dataset including 6 rows of input and output features. Likewise, we have the same dataset layout with process P_1.

After training, each process has its own prediction model. Before the next iterations begin, these models are loaded in advance at $Tcomm$ and predict the total load values.

To guide proactive task offloading, prediction results on each process need to be exchanged. After that, each can adapt these predicted values to a proactive task offloading algorithm introduced in the next subsection.

For an overview, we summarize the input and output parameters of both examples, `MxM` and Sam(oa)^2 in Table 4.1. The overview emphasizes that our load prediction model relies on the influence features we can obtain to configure prediction models. Therefore, different applications might need different configurations before running.

Regarding what is prepared to train an online prediction model, Figure 4.5 shows the flow from input to output. In the middle, we highlight possible machine learning models ($MODEL$) that can be applied to predict the target. These are machine learning regression algorithms [Jam+13], e.g., Linear, Ridge, Bayesian, LARS regression, etc. Figure 4.5 (A) shows the models for `MxM`, and (B) for Sam(oa)^2.

Technically, these algorithms are lightweight to implement. With the requirements of load prediction at runtime, they are simple to customize. Besides that, the main point of load prediction for balancing problems is not how a predicted value of L or w is highly accurate, while we mainly focus on how the imbalance between processes along with an acceptable accuracy.

Table 4.1.: The input-output features for training a load prediction model in `MxM` and Sam(oa)^2.

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

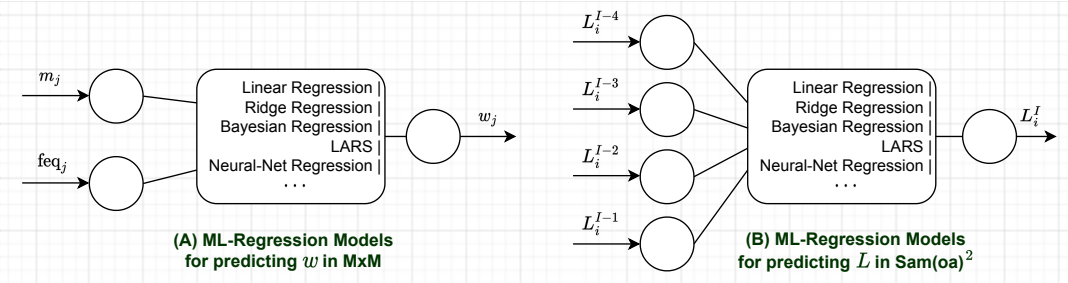


Figure 4.5.: Machine learning regression models and algorithms that can be applied in `MxM` and Sam(oa)^2.

4.3.3. Proactive Task Offloading Algorithm

After loading the prediction, the prediction output is transferred to the algorithm, which is used to guide task offloading. We call it "proactive task offloading" algorithm. The prediction results are simplified for the algorithm's input. In general, the input and output include:

- `Input`: includes array `L` and array `N`. Array `L` is the predicted load values of processes, which can be specified as [L_0, ..., L_{P-1}]. Array `N` is the number of assigned tasks on each process, depending on a distribution of tasks before execution. The length of array `L` and `N` is the number of involved processes, P. Each process has array `L` after it exchanges the predicted load value (L_i) with the others. Following that, we can calculate imbalance ratio (R_{imb}).
- `Output`: is the reference number of how many tasks should be offloaded on each process. Obviously, if a process is underloaded, its reference number of offloaded tasks is zero. If this reference number is > 0, the corresponding process should offload tasks proactively.

Algorithm 2 shows the proactive task offloading algorithm. As mentioned above, array `L` and array `N` contain the predicted load values of processes[2] and the given number of assigned tasks per process. The length of each array is `P`, where `P` is the number of involved processes. For the output, this algorithm generates a result named `TB`, which can be addressed as a 2D array (a table, or a matrix) to record the following values.

- The number of remaining tasks in original processes.
- The number of offloaded tasks from the original process to others.
- The indices of each element in `TB` indicate which process should offload tasks to which process.

Step-by-step, we create a new array $\hat{\text{L}}$ at Line 1 to store the sorted load values of array `L`, and calculate the average load value (L_{avg}) based on $\sum_{i=0}^{\text{P}-1} \frac{\text{L}[i]}{\text{P}}$ at Line 2. The average is considered an optimal balanced value. To track the amount of load that receives from the offloaded tasks, Algorithm 2 allocates array `R`. We call this amount of load remote load values, and each element of array `R` is the remote load value of a corresponding process.

To track the number of tasks, array `TB` records the number of local tasks (remaining tasks in an original process) and remote tasks (offloaded tasks to other processes if yes). We use `TB` as a tracking or recording table that we can technically implement it as a matrix. The size of `TB` is `P`×`P`. However, we just need the upper or lower triangle part, where its diagonal represents the number of local tasks, and the others indicate the number of offloaded tasks. For example, if the value of `TB[i,j]` > 0 ($i \neq j$), process P_i should offload a number of `TB[i,j]` tasks to process P_j.

In detail, Algorithm 2 is described by two main loops. The outer loop processes each victim. Note that "victim" here indicates the process which receives offloaded tasks from the others, while "offloader" indicates the process which migrates tasks. Compared to the average load value, a victim is

[2] "Process" and "rank" might be used interchangeably to describe the text in Algorithm 2

Algorithm 2: Proactive Task Offloading

Input : Array `L`, `N`, where each element `L[i]`, `N[i]` indicates the predicted load value and the number of assigned tasks on process P_i.

Output: Table `TB` is a 2D array, which can be represented as a matrix. The size of `TB` is `P`×`P`. Each element indicates the number of tasks that a process should offload.

```
1  New Array L̂ ← Sort L by the load values
2  L_avg ← Σ_{i=0}^{P-1} L[i]/P  // Estimate the average load value
3  New Array R; TB  // R has P elements denoting the total load of remote tasks per
     rank, TB has P×P elements as mentioned
4  for i ← 0 to P-1 do
5      if L̂[i] < L_avg then
6          Δ_under ← L_avg − L̂[i]  // Calculate the load value under average
7          for j ← P-1 to 0 do
8              if L̂[j] > L_avg then
9                  Δ_over ← L̂[j] − L_avg  // Calculate the load value over average
10                 ŵ ← Estimate the load per task and ASSERT (Δ_over ≥ ŵ)
11                 if Δ_over ≥ Δ_under then
12                     N_off, L_off ← Estimate the number of tasks to offload and the amount of load
                         for these remote tasks by ŵ, Δ_under
                   else
13                     N_off, L_off ← Estimate the number of tasks to offload and the amount of load
                         for these remote tasks by ŵ, Δ_over
14                 Update Δ_under, L̂ at the index i and j based on N_off, L_off
15                 Update N[j], R[j]; TB at the index (i, j), (j, i), (j, j)
16                 Break if ABS (Δ_under, L_avg) < ŵ
17 return TB
```

underloaded ($\hat{\mathtt{L}}[i] < \mathtt{L}_{avg}$), while an offloader is overloaded ($\hat{\mathtt{L}}[i] > \mathtt{L}_{avg}$). Typically, the underloaded value between process P_i and $\mathtt{L}_{avg}$ is calculated at Line 6, named Δ_{under}. This implies that process P_i needs to be filled with a load of Δ_{under} to balance the load. The inner loop goes backward for each offloader. The overloaded load (Δ_{over}) between process P_j and $\mathtt{L}_{avg}$ is then calculated at Line 9.

To compute the number of tasks for offloading, we need to know the load value w of each task. With an online load prediction model, if our prediction target is the load w, then the value w of each task is obviously available. Otherwise, if our prediction target is the total load L of each process, then the value w can be estimated as an average between L and the given number of assigned tasks per process. Therefore, we call $\hat{w}$ an estimated load value of each task (Line 10 in Algorithm 2). This depends on application characteristics to estimate w and also depends on user configuration before execution.

Afterward, the number of offloaded tasks ($\mathtt{N}_{\mathtt{off}}$) and the total offloaded load ($\mathtt{L}_{\mathtt{off}}$) are calculated. We cannot fix the calculation of $\mathtt{N}_{\mathtt{off}}$. Instead, our algorithm shows that $\mathtt{N}_{\mathtt{off}}$ offers several possibilities to calculate. This should be tuned in a proper way because we know load information at runtime

relying on the load prediction model. Basically, we can divide Δ_{under} by the task execution time, w, to estimate $\mathtt{N_{off}}$, and simultaneously multiply it with a coefficiency. The coefficiency implies that there are some scenarios that we have to consider for tuning the calculation of $\mathtt{N_{off}}$ and $\mathtt{L_{off}}$. For example:

- Scenario 1: the imbalance is caused by performance slowdown in some processes, where the number of assigned tasks on each process is the same, while the load values w of tasks differ, depending on where the tasks are executed. Following that, the values of $\mathtt{N_{off}}$ and $\mathtt{L_{off}}$ after calculation need to be adjusted, e.g., 2%, 5%, or 7% more because the tasks on the original process moving to the remote process can be performed faster.
- Scenario 2: the execution speed of all processes is stable. The difference in task distribution causes imbalance. Then, the value of $\mathtt{N_{off}}$ can be calculated by Δ_{under} and the predicted value w.

In further steps of the algorithm, the values of Δ_{under}, $\hat{\mathtt{L}}$, `N`, `R`, `TB` will be updated at the corresponding indices. At line 16, the absolute values between Δ_{under} and $\mathtt{L}_{avg}$ are 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 for another candidate (another offloader). In terms of complexity, if we have `P` processes in total, where `Q` is the number of victims, `P-Q` will be the number of offloaders; then the algorithm takes O`(Q(P-Q))` steps. Our implementation is published and described in detail on a GitHub repository, referring to this link at footnote[3].

4.3.3.1. Example: `MxM` matrix multiplication

To demonstrate the algorithm, we use the example of the `MxM` matrix multiplication again. For reproducibility purposes, we artificially generate the imbalance of `MxM` by distributing different numbers of tasks on different processes. The execution speed of all processes is kept stable, and all tasks have uniform load (similar load). The level of imbalance here can be easily varied by the number of tasks per process, intending to create a specific imbalance scenario. Particularly, we assume having 8 MPI processes. The example in this sub-section is shown with an imbalance ratio of ≈ 4.0. We distribute the number of tasks on each rank before execution, such as $T_0 = 800$, $T_1 = 100$, $T_2 = 50$, $T_3 = 50$, $T_4 = 50$, $T_5 = 50$, $T_6 = 90$, $T_7 = 90$. A task is defined by a compute kernel of `MxM`, and the program execution is set iteratively by 5 iterations in total. The first iteration is left to predict the load values. In the following steps, we assume that the prediction results per process are already obtained and available to input the algorithm.

Figure 4.6 illustrates the initialization step (denoted by `(1)`), first loop (denoted by `(2)`) and last loop (denoted by `End algorithm`) of Algorithm 2 to visualize how we can calculate the recommended number of tasks for offloading. From left to right are three columns of the illustrated arrays, each representing a working step.

The initialization step `(1)` is shown in the first column. From here, the input is shown with array `L` and array `N`. Array `L` contains the local load values of each process, which are predicted and exchanged from the prediction module. Array `N` contains the given numbers of assigned tasks on each process. Here, process P_0 is the most overloaded process with a total execution time of 18.4, and process P_1 occupies the second position. Corresponding to the number of assigned tasks, process P_0 has the largest number of tasks (800), and its load stands the biggest value. In the first stage, array `L` is sorted by the load values and named $\hat{\mathtt{L}}$. With $\hat{\mathtt{L}}$, we create another array `R` to keep tracking the remote load when tasks are migrated around from one process to another. Their local load values will be changed alongside their remote load values. Table `TB` tracks the number of local and remote tasks, where the diagonal line accounts for the local tasks. The upper triangle part of `TB` (cells are shaded grey) accounts for the number of remote tasks that should be offloaded. This example shows one of the extreme imbalance cases, where we have one overloaded process, and it has to migrate tasks to other processes for balancing.

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

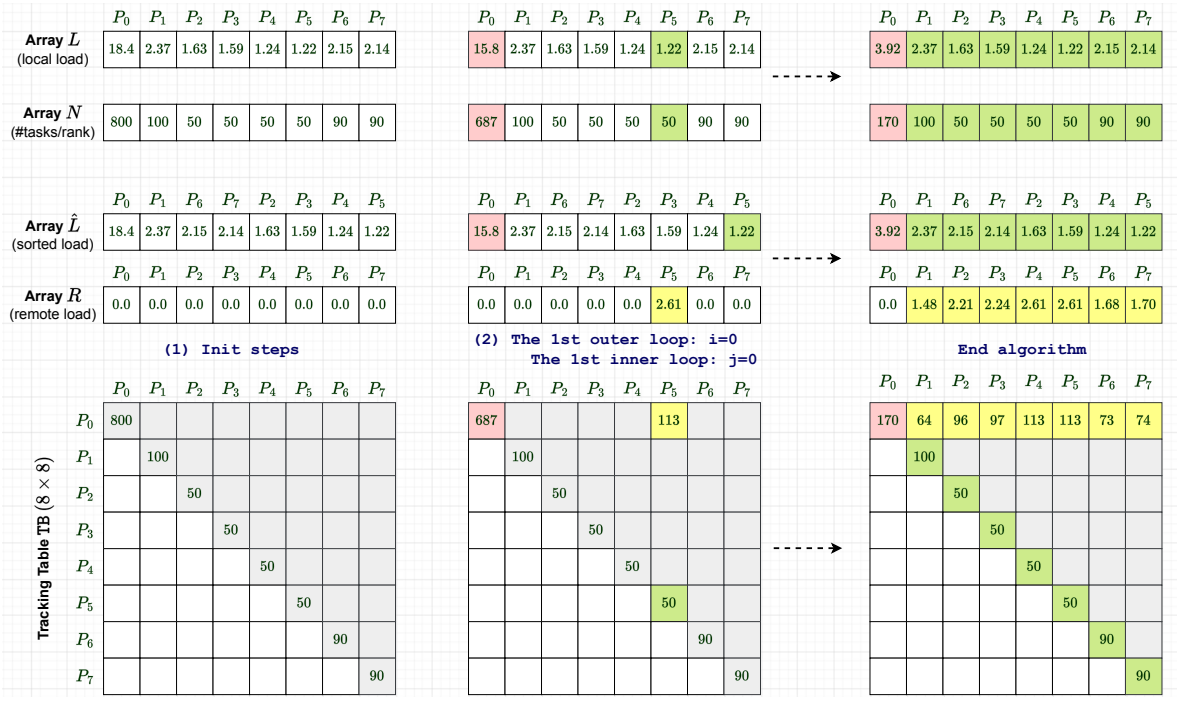


Figure 4.6.: An example of applying the proactive task offloading algorithm on `MxM` matrix multiplication.

In the second step `(2)`, the outer loop of Algorithm 2 is processed. We traverse the first victim, process P_5. At this time, the P_5's predicted load is 1.22. Compared to the average load (≈ 3.84), process P_5 needs around a load of 2.62 to be balanced. Then, we go to the inner loop, traversing the first offloader, which has the load value $L > \mathtt{L}_{avg}$. Obviously, process P_0 is an appropriate offloader, so we can calculate how much overloaded it holds compared to the average load. At this time, we estimate that process P_0 has enough tasks to offload to process P_5, and a load of ≈ 2.62 is equivalent to 113 tasks from process P_0. Afterward, the arrays such as `L`, `N`, `R`, and `TB` are updated. As Figure 4.6 shows, `TB` is updated at the indices $[0, 0]$, $[5, 5]$, and $[0, 5]$, where the element `TB`$[0, 5]$ highlights the number of tasks that we should offload from process P_0 to P_5.

Ultimately, in the last step (`End algorithm`), we see a fully updated `TB`. This case has only process P_0 changing local tasks and local load, while the others only receive remote tasks. The results suggest offloading tasks earlier at the beginning of the upcoming iteration. For instance, process P_0 should offload 64 tasks to P_1, 96 to P_2, and so on. The ML-based task offloading method shows that we can proactively offload tasks early with a guided plan. When we have prediction results, Algorithm 2 is applied the same step-by-step. Therefore, we do not need to show another example of how this algorithm works for Sam(oa)^2.

4.3.3.2. Task Migration Strategies

Following the output of Algorithm 2, we can refer to how many tasks should be offloaded as well as migrated from which process to which potential victim proactively. While "reactive" is based on the most current status to react immediately to migrating tasks, we can anticipate a relative plan for migrating tasks with respect to "proactive". Based on the result of the example shown in Figure 4.6, we know that process P_0 should offload 64 tasks to process P_1, 96 to process P_2 at the beginning when a new iteration starts executing tasks. Following this point, the questions are: Which process should migrate the tasks first? And how should we migrate the tasks according to a reference number such as 64, 96? Therefore, with the ML-based task offloading method, we can also define a better task migration strategy.

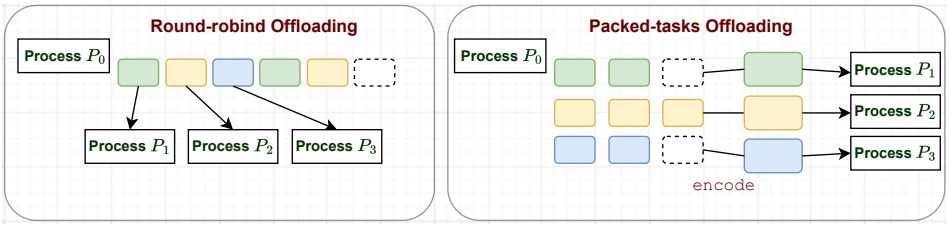


Figure 4.7.: Two related strategies for task migration.

In Figure 4.7, we propose two task migration strategies in the context of ML-based task offloading. These two migration strategies can be useful if one process sends tasks to multiple victims. Specifically, we call them:

- round-robin task offloading
- packed-tasks offloading

Round-robin sends task by task; for example, we know that process P_0 needs to offload 64 tasks to process P_1, 96 tasks to process P_2, etc. Then, the round-robin strategy will take the 1^{st} task to P_1, the 2^{nd} task to P_2, and repeat the progress until all tasks are sent.

In contrast, packed-tasks offloading encodes all 64 tasks, which are planned to be sent to process P_1, as a package and sends the package at once. After that, we continue to encode 96 tasks, send the whole package to process P_2 at once, and so on.

4.4. Extension: Co-scheduling Tasks across Multiple Applications

This section introduces an extension based on our proactive load balancing approach, "co-scheduling tasks across multiple applications," which is considered a method for balancing the load not only within an application but multiple applications. *Tcomm* is still dedicated to handling task migration, but the scope is extended to multiple applications running simultaneously. The principle is mainly migrating tasks from one process to another, but we perform on different applications. Therefore, this method refers to be called "co-scheduling" tasks.

Tcomm plays the role of a communication channel, where one process in a program[4] can share information with another process running in another program. Following that, the imbalance in a single application can now share the load not only among its processes, but also the other involved applications. There are three main stages:

1. Task characterization and load prediction: *Tcomm* is also deployed to characterize tasks and predict their load values. This stage provides a knowledge to estimate how many tasks we can migrate.
2. Idle slot selection: We can detect idle slots in the application based on task characteristics and load prediction. This phase helps estimate how long a task will take and how much idle time can be suitable for task migration.
3. Prediction exchange and task migration: *Tcomm* exchanges the prediction information between processes among different applications. If there is an acceptance for sharing tasks, we migrate several tasks from a busy process of the current program to a process with idle slots of the other program.

To enable these stages for co-scheduling tasks, (1) tasks need to be migratable among processes on both sides of involved applications. This implies that different applications need to be enabled for sending or receiving tasks in terms of configuration in advance. If one application meets idle slots, it

[4] A program implies that an application is launched at runtime.

is feasible to migrate tasks across applications. (2) Our method might not be relevant for task-based applications that feature dynamic task creation because we do not know how many tasks are created before execution.

We describe the method and its steps in Algorithm 3. The input is an array of process indices of involved applications [P_{10}, P_{11}, ..., P_{ij}], where i indicates the index of application, j indicates the index of process belonging to that application. The expected outputs include an idle-time array of processes sorted by the idle values ($\texttt{IDLE}^{'}$), process victims ($P_{\texttt{victim}}$) for migrating tasks, and the estimated number of tasks for migrating at once ($\texttt{num}_{\texttt{offload}}$). We clarify the method with two procedures in Algorithm 3:

- `estimate_IDLE()`
- `proact_coordinate_TASK()`

First, the procedure `estimate_IDLE()` indicates the stage of online load prediction and idle-time estimation. We assume the prediction result is already available at this function. The data collection techniques and training models are similar to the procedure mentioned in Section 4.3. *Tcomm* is triggered to train a prediction model asynchronously while other threads are executing tasks. From here, we just need to load the trained model. Therefore, the procedure shows the predicted load values of the current process (*pid*) at Line 2. Afterward, we share this value with other processes in the original and other applications to estimate idle time. This mainly intends for different types of tasks. For example, we distinguish computation tasks and communication/IO tasks, where communication/IO tasks are just waiting for communication procedures. Then, we can fill the waiting gaps with tasks from other processes or from other applications. Following the exchange of predicted values, each process will have a list of predicted loads of other processes in its application. We can calculate the average value of idle time based on the maximum predicted load value, resulting in an array `IDLE`. After sorting `IDLE`, we have a new array, $\texttt{IDLE}^{'}$. The purpose of $\texttt{IDLE}^{'}$ is to conduct a general estimation of idle time per process, enabling estimation of how many tasks should be shared if possible. The values of $\texttt{IDLE}^{'}$ are the input for the next procedure.

Second, the procedure `proact_coordinate_TASK()` denotes how tasks are migrated to co-scheduling among the involved processes across different applications. There are two sides to the corresponding operations: one side of processes having idle slots and one side of processes being busy. At Line 9, 10, when an idle slot is detected, this information is distributed by the function `distribute()`. Otherwise, a process checks for receiving idle status from the others. If the idle status is received, a busy process can request offloading tasks (ranging from Line 11 to 17). Following that, the busy process waits for a confirmation to proceed with further task offloading if the number of expected tasks and idle slot is matched (indicated by Line 18, 19).

More intuitively, Figure 4.8 demonstrates the operations of Algorithm 3. From top to bottom, we can loosely see three blocks denoted by **Applications**, **Task Characterization** & **Load Prediction**, and **Runtime** & **Coscheduling Tasks**. The scenario in this figure also illustrates the first use case of our co-scheduling method. In detail, each block is described as follows.

- **Applications**: Two task-based applications are shown in the row of **Applications**, `App.1` and `App.2`. We assume that their tasks include compute tasks and IO/communication tasks. IO/communication tasks are considered idle slots because they are not compute-intensive but waiting for IO/communication. The CPU cores processing these tasks are idle for a period and can be switched to other compute tasks.
- **Task Characterization** & **Load Prediction**: We address the stage of profiling tasks and predicting their load values. As illustrated, we give two processes per application; each process spawns multiple threads for executing tasks. Their IDs are indexed by the notation P_{ij}, where i and j indicate the index of application and process as mentioned above, e.g., P_{10} indicates process P_0 belonging to application 1. Suppose the prediction's results are ready; we have the predicted values of total load ($L_{ij}^{'}$), execution time per task ($w_{ij}^{'}$), and the idle time ($\texttt{IDLE}_{ij}$). For instance, application 1 is shown with $L_{10}^{'}$, $L_{11}^{'}$, $w_{10}^{'}$, $w_{11}^{'}$, $\texttt{IDLE}_{10}$, and $\texttt{IDLE}_{11}$.

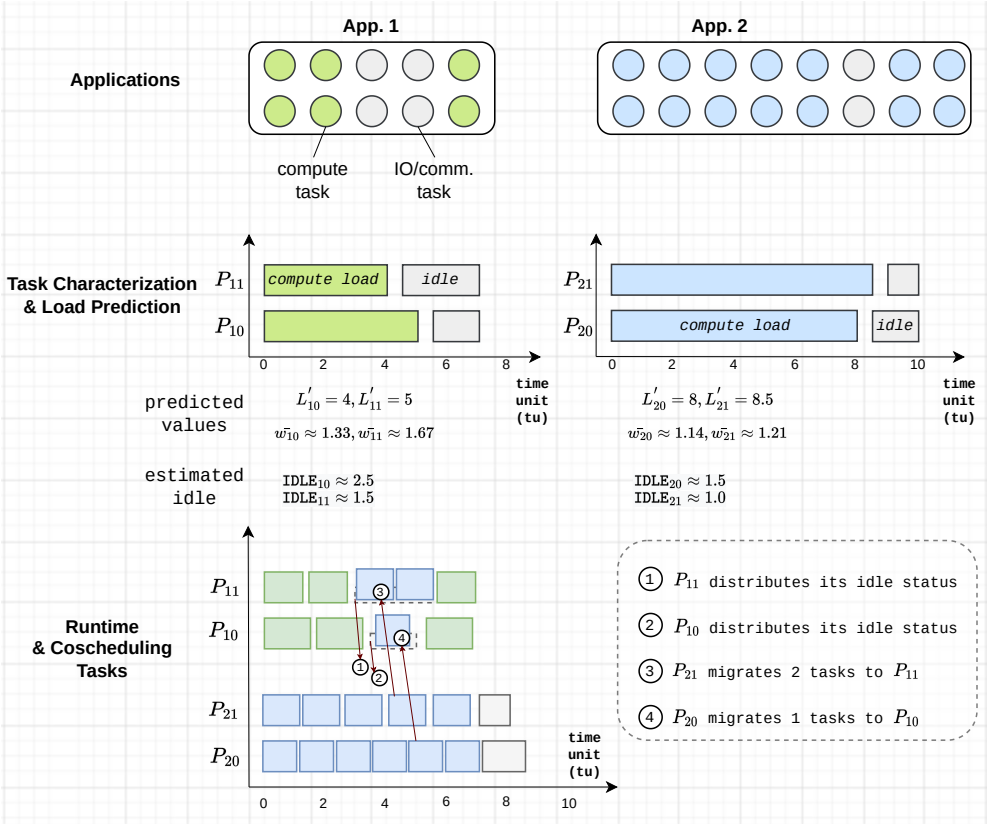


Figure 4.8.: Co-scheduling strategy and the first use between two CPU applications.

- **Runtime** & **Coscheduling Tasks**: We show the illustrated events corresponding to the steps of procedure `proact_coordinate_TASK()` in Algorithm 3. In this use case, `App.1` has idle slots, `App.2` is busier with more compute tasks. At the operations ①, ②, processes P_{11} and P_{10} of `App.1` distribute information about their idle status. In principle, after shaking-hand steps to reach an acceptance, process P_{21} and P_{20} of `App.2` migrates tasks to `App.1` to take advantage of the idle slots of `App.1` (shown as the operations ③, ④).

The operations ①, ②, ③, ④ are detailed in Figure 4.9. We have hand-shaking steps before tasks are offloaded. To ease the coordination between the processes of two applications, we distribute idle status associated with its estimated value, bandwidth information (B), and the current status of the queue. Bandwidth B can be measured in advance and averaged gradually from the information exchange of previous steps. We attach it to calculate transmission time before offloading tasks. Furthermore, we need information about the current queue length to check how many remaining tasks there are and the queue lengths between the idle and busy processes. For example, P_{11} distributes a message of $\texttt{IDLE}_{11}$, B_{11}, Q_{11} at operation ①. On the side receiving idle status, the arrived information is checked first with the difference in queue length to ensure that there are still available tasks for migration. Then, we calculate a suitable number of tasks that should be offloaded from the busy process based on `IDLE`. The calculated amount of tasks should not exceed the capability of average bandwidth B through a ratio, $\frac{\texttt{IDLE} \times B}{\texttt{data\_size}/task}$. Thereby, we can calculate how many tasks should fit the gap. For example, P_{21} sends a request to match with P_{11} at 1.1 after calculating the number of available tasks. If P_{11} agrees, it sends a confirmation message at 1.2, and tasks are offloaded afterward. The selected victim for co-scheduling tasks during an idle period is ranked and prioritized from the sorted array of $\texttt{IDLE}'$.

Several realistic HPC applications can represent the use case shown in Figure 4.8. They have become more popular with large-scale simulations, such as environment issue simulations. More importantly, these applications must run with various scenarios and long-term tuning of the parameters

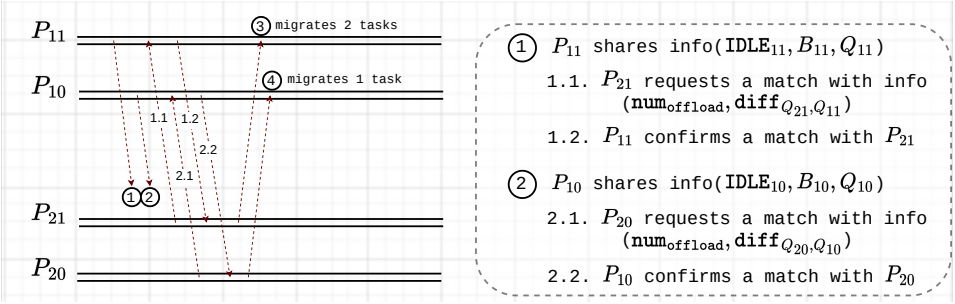


Figure 4.9.: A co-scheduling protocol for exchanging tasks with idle slots.

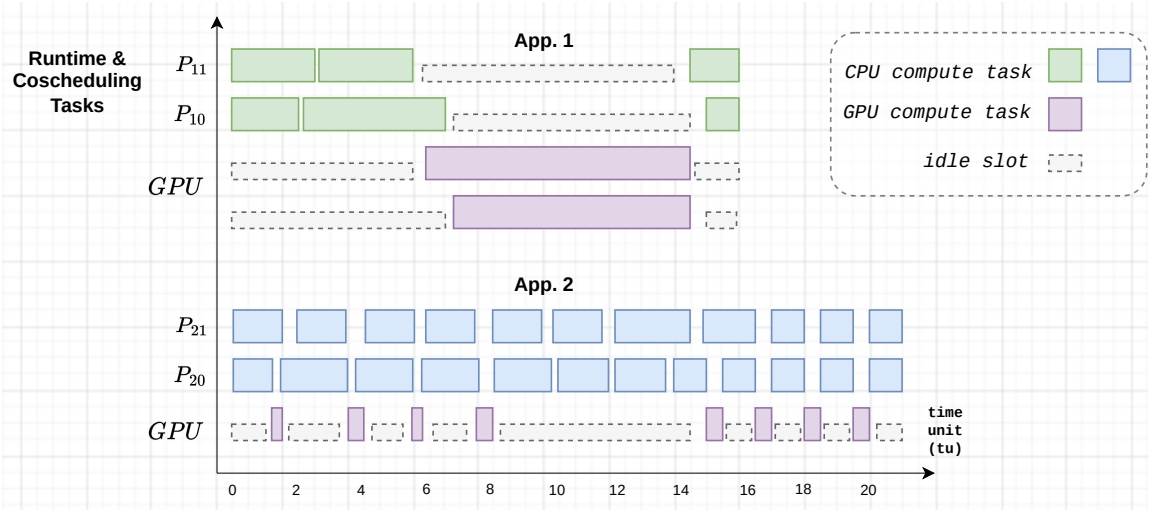


Figure 4.10.: The second use case of proactive coscheduling tasks.

for an optimal performance setup in HPC clusters. Hence, the idea of co-scheduling tasks in this method can be beneficial when one program has available idle slots, which are long enough to be replaced by tasks from another program. Our method can make both programs efficient and achieve better computing resource utilization.

For today's aspects with ML/DL applications, we illustrate another use case in Figure 4.10. The x-axis shows time progress, while the y-axis highlights each application's processes and GPU region. The top is application 1 (`App.1`), and the bottom is application 2 (`App.2`). Vertically, the process labels imply tasks executing on the CPU side (`CPU compute task`), while the GPU labels imply tasks executing on the GPU side (`GPU compute task`), and the blocks with dashed borders indicate idle slots. The main idea for applying our method to this use case includes:

- An application has two types of tasks, `CPU compute task` and `GPU compute task`. Assuming when GPU tasks on an application are running, the CPU side is idle.
- When the CPU side of an application is idle, CPU tasks from another application can be migrated to fill the idle slots.

This use case becomes more familiar with heterogeneous architectures, i.e., CPU-GPU. In Figure 4.10, we assume a distributed memory system of CPU-GPU nodes. We can share tasks during idle slots if tasks are well-defined.

In the following chapters, we will show the implementation of our two methods and this extension. Significantly, with this extension for co-scheduling tasks, we cannot interfere with the execution of jobs in HPC clusters due to permission. Therefore, the experiments are emulated by defining different tasks in an executable, representing multiple applications that run simultaneously. Different tasks

from different applications are in the same executable. For example, there is a whole job executing on 4 compute nodes, where two nodes will host the first application with its tasks, and the remaining nodes will host the second application. In such a submitted job, the two applications run simultaneously, enabling MPI to provide the same communication environment between them.

Algorithm 3: Proactive Co-scheduling Tasks

Input : Array of involved processes, P elements in range [P_{10}, P_{11}, ..., P_{ij}], where i indicates application index, j is the process index in that corresponding application.

Output: $\texttt{IDLE}^{'}$: Array of sorted idle time corresponding to the process indices;
$P_{\texttt{victim}}$: Process victims for offloading tasks;
$\texttt{num}_{\texttt{offload}}$: The number of tasks for offloading at a point in time.

Procedure `estimate_IDLE(`*Array* $P[]$`):`

```
    /* Get load prediction */
1   pid ← check process id
2   L'_pid ← get load prediction
    /* Exchange load prediction */
3   Array L' ← get predicted load values of other processes
4   L'_max ← maximum load
5   Array IDLE ← estimating idle slots based on L'_max
6   IDLE' ← sorting the array IDLE
    Return: IDLE'
```

Procedure `proact_coordinate_TASK(`*Array IDLE*$^{'}$`):`

```
    /* Check runtime information */
7   pid ← check process id
8   B ← check average bandwidth information for task migration
    /* At the side of idle processes */
9   if check(idle) then
10      distribute(IDLE_pid, B_pid, Q_pid) // share idle info around
    else
        /* At the side of busy processes */
11      if receive(idle) then
12          Assign pid_idle, pid_busy
13          assert(diff_{Q_pid} > α) // α is a constant limiting the minimum number of
             remaining tasks that we can make co-scheduling with other processes
14          max_offload ← IDLE_{pid_idle} × B̄ / data_size/task
15          num_offload ← IDLE_{pid_idle} / w'_{pid_busy}
16          assert(num_offload < max_offload and Q_{pid_busy})
17          send(request[num_offload, Q_{pid_busy}])
18      if receive(confirm) then
            // offload tasks afterward if we receive an acceptance
19          offload(num_offload)
    Return: P_victim, num_offload
```

5. Proof of Concepts

This chapter shows the implementation of our proactive load balancing approach. The implementation is built upon a task-based parallel framework named Chameleon [Kli+20a], which supports task-based applications running on shared and distributed memory. In our proactive load balancing approach, integrating an online load prediction model cannot be fixed in a specific application. Besides that, proactive task offloading methods are intended to be customized on the user side. Therefore, we design the implementation as a plugin tool upon Chameleon. Users can be flexible to adjust prediction model and task migration strategy before execution, even load balancing algorithm. This chapter is outlined as the following:

1. We describe the Chameleon framework in Section 5.1.
2. We present our implementation designed as a plugin tool upon Chameleon in Section 5.2.
3. We show an example of putting Chameleon and the plugin tool together in Section 5.3.

5.1. Chameleon Framework

Chameleon is implemented in C++ based on hybrid MPI+OpenMP. In an MPI process, we can deploy multiple OpenMP threads for executing tasks and a dedicated thread (called *Tcomm*) for overlapping communication & computation. *Tcomm* is deployed using POSIX thread (`pthread`). Task-based parallel applications with Chameleon can run efficiently on shared and distributed memory systems. We mainly use *Tcomm* to deploy our proactive load balancing approach, where our implementation can be called a proactive load balancing tool (or proactive LB tool for short).

Note that in Chameleon itself, this framework already provides the reactive load balancing approach. As explained in Chapter 2, Section 2.4, reactive load balancing follows a push-oriented mechanism, where tasks are offloaded from overloaded to underloaded processes. In other words, we can say that tasks are offloaded from a slow to a fast process. This is different from a pull-oriented mechanism in work stealing.

The Chameleon framework is described through its task system and how tasks are managed, as the following subsections:

1. Migratable task definition: explains the structure of tasks defined by task properties and relevant methods in terms of implementation. For task migration at runtime, tasks must be migratable. Therefore, this subsection will show the definition of a migratable task in Chameleon.
2. Task execution and communication thread: explains the structure and implementation of the task manager in Chameleon. This implies the mechanism by which tasks are scheduled and mapped to resources for execution. Importantly, this subsection introduces an overview of the dedicated thread for communication and task migration.

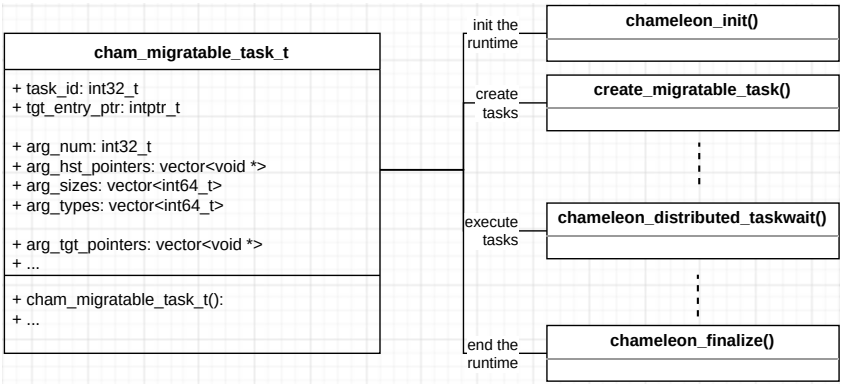


Figure 5.1.: A combined communication diagram between the task class and main APIs of the Chameleon runtime.

5.1.1. Migratable task definition

The implementation of Chameleon is a custom `libomptarget` plugin in LLVM OpenMP runtime [LLV13]. Hence, Chameleon enables declaring a task using directives, such as `#pragma omp target`. For the data environment of tasks, we can use the clause `map` to specify. When a task is created, the compiler will manage appropriate calls, referring to the task entry function (code) and its data environment (data). Properties in implementation represent the code and data in a task.

Figure 5.1 shows a definition of class `cham_migratable_task_t` associated with the task properties, defining a task in Chameleon. This class is shown on the left side, detailing some essential attributes and a constructor function (named `cham_migratable_task_t()`). On the right side, we illustrate the link between the initialization of a task and the main API functions in Chameleon.

Each task has a unique ID (`task_id`, type `int32_t`). When a task is migrated to another process, its ID stays the same for simply distinguishing between local and remote tasks. The following are some other important properties:

- `arg_num`: indicates the number of input arguments belonging to a task.
- `arg_hst_pointers`: indicates a vector of pointers that refers to the arguments of a task. Following this, there are related properties, such as argument sizes (`arg_sizes`), argument types (`arg_types`).
- `arg_tgt_pointers`: points to the remote side or remote process where a task will be executed if it is migrated.

When a task is offloaded onto a remote side, the same hybrid binary of the task is executed. Besides that, we address each task by an offset from the start of the loaded library to the corresponding task entry function. It is used to determine the correct entry point on a remote MPI process.

The class `cham_migratable_task_t` allows us to pack and migrate tasks from one process to another. In practice, tasks can be generated by `create_migratable_task()`. The working flow of a task-based application written by Chameleon follows the main API functions shown in Figure 5.1, including:

- `chameleon_init()`: initializes Chameleon.
- `create_migratable_task()`: creates migratable tasks. Users specify tasks and the number of tasks.
- `chameleon_distributed_taskwait()`: distributes and executes tasks. The scale of computing resources, such as the number of compute nodes, processes, and threads per process, is configured by users.

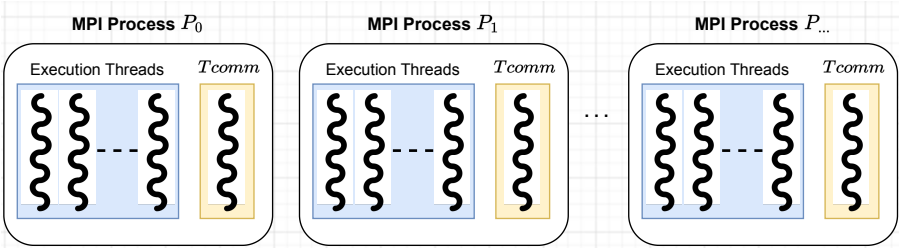


Figure 5.2.: Chameleon runtime with the dedicated thread for overlapping communication and computation.

- `chameleon_finalize()`: finalizes the execution by Chameleon.

Behind these functions, tasks are managed to run in parallel (even on shared or distributed memory systems). From the side of users, we can consider applications as black boxes. Chameleon takes control of task execution. After that, the application flow is given back to users. The next subsection will explain the execution of tasks in Chameleon.

5.1.2. Task execution and communication thread

Like other task-based programming models, task execution in Chameleon is hierarchical. After tasks are created and distributed across processes, each will manage the execution of assigned tasks through its threads. Chameleon distinguishes OpenMP threads for executing tasks (called execution threads or worker threads) and a dedicated POSIX thread for communication between processes (named $Tcomm$). When imbalance occurs at runtime, the dedicated thread can migrate tasks to balance the load.

To ensure the execution of all local and remote tasks are synchronized, Chameleon requires a global synchronization barrier. Therefore, all communication between processes has to reach the synchronization barrier. In a process, all execution threads are triggered when `chameleon_distributed_taskwait()` is invoked. The dedicated thread is triggered when reactive load balancing is enabled. After tasks are assigned to each process, they are queued before scheduling to execution. There are two types of task queues: one queue of local tasks and one queue of remote tasks. The framework prioritizes the execution of remote tasks before working on local tasks because the original process might request the remote side to send back the results of their tasks.

Regarding the progress of execution threads and $Tcomm$, they are desbribed as follows:

- The execution threads are created by OpenMP runtime, while $Tcomm$ is created by a POSIX thread (*pthread*). When $Tcomm$ is enabled, it monitors load status and exchanges the status continuously. The load status accounts for execution speed. In particular, Chameleon uses the queue length that indicats the number of remaining tasks. This determines which process is slow, which is fast, and imbalance ratio to make decisions for reactive load balancing.
- When an imbalance is detected, the operations performed on $Tcomm$ include:
  - Select a victim to migrate tasks (offload tasks).
  - Decide the number of tasks for migrating at once, where this number can be safely set one task at a time, or users can adjust it.
  - Enable to receive back the results of migrated tasks.
  - Stop or cancel all communication progress when the execution of tasks reaches synchronization barriers.

Figure 5.2 illustrates the overview of processes and threads created in Chameleon. Horizontally, suppose we launch P processes indexed from 0 to $P-1$. One process is mapped to one CPU socket, where a CPU has multiple cores. The execution threads in a process are pinned to these cores, where the last core is off for running the dedicated thread $Tcomm$. In Figure 5.2, we highlight execution

threads with blue background, while *Tcomm* with yellow background. A compute node in HPC clusters today often has two CPU sockets. Hence, to facilitate the execution of Chameleon with full cores, we can deploy two processes per node, and all threads map to all cores per a CPU socket.

For our proactive load balancing scheme, we can exploit *Tcomm* to perform task characterization and load prediction, instead of only monitoring load. After training a prediction model, *Tcomm* provides prediction knowledge about load values and distributes the values. Benefiting from predicted load values, we estimate which process is a potential victim and how many tasks should be migrated at a time. The implementation of our approach is designed as a plugin tool (refers to a callback tool like the OpenMP Tool interface [Eic+13]). *Tcomm* will manage callback functions declared in the tool, allowing users to predefine events and functions needed for proactive load balancing. The next subsection shows our implementation as a Chameleon plugin tool.

5.2. Proactive Load Balancing as a Plugin Tool

Users can interfere with the Chameleon operations via callback events that are designed similarly to the OpenMP Tool interface (OMPT)[Eic+13] [SK22]. Note that the term "tool" here indicates a callback tool used to interfere with the primary operations in Chameleon. Users can define the tool, so we sometimes call it as a plugin or user-defined tool. The idea is to implement the tool outside the main library to support different purposes, such as profiling tasks, load prediction, and load balancing algorithms. Therefore, we can create different tools.

Chameleon is developed with callback signatures and specific data types to define callback functions and their parameters in the tool. For the returned values of the callback functions, we can yield them to input the API functions in Chameleon. The tool can be initialized by linking and loading directly into Chameleon.

Figure 5.3 shows a flowchart to explain how a callback tool is loaded and executed alongside the Chameleon framework. This flowchart refers to the OpenMP Tool scheme, where the Chameleon runtime is similar to the OpenMP runtime. During execution, the Chameleon runtime will examine the `tool-var` ICV[1] as one of the first initialization steps. If the `tool-var` value is `null` or `disabled`, Chameleon will continue without checking the tool's presence, and the tool interface will be disabled (so-called `inactive`). Otherwise, if the tool interface is `active`, then change to `Pending` and `Find next tool`. If Chameleon finds the tool, it calls `cham_start_tool`. At this stage, there are two options to define `cham_start_tool`:

- By statically linking the definition of `cham_start_tool` into Chameleon.
- By introducing a dynamic-linked library that includes the definitions of `cham_start_tool` into the application's address space.

We apply the second option with the illustration shown in Figure 5.3. Our definition of the tool is separate and shown as the interruptable region. The region shows `tool.h` and `tool.cpp`, where we define a data structure for profiled-tasks information (`prof_task_info_t`). With `prof_task_info_t`, we highlight some major properties such as `ttid` (task ID for tracking tasks at the tool side), `num_args` (the number of input arguments in a task), and other time parameters like `que_time` (when a task is queued), `sta_time` (when a task is scheduled to execute), `end_time` (when a task is finished), `mig_time` (when a task is migrated to another process), `exe_time` (task execution time which can be calculated by `sta_time` and `end_time`), `core_freq` (the profiled information about core frequency where a task is assigned). The frequency can be measured directly before a task is executed. Explicitly, we present some primary methods in the tool that are invoked to characterizing tasks, training and loading prediction models. For example,

- `on_cham_callback_task_create()`: is called after tasks are created. Thus, the tasks' features, including arguments and input data sizes, can be characterized.

[1] Internal control variables (ICVs) that control the behavior of the runtime. ICV stores information such as the address of the compiled tool externally.

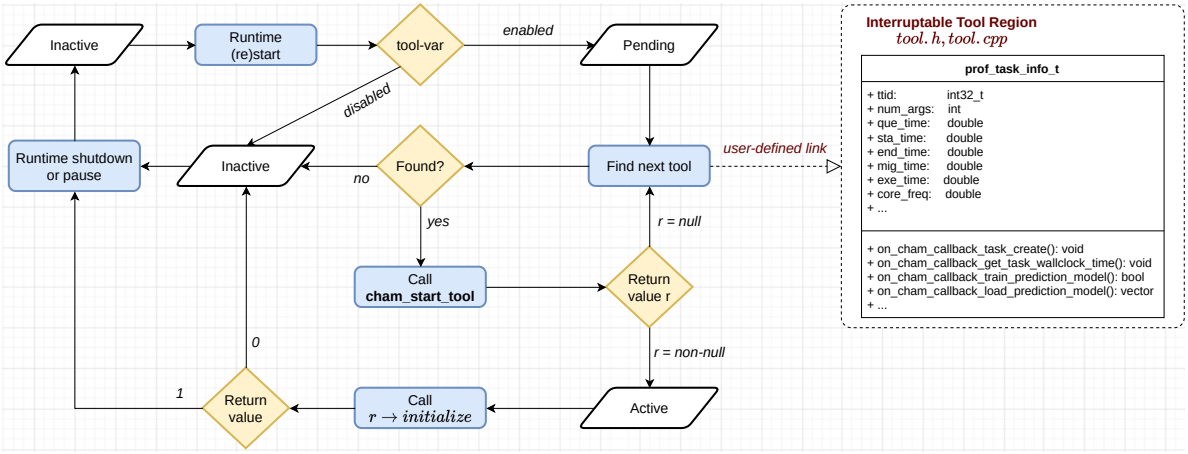


Figure 5.3.: A flowchart and class linking to show how a plugin tool is loaded.

- `on_cham_callback_get_task_wallclock_time()`: is called after tasks are finished. We can record `end_time` and calculate `exe_time`.
- `on_cham_callback_train_prediction_model()`: is called to define a prediction model with input and output layers. The model is trained to predict the load values based on what we hint to characterize. In iterative applications, this callback can be triggered after several finished iterations. For example, we can configure a certain number of iterations before execution, such as 5, 10 iterations, to collect data and train a prediction model.
- `on_cham_callback_load_prediction_model()`: is called to load a trained model. This callback is triggered after the return of `on_cham_callback_train_prediction_model()` is available.
- Additional callback functions can be defined and added by users.

In the case without online training, a pre-trained model for predicting load can be trained from the historical data. Then, we load it directly by the tool. The mechanism of this callback tool is intended to be flexible for developing different load balancing methods.

5.3. Task-based Parallel Application as a Black Box

As an example, we also use the synthetic `MxM` matrix multiplication to explain how the plugin tool and Chameleon work together. `MxM` is reproducible to conduct different imbalance scenarios. The tasks are defined by `MxM` compute kernels, indicating the function of calculating dense matrix multiplication $C = A \times B$. The input arguments include matrix A and matrix B, while the output is matrix C, where their sizes reflect the wallclock execution time of tasks. We reveal a snippet code of `MxM` to introduce writing a task-based application with Chameleon [Kli+20a].

Listing 5.1: A snippet code for implementing `MxM` compute tasks in Chameleon

```
1  // MxM compute kernel defined as task
2  void mxm_kernel(double *A, double *, double *C, int mat_size);
3
4  // Main function
5  inf main(int argc, char *argv[])
6  {
7    // Chameleon, MPI initialization and matrix allocation, ...
8    int N = matrix_size;
9
10   // define tasks
11   #pragma omp parallel
12   {
13     #pragma omp for nowait
14     for (int i = 0; i < num_tasks; i++){
```

```
15        double *A = matrices_a[i];
16        double *B = matrices_b[i];
17        double *C = matrices_c[i];
18
19        #if USE_OPENMP_TARGET_CONSTRUCT // OpenMP target approach
20          # pragma omp target map(tofrom: C[0: N*N]) map(to: A[0: N*N], B[0: N*N])
21          mxm_kernel(A, B, C, N);
22
23        #else // Chameleon API approach
24          map_data_entry_t *args = new map_data_entry_t[4];
25          args[0] = chameleon_map_data_entry_create(A,
26                          N*N*sizeof(double),
27                          MAPTYPE_INPUT);
28          args[1] = chameleon_map_data_entry_create(B,
29                          N*N*sizeof(double),
30                          MAPTYPE_INPUT);
31          args[2] = chameleon_map_data_entry_create(C,
32                          N*N*sizeof(double),
33                          MAPTYPE_OUTPUT);
34          args[3] = chameleon_map_data_entry_create(lit_size,
35                          sizeof(void *),
36                          MAPTYPE_INPUT|MAPTYPE_LITERAL);
37          cham_migratable_task_t * cur_task =
38                          chameleon_create_task((void *)& mxm_kernel, 4, args);
39          chameleon_add_task(cur_task);
40        #endif
41      }
42
43    // trigger task execution and reactive load balancing in the background
44    chameleon_distributed_taskwait();
45    }
46
47    // Chameleon, MPI finalization and clean up ...
48    ...
49  }
```

Listing 5.1 shows a snippet code example of defining `MxM` compute tasks. The task code entry points to the function `mxm_kernel()`, where the inputs of a task include four arguments, i.e., matrix A, B, C, and `lit_size`. Argument `lit_size` implies the literal size of matrices. Following here, tasks are generated by the function `chameleon_create_task()` and queued by `chameleon_add_task()`. To execute tasks, we call `chameleon_distributed_taskwait()`. This function implicitly performs task execution in parallel.

Figure 5.4 presents the interaction workflow between the application, Chameleon, and tool. The workflow has three columns: Task-based Applications, Chameleon Runtime, and User-defined Tools. Corresponding to the snippet code of `MxM`, the side of application starts with initializing Chameleon (shown as arrow `init_chameleon`), then creating tasks (arrow `create_tasks`), executing them (arrow `execute_tasks`), and finalizing Chameleon (arrow `finalize_chameleon`). At the side of Chameleon

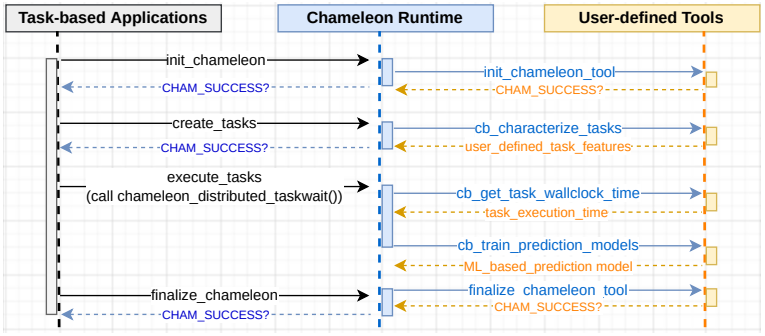


Figure 5.4.: A sequence diagram of runtime behaviors between applications, Chameleon, and tool.

during execution, it controls running tasks in the middle. If the tool is enabled (shown as arrow `init_chameleon_tool`), its callback events are triggered for characterizing tasks with input arguments (arrow `cb_characterize_tasks`), task execution time (arrow `cb_get_task_wallclock_time`). Then, the prediction model is trained afterward (arrow `cb_train_prediction_models`). After training, the prediction knowledge is obtained, and we can transfer predicted load information back to the Chameleon runtime for further proactive load balancing methods.

Listing 5.2: A snippet code for implementing a prediction model in the callback tool with Chameleon.

```
1
2  // ----------------------------------------
3  // tool.cpp
4  // ----------------------------------------
5  static bool
6  on_cham_t_callback_train_prediction_model(int32_t iter, int prediction_mode)
7  {
8      bool is_trained = false;
9      int rank = cham_t_get_rank_info()->comm_rank;
10
11     // Mode: prediction by time-series features
12     if (prediction_mode == TIME_SERIES_DAT){
13         ...
14     }
15     // Mode: prediction by task-characterization features
16     else if (prediction_mode == TASK_FEATURES){
17         // call the defined model
18         is_trained = online_mlpack_training_task_features(profiled_task_list, iter);
19     }
20     else {
21         ...
22     }
23
24     return is_trained;
25 }
26
27 // ----------------------------------------
28 // tool.h
29 // ----------------------------------------
30 bool online_mlpack_training_task_features(prof_task_list_t& tasklist_ref, int iter)
31 {
32     // get the data of profiled tasks
33     int num_tasks = tasklist_ref.ntasks_per_rank*(iter);
34
35     // normalize and generate a dataset for training
36     // declare size of dataset
37     int n_rows_X = 1;
38     int n_cols_X = num_tasks;
39     int n_rows_Y = 1;
40     int n_cols_Y = n_cols_X;
41
42     // tranform input features
43     arma::mat trainX(n_rows_X, n_cols_X);
44     arma::mat trainY(n_rows_Y, n_cols_Y);
45     for (int i = 0; i < num_tasks; i++){
46         trainX.col(i) = tasklist_ref.task_list[i]->args_list[0];
47         trainY.col(i) = tasklist_ref.task_list[i]->exe_time;
48         ...
49     }
50
51
52     // declare and apply linear regression as the prediction model
53     mlpack::regression::LinearRegression lr_pred_model(trainX, trainY);
54
55     ...
56 }
```

We refer to Listing 5.2 to show an implementation example of the callback tool. It is also a snippet code example defining a prediction module for `MxM` matrix multiplication tasks. Regarding

`MxM`, suppose the example is run with 8 processes in total. We can simply create imbalance scenarios by two options:

- Imbalance caused by a given task distribution: For example, such a high imbalance scenario $R_{imb} \approx 4.0$, tasks are uniform load. One process has 800 tasks, one with 100 tasks, four with 50 tasks, and two with 90 tasks.
- Imbalance caused by performance slowdown: For example, we can configure noise to emulate the performance slowdown. Suppose $R_{imb} \approx 1.5$, each process can be assigned the same number of tasks (100 tasks). Two processes are set up with the noise to make the execution slower.

The implementation example of the tool is mainly overviewed, referring to two callback functions:

- `on_cham_t_callback_train_prediction_model()`: suppose it is specified in file `tool.cpp`.
- `online_mlpack_training_task_features()`: suppose it is specified in file `tool.h`.

The function `on_cham_t_callback_train_prediction_model()` indicates the callback event triggered to train a pre-defined model. Depending on the users' configuration, we can specify a time when this callback is triggered. Here, we set this callback to start after the first iteration is finished. As one of the arguments, `prediction_mode` indicates that the way of generating a dataset for training here can be defined by time-series fashion or by direct task features. In `MxM`, we use direct task features because the matrix size influences MxM execution time.

The function `online_mlpack_training_task_features()` indicates where users can define prediction models. There are two arguments: `tasklist_ref` and `iter`, where `iter` is to check when or which iteration the model should be training. `tasklist_ref` indicates the profiled task list, in which we store the recorded information about input features (matrix sizes) of all tasks up to the current iteration. From line 37 to line 47, we extract these input features and synthesize the dataset. The input feature points to an argument array of tasks, i.e., matrix sizes (`args_list[0]`). The output label, in this case, is `exe_time` wallclock execution time w of tasks. Afterward, the model is triggered to be trained.

The API used to implement our machine learning model is inherited from the third-party libraries: Armadillo [SC16] [SC18] for data preprocessing and mlpack C++ [Cur+13] [Cur+23] for machine learning models. Here, we use linear regression to predict the load of each `MxM` task. When our prediction model is trained, it can be loaded by other callback events to predict the load values of tasks for the next execution phases. Ultimately, prediction results will be input for the proactive task offloading algorithm shown in Subsection 4.3.3 on Page 69.

Today, machine learning support libraries have become more popular, and we do not need to implement them from scratch. Mlpack C++ [Cur+23] is an example that we can adjust the baseline model to fit our case. As we consider machine learning a tool for learning and providing prediction knowledge, we use it for dynamic load balancing and hinge our approach into different task offloading methods.

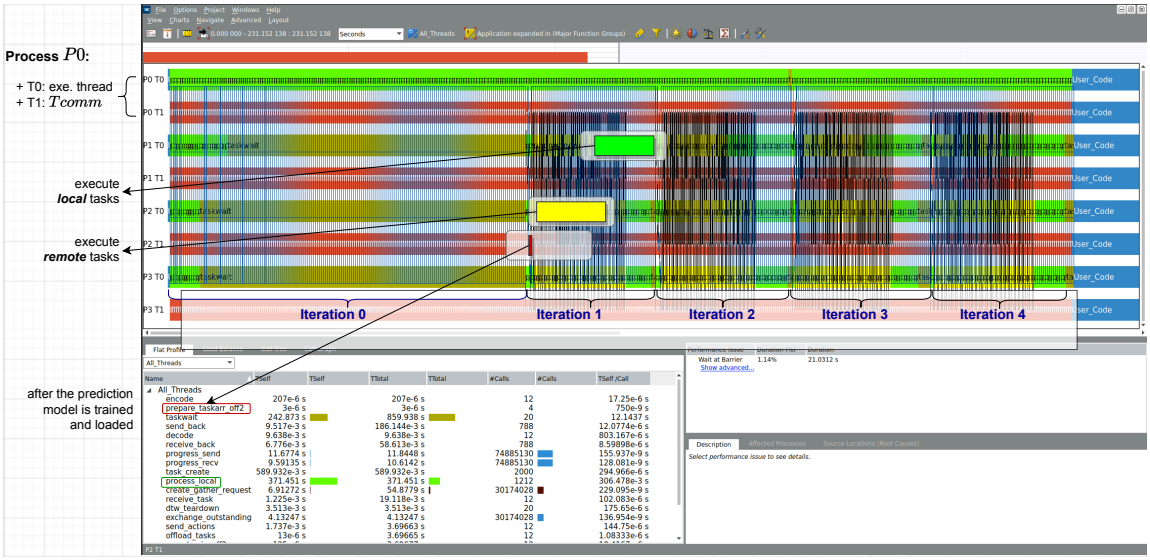


Figure 5.5.: A visualization of Chameleon and the callback tool based on Intel Trace Analyzer and Collector.

Figure 5.5 shows a visualization of running Chameleon and our callback tool through most of the main operations profiled during execution. The visualization is conducted by Intel Trace Analyze and Collector (ITAC) [WA04]. Due to the trace's scale and its overhead, this experiment is only a small test with 4 MPI processes. The area with colors shows execution progress following the horizontal axis. On the left side, we can see processes labeled by $P0$, ..., $P3$, the notations of task execution through colors, and the profiled information in the box at the bottom left corner. The experiment is shown over 5 iterations, denoted by Iteration 0, ... , Iteration 4.

Each process has two threads denoted by $T0$ and $T1$, where one execution thread is created by OpenMP and one dedicated thread by POSIX thread. For example, Process $P0$ has $T0$ and $T1$, where $T0$ is the execution thread and $T1$ is $Tcomm$, as shown In Figure 5.5. The green region indicates local tasks during execution, while the yellow one indicates remote tasks that are offloaded and executed on the remote side. In the first iteration, we can see that $P0$ is the bottleneck process with the most overloaded value without load balancing. During this iteration, $Tcomm$ on each process characterizes task features and records the load value per task. At the beginning of Iteration 1, we mark a traced event called "`prepare_taskarr_off2`" to denote that the dataset is ready for training a prediction model after Iteration 0. This callback event highlights the loaded prediction model for the upcoming iterations. Following that, in Iteration 1, 2, 3, 4, remote tasks are visualized with the execution on underloaded processes, and the imbalance is seen as significantly reduced.

6. Evaluation

Chapter 6 describes performance evaluation and experiments, comparing our load balancing methods based on the proactive approach with the existing balancing methods. The metrics for evaluation include imbalance ratio and speedup in completion time. We evaluate the speedup by the ratio between baseline (no balancing) and applied methods, such as work stealing, reactive task offloading, proactive task offloading. In detail, this chapter is outlined by

1. Presenting the environmental setup in Section 6.1.
2. Presenting the evaluation of online load prediction that provides knowledge for proactive task offloading methods, in Section 6.2.
3. Presenting the comparison results between our methods and the existing methods in Section 6.3.
4. Presenting the evaluation of co-scheduling tasks across multiple applications in Section 6.4, which is considered as an extension based on our proactive load balancing approach.

These experiments are performed in three different HPC clusters using synthetic micro-benchmarks and a realistic HPC application.

6.1. Environmental Experiments

We configure and perform experiments on three different HPC systems at the Leibniz Supercomputing Centre (LRZ), namely CoolMUC2, SuperMUC-NG and BEAST. The overview of these systems is already mentioned in Chapter 3 on Page 41. However, to give a detailed specification of hardware and software stack, we present Table 6.1, describing more information such as processor type, the number of CPU sockets (# sockets), number of cores per socket (# core per socket), memory volume, and the maximum bandwidth (Max. bandwidth - B) measured by using the OSU benchmark [Pan+21].

The main difference between these systems is the interconnection, distinct by different technologies. CoolMUC2 uses FDR14 Infiniband, an older technology with a bandwidth of $\approx$ 6.52 GB/s. SuperMUC-NG is higher with 12.07 GB/s, and BEAST is a better system featured with a new Infiniband technology $\approx$ 22.03 GB/s. Especially in the BEAST system, it has different types of computing architectures, featuring two nodes of AMD processors (Rome EPYC 7742), two nodes of ARM ThunderX2, eight nodes or a so-called small cluster with Fujitsu A64FX, and two nodes of Intel CooperLake CPU. However, we do not use all of the nodes on BEAST, only selecting several nodes with the same architectures to run the experiments.

Table 6.1.: The specification of systems for runing experiments.

Specification	**CoolMUC2**	**SuperMUC-NG**	**BEAST**
Processor	Intel Xeon E5-2697 v3 2.60GHz	Intel Skylake Xeon Platinum 8174 2.40GHz	featured with different processors, e.g., AMD Rome EPYC 7742, ARM ThunderX2, Fujitsu A64FX, Intel CooperLake
# sockets	2	2	2(AMD), 2(ARM), 1(A64FX), 4(Intel)
# cores per socket	14	28	64(AMD), 32(ARM), 48(A64FX), 24(Intel)
Memory	64	96 (thin nodes), 768 (fat nodes)	512(AMD), 512(ARM), 32(A64FX), 768(Intel)
Network Infrastructure	FDR14 Infiniband	OmniPath 100Gb/s	HDR InfiniBand 200Gb/s
Max. bandwidth (B) (measured with OSU Benchmark [Pan+21])	6.52 GB/s	12.07 GB/s	22.03 GB/s
Operating System (OS)	SLES15 SP1 Linux	Suse Linux (SLES)	Suse SLES (AMD, Intel nodes), CentOS 8 (ARM, A64FX nodes)

Regarding micro-benchmarks, we use the following kernels to define tasks.

- `MxM`: matrix multiplication, which is mentioned as an example for our illustration in the previous chapters. A task is defined by a matrix multiplication compute kernel.
- `Nbody`: N-body simulation, which is a simulation of a dynamical system of particles. Tasks in `Nbody` are defined by different compute kernels, such as Nbody solver and force calculation.
- `Jacobi`: Jacobi solver, which refers to the Jacobi method in numerical linear algebra. The compute kernels in the Jacobi solver define tasks.

Regarding the realistic use case, we use Sam(oa)^2 [MRB16] as introduced in Chapter 4, Page 66. Sam(oa)^2 is a simulation framework developed through the concept of Space-filling curves and Adaptive Mesh Refinement for Oceanic Applications in HPC. This framework features hybrid MPI+OpenMP parallelization based on the Sierpinski order. The Sam(oa)^2 authors focus on providing a dynamically adaptive solver for 2D Partial Differential Equations (PDEs).

In terms of scientific simulation, there are two popular scenarios using Sam(oa)^2:

- simulating multiphase flow in porous media.
- simulating environmental issues, such as tsunami wave propagation and earthquake, based on solving shallow water equations.

In our experiments, we deploy Sam(oa)^2 as a real use case, simulating related oceanic problems such as tsunami wave propagation, oscillating lake. The design of Sam(oa)^2 aims at memory efficiency, where the complexity of the stack&stream approach is hidden while providing flexibility for multiple discretization methods. There are different variants of Sam(oa)^2 researched for a certain optimization purpose. In this thesis, we refer to using a new optimized version called ADER-DG [RDB18] because it is said to be optimized with the application of the ADER Discontinuous Galerkin method for simulating tsunamis. Similar to Sam(oa)^2, there are several libraries and frameworks, e.g., Peano [Bun+10], PDELab in DUNE [BB07].

Related to task definition and execution, the Sam(oa)^2 framework employs the workflow of traversing grid. The grid is sub-partitioned into sections. All sections are distributed in distributed memory systems over multiple processes. With task-based parallel programming models, threads in a process

are execution units that traverse these sections. Traversing a section is considered as a computational task. These computations are characterized as independent tasks in Chameleon. Before execution, the canonical approach in Sam(oa)^2 itself divides the whole grid into equal parts over processes. Following that, tasks within a process are uniform or equal load, but the tasks from different processes might not have the same load values. Furthermore, Sam(oa)^2 already has several balancing techniques built in the framework relying on its cost models. However, our context is assumed to encounter performance slowdown; therefore, this leads to a new imbalance at runtime.

6.2. Online Load Prediction Evaluation

This section shows an evaluation of our online load prediction scheme described in Subsection 4.3.1. The experiments consist of `MxM` matrix multiplication and Sam(oa)^2. For the evaluation, we specify two criteria:

- Accuracy: the accuracy of prediction models. The metric accounting for accuracy is loss, which alludes to the difference between real and predicted values.
- Overhead: the overhead of training and inferencing prediction models. The unit accounting for overhead is time in microseconds, milliseconds, or seconds, depending on the length of completion time.

There are different metrics to calculate the loss in machine learning [NA21]. We use the following loss metrics:

- MAE: Mean Absolute Error. MAE calculates the absolute difference between actual and predicted values.
- MSE: Mean Squared Error. MSE calculates the squared difference between actual and predicted values.
- RMSE: Root Mean Squared Error. RMSE is a simple square root of MSE.
- R2score: R Squared. R2score is a metric that explains the performance of prediction models, calculating how much a regression line is better than a mean line in prediction.

For the overhead of training and inferencing, we compare the application completion time with the training and inferencing times. This overhead is important because if its value is too high, not only the load balancing module is affected but also the execution of our application.

Our evaluation is divided into two groups of experiments, detailed in the following subsections.

1. Varying the scale parameters: the number of compute nodes and the number of threads per process. This group of experiments aims at the feasibility of applying machine learning in online load prediction by evaluating the accuracy. Simultaneously, we show that the accuracy and cost of training and loading the prediction model, are not greatly affected when increasing the computational scale.
2. Varying the machine learning algorithms: Linear Regression, Ridge, Bayesian, and LARS (Least Angle Regression (Stage-wise/laSso)). These algorithms are classified as regression algorithms in machine learning [Bon17]. This group of experiments focuses not on accuracy but on flexibility in user-defined prediction models. We show that load prediction models can change based on users and specific applications. Thereby, in this group, we highlight that flexibility in applying different machine learning algorithms is feasible. Simultaneously, accuracy can be guaranteed when we understand our applications as well as prediction models.

6.2.1. Varying the scale parameters: the number of compute nodes and the number of threads per process

We perform three experiments that revolve around adjusting the number of compute nodes and the number of threads per process. These experiments include:

Table 6.2.: Evaluation of online load prediction for `MxM` matrix multiplication over the scale of compute nodes.

Node scales	**avg(MSE loss)**	**training time (ms)**	**inference time (ms)**
2	0.00059	458.209	0.1357
4	0.00038	478.609	0.1406
8	0.02888	502.514	0.1432
16	0.00470	320.179	0.1518
32	0.00618	277.506	0.1492
64	0.02130	346.579	0.1613

- Experiment 1: is performed with `MxM` matrix multiplication, where we vary the number of compute nodes from 2 to 64. This experiment is performed on CoolMUC2, and the results are shown in Table 6.2. It is configured by the same number of tasks assigned to each process. To increase objectivity, we randomize the argument size of tasks, resulting in randomized wallclock execution time.
- Experiment 2: is also performed with `MxM`, but we vary the number of threads per process. This experiment is performed with two nodes on the BEAST system because BEAST has AMD nodes featuring a multicore architecture with 64 physical cores per processor. This allows deploying multiple threads in a process. The experiment results are shown in Table 6.3.
- Experiment 3: is performed with the realistic application, Sam(oa)^2, where we run Sam(oa)^2 to simulate oscillating lake over 100 time steps (iterations). This experiment is used to evaluate the accuracy of online load prediction in a real use case. We again vary the number of compute nodes, and the results are shown in Figure 6.1.

In addition, the experiments with `MxM` can be reproduced with different imbalance scenarios through the number of assigned tasks per process and task data size. However, to evaluate load prediction, we only focus on the load of tasks and configure the same number of tasks per process, where task argument size is randomized to create randomized load values.

Table 6.2 shows the prediction results of Experiment 1. It has four columns, where the compute node scale is shown in the column "Node scales", varied from 2 to 64 nodes. Accordingly, the next columns "avg(MSE loss)", "training time", and "inference time" show the results of loss values calculated by MSE, overhead by training time, and inferencing time in milliseconds.

On CoolMUC2, each node is deployed 2 MPI processes. Each process spawns 13 OpenMP threads to execute tasks and 1 thread (pthread) to dedicate communication ($Tcomm$). Given a distribution of tasks, we assign 200 tasks on each process. For the prediction model in this case, we use the task's argument sizes and CPU core frequencies for the input layer. Task wallclock execution time is configured for the output label. Note that the core frequencies of compute nodes on CoolMUC2 are safely configured "fixedly"; therefore, they do not influence the prediction model. In this case, only matrix sizes affect the model's accuracy. Regarding the machine learning algorithm, $Tcomm$ is configured to use linear regression to train the prediction model. The results of Experiment 1 in Table 6.2 show that:

- The loss values over different node scales remain low and stable (around ≈ 0.02). This highlights the feasibility of predicting load during execution, especially supporting load balancing.
- The training and inferencing time are also low and stable. The maximum training overhead is $\approx$ 500ms, and the inferencing overhead is under 1ms.

In our approach, the operations of $Tcomm$ are separate from the others; therefore, the overhead of deploying machine learning keeps little change over different experiments. Particularly, over node scales, we can see that training and inferencing time remain approximately. In fact, the cost of training might be a trouble if the dataset is large and complex. However, regarding online load prediction,

Table 6.3.: Evaluation of online load prediction for `MxM` over the scale of threads per process.

#threads per process	**real R_{imb}**	**predicted R_{imb}**	**training time (ms)**	**inference time (ms)**
2	0.0311	0.0108	18.156	0.0257
4	0.2706	0.3863	18.218	0.0103
8	0.5384	0.5079	19.804	0.0215
16	0.3502	0.2298	19.793	0.0199
32	0.5815	0.5442	19.807	0.0192
64	0.4245	0.3111	20.951	0.0229

we need to select influence factors that can be satisfied to help training. Therefore, we show that the number of parameters suitable for predicting load is not many, and a lightweight machine learning model to predict the load at runtime is possible.

Table 6.3 shows the results of Experiment 2, varying the number of task-execution threads per process. It has five columns, where the first column indicates the number of threads, "#threads per process". The next columns show "real R_{imb}", "predicted R_{imb}", "training time", and "inference time" respectively. The real R_{imb} values are calculated by the real (observed) total load values of processes, while the predicted R_{imb} values are calculated by the predicted total load values of processes. Notably, this experiment presents the load prediction of each task (w) in `MxM`; therefore, the predicted total load values per process can be calculated by summing up all values w that belong to a process. The results of Experiment 2 in Table 6.3 highlight that:

- The difference between real and predicted R_{imb} is low and optimistic, except for the case with the smallest scale, 2 threads per process. This is because the real imbalance ratio is already too small. Other cases show again the feasibility of online load prediction over changing the number of threads executing tasks.
- The training and inferencing time on BEAST are smaller and more stable than in Experiment 1 (on CoolMUC2). This is partly from executing on BEAST; the core frequencies are not fixed, and the prediction model has an input parameter of core frequencies, which increases the accuracy of the machine learning model.

Overall, $Tcomm$ is asynchronously run with the other execution threads. Its overhead does not impact system-level configuration, e.g., node or thread scales. In terms of prediction accuracy, we show that considering influence features for online load prediction should be revised by users. Depending on a definition of tasks, users can define influence features better.

Figure 6.1 shows the results of Experiment 3, running Sam(oa)^2 over 100 time steps (iterations or iter for short). This experiment is performed on CoolMUC2, where the number of compute nodes is varied from 2 to 16. Unlike `MxM`, the predicted load of each process is based on the load value in previous iterations. We use 20 first iterations for generating the dataset, then $Tcomm$ on each process trains the model afterward. The trained model is loaded at the 21^{st} iteration. Figure 6.1 has two plots, where the left one shows a box plot of MSE loss values over different scales of compute nodes, and the right one illustrates the comparison between real and predicted load values in the case of 16 nodes. In Figure 6.1 (left), the x-axis indicates the labels of node scale, while the y-axis indicates MSE values. In Figure 6.1 (right), the x-axis indicates the iteration indices from 0 to 99, and the y-axis indicates the process labels along with the load values in seconds. The results in this experiment highlight that:

- MSE loss again shows the feasibility when we predict the load values at runtime by a different prediction model. Particularly, it relies on the load in the previous iterations. The smaller loss determines a better prediction.
- The comparison between real and predicted load values emphasizes the applicability of machine learning when training prediction models at runtime. The iterations indexed from 0 to 19 are

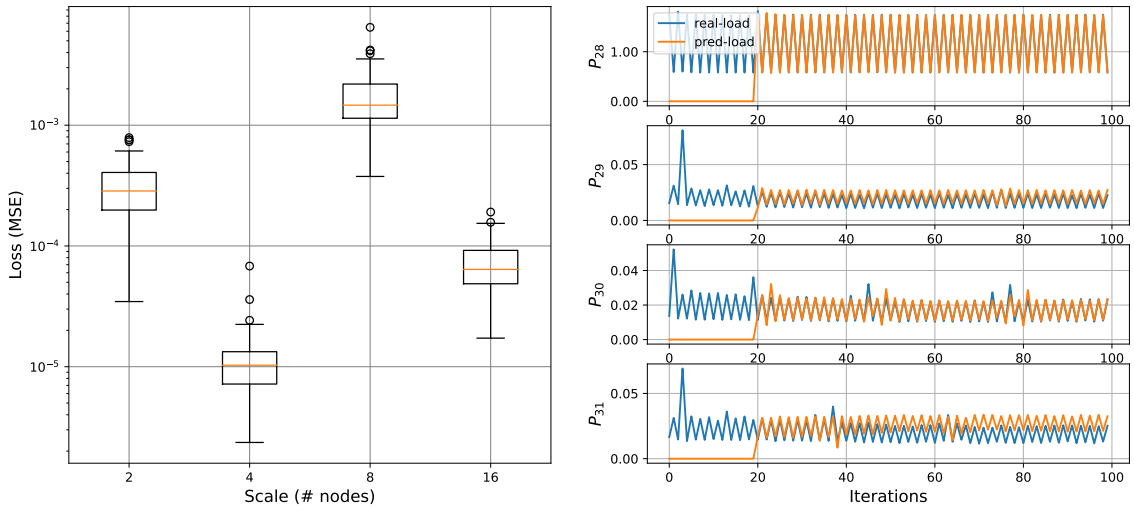


Figure 6.1.: Evaluation of online load prediction on Sam(oa)^2 when simulating oscillating lake.

Table 6.4.: The evaluation of load prediction using different machine learning regression algorithms.

	`MxM` Matrix Multiplication			
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

used to generate the dataset, in which we see the predicted values are zero. The results in Figure 6.1 (right) correspond to the 16-nodes scale, where each node is deployed 2 processes, and we have 32 processes in total. This figure only shows the results of 4 in 32 processes, i.e., P_{28}, P_{29}, P_{30}, P_{31}, because the results of other processes are similar. Thus, it is not necessary to illustrate them all. Besides that, the presented results of 4 processes also fit better with Figure 6.1 on the left. Starting from iteration 20, the total load values of processes are predicted. The blue line denotes observed values (real values), while the orange line shows predicted values.

These results show a positive outcome in general. Besides that, we conclude that the major point of load balancing is the load difference between the involved processes. Therefore, a very high-accuracy prediction is not a challenge; instead, we can rely on the load difference among processes to guide load balancing. Unlike several problems in computer vision or natural language processing, the high-accuracy prediction models are expected. In our context, the experiments above show that using load prediction results at runtime to balance the load is feasible.

6.2.2. Varying the machine learning algorithms

We perform two experiments to evaluate the possibility of changing prediction algorithms, one with `MxM` matrix multiplication and one with Sam(oa)^2. Generally, using machine learning to predict load depends on input features and algorithms. It is difficult to decide how much data is adequate enough to determine the training configuration. Therefore, the experiments in this subsection again refer to user-defined prediction models. We can flexibly adjust the learning algorithms that do not affect

Table 6.5.: The overview of different load balancing methods in comparison.

No.	Method	Description
1	`baseline`	No load balancing.
2	`random_ws`	Randomized work stealing.
3	`react_off`	Reactive task offloading only.
4	`react_rep`	A-priori speculative task replication only.
5	`react_off_rep`	Mixed reactive task offloading and replication.
6	`proact_fb`	Proactive load balancing with feedback task offloading.
7	`proact_off1`	Proactive load balancing with round-robind task offloading.
8	`proact_off2`	Proactive load balancing with packed-tasks offloading.

the execution of our application. Our approach is implemented as a plugin tool upon a task-based framework/library, which facilitates user-defined prediction models.

Table 6.4 shows the loss evaluation by applying different regression algorithms, including Linear Regression, Ridge, Bayesian, and LARS (Least Angle Regression (Stage-wise/laSso)) as mentioned on Page 91. These algorithms are used because they are suitable for prediction problems with continuous output values instead of classification problems. The average loss values are calculated by MAE, MSE, RMSE, and R2scores [NA21]. The first column in Table 6.4 indicates the algorithms, followed by the result columns of MAE, MSE, RMSE, and R2scores. The results of both `MxM` and Sam(oa)^2 highlight positive results, with low loss values. For the case of large-scale applications, it is feasible to use *Tcomm* for augmenting the dataset and re-training the models at runtime. Then, we can improve the prediction accuracy. The next experiments show the evaluation when adapting prediction outputs to support dynamic load balancing in our proactive approach.

6.3. Evaluation of Proactive Load Balancing

We present two groups of experiments to compare our proactive load balancing approach with the previous approaches. In particular, there are three proposed methods within our approach, including feedback task offloading (`proact_fb`), proactive task offloading with the task migration strategy 1 (`proact_off1`), and proactive task offloading with the task migration strategy 2 (`proact_off2`). All methods are explained below and summarized in Table 6.5.

- `baseline`: no load balancing. Applications might have default pre-partitioning algorithms for task distribution and load balancing.
- `random_ws`: randomized work stealing.
- `react_off`: reactive task offloading, which refers to the default method of reactive load balancing approach.
- `react_rep`: reactive task replication, which is a variant of `react_off` in that we replicate the tasks reactively instead of migrating tasks.
- `react_off_rep`: a combination of reactive task offloading and replicating, which is another variant of `react_off`.
- `proact_fb`: feedback task offloading, which is a method of our proactive load balancing approach based on the statistic of load balancing in a previous execution phase.
- `proact_off1`: proactive task offloading with a task migration strategy called round-robin.
- `proact_off2`: proactive task offloading with a task migration strategy called packed-tasks.

The evaluation metrics include imbalance ratio and speedup. A smaller imbalance ratio is better, while a larger speedup is better. The speedup value is calculated by the ratio between the completion time of baseline (no load balancing) and the completion time of applied load balancing methods. The two groups of experiments are overviewed as follows.

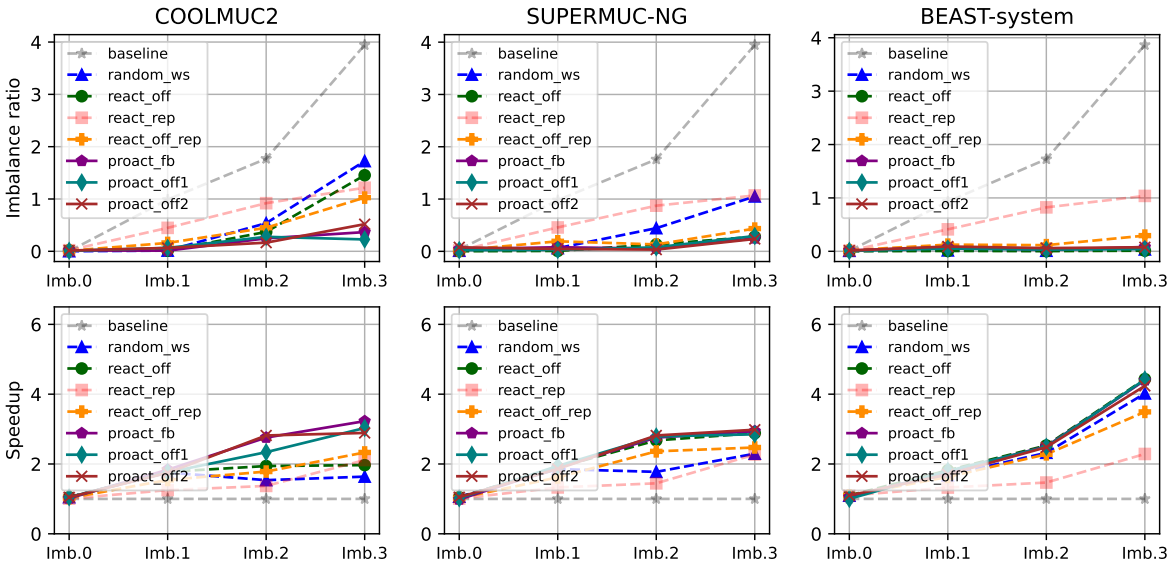


Figure 6.2.: A comparison of different load balancing methods on `MxM` through imbalance ratio and speedup.

- Group 1: experiments again with `MxM` matrix multiplication on three different HPC systems, CoolMUC2, SuperMUC-NG, and BEAST. `MxM` is executed with different imbalance scenarios. We analyze and present the results of Group 1 in Subsection 6.3.1.
- Group 2: experiments with Sam(oa)^2 on CoolMUC2, SuperMUC-NG, and BEAST. The imbalance is derived from the simulated use cases in Sam(oa)^2 and the scale of compute nodes or processes used. We analyze and present the results of Group 2 in Subsection 6.3.2.

6.3.1. Experiments with `MxM` matrix multiplication

The synthetic `MxM` is reproducible, where a task is defined by an `MxM` compute kernel. Given the tasks are independent and have uniform load if the matrix sizes are configured the same. We create different imbalance scenarios by varying the number of `MxM` compute tasks per MPI process as a given distribution. There are 4 cases generated from no imbalance to high imbalance ratios (`Imb.0` - `Imb.3`). Due to permission on the systems CoolMUC2 and SuperMUC-NG at LRZ, we cannot adjust the core frequency to emulate performance slowdown, and the frequency is configured at a fixed level. Thereby, matrix sizes are the input parameters that mainly affect the accuracy of online load prediction on these two systems. Core frequency values fluctuate in the BEAST system, so the load prediction model still works with the input features of both application and system. For each run, `MxM` is set up 5 iterations as 5 execution phases of computing tasks, where the first iteration is used for load prediction, and the remaining iterations are used to evaluate load balancing methods.

The experimental results are presented in Figure 6.2. There are two rows of results corresponding to the sub-figures (sub-images), where the first row denotes imbalance ratios, the second row denotes speedup. From left to right, each sub-figure shows the corresponding results of each system, including CoolMUC2, SuperMUC-NG, and BEAST. Regarding the coordinates, the x-axis indicates imbalance scenarios in response to `Imb.0`, `Imb.1`, `Imb.2`, `Imb.3`, while the y-axis indicates the evaluation values, imbalance ratio or speedup.

For evaluating imbalance ratios, we can see that the reactive load balancing methods `react_off` and `react_off_rep` are competitive. However, these methods show that the imbalance can still be improved with a high imbalance scenario such as `Imb.3`. For example, the values of R_{imb} in this case are still high on CoolMUC2, $\approx$ 1.7 with `random_ws`, 1.5 - 1.1 with `react_off` and `react_off_rep`.

Table 6.6.: Scenario 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 6.7.: Scenario 3: Imbalance Ratio (`imb.3`)

AVERAGE	COOLMUC2	SUPERMUC-NG	BEAST-system
∑(#migrated tasks) over processes	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

With our approach, the proactive load balancing methods `proact_fb`, `proact_off1`, and `proact_off2` reduce the case `Imb.3` under 0.6. On SuperMUC-NG and the BEAST system, the overhead of load balancing and task migration is mitigated by better computation speed and interconnection network compared to CoolMUC2. This is beneficial for a compute-bound microbenchmark like `MxM`. Especially, the results in BEAST show that the reactive load balancing methods are still robust.

Corresponding to the imbalance ratio results, we show the speedup results on the second row of Figure 6.2. The three proactive load balancing methods gain an improvement in CoolMUC2 1.2× speedup compared to `react_off` and 1.4× to `random_ws` in average; 1.64× speedup compared to `react_off` and 1.97× to `random_ws` in maximum. In SuperMUC-NG, we do not gain much improvement in average, where the methods `proact_fb`, `proact_off1`, and `proact_off2` only get 1.2× compared to `random_ws`, and approximate the speedup of reactive methods. Similarly, in the BEAST system, the performance of proactive methods also approximates the reactive methods. Reactive load balancing still shows resistance for different imbalance scenarios on BEAST.

To explain how the reactive methods are still robust on BEAST, we present the profiled data of balancing operations. This is the detailed information of `react_off` (which denotes reactive task offloading and the best reactive method in the experiments above) in the cases `imb.2` and `imb.3`, shown in Table 6.6 and Table 6.7 respectively. Each table has four columns, where the first column highlights three profiled information as explained below. The remaining columns are the profiled information that `react_off` is run on the corresponding systems, CoolMUC2, SuperMUC-NG, and BEAST.

- ∑(#migrated tasks) over processes: the average sum of how many tasks are migrated. This value gives an overview of the number of tasks moved around processes.
- #balancing calculation calls: the average number of reactive balancing calculation calls. This value denotes the operations for checking queue status and calculating imbalance conditions performed by $Tcomm$.
- cost/balancing calculation (μs): the average cost per balancing calculation call in microseconds (μs). This value denotes the overhead of running the operation that calculates imbalance ratios and detect an imbalance.

These three values are first calculated by average over the number of iterations, then second by average over the number of processes. A large or small number of ∑(#migrated tasks) over processes does not mainly indicate good or bad migration because there could be some wrong task migration at an incorrect reactive decision. However, this value can show the availability and capability of each system in migrating tasks. With #balancing calculation calls, a larger number shows $Tcomm$ works more active in checking imbalance status.

As the results in both Table 6.6 and Table 6.7, we can see that the balancing cost on BEAST is ≈ 2× - 3× lower than on SuperMUC-NG and CoolMUC2. The number of migrated tasks is also higher. The BEAST system has benefited from the migration throughput as well as the lower cost of

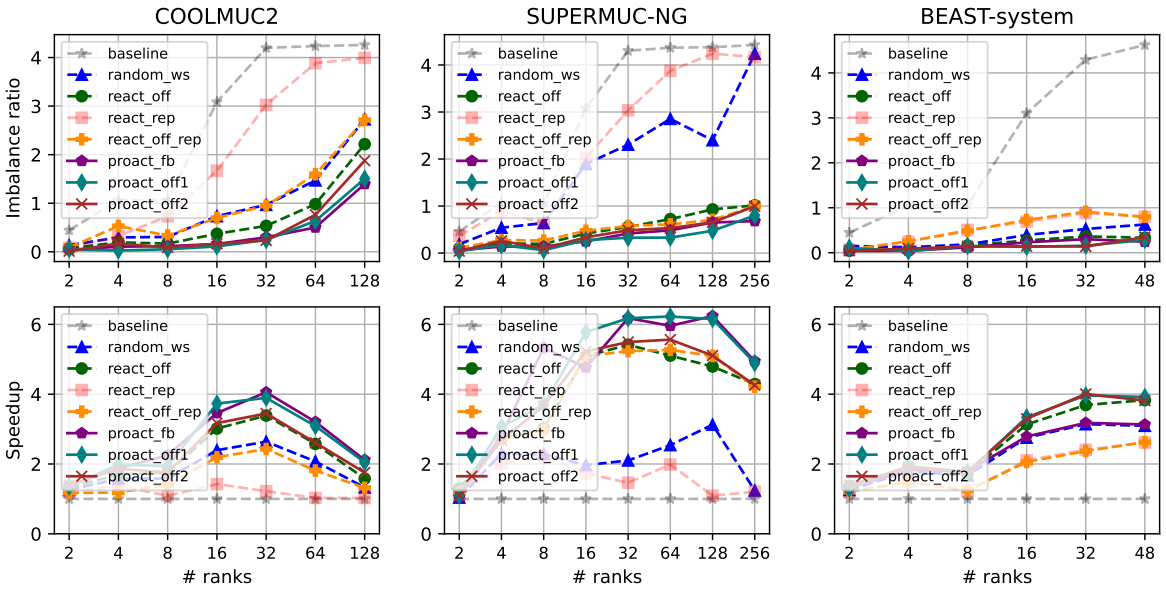


Figure 6.3.: Comparison of different load balancing methods on Sam(oa)^2 simulating the scenario of oscillating lake.

balancing operations. *Tcomm* is more reactive; therefore, even in the case of wrong task migration, it can migrate tasks back around. Furthermore, the interconnection network on BEAST 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.

6.3.2. Experiments with Sam(oa)^2

Sam(oa)^2 is set up by running a 100 time-steps simulation. The simulation scenario is again oscillating lake, where tasks are defined by traversing sections (like cells in the grid) that we have explained in Chapter 4, Section 4.3 on Page 66. The load of tasks in a process can be changed dynamically due to the adaptive mesh refinement algorithm [RDB18], which is related to wetting, drying, and the limiter applied to several mesh cells. We use standard configurations, where tasks are fine-grained, and the number of tasks per process depends on simulation scenario scale, i.e., the number of sections per mesh region.

In Sam(oa)^2, there are several parameters[1] to set up a simulation scenario, e.g., the number of sections (`sections`), number of threads (`threads`), number of nodes (`nodes`), number of iterations (`nmax`), boundary size (`boundary_size`), etc. On the side of a task-based programming framework, application is considered as a black box. Hence, we are simly concerned with the parameters in Sam(oa)^2 that define tasks and affect imbalance with the scale of execution. Such the default configuration, we set up `sections` $= 16$, `threads` $= 13$, `nodes` $= 32$, `nmax` $= 50$. In detail,

- `sections`: the number of sections created and executed within a thread.
- `threads`: the number of threads spawn in a process. Multiplying `threads` with `sections` determines the number of tasks defined and assigned to a process. For example, here we have $13 \times 16 = 208$ tasks per process.
- `nodes`: the number of compute nodes that are required and allocated on HPC clusters. If we multiply the number of tasks per process with `nodes`, the total number of tasks is $208 \times 32 = 6656$ tasks in total.

[1] https://github.com/meistero/Samoa

- `nmax`: the number of time steps that run the simulation. Depending on a simulated scenario and algorithm used in Sam(oa)^2, this value can regulate the number of actual iterations run in practice, e.g., 2 × `nmax` is the number of actual iterations. Therefore, this case has 100 iterations.

By characterizing the behavior of Sam(oa)^2, we observe that the load of tasks is uniform, and task distribution is fine-grained. Nonetheless, the load values can be susceptible to numerical algorithms. The load value per task is not large but prone to fluctuations. Therefore, many tasks per node can be affected by fluctuations in system performance slowdown, leading to an imbalance.

For the evaluation, we vary the number of processes on each system with two MPI processes per node, and each process uses full cores of a CPU socket. The experiments can show scalability and adaptation in various methods, especially with different interconnection networks on three systems. Similar to the format of Figure 6.2 in the previous subsection, the experimental results are shown in Figure 6.3. There are also two rows showing the results. The first row points to imbalance ratio, while the second row points to speedup. Each sub-figure has the x-axis showing the scale of processes used to run the experiment, while the y-axis shows the imbalance and speedup values.

Overall, the reactive and proactive load balancing methods perform better than work stealing. With reactive load balancing, `react_off` is shown as the best method in most of the cases, while (`react_rep`) is worse because of the cost of speculative task replication. Nevertheless, their combination `react_off_rep` could help the cases from 16 processes (MPI ranks) on CoolMUC2 and BEAST. Based on profiling the execution behavior of `react_rep`, we notice that the speculative task replication strategy has difficulty dealing with the imbalance of consecutive underloaded processes. For instance, without prior knowledge, we have to fix a certain number of replicated tasks and the process victim for replication.

With our proactive approach, we use online prediction to provide load information guiding task offloading. The methods `proact_fb`, `proact_off1`, and `proact_off2` show an improvement in the high imbalance cases (≥ 8 processes), where `proact_fb` and `proact_off1` are competitive. The benefit that makes our methods better is predicted load information at runtime. This helps reduce the number of wrong task migration decisions. Simultaneously, we can better estimate the appropriate number of tasks and potential processes for migration.

Especially on CoolMUC2 and SuperMUC-NG, `proact_fb` and `proact_off1` achieve the best performance. Differing from the results of `MxM`, we gain the best performance with `proact_fb` and `proact_off1` on SuperMUC-NG. In particular, the performance gain is 1.2× speedup compared to `react_off` and 2.3× to `random_ws` in average; 1.7× speedup compared to `react_off` and 3.9× to `random_ws` in maximum. On CoolMUC2, Sam(oa)^2 with our methods gains around 1.2× speedup compared to `react_off` and 1.4× to `random_ws` in average; 1.3× speedup compared to `react_off` and 1.6× to `random_ws` in maximum. Similar to the results of `MxM`, the performance improvement of proactive load balancing on BEAST is insignificant and approximates the reactive methods.

With two proactive task offloading methods, `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.

In principle, proactive task offloading must depend on the accuracy of load prediction models. However, the features characterized by an online scheme at runtime can reflect the execution behavior adaptively. Besides, our expected accuracy of prediction models does not need to be too high, which means an acceptable result to estimate the load difference among processes is enough for load balancing. Therefore, the proactive load balancing scheme is feasible to perform proactive task offloading.

Table 6.8.: An experiment of co-scheduling tasks across two applications under randomized task generation and imbalance ratios.

Imb.Case	#tasks		#shared tasks	C_{max}	C'_{max}	Overhead	Speedup
	App.1	App.2					
C1-$R_{imb} = 0.06$	58	51	3	21.935	17.253	8.46%	1.27×
C2-$R_{imb} = 0.38$	100	45	27	37.799	22.924	8.27%	1.65×
C3-$R_{imb} = 0.04$	105	113	4	42.982	34.286	7.99%	1.25×
C4-$R_{imb} = 0.29$	121	66	27	46.601	29.483	8.43%	1.58×
C5-$R_{imb} = 0.51$	140	46	47	52.522	29.259	8.38%	1.79×
C6-$R_{imb} = 0.27$	154	89	32	57.976	38.180	8.38%	1.52×
C7-$R_{imb} = 0.31$	190	101	44	72.711	43.701	9.36%	1.66×

6.4. Evaluation of Co-scheduling Tasks across Multiple Applications

As addressed in Chapter 4, Section 4.4, the experiments of co-scheduling tasks across multiple applications have to be emulated on HPC systems. We define different tasks of different applications in the same executable to emulate multiple runs simultaneously. There are three experiments to show the applicability and performance of our co-scheduling extension.

- Experiment 1: aims to show the feasibility of co-scheduling tasks across two applications. We use two types of `MxM` compute tasks accounting for two `MxM` applications. The results are shown in Table 6.8.
- Experiment 2: aims to vary the proportion of idle slots between two applications. This experiment can show the advantage when tasks from one application can be co-scheduled to another, corresponding to the idle proportion. We also use two types of `MxM` compute tasks accounting for two different `MxM` applications. The results are shown in Figure 6.4.
- Experiment 3: aims to show the benefit of co-scheduling tasks across different microbenchmarks, including `MxM`, Cholesky, Jacobi, and Nbody. The results of this experiment are shown in Figure 6.5

In Experiment 1, a given imbalance scenario is set up before two applications run, where one is intended to configure fewer compute tasks but more idle slots, while one has more compute tasks. The number and data size of tasks are randomized to generate non-uniform load and random imbalance levels. Table 6.8 details the experiment results with eight columns. The first column indicates the label of imbalance cases, marked from C1 to C7 and associated with the values of R_{imb}. The second and third columns show the number of tasks that are randomly created on each application, "*App.*1" and "*App.*2". For example, in case C1, *App.*1 has 58 tasks, while *App.*2 has 51 tasks, R_{imb} is 0.06. The column "#shared tasks" indicates the number of tasks that are co-scheduled across *App.*1 and *App.*2 to make them both improve the performance. Following that, the remaining columns show the evaluation metrics in:

- C_{max}: the completion time without co-scheduling tasks. This value is measured by the maximum completion time between *App.*1 and *App.*2.
- C'_{max}: the completion time with co-scheduling tasks. Similarly, this value is also measured by the maximum completion time between *App.*1 and *App.*2.
- Overhead: the consumped time is measured when the co-scheduler is triggered and spent time to take action on task migration.
- Speedup: the ratio between C_{max} and C'_{max}.

By applying co-scheduling tasks, the overloaded application sends an estimated number of tasks to another application. Note that task input arguments are sent from the side of the original processes

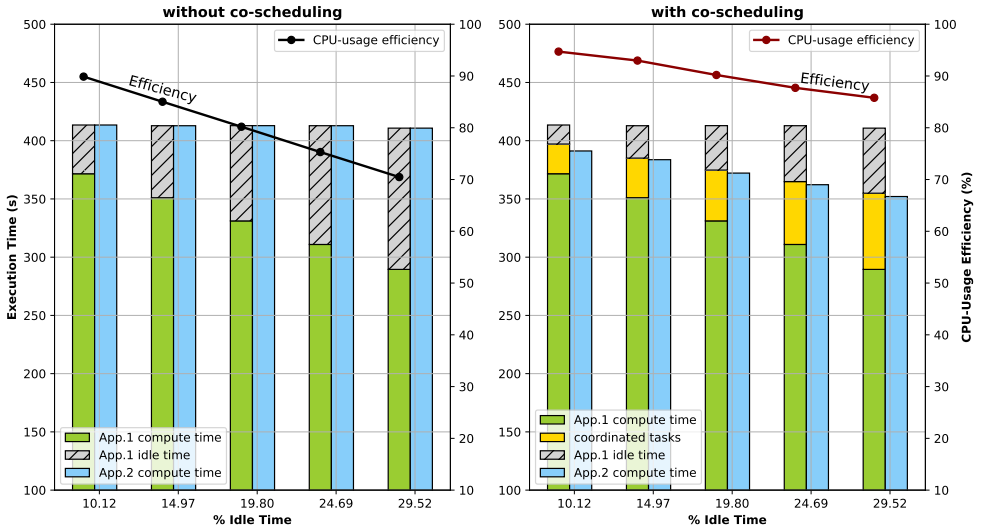


Figure 6.4.: An experiment of co-scheduling tasks across two applications under varied idle slots.

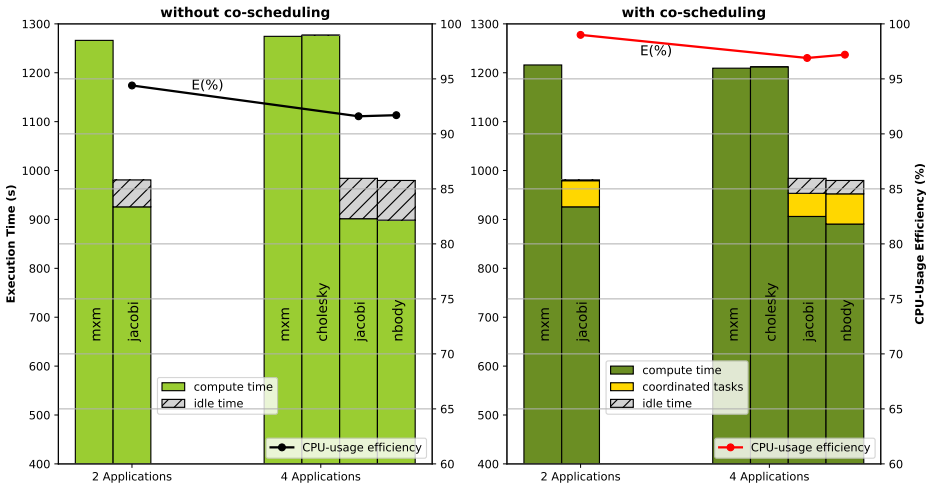


Figure 6.5.: An experiment of co-scheduling tasks across different applications.

in *App*.1, while only task results are sent back when the execution is done on the remote side, *App*.2. As a result, Table 6.8 shows an improvement with the values C'_{max} compared to C_{max}. Between *App*.1 and *App*.2, the application with idle slots can be filled up by compute tasks received from the other application.

Regarding the overhead, our approach shows that the cost is under 10% in this case, compared to the completion time of applications. With the isolated $Tcomm$, co-scheduling tasks across two applications is feasible. $Tcomm$ is asynchronously managed; therefore, we can overlap communication with computation, and the overhead has a small impact on other threads. The speedup columns show positive results, particularly in the high imbalance cases such as C4, C5, C7. Generally, these test cases are considered as reproducible test cases for co-scheduling tasks between two applications. When one is busier with more compute tasks, the other can share available tasks.

In Experiment 2, we still use two types of `MxM` compute tasks accounting for two `MxM` applications running simultaneously. One application has I/O tasks representing idle slots, where we can vary the length of idle slots. This experiment evaluates how many tasks can be migrated to fill the idle slots for co-scheduling. Furthermore, it shows how much improvement we can obtain. The results are shown

in Figure 6.4 with two figures: one on the left indicates the experiments without co-scheduling, and one on the right indicates the experiments with co-scheduling. In each figure, the x-axis shows the proportion of idle slots over the completion time, from ≈ 10% to ≈ 30%. The y-axis is a double axis, where the left one denotes application execution time or completion time, the right one denotes CPU usage efficiency (in percentage %). The CPU usage efficiency is calculated by $E = \frac{\sum CPU_{\texttt{compute\_time}}}{C_{max} \times \texttt{num\_cores}}$, where the sum means all computation time measured by all CPU cores, C_{max} is the completion time in parallel, and `num_cores` is the number of used CPU cores.

Principally, the ideal CPU utilization efficiency is 100%, but we cannot reach this maximum due to idle slots. Therefore, co-scheduling tasks can reduce idle slots in *App.*1 and increase the CPU usage efficiency in *App.*2. In Figure 6.4 (left), the area of idle in *App.*1 is marked with the color in grey. In Figure 6.4 (right), we can see that this area of *App.*1 is filled with the execution time of migrated tasks from App.2 (denoted by `coordinated tasks`). Corresponding to the length of idle slots, the number of offloaded tasks is estimated by the proposed algorithm in Chapter 4, Section 4.4.

In Experiment 3, we perform co-scheduling on different micro-benchmarks representing tasks from different applications. The results are shown in Figure 6.5 also with two figures, where one on the left side indicates the experiments without co-scheduling, while one on the right side indicates the experiments with co-scheduling. Each figure highlights two cases, a case with two applications (`MxM` and Jacobi) and a case with four applications running simultaneously (`MxM`, Jacobi, Cholesky, and Nbody). The x-axis of each figure denotes the case labels associated with the number of involved applications. Similar to Figure 6.4, the y-axis (left) denotes completion time in seconds (C_{max}), while the y-axis (right) denotes CPU usage efficiency in %. The length of stacked bars in Figure 6.5 shows the length of C_{max}, where the grey area with stripes indicates the accumulated time of idle slots. The yellow area indicates execution time occupied by another application's remote tasks (migrated tasks). The value of line charts indicates CPU usage efficiency (higher is the better), following the y-axis on the right side.

Note that the results in both figures are the average results collected by at least 5 times runs per experiment. Thereby, the height of stacked bars corresponding to the cases of without and with co-scheduling cannot be 100% equal due to noise at system and runtime. As a result, the applications with more compute tasks can share computation slots with the others, e.g., tasks in `MxM` and Cholesky are shared to coordinate with tasks in Jacobi and Nbody. The idle slots are replaced with remote task execution. After execution, the results of remote tasks are sent back to the processes of the original applications. The line chart in this experiment shows an average of 5% in CPU usage efficiency compared to the cases without co-scheduling tasks.

Our co-scheduling method can work with task-based parallel applications when we know a given distribution of tasks before execution. In the case, if tasks are generated dynamically and unpredictably, this co-scheduling method might have some risks. Due to abnormal behaviors, automatic task generation makes it difficult to manage the movement of tasks among different applications. Nevertheless, our work shows that emerging task-based parallel models can facilitate parallel execution via task abstraction. With the co-scheduling scheme, we can support portability in

- Predicting idle slots as well as task execution time.
- Offloading tasks proactively from a busy process to an idle one, following a co-scheduling algorithm.
- Getting benefits in hiding idle and increasing computing resource usage by exploiting the repeated behaviors of scientific simulations in HPC.

7. Conclusions and Future Work

Load balancing is a classic problem in HPC, which largely depends on a specific context and an imbalance cause. Our work focuses on the imbalance of task-based parallel applications when running in distributed memory systems.

The context can be determined when our application has a number of compute tasks assigned to typical execution units called processes. With today's multicore computing architectures and task-based parallel programming models, a process can be mapped to a multicore processor, in which we can create multiple threads mapped to cores where tasks are executed.

Regarding the imbalance, the main cause is performance slowdown that occurs on some processes, causing a slower execution than usual. Moving tasks from slow to fast processes is the only way to balance the load. However, the challenge is communication overhead since task migration is costly. This requires an appropriate approach to migrate as many tasks as appropriate from/to a potential process.

Through the research process, we obtained some key points:

- The classic approach suitable for our context is work stealing, which is widely studied in shared memory systems. However, communication in distributed memory systems hinders this approach because stealing a task can be time-consuming. Therefore, communication overhead can limit the number of tasks that can be stolen.
- An improvement of work stealing in distributed memory is reactive load balancing, which is a state-of-the-art approach. Reactive load balancing leverages task-based programming models and multicore architectures to perform reactive task migration. Unlike the pull-oriented mechanism of work stealing, the reactive approach uses a dedicated thread in each process to continuously check the execution status. Following that, we can speculatively detect an imbalance earlier, and tasks can be migrated earlier. However, this approach is quite risky in the case of high imbalance because reactive task migration is only based on the most current execution status at a time and is quite speculative. The number of times migrating incorrect tasks can occur frequently in some cases, such as when the number of tasks is large and the load difference is high.
- The factors affecting dynamic load balancing include both application and system characteristics. In particular, an application can be characterized through the number of tasks, execution scale (e.g., the number of compute nodes and processes), data size of tasks. The system's characteristics include communication overhead, performance variability.
- Both work stealing and reactive load balancing can be slow at runtime due to their characteristics. For instance, work stealing starts stealing tasks when one of the processes is idle, while reactive load balancing operations are quite speculative.
- An important point about dynamic load balancing in general and about our context in particular is the lack of load information. This is understandable since most current approaches are based on little information, such as the execution status inferred from the queue length of each process.

The key points above motivate our contributions, including:

1. A new performance model: We show that an appropriate performance model is essential to analyze the operations and limitations of the current approaches.
2. A new proactive load balancing approach: We show a new idea called "proactive load balancing", where the load knowledge can be generated at runtime through online load prediction.

Our model is based on discrete-time modeling towards simulating reactive load balancing behavior through the operations, such as monitoring execution status, exchanging status, checking imbalance, and offloading tasks. These are continuously and spontaneously performed at runtime. To facilitate analyzing the performance model, we combine:

- the operations of monitoring, exchanging, and checking into a balancing cost parameter, which implies an average cost of balancing operations.
- the operations of offloading tasks into a task migration cost parameter, which is affected by delay time in distributed memory.

Most previous performance models typically focus on the task migration cost parameter because transmission time in distributed memory systems is high. However, our proposed model shows that the cost of balancing operations also keeps a high impact. Although each operation costs insignificantly, a large number of operations can produce a large impact. When the number of tasks and the imbalance are high, these balancing operations lead to large delays before a task can be migrated.

Work stealing and reactive load balancing work without prior knowledge at runtime; hence, the balancing operations are performed continuously and repeatedly. Only relying on the execution status at a certain time to decide task migration is not enough. This perspective motivates our proactive approach. The load prediction helps determine imbalance levels, which processes are overloaded and underloaded. Following that, we can better estimate the number of tasks that should be migrated. We confirm the benefits of proactive load balancing through two methods:

- Feedback task offloading
- ML-based task offloading

Our implementation is designed as a plugin tool of a task-based programming framework named Chameleon. Through the experiments on microbenchmarks and a real application called Sam(oa)^2, we show a speedup improvement on average between 1.7× - 3.5× in significant imbalance cases.

Furthermore, our approach supports a new idea: co-scheduling tasks across multiple applications. The scope of scheduling tasks is not only in a single application but also in multiple applications running simultaneously. We can share tasks across multiple applications to make them better in using resources and balancing load.

Nevertheless, our work has limitations related to the proposed performance model and the proactive load balancing approach. This implies future research directions that are shown in the following part.

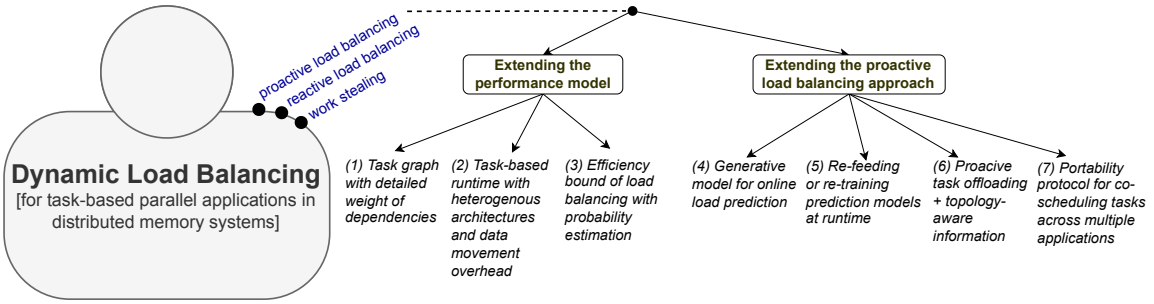


Figure 7.1.: An overview of future work for further research directions.

In the future, the following potential work can be considered. Figure 7.1 shows seven research directions based on two branches:

- Extending the performance model: Our model revolves around the influence parameters, including the number of tasks (T), processes (P), performance slowdown factor ($Slow$), execution speed of a process (S_P), overhead in balancing operations ($O_{\text{balancing}}$), and delay time in task migration (d). However, other factors and aspects should be oriented in the future, following the sub-branches (1), (2), (3).
- Extending the proactive load balancing approach: Our approach exploits one core in a multicore processor to dedicate a thread for communication and load balancing. When this thread can be used to perform different things at runtime, such as load prediction and communication between different applications running at the same time, then we can improve it following the sub-branches (4), (5), (6), (7).

The details of each sub-branch are as follows:

(1) Task-based applications with dependencies: implies a graph of tasks. Assuming the edges in the graph represent waiting time (weight) among tasks, the effect of these weights is challenging task migration for dynamic load balancing.

(2) System with heterogeneous architectures: implies the problem of modeling heterogeneous tasks and different communication overhead in task migration.

(3) Using probability to estimate the limit of task migration in reactive and proactive load balancing: implies a mathematical proof based on probability to analyze the limit of task migration.

(4) Generative model for online load prediction: indicates a generative model for predicting load automatically. A generative model can reduce user's effort in guiding and collecting influence features at runtime.

(5) Re-feeding and re-training prediction model at runtime: indicates the methods that we can use to perform auto-feeding and auto-training for online load prediction.

(6) Proactive load balancing + topology information: indicates an integration of topology information with proactive task offloading. "Proactive" provides an appropriate number of tasks and processes to perform task migration. However, if the knowledge of hardware topology can support information about which process is fast in communication, then we can improve migration strategies further.

(7) Portability for co-scheduling tasks across multiple applications: indicates a generalized protocol in distributed memory systems, which supports enabling co-scheduling tasks in a simplified method.

All in all, we conclude that optimizing dynamic load balancing in distributed systems requires consideration between strategy and time. Strategy concerns passive and active, operations and their cost, while time is associated with tasks and the cost of migrating tasks. With current computing architectures and new programming models, such as task-based programming, our approach shows feasibility and efficiency. In particular, we leverage machine learning as a load-balancing support tool. For a long-term vision, this work is compatible with automatic tuning as well as automatic scheduling for high-performance applications based on human rules and policies.

Intentionally left blank.

Appendix A.

Supplement – Performance Modeling

A.1. Related Performance Models for Work Stealing

In this appendix, we summarize the previous performance models for work stealing. The first model is work stealing without communication latency [Tch+10] [TGT13]. The second model is also work stealing but with latency proposed by [Gas+21] in 2021.

The main purpose of performance modeling is to show an upper bound of balancing efficiency, such as the number of tasks that can be migrated. Extended from the original work [BL99], Tchiboukdjian et al. have proposed a good model for work stealing without latency since 2010 [Tch+10] [TGT13].

To summarize the related performance models, we illustrate an example of work stealing behaviors in Figure A.1. The x-axis shows the progress over execution time along with four processes indexed from P_0 to P_3. The triangles represent stealing operations at a time. This is an ideal case for load balancing when the overhead of sending and receiving tasks is almost instantaneous. This example is known as task execution in shared memory systems. As we know, in work stealing, idle processes try to steal tasks from busy processes. Communication is assumed to be very fast in sending and receiving tasks.

To demystify the related models, we summarize the terminologies and notations that their authors used in Table A.1; comparing to the notations used in this thesis. We try to address them separately to avoid a conflict. For example, this thesis formulates a given distribution of T on P processes, where each holds a subset of T_i tasks. In the related performance models, the authors use w_i to indicate the number of tasks in process P_i [TGT13]. We try to make this consistent by using T_i as the number of tasks in process P_i. In the following sections, there might be some additional notations about the queue information, e.g., Q_i represents the queue status denoted by the values of the number of tasks. Mapping them to the time progress, this may have the field of time (t). For instance, $T_i(t)$ means the number of tasks in process P_i at time t, or $T(t)$ without i means the total number of tasks (for all processes) at time t. The text might re-mention these notations in some specific paragraphs when we want to show further explanation.

A. A Tight Analysis of Work Stealing [Tch+10; TGT13]

In [TGT13], Tchiboukdjian et al. considered a parallel platform with P processes. At time t, $T_i(t)$ represents the amount of works[1] on process P_i. Compared to $t = 0$, $T_i(t)$ will be decreased by time progress. For example, at $t = 0$ the workload is distributed on each process, $T_i(t_0) = 100$ means having 100 tasks before running. Then, at $t = 10$ we assume 10 tasks have been done on process P_i, and $T_i(t_{10})$ would be 90 as the remaining tasks. Taking Figure A.1 as an example, at $t = 10$ process P_3 has done 10 tasks, and its queue remains 90 tasks, $T_0(t_{10}) = 90$. In contrast, process P_2 is faster

[1] Works are considered as tasks. Therefore, they might be used interchangeably.

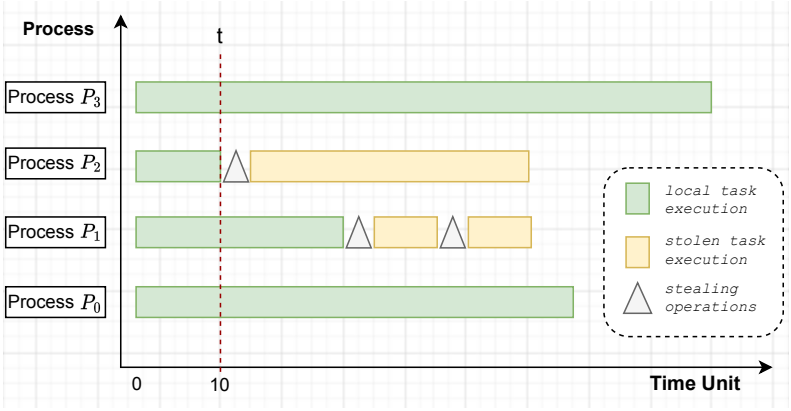


Figure A.1.: An illustration of work stealing without latency.

and it has done all tasks at $t = 10$, therefore, $T_2(t_{10}) = 0$. In general, the execution behavior can be simplified as follows, where a processor also refers to a process.

- when $T_i(t) > 0$, Processor i is active and execute tasks, $T_i(t+1) \leq T_i(t)$.
- when $T_i(t) = 0$, Processor i is idle and intends to steal tasks from a random processor j.

If work stealing is applied, some tasks are migrated, then $T_i(t)$ will increase or decrease, depending on how many steal requests and tasks are performed at a time. Tchiboukdjian et al. [Tch+10] [TGT13] assumed that between time t and $t+1$, there are $P-\alpha(t)$ steal requests, P denotes the total number of processes and $\alpha(t)$ denotes the number of active processes, $\alpha(t) \in [0, m]$. When $\alpha(t) = 0$, it means all queues are empty as well as the execution is complete. Corresponding to the sucess steal requests, process P_j transfers an amount of works to process P_i and $T_i(t+1) + T_j(t+1) \leq T_j(t)$. Similarly, when $\alpha(t) = 0$, the execution terminates if $\forall i \in P, T_i(t) = 0$. At time t, the total number of tasks of all processes is denoted by $T(t) = \sum_{i=1}^{P} T_i(t)$. After that, process P_2 steals some tasks from the others to help balance the load. The triangle shows stealing operations, and there is obviously stealing overhead in practice. However, we show this model with an assumption about no latency, in which the use cases might be considered in shared memory.

In the baseline without work-stealing, the total execution time depends on the process with a maximum load, $C_{max} = max_{i \in P} T_i(t_0)$. When we apply work stealing, C_{max} can be reduced. Following that, the main question is:

What is the upper bound of makespan when work stealing is applied?

The work [Tch+10] has proposed a potential function $\Phi(t)$ to model the performance of work stealing by studying the potential decrease. Assuming that tasks are unit independent, $\Phi(t)$ is defined by

$$\Phi(t) = \sum_{i=1}^{P} (T_i(t) - \frac{T(t)}{P})^2 = \sum_{i=1}^{P} T_i(t)^2 - \frac{T(t)^2}{P} \tag{A.1}$$

where, the potential represents the load imbalance of execution. For example, as expected at time t the average load among P processes is $\frac{T(t)}{P}$. If we sum the difference at time t between the load value of each process and the average load value, the result shows the load difference level.

Therefore, the decrease of $\Phi(t)$ depends on the number of steal requests and execution scenario. The above has mentioned an estimated formula for the number of steal requests $(P - \alpha(t))$. $\alpha(t)$ is a complicated random process that chooses the number of active processors at each step. Such an expectation in work-stealing, the more work requests it creates, the more the potential will decrease. The performance analysis model is performed in three steps:

No.	Notations	Description	Note
1	T	the number of tasks indexed $\{0, ..., (T-1)\}$	In the related works, people used W instead of T
2	P	the number of processes, $\{0, ..., (P-1)\}$	The related works used m identical processors instead of P
3	T_i	the set of assigned tasks in process P_i	The related works used w_i instead of T_i
4	L_i	the total load of process P_i	
5	w_i	the wallclock execution time of a task, i.e., task i	
6	W_i	the wallclock execution time of process P_i	
7	S_{P_i}	processing speed (performance model) of process P_i	
8	$Slow_i$	slowdown coefficient of process P_i at runtime	
9	W_{par}	the total wallclock execution time (makespan)	Also called completion time (C)
10	R_{imb}	imbalance ratio	
11	λ	communication or network latency	
12	d	delay or transmission time	
13	B	network bandwidth	
14	$O_{ij}(t)$	the number of offloaded tasks from process P_i to P_j	The authors in [Tch+10] used $R(t)$ as the number of task requests in global after time t

Table A.1.: Used notations in the thesis comparing to the notations from related works.

1. Define $\Phi(t)$.

2. Compute the expected decrease of $\Phi(t)$ between time step t and $t+1$, $\Delta\Phi(t) \stackrel{def}{=} \Phi(t) - \Phi(t+1)$. However, at a specific time t, how can we estimate the decrease? The authors define another term, named $\delta_i^k(t)$, to compute the expected decrease,

$$\sum_{i=1}^{P} \sum_{k=0}^{P-1} E[\delta_i^k | i \text{ receives } k \text{ requests}] P\{i \text{ receives } k \text{ requests}\} \tag{A.2}$$

, where the first sum goes through all P processes and the second sum goes from 0 to $(P-2)$ as the maximum requests a process can get at a time. $E[X|Y]$ denotes the expectation of X knowing Y. Applying this to the example in Figure A.2 we have

$$E[\Delta\Phi(t)] = \sum_{i=0}^{3} \sum_{k=0}^{3} E[\delta_i^k | i \text{ receives } k \text{ requests}] P\{i \text{ receives } k\text{requests}\} \tag{A.3}$$

Following that, the authors showed that there exists a function called $h(\alpha) \in (0; 1]$ such that $E[\Delta\Phi(t)|\Phi(t) = \Phi, \alpha(t) = \alpha] \geq h(\alpha)\Phi$.

3. The work [Tch+10] obtained a bound on the expected number of steal requests $E[R]$ (R is the number of steal requests), and the expected makespan $E[C_{max}]$ can be further obtained from $E[R]$.

Tchiboukdjian et al. [Tch+10] concluded that the expected number of steal requests R until $\Phi(t) \leq 1$ is bounded by $E[R] \leq \lambda \cdot P \cdot log_2\Phi(0)$, where λ in this case is $max_{1\leq\alpha\leq P-1}\frac{P-\alpha}{-P\cdot log_2(1-h(\alpha))}$ and $\Phi(0)$ is the potential at $t = 0$. At the end, the expected value of C_{max} will be bounded by

$$E[C_{max}] \leq \frac{T}{P} + \frac{2}{1 - \log_2(1 + \frac{1}{\epsilon})} \log_2 W + 1 \tag{A.4}$$

Before [Tch+10; TGT13], some previous works also attempted to find an upper bound. One of the original studies from Blumofe and Leiserson [BL99] showed that the efficiency of work stealing is bounded by $E(C_{max}) \leq \frac{T}{P} + O(D)$, where D is the length of the critical path in the case of dependent tasks (a task graph). The analysis is further improved by [ABP01] using potential functions. The authors used an amortization argument based on a potential function that decreases the work stealing algorithm processes. The idea is to divide the execution into phases and show that the potential decreases by at least a constant fraction with a constant probability. For the case of varying the speed of P processors or heterogeneous processors, Bender and Rabin [BR02] provided a new analysis of an old scheduling algorithm called *maximum utilization scheduler*. The authors showed a given context for P processors with speed π_i steps/time. These studies are constrained in the context of only one source of tasks, which could not easily model the basic case of independent tasks, and communication overhead is not explicitly counted.

B. A Tighter Analysis of Work Stealing with Latency [Gas+21]

For a performance model with communication latency, Nicolas Gast et al. [Gas+21] proposed a new analysis model on how communication latency impacts stealing operations in terms of task-parallel applications. The model has been created for distributed memory clusters with P identical processors. The latency value is denoted by λ. This research is inherited from the previous work [TGT13] and aimed at an upper bound for an expected makespan.

From the previous study [TGT13], the authors also make a consistent assumption about work stealing algorithms as follows:

- The total number of processes is defined as P identical processors.
- $T_i(t)$ denotes the amount of tasks on processor i at time t, while the total tasks at time t is $T(t) = \sum_{i=1}^{P} T_i(t)$.
- A task corresponds to a unit of execution time.
- Regarding communication latency, all operations take the same $\lambda \in N$ time unit.
- Single Task Transfer: a processor can send some tasks to at most one processor at a time.
- Steal Threshold: if the victim has less than λ task units to execute, the stealing request will be failed.
- Task to be stolen: the victim sends to the thief half of its tasks at a time. For example, the total tasks of process P_i at time t is $T_i(t) = \frac{T_i(t-1)-1}{2}$.

Figure A.2 demonstrates a scenario of work stealing with latency. The x-axis represents the time progress of execution with four involved processes. Process P_2 is assumed idle and sends a steal request to process P_3. Since process P_3 accepts, task is sent from P_3 to P_2. Nicolas Gast et al. [Gas+21] name λ as the latency, and one stealing action takes a round-trip time 2λ. This occurs similarly between process P_1 and P_0.

As the round-trip time of 2λ and the total amount of tasks T on P processes, Nicolas Gast et al. introduce a straightforward bound of makespan as A.5.

$$\begin{aligned} P{\cdot}C_{max} &\leq T + 2\lambda \cdot \#\text{task requests} \\ \Leftrightarrow C_{max} &\leq \frac{T}{P} + 2\lambda \cdot \frac{\#\text{task requests}}{P} \end{aligned} \tag{A.5}$$

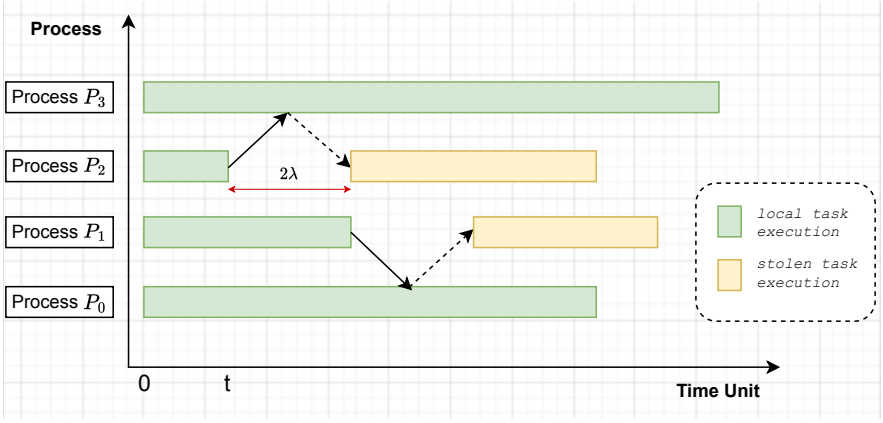


Figure A.2.: An illustration of work-stealing with latency effect.

However, to be closer to an optimal bound, this work [Gas+21] approached a potential function bounding on the number of work stealing requests[2]. The authors reconsidered the time division as periods of duration λ to analyze the impact of latency. To not abuse notations, we use $T_i(k)$ and $s_i(k)$ to denote the current number of tasks and the number of stolen tasks from process P_i at time k; then the quantities will be $T_i(k\lambda)$, $s_i(k\lambda)$. In the interval $(\lambda(k-1), \lambda k]$, the total number of incoming work stealing requests is defined by $r(k) = \sum_{i=1}^{\lambda} R((k-1)\lambda + i)$, where $0 \leq r(k) \geq P$. The probability, that a process receives ≥ 1 requests in the interval $(\lambda(k-1), \lambda k]$, is $q(r(k))$.

Nicolas Gast et al. [Gas+21] showed the analysis of the potential decrease, which is defined in Equation A.6 at time-step $k\lambda$.

$$\Phi(k) = 1 + \frac{1}{\lambda^2} \sum_{i=1}^{P} (T_i(k)^2 + 2s_i(k)^2) \tag{A.6}$$

where, $\Phi(k)$ gets through all processes $\in P$, $T_i(k)$ and $s_i(k)$ represents the number of remaining tasks as well as the number of stolen tasks in process P_i at time k. Let $\Phi(0)$ be the potential at time t_0 and τ be the first time step at which the potential reaches 1. Then, Nicolas Gast et al. proved that the number of incoming steal requests until τ, $R = \sum_{k=0}^{\tau=1} r(k)$ satisfies:

$$\begin{aligned} &E[R] \leq P\gamma \log_2 \Phi(0) \\ &\mathbb{P}[R \leq P\gamma(\log_2 \Phi(0) + x)] \leq 2^{-x} \end{aligned} \tag{A.7}$$

Where, $E[R]$ is the expected number of total incoming steal requests and $\mathbb{P}$ indicates the probability when $R \leq P\gamma(\log_2 \Phi(0) + x)$, and γ is a constant such that $\gamma < 4.03$. Applying to C_{max}, the study concluded as Equation A.8 shows.

$$\begin{aligned} &E[C_{max}] \leq \frac{T}{P} + 4\lambda\gamma \log_2 \frac{P}{\lambda} + 2\lambda\gamma \\ &\mathbb{P}[C_{max} \geq \frac{T}{P} + 4\lambda\gamma \log_2 \frac{P}{\lambda} + x] \leq 2^{\frac{-x}{2\lambda\gamma}} \end{aligned} \tag{A.8}$$

Further discussion: The related models above are relevant in terms of work stealing with or without latency. However, latency λ is considered as a constant, and the number of steal requests must contribute relatively, such as task execution time. These constraints might be limited by the cases that task data sizes differ and transmission time considerably impacts the efficiency of stealing operations in practice. In contrast, this thesis analyzes the performance in a different direction. We introduce

[2]Work-stealing requests mean the number of requests for stealing tasks. Therefore, it is also called #task requests shown in Equation A.5 or steal requests

a model associated with transmission time (delay time) in HPC. The delay happens when tasks are migrated in distributed memory. First, the delay values depend on the size of task data in movement and the current status of interconnection, e.g., latency and bandwidth. Second, we show that this challenge can negatively impact the decision time of taking stealing or balancing actions. This is why it delays dynamic load balancing in distributed memory.

Appendix B.

Supplement – Implementation in C++

B.1. Artifact Description

Appendix C.

Supplement – Thesis Structure as a Taskflow

C.1. Structuring the thesis as a task flow

This Appendix shows the structure of this thesis outlined as a task flow, where the thesis's main points, keywords, or key paragraphs are described as a task. There are five figures shown here, including:

- Figure C.1: the taskflow of Chapter 1 - Introduction.
- Figure C.2: the taskflow of Chapter 2 - From work stealing to reactive load balancing, associated with related works.
- Figure C.3: the taskflow of Chapter 3 - Performance modeling and analysis.
- Figure C.4: the taskflow of Chapter 4 - A proactive approach for dynamic load balancing.
- Figure C.5: the taskflows of Chapters 5, 6, 7 - Corresponding to proof of concepts, evaluation, and conclusions in this thesis.

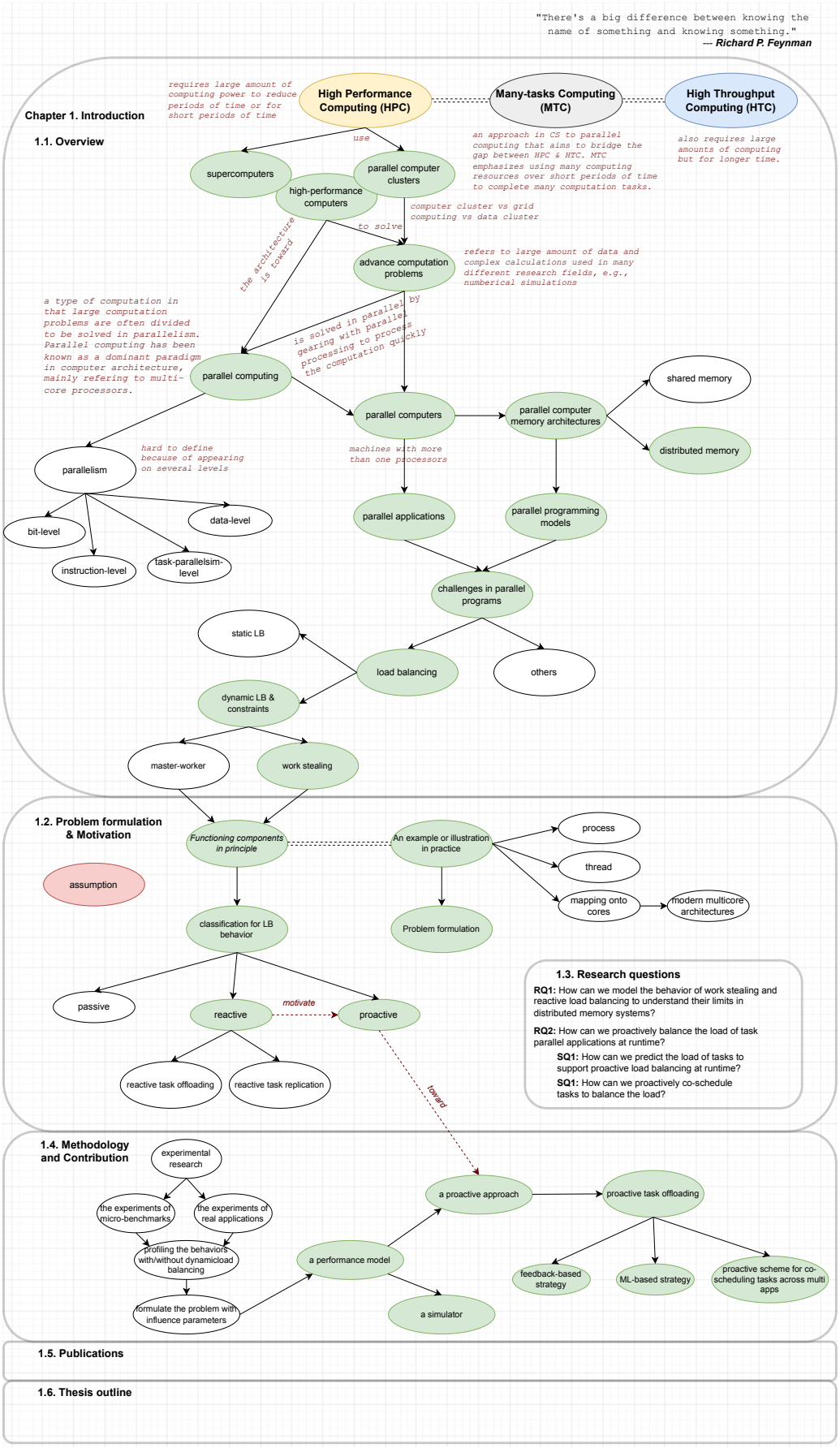


Figure C.1.: Terms and key points in Chapter 1 represented as a taskflow.

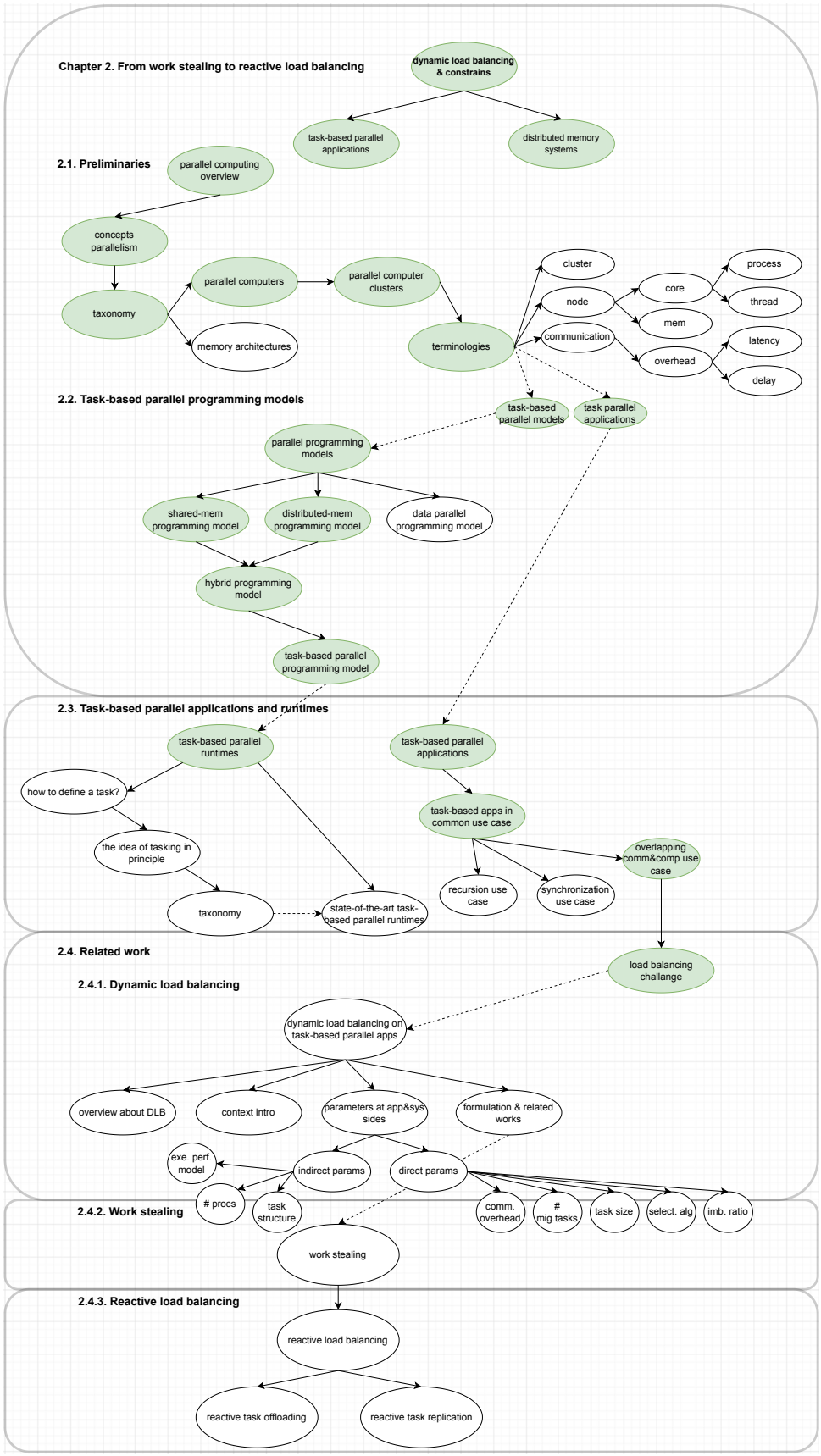


Figure C.2.: Terms and key points in Chapter 2 represented as a taskflow.

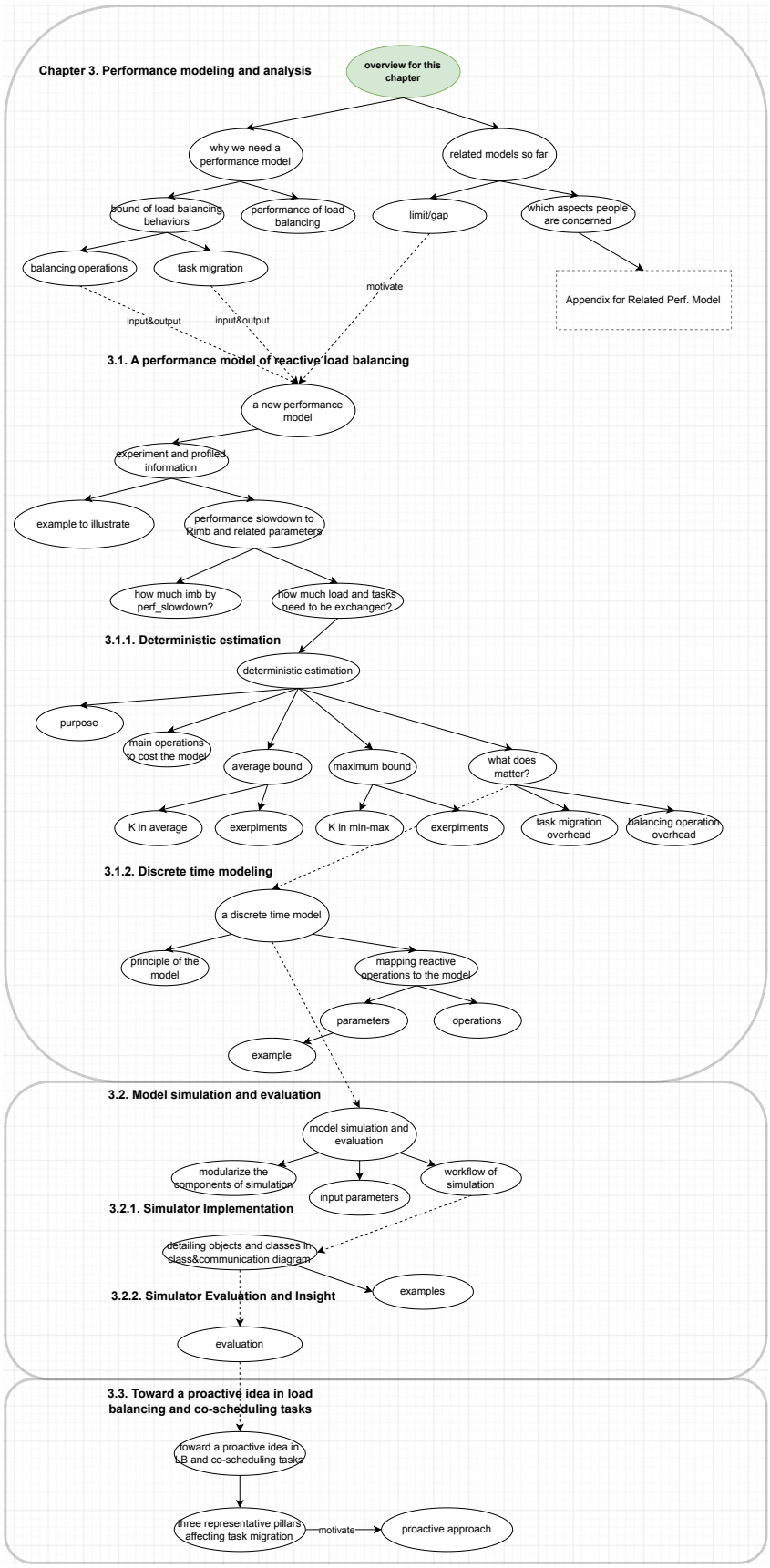


Figure C.3.: Terms and key points in Chapter 3 represented as a taskflow.

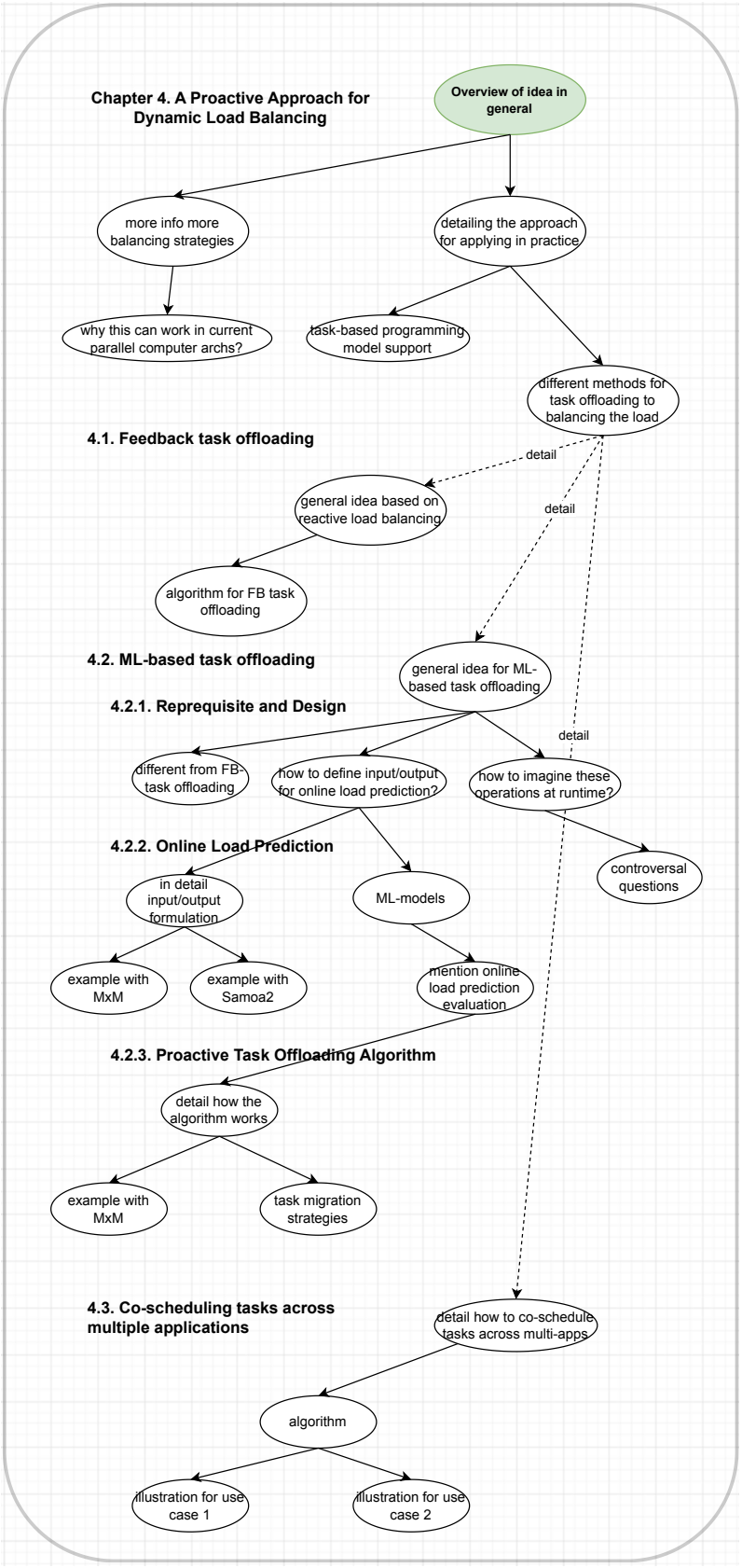


Figure C.4.: Terms and key points in Chapter 4 represented as a taskflow.

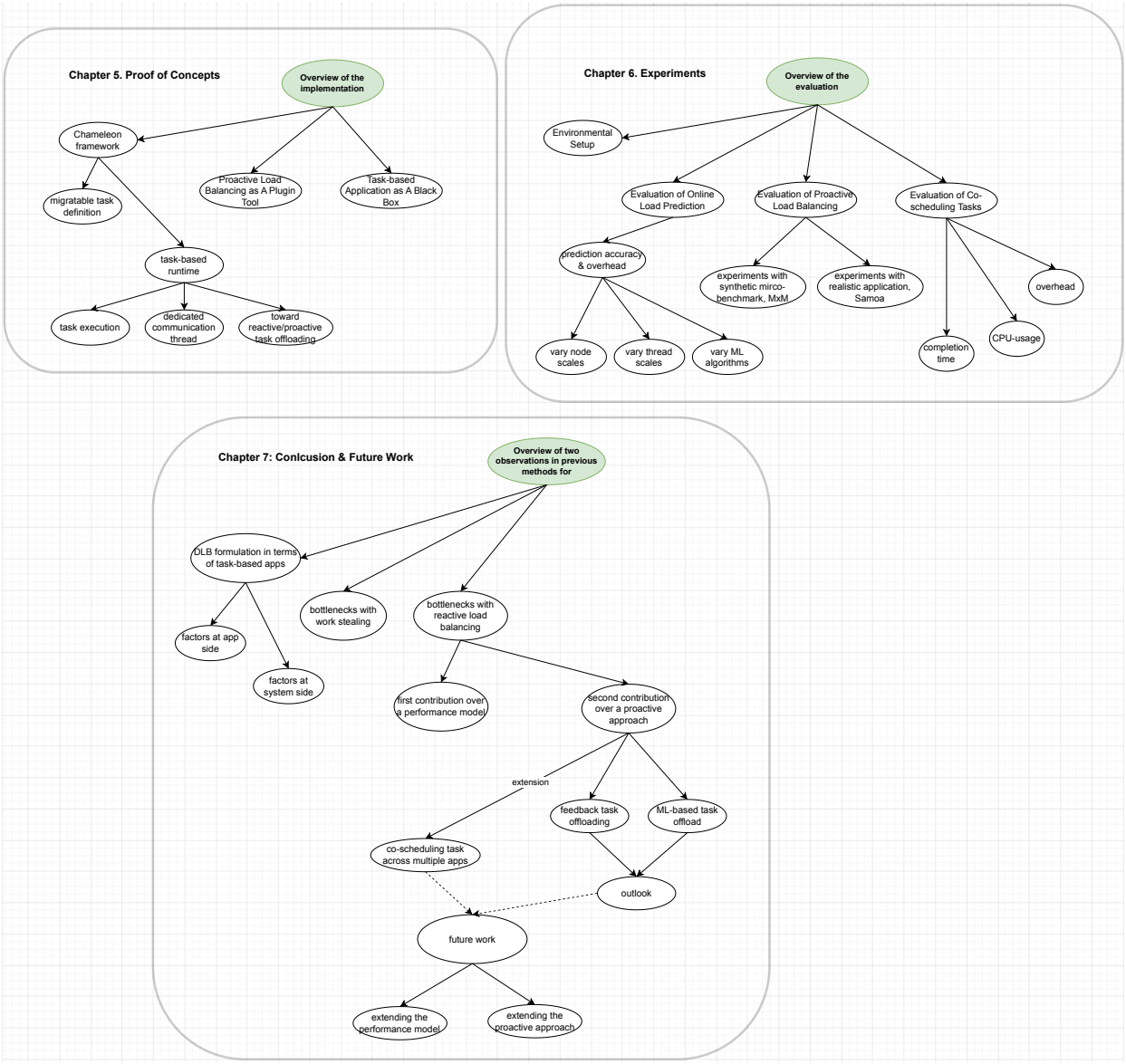


Figure C.5.: Terms and key points in Chapter 5, 6, 7 represented as a taskflow.

Acronyms and Abbreviations

AMR Adaptive Mesh Refinement 123,
Glossary: Adaptive Mesh Refinement

BSP Bulk Synchronous Parallel 123,
Glossary: Bulk Synchronous Parallel

CRediT Contributor Roles Taxonomy 123,
Glossary: Contributor Roles Taxonomy

DLB Dynamic Load Balancing 123,
Glossary: Dynamic Load Balancing

FLOPS FLoating point Operations Per Second 123,
Glossary: FLoating point Operations Per Second

HPC High Performance Computing 123,
Glossary: High Performance Computing

HPX High Performance ParalleX 123,
Glossary: High Performance ParalleX

I/O Input/Output 123,
Glossary: Input/Output

MD Molecular Dynamics 123,
Glossary: Molecular Dynamics

MPI Message Passing Interface 123,
Glossary: Message Passing Interface

MPI+X MPI+X 123,
Glossary: MPI+X

Glossary

Adaptive Mesh Refinement In numerical simulation or analysis, Adaptive Mesh Refinement is a method of adapting the accuracy of a solution dynamically. When solutions are calculated numerically, the computation domain are often limited to pre-determined quantified grids or meshes. 123

Bulk Synchronous Parallel Bulk synchronous parallel (BSP) is considered as a bridging model for designing parallel algorithms. 123

Contributor Roles Taxonomy CRediT (Contributor Roles Taxonomy) is a high-level taxonomy, including 14 roles, that can be used to represent the roles typically played by contributors to research outputs (https://credit.niso.org/). 123

Dynamic Load Balancing DLB can be understood as a problem or a solution. The problem is defined by parallel execution on multiple processors/nodes, where their load might be unbalanced at runtime. DLB solutions also refer to the approaches to balancing the load. 123

FLoating point Operations Per Second Floating point operations per Second (FLOPS) is the metric used for issued 64 bit floating-point operations per second. Precision used is 64 bit in accordance with IEEE754. SI-prefixes do apply. 123

High Performance Computing High-Performance Computing (HPC) refers to using the computing power of supercomputers or clusters to solve advanced computation problems. Additionally, the current trend also towards the challenges related to big data processing. The main target is to make the challenges of science and engineering problems computable in a reasonable time. 123

High Performance ParalleX The HPX runtime system is the runtime system for the parallel execution model ParallelX. 123

Input/Output I/O in this work refers to any information transfer from the main computational devices, CPU and Memory, to a remote system, such as other nodes of other clusters or persistent memory (storage) located outside the node. 123

Message Passing Interface MPI is the standard interface called message passing interface for process communication in HPC systems. Data is moved from the address space of one process to another process through cooperative operations. This standard has gone through a number of revisions and the most recent version is MPI-4.x. 123

Molecular Dynamics The term indicates molecular dynamic simulations in HPC. It is considered as a computer simulation method for analyzing the physical movements of atoms and molecules. 123

MPI+X It is defined as a hybrid programming model between MPI process + thread. Or we have known that X can be OpenMP threads, Pthread, or other kind of threads. 123

Bibliography

The Bibliography is split into four sections:

- Published Resources are labeled by `<author prefix>` + `<year>`.
- Web Resources are labeled by an `<author>/<organizaiton>` Identifier (without year).
- Meetings/Seminars/Workshops are labeled using `<Venue Identifier>` + »'« + `<year>`.

Published Resources

[Acu+14] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, et al. "Parallel Programming with Migratable Objects: Charm++ in Practice". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'14. New Orleans, Louisana, 2014, pp. 647–658. DOI: https://doi.org/10.1109/SC.2014.58.

[AMK16] Bilge Acun, Phil Miller, and Laxmikant V. Kale. "Variation Among Processors Under Turbo Boost in HPC Systems". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS'16. Istanbul, Turkey: ACM, 2016. DOI: https://doi.org/10.1145/2925426.2926289.

[Adl+95] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. "Parallel Randomized Load Balancing". In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '95. Las Vegas, Nevada, USA: ACM, 1995, pp. 238–247. DOI: https://doi.org/10.1145/225058.225131.

[ABP01] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Theory of Computing Systems* 34.2 (2001), pp. 115–144. DOI: https://doi.org/10.1007/s00224-001-0004-z.

[Ate+19] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. "HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP'19. Kyoto, Japan: ACM, 2019, pp. 1–10. DOI: https://doi.org/10.1145/3337821.3337907.

[Aug+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198. DOI: https://doi.org/10.1002/cpe.1631.

[Ayg+09] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. "The Design of OpenMP Tasks". In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2009), pp. 404–418. DOI: https://doi.org/10.1109/TPDS.2008.105.

[Bak+18] Seonmyeong Bak, Harshitha Menon, Sam White, Matthias Diener, and Laxmikant Kale. "Multi-Level Load Balancing with an Integrated Runtime Approach". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID'18. Washington, DC, USA, 2018, pp. 31–40. DOI: https://doi.org/10.1109/CCGRID.2018.00018.

[BB99] Mark Baker and Rajkumar Buyya. "Cluster Computing at a Glance". In: *High Performance Cluster Computing - Architectures and Systems*. Prentice Hall PTR, 1999, pp. 3–47. ISBN: 0130137847. URL: https://dl.acm.org/doi/10.5555/520257.

[BK18] Saman Barghi and Martin Karsten. “Work-Stealing, Locality-Aware Actor Scheduling”. In: *2018 IEEE International Parallel and Distributed Processing Symposium*. IPDPS’18. Vancouver, BC, Canada, 2018, pp. 484–494. DOI: https://doi.org/10.1109/IPDPS.2018.00058.

[Bau14] Michael Edward Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD Thesis. Stanford University Archives, 2014. URL: http://purl.stanford.edu/kk063hx7516.

[BR02] Michael A. Bender and Michael O. Rabin. “Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk”. In: *Theory of Computing Systems* 35.3 (2002), pp. 289–304. DOI: https://doi.org/10.1007/s00224-002-1055-5.

[Ben91] Kouider Benmohammed-Mahieddine. “An Evaluation of Load Balancing Algorithms for Distributed Systems”. PhD thesis. 1991. URL: https://etheses.whiterose.ac.uk/4395/.

[BDM09] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. “A survey of multicore processors”. In: *IEEE Signal Processing Magazine* 26.6 (2009), pp. 26–37. DOI: https://doi.org/10.1109/MSP.2009.934110.

[BB07] Markus Blatt and Peter Bastian. “The Iterative Solver Template Library”. In: *Applied Parallel Computing. State of the Art in Scientific Computing*. PARA’06. Umea, Sweden, 2007, pp. 666–675. DOI: https://doi.org/10.1007/978-3-540-75755-9_82.

[BL99] Robert D Blumofe and Charles E Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *Journal of The ACM (J. ACM)* 46.5 (1999), pp. 720–748. DOI: https://doi.org/10.1145/324133.324234.

[Bok81] Shahid H. Bokhari. “On the Mapping Problem”. In: *IEEE Transactions on Computers* C-30.3 (1981), pp. 207–214. DOI: https://doi.org/10.1109/TC.1981.1675756.

[Bok87] Shahid H. Bokhari. *Assignment Problems in Parallel and Distributed Computing*. Springer New York, NY, 1987. ISBN: 978-1-4613-2003-6. DOI: https://doi.org/10.1007/978-1-4613-2003-6.

[Bok88] Shahid H. Bokhari. “Partitioning Problems in Parallel, Pipeline, and Distributed Computing”. In: *IEEE Transactions on Computers* 37.1 (1988), pp. 48–57. DOI: https://doi.org/10.1109/12.75137.

[Bon17] Giuseppe Bonaccorso. *Machine Learning Algorithms*. Packt Publishing, 2017. ISBN: 9781789347999. URL: https://packt.link/free-ebook/9781785889622.

[BNG92] Nicholas S. Bowen, Christos N Nikolaou, and Arif Ghafoor. “On The Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems”. In: *IEEE Transactions on Computers* 41.3 (1992), pp. 257–273. DOI: https://doi.org/10.1109/12.127439.

[Bun+10] Hans-Joachim Bungartz, Miriam Mehl, Tobias Neckel, and Tobias Weinzierl. “The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids”. In: *Computational Mechanics* 46.1 (2010), pp. 103–114. DOI: https://doi.org/10.1007/s00466-009-0436-x.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997. ISBN: 0201633922.

[CCZ07] Bradford L. Chamberlain, D. Callahan, and H.P. Zima. “Parallel Programmability and the Chapel Language”. In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: https://doi.org/10.1177/1094342007078442.

[Cha+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”. In: *ACM SIGPLAN Notices (SIGPLAN Not.)* 40.10 (2005), pp. 519–538. ISSN: 0362-1340. DOI: https://doi.org/10.1145/1103845.1094852.

[CGG14] Quan Chen, Minyi Guo, and Haibing Guan. “LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures”. In: *Proceedings of the 28th ACM International Conference on Supercomputing*. ICS’14. Munich, Germany, 2014. DOI: https://doi.org/10.1145/2597652.2597665.

[Chr+05] Anthony T Chronopoulos, Satish Penmatsa, Ning Yu, and Du Yu. “Scalable Loop Self-scheduling Schemes for Heterogeneous Clusters”. In: *International Journal of Computational Science and Engineering* 1.2-4 (2005), pp. 110–117. DOI: https://doi.org/10.1504/IJCSE.2005.009696.

[Cur+13] Ryan R. Curtin, James R. Cline, N. P. Slagle, William B. March, Parikshit Ram, Nishant A. Mehta, and Alexander G. Gray. “MLPACK: A Scalable C++ Machine Learning Library”. In: *Journal of Machine Learning Research* 14.24 (2013), pp. 801–805. URL: http://jmlr.org/papers/v14/curtin13a.html.

[Cur+23] Ryan R. Curtin, Marcus Edel, Omar Shrit, Shubham Agrawal, Suryoday Basak, James J. Balamuta, Ryan Birmingham, et al. “mlpack 4: a fast, header-only C++ machine learning library”. In: *Journal of Open Source Software* 8.82 (2023). DOI: https://doi.org/10.21105/joss.05026.

[Cyb89] George Cybenko. “Dynamic Load Balancing for Distributed Memory Multiprocessors”. In: *Journal of Parallel and Distributed Computing* 7.2 (1989), pp. 279–301. DOI: https://doi.org/10.1016/0743-7315(89)90021-X.

[DM98] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: https://doi.org/10.1109/99.660313.

[Dar01] Frederica Darema. “The SPMD Model: Past, Present and Future”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. The 8th European PVM/MPI Users’ Group Meeting. Santorini/Thera, Greece, 2001, pp. 1–1. DOI: https://doi.org/10.1007/3-540-45417-9_1.

[De +15] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. “Partitioned Global Address Space Languages”. In: *ACM Comput. Surv.* 47.4 (2015). DOI: https://doi.org/10.1145/2716320.

[DL18] Li Deng and Yang Liu. *Deep Learning in Natural Language Processing*. Springer Singapore, 2018. ISBN: 978-981-10-5209-5. DOI: https://doi.org/10.1007/978-981-10-5209-5.

[Der+15] Said Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter Atos. “The BXI Interconnect Architecture”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. Santa Clara, CA, USA: IEEE, 2015, pp. 18–25. DOI: https://doi.org/10.1109/HOTI.2015.15.

[Din+09] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. “Scalable Work Stealing”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC’09. Portland, Oregon, 2009. DOI: https://doi.org/10.1145/1654059.1654113.

[Dre+14] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. “Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.3 (2014). DOI: https://doi.org/10.1145/2641764.

[Dur+11] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. “Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures”. In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193. DOI: https://doi.org/10.1142/S0129626411000151.

[DGC05] Alejandro Duran, Marc Gonzàlez, and Julita Corbalán. “Automatic Thread Distribution for Nested Parallelism in OpenMP”. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS’05. Cambridge, Massachusetts: ACM, 2005, pp. 121–130. DOI: https://doi.org/10.1145/1088149.1088166.

[Eic+13] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. "OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis". In: *OpenMP in the Era of Low Power Devices and Accelerators*. IWOMP'13. Canberra, Australia: Springer Berlin Heidelberg, 2013, pp. 171–185. DOI: https://doi.org/10.1007/978-3-642-40698-0_13.

[EL99] Rudolf Eigenmann and David J. Lilja. "Von Neumann Computers". In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Ltd, 1999. ISBN: 9780471346081. DOI: https://doi.org/10.1002/047134608X.W1704.

[Eij10] Victor Eijkhout. *Introduction to High-Performance Scientific Computing*. 2010. URL: https://theartofhpc.com/istc.html.

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: https://doi.org/10.1109/TC.1972.5009071.

[Fre+21] Vinicius Freitas, Laércio L Pilla, Alexandre de L Santana, Márcio Castro, and Johanne Cohen. "PackStealLB: A scalable distributed load balancer based on work stealing and workload discretization". In: *Journal of Parallel and Distributed Computing* 150 (2021), pp. 34–45. DOI: https://doi.org/10.1016/j.jpdc.2020.12.005.

[GCL09] Marta Garcia, Julita Corbalan, and Jesus Labarta. "LeWI: A Runtime Balancing Algorithm for Nested Parallelism". In: *2009 International Conference on Parallel Processing*. Vienna, Austria, 2009, pp. 526–533. DOI: https://doi.org/10.1109/ICPP.2009.56.

[Gas+21] Nicolas Gast, Mohammed Khatiri, Denis Trystram, and Frédéric Wagner. "Analysis of Work Stealing with Latency". In: *Journal of Parallel and Distributed Computing* (2021), pp. 119–129. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2021.03.010.

[Gro+96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel Computing* 22.6 (1996), pp. 789–828. DOI: https://doi.org/10.1016/0167-8191(96)00024-5.

[HW80] Robert H. Halstead and Stephen A. Ward. "The MuNet: A Scalable Decentralized Architecture for Parallel Computation". In: *Proceedings of the 7th Annual Symposium on Computer Architecture*. ISCA'80. La Baule, USA, 1980, pp. 139–145. DOI: https://doi.org/10.1145/800053.801919.

[Hay+04] Majeed M. Hayat, Sagar Dhakal, Chaouki T. Abdallah, J. Douglas Birdwell, and John Chiasson. "Dynamic Time Delay Models for Load Balancing. Part II: A Stochastic Analysis of the Effect of Delay Uncertainty". In: *Advances in Time-Delay Systems*. Paris, France: Springer, 2004, pp. 371–385. DOI: https://doi.org/10.1007/978-3-642-18482-6_27.

[HIB10] Steven Hofmeyr, Costin Iancu, and Filip Blagojević. "Load Balancing on Speed". In: *ACM SIGPLAN Notices (SIGPLAN Not.)* 45.5 (2010), pp. 147–158. DOI: https://doi.org/10.1145/1837853.1693475.

[Hoq+17] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. "Dynamic Task Discovery in PaRSEC: A Data-Flow Task-Based Runtime". In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA'17. Denver, Colorado: ACM, 2017. DOI: https://doi.org/10.1145/3148226.3148233.

[Hua+21] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. "Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.8 (2021), pp. 1687–1700. DOI: https://doi.org/10.1109/TCAD.2020.3025075.

[JNW08] Bruce Jacob, Spencer W. Ng, and David T. Wang. "CHAPTER 2 - Logical Organization". In: *Memory Systems*. Morgan Kaufmann, 2008, pp. 79–115. ISBN: 978-0-12-379751-3. URL: https://doi.org/10.1016/B978-012379751-3.50004-7.

[Jam+13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer New York, NY, 2013. ISBN: 978-1-4614-7138-7. DOI: https://doi.org/10.1007/978-1-4614-7138-7.

[Kai+14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS'14. Eugene, OR, USA: ACM, 2014. DOI: https://doi.org/10.1145/2676870.2676883.

[KMW32] William Ogilvy Kermack, A. G. McKendrick, and Gilbert Thomas Walker. "Contributions to the mathematical theory of epidemics. II. The problem of endemicity". In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 138.834 (1932), pp. 55–83. ISSN: 2053-9150. DOI: https://doi.org/10.1098/rspa.1932.0171.

[Kli+20a] Jannis Klinkenberg, Philipp Samfass, Michael Bader, Christian Terboven, and Matthias S Müller. "CHAMELEON: Reactive Load Balancing for Hybrid MPI+OpenMP Task-Parallel Applications". In: *Journal of Parallel and Distributed Computing* 138 (2020), pp. 55–64. DOI: https://doi.org/10.1016/j.jpdc.2019.12.005.

[Kli+20b] Jannis Klinkenberg, Philipp Samfass, Michael Bader, Christian Terboven, and Matthias S. Müller. "Reactive Task Migration for Hybrid MPI+OpenMP Applications". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PPAM'19. Bialystok, Poland: Springer, 2020, pp. 59–71. DOI: https://doi.org/10.1007/978-3-030-43222-5_6.

[Kor15] Christopher Kormanyos. "C++ Multitasking". In: *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer Berlin, Heidelberg, 2015, pp. 199–210. ISBN: 978-3-662-47810-3. DOI: https://doi.org/10.1007/978-3-662-47810-3_11.

[Kum+94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., 1994. ISBN: 0805331700. URL: https://dl.acm.org/doi/10.5555/156619.

[Kum+16] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. "Optimized Distributed Work-Stealing". In: *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. Salt Lake City, UT, USA, 2016, pp. 74–77. DOI: https://doi.org/10.1109/IA3.2016.019.

[Kum+15] Vivek Kumar, Alina Sbîrlea, Ajay Jayaraj, Zoran Budimlić, Deepak Majeti, and Vivek Sarkar. "Heterogeneous work-stealing across CPU and DSP cores". In: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA, 2015. DOI: https://doi.org/10.1109/HPEC.2015.7322452.

[LSD19] D. Brian Larkins, John Snyder, and James Dinan. "Accelerated Work Stealing". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP'19. Kyoto, Japan: ACM, 2019. DOI: https://doi.org/10.1145/3337821.3337878.

[LBC91] S.Y. Liem, D. Brown, and J.H.R. Clarke. "Molecular dynamics simulations on distributed memory machines". In: *Computer Physics Communications* 67.2 (1991), pp. 261–267. DOI: https://doi.org/10.1016/0010-4655(91)90021-C.

[LKK12] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. "Work Stealing and Persistence-Based Load Balancers for Iterative Overdecomposed Applications". In: *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC'12. Delft, The Netherlands: ACM, 2012, pp. 137–148. DOI: https://doi.org/10.1145/2287076.2287103.

[LK87] Frank C. H. Lin and Robert M. Keller. “The Gradient Model Load Balancing Method”. In: *IEEE Transactions on Software Engineering* SE-13.1 (1987), pp. 32–38. DOI: https://doi.org/10.1109/TSE.1987.232563.

[MRB16] Oliver Meister, Kaveh Rahnema, and Michael Bader. “Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells”. In: *ACM Transactions on Mathematical Software (TOMS)* 43.3 (2016). DOI: https://doi.org/10.1145/2947668.

[Mel+09] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, and Guohua Jin. “A New Vision for Coarray Fortran”. In: *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. PGAS’09. Ashburn, Virginia, USA, 2009. DOI: https://doi.org/10.1145/1809961.1809969.

[Men+12] Harshitha Menon, Nikhil Jain, Gengbin Zheng, and Laxmikant Kalé. “Automated Load Balancing Invocation Based on Application Characteristics”. In: *2012 IEEE International Conference on Cluster Computing*. CLUSTER’12. Beijing, China, 2012, pp. 373–381. DOI: https://doi.org/10.1109/CLUSTER.2012.61.

[MK13] Harshitha Menon and Laxmikant Kalé. “A Distributed Dynamic Load Balancer for Iterative Applications”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC’13. Denver, CO, USA, 2013, pp. 1–11. DOI: https://doi.org/10.1145/2503210.2503284.

[NA21] M. Z. Naser and Amir H. Alavi. “Error Metrics and Performance Fitness Indicators for Artificial Intelligence and Machine Learning in Engineering and Sciences”. In: *Architecture, Structures and Construction* (2021). DOI: https://doi.org/10.1007/s44150-021-00015-8.

[NST96] David M. Nicol, Rahul Simha, and Don Towsley. “Static Assignment of Stochastic Tasks using Majorization”. In: *EEE Transactions on Computers* 45.6 (1996), pp. 730–740. DOI: https://doi.org/10.1109/12.506428.

[NC99] Jarek Nieplocha and Bryan Carpenter. “ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems”. In: *Parallel and Distributed Processing*. Puerto Rico, USA: Springer Berlin Heidelberg, 1999, pp. 533–546. DOI: https://doi.org/10.1007/BFb0097937.

[Pan+21] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. “The MVAPICH project: Transforming research into high-performance MPI library for HPC community”. In: *Journal of Computational Science* (2021). DOI: https://doi.org/10.1016/j.jocs.2020.101208.

[Pap+14] Jean-Charles Papin, Christophe Denoual, Laurent Colombet, and Raymond Namyst. “Dynamic Load Balancing with Pair Potentials”. In: *Euro-Par 2014: Parallel Processing Workshops*. Porto, Portugal: Springer, 2014, pp. 462–473. DOI: https://doi.org/10.1007/978-3-319-14313-2_39.

[Pap+21] Jean-Charles Papin, Christophe Denoual, Laurent Colombet, and Raymond Namyst. “SPAWN: An Iterative, Potentials-Based, Dynamic Scheduling and Partitioning Tool”. In: *International Journal of Parallel Programming* 49.1 (2021), pp. 81–103. DOI: https://doi.org/10.1007/s10766-020-00677-9.

[PTA13] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. “On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks”. In: *2013 42nd International Conference on Parallel Processing*. Lyon, France, 2013, pp. 100–109. DOI: https://doi.org/10.1109/ICPP.2013.19.

[PD21] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach (The Morgan Kaufmann Series in Networking)*. Sixth Edition. Elsevier, 2021. ISBN: 978-0-12-818200-0. DOI: https://doi.org/10.1016/C2018-0-01477-2.

[PCN18] Raphaël Prat, Laurent Colombet, and Raymond Namyst. "Combining Task-Based Parallelism and Adaptive Mesh Refinement Techniques in Molecular Dynamics Simulations". In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP'18. Eugene, OR, USA, 2018. DOI: https://doi.org/10.1145/3225058.3225085.

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Weimar, Germany: IEEE, 2009, pp. 427–436. DOI: https://doi.org/10.1109/PDP.2009.43.

[Rab+06] Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller. "Hybrid MPI and OpenMP Parallel Programming". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. The 13th European PVM/MPI User's Group Meeting. Bonn, Germany, 2006, pp. 11–11. DOI: https://doi.org/10.1007/11846802_10.

[RWS88] Sanjay Ranka, Youngju Won, and Sartaj Sahni. "Programming A Hypercube Multicomputer". In: *IEEE Software* 5.5 (1988), pp. 69–77. DOI: https://doi.org/10.1109/52.7944.

[RDB18] Leonhard Rannabauer, Michael Dumbser, and Michael Bader. "ADER-DG with a-posteriori finite-volume limiting to simulate tsunamis in a parallel adaptive mesh refinement framework". In: *Computers & Fluids* 173 (2018), pp. 299–306. DOI: https://doi.org/10.1016/j.compfluid.2018.01.031.

[RLP11] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. "Work Stealing for Multi-core HPC Clusters". In: *Euro-Par 2011 Parallel Processing*. Bordeaux, France: Springer, 2011, pp. 205–217. DOI: https://doi.org/10.1007/978-3-642-23400-2_20.

[Reg+22] Mustapha Regragui, Baptiste Coye, Laércio L. Pilla, Raymond Namyst, and Denis Barthou. "Exploring Scheduling Algorithms for Parallel Task Graphs: A Modern Game Engine Case Study". In: *Euro-Par 2022: Parallel Processing*. Glasgow, UK, 2022, pp. 103–118. DOI: https://doi.org/10.1007/978-3-031-12597-3_7.

[Ria+11] Ioannis Riakiotakis, Florina M Ciorba, Theodore Andronikos, and George Papakonstantinou. "Distributed Dynamic Load Balancing for Pipelined Computations on Heterogeneous Systems". In: *Parallel Computing* 37.10 (2011), pp. 713–729. DOI: https://doi.org/10.1016/j.parco.2011.01.003.

[Rob13] Arch D. Robison. "Composable Parallel Patterns with Intel Cilk Plus". In: *Computing in Science & Engineering* 15.2 (2013), pp. 66–71. DOI: https://doi.org/10.1109/MCSE.2013.21.

[RSU91] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. "A Simple Load Balancing Scheme for Task Allocation in Parallel Machines". In: *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA'91. Hilton Head, South Carolina, USA: ACM, 1991, pp. 237–245. DOI: https://doi.org/10.1145/113379.113401.

[Sam+21] Philipp Samfass, Jannis Klinkenberg, Minh Thanh Chung, and Michael Bader. "Predictive, Reactive and Replication-Based Load Balancing of Tasks in Chameleon and Sam(Oa)2". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC'21. Geneva, Switzerland: ACM, 2021. DOI: https://doi.org/10.1145/3468267.3470574.

[SC16] Conrad Sanderson and Ryan Curtin. "Armadillo: a template-based C++ library for linear algebra". In: *Journal of Open Source Software* 1.2 (2016), p. 26. DOI: https://doi.org/10.21105/joss.00026.

[SC18] Conrad Sanderson and Ryan Curtin. "A User-Friendly Hybrid Sparse Matrix Class in C++". In: *Mathematical Software*. ICMS'18. South Bend, IN, USA, 2018, pp. 422–430. DOI: https://doi.org/10.1007/978-3-319-96418-8_50.

[Seb+06] Nicu Sebe, Ira Cohen, Ashutosh Garg, and Thomas S Huang. *Machine Learning in Computer Vision*. Springer Dordrecht, 2006. ISBN: 978-1-4020-3275-2. DOI: https://doi.org/10.1007/1-4020-3275-7.

[SMB06] Alexander Spiegel, Dieter an Mey, and Christian Bischof. "Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing". In: *Applied Parallel Computing. State of the Art in Scientific Computing*. PARA'06. Umea, Sweden: Springer Berlin Heidelberg, 2006, pp. 433–441. DOI: https://doi.org/10.1007/11558958_51.

[SAB18] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. *High Performance Computing: Modern Systems and Practices*. Elsevier, 2018. ISBN: 978-0-12-420158-3. URL: https://doi.org/10.1016/C2013-0-09704-6.

[SB77] Herbert Sullivan and Theodore R. Bashkow. "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I". In: *Proceedings of the 4th Annual Symposium on Computer Architecture*. ISCA'77. ACM, 1977, pp. 105–117. DOI: https://doi.org/10.1145/800255.810659.

[SK22] Bronis R. de Supinski and Michael Klemm. *OpenMP Technical Report 11: Version 6.0 Preview 1*. Tech. rep. OpenMP Architecture Review Board. 2022. URL: https://www.openmp.org/wp-content/uploads/openmp-TR11.pdf.

[TGT13] Marc Tchiboukdjian, Nicolas Gast, and Denis Trystram. "Decentralized List Scheduling". In: *Annals of Operations Research* 207.1 (2013), pp. 237–259. DOI: https://doi.org/10.1007/s10479-012-1149-7.

[Tch+10] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. "A Tighter Analysis of Work Stealing". In: *Algorithms and Computation*. ISAAC'10. Jeju Island, Korea: Springer Berlin Heidelberg, 2010, pp. 291–302. DOI: https://doi.org/10.1007/978-3-642-17514-5_25.

[Tho+18] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, et al. "A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing". In: *Journal of Supercomputing* 74.4 (2018), pp. 1422–1434. DOI: https://doi.org/10.1007/s11227-018-2238-4.

[Tun+19] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. "Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 883–896. DOI: https://doi.org/10.1109/TPDS.2018.2870403.

[Val90] Leslie G Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (1990), pp. 103–111. DOI: https://doi.org/10.1145/79173.79181.

[VWR13] B. Vikranth, Rajeev Wankar, and C. Raghavendra Rao. "Topology Aware Task Stealing for On-chip NUMA Multi-core Processors". In: *Procedia Computer Science* 18 (2013). DOI: https://doi.org/10.1016/j.procs.2013.05.201.

[Wei+18] H. Weisbach, B. Gerofi, B. Kocoloski, H. Härtig, and Y. Ishikawa. "Hardware Performance Variation: A Comparative Study Using Lightweight Kernels". In: *High Performance Computing. ISC High Performance 2018. Lecture Notes in Computer Science*. ISC'18. Frankfurt, Germany: Springer, 2018, pp. 246–265. DOI: https://doi.org/10.1007/978-3-319-92040-5_13.

[WP08] Thomas Willhalm and Nicolae Popovici. "Putting Intel® Threading Building Blocks to Work". In: *Proceedings of the 1st International Workshop on Multicore Software Engineering*. IWMSE'08. Leipzig, Germany, 2008, pp. 3–4. DOI: https://doi.org/10.1145/1370082.1370085.

[WA04] M. Wrinn and R. Asbury. "MPI tuning with Intel/spl copy/ Trace Analyzer and Intel/spl copy/ Trace Collector". In: *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*. San Diego, CA, USA: IEEE, 2004, p. 4. DOI: https://doi.org/10.1109/CLUSTR.2004.1392595.

[XL97] Chengzhong Xu and Francis C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Springer New York, NY, 1997. ISBN: 978-0-585-27256-6. DOI: https://doi.org/10.1007/b102252.

[Yos+22] Kohei Yoshida, Rio Sageyama, Shinobu Miwa, Hayato Yamaki, and Hiroki Honda. "Analyzing Performance and Power-Efficiency Variations among NVIDIA GPUs". In: *Proceedings of the 51st International Conference on Parallel Processing*. ICPP'22. Bordeaux, France: ACM, 2022, pp. 1–10. DOI: https://doi.org/10.1145/3545008.3545084.

[ZL18] Afshin Zafari and Elisabeth Larsson. "Distributed Dynamic Load Balancing for Task Parallel Programming". In: *Computing Research Repository* 1801.04582 (2018). URL: https://arxiv.org/abs/1801.04582.

[ZO11] Yao Zhang and John D. Owens. "A quantitative performance analysis model for GPU architectures". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. San Antonio, TX, USA: IEEE, 2011, pp. 382–393. DOI: https://doi.org/10.1109/HPCA.2011.5749745.

[Zhe+14] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. "UPC++: A PGAS Extension for C++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS'14. Phoenix, AZ, USA, 2014, pp. 1105–1114. DOI: https://doi.org/10.1109/IPDPS.2014.115.

Online Resources

[FZJ] Forschungszentrum Jülich (FZJ). *MPMD: Multiple Program Multiple Data Execution Model*. URL: https://apps.fz-juelich.de/jsc/hps/jureca/mpmd.html (visited on 2023-03-09).

[LLN] HPC Livermore Computing (LLNL). *Introduction to Parallel Computing Tutorial*. URL: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial (visited on 2023-03-07).

[ACB21] Olivier Aumage, Paul Carpenter, and Siegfried Benkner. *Task-Based Performance Portability in HPC*. 2021. URL: https://doi.org/10.5281/zenodo.5549731.

[Bav20] Leibniz Supercomputing Centre of the Bavarian Academy of Sciences. *SuperMUC-NG*. 2020. URL: https://doku.lrz.de/supermuc-ng-10745965.html (visited on 2023-06-07).

[Div96] NASA Advanced Supercomputing (NAS) Division. *NAS Parallel Benchmarks*. 1996. URL: https://www.nas.nasa.gov/software/npb.html (visited on 2023-03-10).

[lam08] Developers at lammps.org. *LAMMPS Molecular Dynamics Simulator*. 2008. URL: https://www.lammps.org (visited on 2023-10-18).

[LLV13] LLVM/OpenMP. *LLVM/OpenMP Runtimes*. 2013. URL: https://openmp.llvm.org (visited on 2023-10-11).

[TOP93] TOP500. *TOP500 List - November 2022*. 1993. URL: https://www.top500.org/ (visited on 2023-03-11).

Index

Intentionally left blank.