High-productivity Abstractions and Efficient Runtime for Dynamically Load-balanced Distributed Programs on Multi/Many-core Clusters

Finnerty, Patrick Martin

(Degree)
博士（工学）

(Date of Degree)
2022-09-25

(Date of Publication)
2024-09-25

(Resource Type)
doctoral thesis

(Report Number)
甲第8467号

(URL)
https://hdl.handle.net/20.500.14094/0100477893

博士論文

High-productivity Abstractions and Efficient Runtime for Dynamically Load-balanced Distributed Programs on Multi/Many-core Clusters

マルチコア / メニーコアクラスタにおける 動的負荷分散プログラムのための 高生産性抽象化と効率的ランタイム実装法

２０２２年７月

神戸大学大学院システム情報学研究科

フィネルティ　パトリック　マルタン

FINNERTY Patrick Martin

Doctoral Dissertation

HIGH-PRODUCTIVITY ABSTRACTIONS AND EFFICIENT RUNTIME FOR DYNAMICALLY LOAD-BALANCED DISTRIBUTED PROGRAMS ON MULTI/MANY-CORE CLUSTERS

マルチコア / メニーコアクラスタにおける 動的負荷分散プログラムのための 高生産性抽象化と効率的ランタイム実装法

July 2022

Graduate School of System Informatics
Kobe University

FINNERTY Patrick Martin

Keep calm and curry カレー on ...

Abstract

Modern supercomputers rely on clusters of many-core processors, bringing large amount of parallelism both within a node and across nodes. Harnessing the potential of such systems is a challenge for application developers, as a large amount of parallelism is available both between and within compute nodes. Modern Partitioned Global Address Space (PGAS) programming languages facilitate this task to a degree by introducing elements representing the distributed nature of the program in the language itself. However, these features alone are not enough to handle the load unbalances that arise in modern applications.

A successful global load balancing scheme is the lifeline-based global load balancer. First implemented in X10, this scheme showed that it can effectively scale up to several thousand compute nodes. A shortcoming of this scheme is that the task granularity, i.e. the number of individual tasks processed in a single batch, greatly influences the performance of the scheme. Without any method to predict what an appropriate setting should be, users of this scheme therefore need to try many settings to find a satisfactory value for their application, wasting much time and valuable computational resources. Instead, we believe this kind of setting adjustments should be performed by the library itself. We integrate a *tuning mechanism* to the scheme which automatically adapts the granularity during execution to guarantee optimum performance. Our grain tuner is implemented as a feedback mechanism and relies on runtime metrics to make its adjustments, with no noticeable overhead. We show that it is capable of handling a variety of tree traversal applications and is robust against changes in implementation.

A limitation of the lifeline-based global load balancer is that it only operates on self-contained tasks, that is, all the data needed to perform the computation is discarded as soon as the task has completed. For cases where the data persists after the computation, other techniques are desired. Current PGAS languages generally support distributed collections, mostly arrays, and allow these arrays to be distributed across the processes taking part in the computation. However, they provide little support for uneven distributions or for dynamic modifications to the distribution of a collection.

In this thesis, we introduce our answer to this issue in the form of *relocatable distributed collections*. The collections we propose are analogous to their shared-memory counterparts but have been fitted with additional features to handle the distributed na-

ture of the computation. In particular, we introduced a dynamic entry relocation system which makes it easy for programmers to dynamically relocate entries of a distributed collection between processes. We introduce the concept of *teamed method* to signify that a method of the collection requires communication or synchronization between processes. We demonstrate the productivity gains brought about by our distributed collections on a complex financial market simulator and a well-known N-Body application.

We then consider the possibility for our distributed collections library to handle the load balancing automatically, relieving application developers from the burden of manually implementing such measures within their programs. Inspired by the lifeline-based global load balancer, we adapt its principles to our distributed collections. We provide a clear context within which it is allowed to operate, keeping the impact on the legibility of programs to a minimum.

Overall, the concepts we introduce make it easier for both newcomers to the field and more experienced programmers to develop dynamic distributed applications. In the future, we think the facilities we introduce will make it possible to more easily develop elastic applications where the number of running processes is dynamically adjusted to match the actual parallelism needs of the application. Indeed a significant hurdle in the development of such applications is the need to relocate the data away from released nodes, or offloading data onto newly joined processes when decreasing or increasing the number of running processes.

Acknowledgments

The research presented here was conducted at Kobe University Graduate School of System Informatics under the guidance of professor Tomio Kamada and Chikara Ohta.

First and foremost, I would like to express my sincere gratitude to Associate Professor Tomio Kamada and Professor Chikara Ohta who supervised me during this work. Not only did they welcome me into the *Future Information Network Engineering* (FINE) laboratory when I was a mere exchange student, they supported me when I expressed my interest in pursuing a PhD. Under their guidance I have now progressed to the point they acknowledge me as a colleague, for which I am most grateful. In several ways, this has been and continues to be a longer collaboration than initially expected.

I would also like to thank Professor Naoyuki Tamura, Professor Mikio Yokokawa, and Professor Takenao Ohkawa from the Graduate School of System of System Informatics for the fruitful discussions and their advice throughout the doctoral course. They helped me consider new problems and brought about new perspectives about my work which I had not considered. I also thank Professor Zhiwei Luo for his help with the title.

I also have to thank Yohsuke Murase, Research Scientist at RIKEN Center for Computational Science, for his help in setting up the program executions on supercomputers, saving me much time and trouble.

I would like to thank my fellow students at CS29 who make for a warm and welcoming environment to work in. Among them, I would like to mention Yoshiki Kawanishi whom I collaborated with on some aspects of the work presented here.

I would also like to express my profound gratitude to my parents for their infallible support in all my endeavors. For some reason, I always seem to get further and further away from my hometown. I promise I will be visiting soon.

Finally, I would like to express my great appreciation to the Japanese Society for the Promotion of Science who funded the research presented here under KAKENHI grants number JP20K11841 and JP18H03232.

Patrick Finnerty
Kobe, Japan
July 15, 2022

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Distributed & parallel computing

Modern supercomputers feature an unprecedented level of parallelism. For instance, the current flagship supercomputer of Japan, Fugaku, is composed of over 158 thousand compute nodes, each node containing a many-core 48 core processor [1]. Harnessing the processing power of such large systems poses a challenge to application developers.

Borrowing the definition from van Steen and Tanenbaum [2]:

> A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

Programming models for distributed programming should therefore present a single coherent system for programmers to be able to reason and create distributed programs with ease.

The second aspect to consider is how to handle the available intra-node parallelism. The emergence of multi- and many-core processors has brought a large amount of parallelism to be exploited within each node. Naturally, a plethora of frameworks, languages, and compiler directives aiming at automatically paralyzing code for shared-memory execution has emerged.

One possibility for programmers is to combine the techniques of distributed computing and intra-node parallelism together in a hybrid manner. This approach means that newcomers to the field of parallel and distributed programming need to familiarize themselves with both distributed programming on one hand, and parallel programming on the other to start developing applications. This comes as a significant burden, effectively requiring from programmers that they become expert in both fields to develop applications for modern supercomputers.

Another approach may consist in using a higher-level framework or a language dedicated to distributed and parallel processing. Currently, a large variety of models and programming languages implementing these models exist, each expressing their own point of view of what a distributed program is. Some models completely hide the distributed

nature of the hardware used to execute the program, while others allow programmers to more finely control how and when independent processes communicate via dedicated primitives.

As we will develop in Chapter 2, the difficulty with such models is that a compromise needs to be struck between control and ease of use. Programming models that completely hide the distributed nature of the program make it difficult for programmers to leverage locality to make efficient programs. On the other hand, techniques that allow programmers to manage the locality of their data will place a significant burden on the programmer when it comes time to implement load balancing measures.

1.2 Motivation

We believe that with the appropriate abstractions and middleware to support them, the barrier to entry into the field of distributed and parallel programming can be significantly reduced. In the work presented in this thesis, we strive to provide the middleware that will help non-expert programmers accustomed to "traditional" shared-memory Object-Oriented Programming transition to distributed computing. We also aim to provide the high-level abstractions that will allow more experienced application developers to create complex and dynamic applications. As such, we prioritize the productivity of the high-level abstractions over pure performance. A program written with the facilities we propose should run reasonably well "out of the box," without the programmer needing to spend much time refining it.

At its most fundamental level, distributed computing involves decomposing a large computation into multiple smaller units and to assign these units to several independent computers to be processed in parallel. Load balancing is the act of assigning these units of work in a manner which matches the compute nodes available processing power, maximizing the program's efficiency. For instance, let us consider an hypothetical computation composed of 20 independent units of work of equivalent load. If we were to distribute these units between 4 compute nodes, a balanced load assignment would consist in attributing 5 units of work to each compute node.

In practice however, things are not that simple. First, it may not be possible to a priori estimate the required computation load of each fragment. As an example, take the enumeration problem of *N-Queens* which consists in finding the number of ways N Queens can be arranged on a N-wide square board without any two Queens threatening one another [3]. A solution to the 8-Queens problem is shown in Figure 1.1. This problem can be solved using a backtrack-search algorithm in which the nodes of the exploration tree consists in placing (or removing, when backtracking) a piece on the board. Sub-trees of the exploration tree can be explored independently and in parallel, but there is no way to precisely estimate the size of a sub-tree. If you were capable of correctly estimating the number of nodes in a sub-tree without exploring it, you would have effectively solved

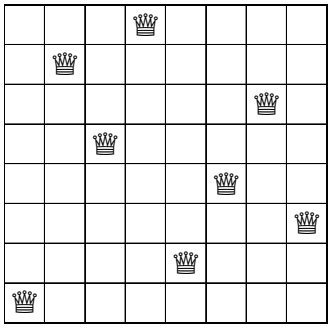


Figure 1.1: One of 92 solutions to the 8-Queens problem

the problem.

Secondly, the performance of the computers hosting the processes used to perform the computation may be uneven, either through different hardware characteristics, or competing processes from other programs running on the same host. As a result, a "programmatically even" load distribution may result in unbalance in practice.

Without any measure taken, some processes will complete the units of work they were assigned early and remain idle until the other processes also complete their part. It does not appear reasonable to account for hardware discrepancies in a program as this would imply modifying the program for every different hardware configuration encountered. As for dealing with processes competing for resources on the same host, it is a dynamic problem by nature and thus cannot be solved by static distribution adjustments.

For situations such as the ones mentioned above, *dynamic load balancing* techniques appear to be an appropriate answer. Dynamic load balancing consists in modifying the distribution of the computation units across processes during the computation as load unbalances are detected. Multiple schemes and techniques can accomplish this goal, such as profiling, and work-stealing schemes. It should be noted that since modern computers rely on multi-core and now many-core processors, a large amount of parallelism is available both within and between hosts. Load balancing schemes therefore need to balance the load both between processor cores and across processors.

In this thesis, we target two challenges facing distributed application developers. The first one consists in finding the appropriate task granularity in a dynamic load balancer. The second one concerns the management of persistent data in a distributed program.

Automatic granularity tuning Programming models, schemes, and techniques necessarily rely on a number of arbitrary settings and parameters. While expert application developers may be able to spend the time to adjust and fine-tune these parameters to obtain optimal performance on specific applications, this is not the case for most people. As one objective of our work is to provide simple abstractions and to guarantee reasonable performance on a wide-range of applications, we find that leaving parameter tuning up to users is not satisfactory. Instead, such tuning steps should be taken by

the middleware itself, adjusting parameters on the fly to adapt to both the underlying hardware and the supported program.

In this thesis, we consider the case of the *multi-worker lifeline-based global balancer* [4, 5, 6]. This framework allows programmers to provide some computation through a simple abstraction. The scheme then takes care of distributing and balancing the load across compute nodes. One setting critical for the performance of this scheme is the *task granularity*, i.e. the number of unit tasks performed together in a single batch. We explore the criteria that will allow us to automatically adjust this setting to guarantee optimal performance.

Relocatable distributed collections Current languages and libraries for distributed programming may allow programmers to distribute data structures such as arrays across processes. However, little support is offered to allow this initial distribution to be modified dynamically as load balances appear.

In this thesis, we introduce the notion of *relocatable distributed collections*. The facilities we introduce allow programmers to explicitly and easily manage the distributed nature of their program. In particular, the high-level dynamic entry relocation system makes it possible for programmers to implement load balancing strategies for their own application with ease. We then explore the possibility for the programmer to temporarily surrender the management of their collections' distribution to the runtime so that they are load-balanced automatically. This allows less-experienced programmers to create dynamically load-balanced programs with minimal effort.

1.3 Contributions

The contributions of this thesis are as follows. First, we claim the successful implementation of a hybrid dynamic load balancer and its application to combinatorial exploration and optimization problems. We integrate a tuning mechanism into the scheme that automatically adjusts the task granularity to guarantee optimal performance. This tuning mechanism is constructed as a feedback mechanism and relies on runtime information to judge whether the granularity of the computation at hand should be either increased, decreased, or kept as is.

Secondly, we claim the implementation of a distributed relocatable distributed collections library. The collections provided are analogous to their shared-memory counterparts, but with extra features handling their distributed nature. The introduction of "teamed methods" offers a clear way to identify functionalities that require communication and/or synchronization between hosts to perform the desired computation. Entries recorded into the distributed collections can be dynamically exchanged between processes through a consistent set of high-level abstractions. This allows programmers to control the object from from process to process with ease. It also allows load balanc-

ing techniques to be implemented by the user. Using these facilities, we introduce a load-balanced version of the PlhamJ financial market simulator.

Finally, we claim the implementation of a *dynamic load balancer integrated into the distributed collection library*. The programming interface we propose allows programmers to temporarily surrender the control over the collections' distribution to the library, which will in turn relocate entries of the underlying collections as unbalances appear. This method allows programmers to automatically balance the load of their computation without having to craft and integrate their own load-balancing strategy into their application.

1.4 Outline

The remainder of this thesis is organized in the following chapters. First, we recall some useful background about programming models and paradigms for distributed and parallel programs in Chapter 2. A particular focus is made on the Asynchronous Partitioned Global Address Space (APGAS) programming model and its use in the Java programming language on which the work presented in this thesis is based.

Then, we discuss the problem posed by task granularity in global load balancers in Chapter 3. The conditions necessary for the successful implementation of a hybrid lifeline-based load balancer are discussed before introducing a tuning mechanism capable of dynamically adjusting the task granularity based on runtime metrics. The scheme presented causes no detectable overhead and is robust against changes in problem implementation.

In Chapter 4, we address the lack of inter-process communication in the APGAS programming model by introducing our *distributed relocatable collection* library. Our contribution comes as a complement to the APGAS for Java library and provides collections which mimic the Java standard library and provide support for common distributed computation patterns through a consistent API.

In Chapter 5, we present the dynamic load balancer integrated into our distributed collection library. The integrated global load balancer we propose relocates entries of distributed collections within a clearly identifiable context. Internally, it borrows ideas of the lifeline-based global load balancer discussed in Chapter 3 and adapts them to the requirements of this situation.

Finally, conclusions and future perspectives are offered in Chapter 6.

Chapter 2

Background

In this chapter, we first outline current techniques, languages, and programming models used in distributed and parallel computing today in Section 2.1. We then lay out the case as to why we believe APGAS for Java is a good candidate to fulfill our objectives and what it still lacks to achieve its full potential in Section 2.2.

2.1 Programming models for distributed & parallel computing

There are many different programming models available to programmers to create parallel and distributed programs. In this section, we lay out the differences between programming models with respect to two specific areas of interest: *locality* and *dynamic distribution management*.

Locality corresponds to how the programming model allows programmers to control what computation is assigned to which process. This refers both to data and task management. As we will see, some models provide a higher-level abstraction of a distributed program which is more detached from any execution consideration, relieving programmers from the need to manage locality. Other models require that all things distributed be managed explicitly.

Dynamic distribution management refers to the capability for data to be redistributed between processes after an initial allotment to processes has been made. Not all programming models and languages used for distributed programming support this kind of facilities, and the involvement of the programmer can vary greatly.

We should point out that any programming language is not precluded to a single category. Frameworks built on top of a particular programming model may belong in another category.

2.1.1 MPI

The Message Passing Interface (MPI) is a standard for communication whose first version was established in 1994 [7]. It has become a de-facto standard for many high-performance programs either used directly or to support higher-level abstractions. MPI itself is merely a standard defining the C calls that are available to programmers, with multiple existing implementation. The standard itself has been revised a couple of times since its inception to introduce new features such as process creation and management, parallel I/O, and one-sided non-blocking communications [8]. The current version at the time of writing stands at *Version 4.0*.

While the most essential use of MPI is done through calls to a C of Fortran interface, multiple projects have introduced a compatibility layer with the Java programming language. Use of the C "native" MPI calls can now be made from a Java program. The conversion layer between the native C calls and the Java language either usually comes in the form of a series of Java objects and methods organizing the MPI functions in a Java-friendly class hierarchy. These Java calls are supported through the use of Java Native Interface (JNI), which allows "native" C code to be called from a Java program. The translation between C and Java arrays is done in this compatibility layer which needs to be compiled and on top of an existing C language MPI implementation, as in *mpiJava* [9] and the *MPJ-Express* [10] projects. Notably, OpenMPI [11] ships with its own Java compatibility layer.

MPI offers a "same program multiple processes" view of a distributed system. This means that the program's `main` will execute on every process spawned for the execution of the program. It is possible to make processes perform different computation by introducing conditions based on the number, or "rank" in MPI terminology, of the running process. Control over which process executes what code over which data is therefore explicit. If data needs to be dynamically relocated between processes, this will require significant programmer investment as no direct support for such features exist in the MPI standard. Intra-host parallelism needs to be implemented with the help of external libraries or compiler directives as MPI is solely focused on inter-host communication.

2.1.2 Charm++

At its core, Charm++ relies on over-decomposing the problem into many objects, "chares" in the language's idiom [12, 13]. Communication is performed by sending "messages" between chares which correspond to a call to a method of this object. The programming model is agnostic to distribution, meaning the programmer does not know how the various chares used in their program will be distributed across the processes actually used to run the program.

The strength of Charm++ lies in its embedded load balancing strategies. Indeed, the Charm++ runtime is capable of profiling the running program and relocating chares

between processes to balance the load between processes. A consequence is that the programmer remains quite removed from any locality concerns.

While the abstraction brought about by the Charm++ programming language are expressive and powerful, they require some getting used to. In particular, the global termination detection scheme which relies on quiescence is a significant hurdle. Also, the fact that the order in which messages are processed by the chares is not deterministic may be problematic for some applications.

2.1.3 Map/Reduce frameworks

The main idea of the Map/Reduce framework is to use inputs of various nature, produce new data through a number of "map" operations, and aggregate this data through "reduce" operations. The programmer is in charge of implementing what the `map` and `reduce` actions on the data source consist in, while the scheduling and execution in a large scale cluster or cloud environment is managed by the system. As this offers a simple framework for programmers, the barrier to entry is significantly reduced compared to other distributed computing approaches. The most popular implementations of this framework are Hadoop [14] and Spark [15].

While the low barrier to entry of Map/Reduce frameworks make them appealing to computer scientists dealing with large datasets that require a large amount of parallelism to process them in reasonable time, the expressiveness of this model is somewhat limited. No notion of locality or distribution management can be expressed by the programmers as the system's runtime is solely in charge of scheduling the tasks on the processors. There is however much research looking into how to make this as efficient as possible [16, 17].

Admittedly, considering Map/Reduce in terms of programming model is quite reductive as their most appealing characteristic lie in their integration in a broader ecosystem for *big data analytics*, comprising interoperability with database management systems, distributed file system, cluster resource management, job schedulers etc [18, 19].

2.1.4 (A)PGAS languages

The Partitioned Global Address Space [20] (PGAS) programming model is implemented by multiple programming languages, including Coarray Fortran [21, 22], Unified Parallel C (UPC) [23, 24], UPC++[25], Xcalable MP [26, 27], and PCJ [28, 29]. The Asynchronous Partitioned Global Address Space (APGAS) extends the PGAS model by introducing asynchronous tasks executing on one of the locality abstractions, with languages such as X10 [30], Habanero-Java [31, 32], and Chapel [33] implementing this model.

At its core, languages that implement these programming models embed an abstraction representing a running process within the language itself. Typically, processes running the program are numbered from 0 to $n-1$ for an n-process execution, analogous

to the notion of *rank* in MPI.

The "partitioned global address space" means that variables located on a process can be accessed from a remote process through facilities provided by the language. Such variables are therefore "global" in the sense that they can be accessed from any process participating in the program through some form of globally unique identifier. Depending on the language, this identifier may take the form of an index in a distributed array, a global pointer, or a generic identifier such as a globally unique key. The address space is *partitioned* between the running processes, meaning that the variables that possess a global address are actually handled by one of the processes. Depending on the process considered, there are therefore "local" memory accesses if the process itself is handling the data, or "remote" memory if the data is handled by another process.

Depending on the language, there are subtle variations on the implemented programming model. For instance, UPC [23] introduces the notion of global pointers which refer to some memory allocated on a process. Inside the program, if access is made to a global pointer and the corresponding memory area belongs to a remote process, the UPC compiler will automatically add the code to make the remote access during compilation. Access to remote memory is therefore implicit to some degree. In the more recent version of the UPC++ [25] language which is derived from UPC, dereferencing global pointers needs to be performed through a specific call, forcing the programmer to explicitly recognize the distributed nature of the program.

Chapel supports distribution of arrays through *Block*, *Cyclic*, and *Cyclic Block* distributions. With an initial array defined, the location of these pieces can be defined using these pre-defined distributions. However, Chapel does not support this features for maps, or *associative domains* per the Chapel idiom. Deitz et al. [34] explored improving the programability and the performance of distributed scans and reductions in Chapel and MPI. In particular, they supplement MPI with a set of preprocessor directives that automatically generate the code to make user-defined parallel and distributed reductions.

PCJ brings the PGAS programming model to Java in a pure Java library, relying on Java annotations to mark the variables that belong to the global address space in particularly elegant manner. The library also provides collective communications operating on the variables of the global address space such as `broadcast`, `scatter`, `reduce` and others in pattern similar to MPI but adapted to their programming model [29].

Another distinction depending on the model is the adoption of a "local view" or a "global view." In *local view*, the program is written from the point of view of one process communicating with the other processes, while in *global view* the program is written for the entire cluster as a single consistent entity. UPC, UPC++, Habanero-Java, and X10 are examples of programming languages that adopt the "local" view, while Coarray Fortran and Chapel adopt the "global view". Xcalable MP can adopt both.

Managing locality with a PGAS language is made easy through the introduction of

```
1  import static apgas.Constructs.*;
2  import apgas.Place;
3  class HelloWorld {
4    public static void main(String[] args) {
5      System.out.println("Running main at " + here() + " of " +
6        places().size() + " places");
7      finish(() -> {
8        for (Place p : places()) {
9          asyncAt(p, () -> System.out.println("Hello from " + here()));
10       }
11     });
12     System.out.println("Bye");
13   }
14 }
```

Listing 2.1: Distributed Hello World in Java

```
1 Running main at place(0) of 4 places
2 Hello from place(0)
3 Hello from place(3)
4 Hello from place(2)
5 Hello from place(1)
6 Bye
```

Listing 2.2: Possible output of the Listing 2.1 program running with 4 processes

some representation of a process within the language. However, dynamic relocation after an initial distribution of a data structure has been made is generally not supported.

One exception is the case of X10 which allows any from of user-defined distribution for its `DistArray` [35] if the programmer so desires it, but modification of an existing array is not supported. Instead, a new array can be created based on the contents of the old one.

2.2 APGAS for Java

The X10 implementation of the APGAS programming model was later ported to Java in the form of a library [36]. The keywords of the X10 language were converted to Java static methods taking lambda-expressions as parameter. With the use of this library, Java effectively becomes an APGAS language.

A distributed *Hello World* program demonstrating the use of the `finish` and `asyncAt` constructs is presented in Listing 2.1. A possible output of this program shown in Listing 2.2. In this example, the `finish` method is called on line 7 to 11, with an asynchronous activity spawned on each place participating in the computation using a for loop and the `asyncAt` method on line 9. The program will not progress beyond the `finish` until every place prints its "hello" message.

First and foremost, Java is a popular programming language featuring an exten-

sive standard library. A large number of third-party libraries is also available on public repositories such as Apache Maven[37]. Unlike programming languages dedicated for distributed computing that often rely on an intermediary compilation to another language before producing the final binaries, the compilation toolchain for Java is rather straightforward with good support in modern IDEs.

Secondly, Java has a clear memory model, independent from any hardware architecture, in which the notions of threads and synchronization scheme between threads operating in shared-memory are defined [38]. In particular, the Java Memory Model allows the JVM to re-order program instructions as long as the "happens-before" relationships between threads that synchronize is preserved. This allows application developers to reason on this model, rather than having to re-adjust for different architectures.

As such, Java combined with the APGAS for Java library makes for an interesting stepping stone into the world of distributed and parallel computing. For newcomers to the field, it provides a familiar environment to Java programmers, with new features to handle the distributed nature of the program. For more experienced programmers, the high-level abstractions of the APGAS for Java library will support dynamic and complex task schedules controlled by the `finish`/`async` constructs. One such application, the lifeline-based global load balancer [4, 5, 6], is the subject of Chapter 3.

Although the APGAS for Java library appears promising, there are a number of features desired for distributed computing not provided. Most notably, no method for communication between activities running on different processes is provided. A simple approach could consist in combining APGAS for Java with a library which supports communication between processes such as MPI. However, this would not be sufficient.

While many common communication patterns such as broadcast or reductions are supported in MPI, these interfaces concern primitive types (`int`, `double` etc.) While this may be satisfactory for numerical simulations, it does not support those same patterns for object-oriented programs. Instead, higher level abstractions and features are desired, with a communication library serving in a support role in the background. We develop our vision for what these higher-level abstractions should be in Chapter 4 and 5.

Chapter 3

Task Granularity Tuning for the lifeline-based multi-worker GLB

The lifeline-based global load balancer [4] is a distributed load balancing scheme capable of dynamically redistributing some computation implemented by users following a specified API. Its scaling and dynamic load balancing capabilities were demonstrated on the Unbalanced Tree Search benchmark [39], a particularly difficult tree traversal due to its inherent irregularity and unpredictability.

In this chapter, we explore two challenges in the implementation of the multi-worker lifeline-based global load balancer. The first one consists in a fairness issue between the various activities used to implement the multi-worker variant of the scheme. The second issue we tackle consists in determining an appropriate setting for the task granularity. While this parameter is important for the general performance of the scheme, there are no methods that can allow users to determine a priori what a good setting would be. We propose a tuning mechanism embedded into the load balancing scheme which dynamically adjusts the setting as the computation takes place based on recently sampled runtime information.

This chapter is organized as follows. We first cover some background information concerning this load balancing scheme in Section 3.1. In Section 3.2, we discuss the conditions necessary for the successful implementation of this scheme in Java. We then cover related work about task granularity tuning in Section 3.3 before introducing our main contribution, the grain tuning mechanism, in Section 3.4. We present our conclusion and perspectives in Section 3.5.

3.1 Background on the lifeline-based global load balancer

The lifeline-based global load balancer is a work-stealing scheme first introduced in X10 [4, 5]. Its main feature consists in defining pre-determined channels for work stealing between places, the so-called "lifelines." When a place runs out of work and is unable to

steal some from a randomly selected victim, it signals its "lifeline" counterparts that it needs some work and passively waits until either work is sent to it, or the computation completes. This mechanism allows the load balancing scheme to maintain high efficiency even for large cluster sizes up to several thousand processors [5].

In the original scheme, there is only one worker thread per process. In a later evolution we discuss in Section 3.1.3, this scheme was extended to support multiple workers on the same process.

3.1.1 Abstraction for programmers

The abstractions provided to the programmer under both of these schemes are summarized in the `Bag` abstraction. In our Java implementation of the load balancer, it comes as an interface that programmers need to implement in their class that contain the data-structures that represent the computation at hand. The methods that programmers need to implement are the following:

- `void process(int, R)`: processes a certain amount of computation, that amount being specified by the integer parameter of the method. An instance of the result type `R` is also provided to the worker to read and/or write information shared with the other workers on the same host during the computation.
- `B split(boolean)`: returns a new bag instance containing a fragment of the computation held by the current bag. The boolean parameter is here to indicate whether or not all of its contents should be given away in the event the instance cannot be split.
- `void merge(B)`: merges the contents of the bag instance given as parameter into this instance.
- `boolean isEmpty()`: indicates if this bag is empty, i.e. if it does not contain any work.
- `boolean isSplittable()`: indicates if this bag can be further split, i.e. if work can be taken from it without emptying it altogether.
- `void submit(R)`: is called when the computation has ended and the result gathering phase begins. This method gives the opportunity to the bag to put its contribution to the final result into the instance provided as parameter.

To balance the load, a fragment of the computation can be obtained from a bag by calling the `split` method before transferring and merging that fragment into another bag instance. The `split` and `merge` methods' implementation is entirely left to the programmer. This grants complete control over the internal data structure used to represent the computation. The library remains oblivious to the data structure used by

the bags, and while programmers are advised to program the `split` method such that half of the contents of the bag are given away, there is no actual mechanism to enforce it. The library guarantees that calls to the `split` and `merge` methods on any bag are made sequentially. Programmers do not have to concern themselves about potential concurrent accesses made to their Bag implementation as they do not occur in the multithreaded lifeline-based global load balancer.

3.1.2 Lifelines

As mentioned above, the key innovation of the lifeline-based laod balancer scheme is that it introduced preferential channels for work-stealing, the so-called lifelines. Taken as a whole, the lifelines of all the processes combined form a directed graph of passive stealing channels, the *lifeline graph*.

In the initial stage, all the lifelines between all the processes are established. The initial bag containing the entire computation is arbitrarily given to the first process (place 0). The worker process of the first process will split its task queue to provide work to the processes which have lifelines established on place 0. These processes will in turn do the same with the processes with lifeline established on them until work trickles throughout the whole cluster. This scheme elegantly solves the problem of termination detection as all asynchronous activities that carry work are transitively spawned from the same superseding `finish`. When all the activities on all the hosts terminate, the enclosing `finish` returns, guaranteeing that global termination was achieved.

It is generally understood that the *lifeline graph* needs to be connected for the work-stealing scheme to be effective. Indeed, using a non-connected lifeline graph would cause work not to trickle from the initial place 0 to the subset of non-connected processes, causing them to remain idle throughout the computation.

Besides connectivity, two more desirable properties for the lifeline graph are listed by the creators of the scheme: bounded out-degree, and low diameter [4]. A bounded out-degree means the number of lifelines established (and received) by each process should be limited so that processes with work do not spend too much time addressing lifeline thieves when work becomes scarce. As for the low-diameter, it means that the maximum number of hops needed to travel between any two nodes in the lifeline graph should be limited. An extreme (but useful) example of a lifeline graph with high-diameter would be the directed loop between all the processes in the program. In a situation with N nodes, the average distance between nodes is $N/2$. When the number of processes used remains small this approach can give satisfactory results. However, when using a larger amount of processes, work takes too long to reach the last node in the graph, resulting in much of the cluster to remain idle for prolonged periods of time.

Although the influence of the lifeline strategy selection has not been studied thoroughly, there is consensus on the fact that the family of *cyclic hypercubes* satisfy the properties mentioned above. The cyclic hypercubes graphs can be defined as follows:

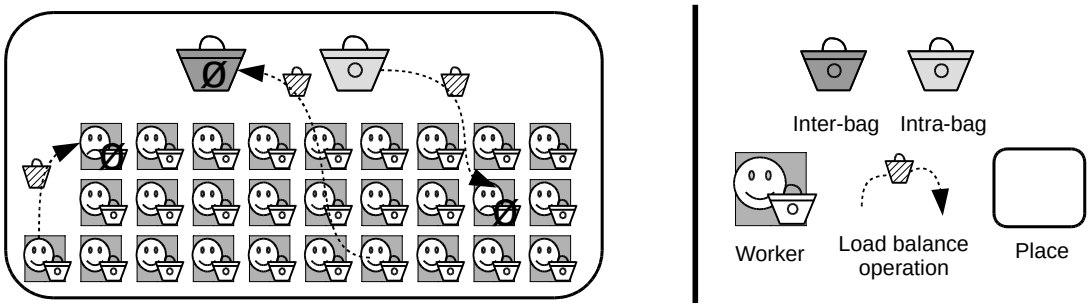


Figure 3.1: Overview of the multithreaded global load balancer opeprations within a place

Given a number of processes $N, N \in \mathbb{N}$, choose a radix $h, h \in \mathbb{N}$ and a power $z, z \in \mathbb{N}$ such that $h^{z-1} < N \leq h^z$. Each vertex p is represented as a number in base h with z digits.

Each vertex has an outgoing edge to every other vertex at a distance $+1$ from it in the Manhattan distance (in modulo h arithmetic). That is, the vertex p labeled $(a1, ..., az)$ has an outgoing edge to every vertex q such that for some $i \in 1..z, q = (a1, ..., (ai + 1)\%h, ..., az)$.

Using radix 2, the graph draws a 2-D square, a 3-D cube, and a 4-D hypercube for 4, 8, and 16 vertices respectively.

3.1.3 Multi-worker GLB

In the original X10 scheme, the lifeline-based global load balancer only supports one worker per host. A later evolution of this scheme, the multithreaded lifeline-based global load balancer [6] (multi-GLB) keeps the same computation abstraction and lifeline mechanism between places but makes each place run multiple worker threads in parallel instead of a single one as per the original scheme.

With this scheme, it is no longer necessary to use multiple places (or processes) per host to use all the cores of the hosts used for the computation. Instead, a single place containing as many workers as there are cores on the underlying processor can be used. This also brings the opportunity for workers on the same host to easily share information as they operate in shared-memory. A graphical representation of the design including the main load balance operations that occur within a host is depicted in Figure 3.1.

Instead of making remote steals when a worker runs out of work, the remote steals are made when all the parallel workers on the host run out of work. Within a place, each worker holds its own dedicated bag instance throughout the computation. Load balance operations are achieved through the use of two additional bag instances that are not processed by any worker. One bag - the *intra-bag* - is primarily used to handle load unbalances within a host, while the second bag - the *inter-bag*- is used to handle steals attempts coming from remote places. The workers on the place collaboratively maintain some work available in both of these bags for a potential thief, be it a local worker or a

remote place.

In its original X10 implementation, this load balancer suffered from a few problems concerning the scheduling of messages. These were resolved in our Java implementation.

3.2 Fairness between activities of the multi-worker load-balancing scheme

3.2.1 Problem statement

For maximum processing power of the multi-GLB scheme, it makes sense to use as many worker activities as there are cores on the underlying host. However, making such a choice will prevent the scheme from operating as intended. The issue stems from the fact that worker activities are long-running tasks.

In the implementation of the X10-style APGAS programming model in Java, asynchronous activities are submitted for execution to a common thread pool on each process. In the multi-worker lifeline-based global load balancer scheme, all the activities (worker activity, lifeline-answer activity, steal activity), are therefore submitted to this unique pool. As per the normal execution policy of thread-pools, tasks submitted execute until completion.

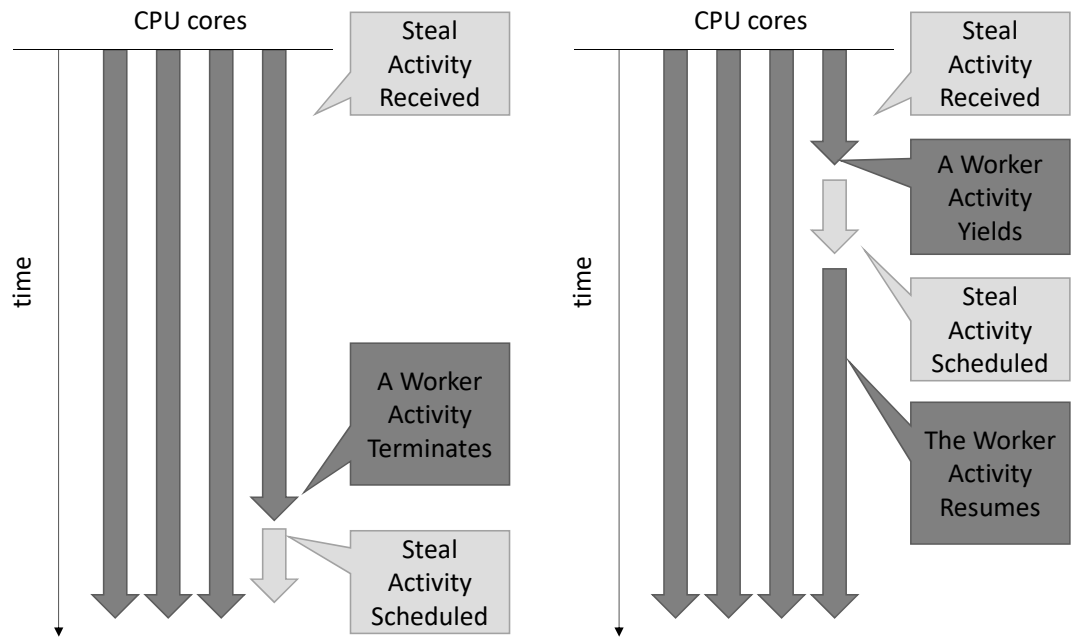
Allowing as many worker activities as there are cores on the underlying host will result in the worker activities monopolizing all the computing resources. As a result, incoming steal requests coming in the form of stealing asynchronous activities will remain staged in the thread pool until one of the worker activities terminates, as illustrated in Figure 3.2a.

This goes against the intent of the work-stealing scheme for two reasons. First, this causes a significant delay between the time the steal request of a remote host is sent and the moment it is actually processed. In the meantime, the thief remains idle. Secondly, the fact that a worker activity terminated on the host is a sign that work is getting scarcer. Therefore, there is good chance that the fragment of work eventually stolen turns out to be relatively small. As a result, the thief will soon run out of work again and restart this ineffective work-stealing process.

In the first implementation of the multi-worker lifeline-based global load balancer in X10 [6], the choice was made to allocate 1 fewer worker than the available number of cores on the running process. This helps scheduling the asynchronous activities coming from remote hosts on the process as there remains one core available at all times for other activities at the cost of reduced computing power.

3.2.2 Yielding worker mechanism

To resolve this scheduling issue, we introduced a yielding mechanism to the routine of worker activities. This mechanism forces a worker activity to stop its progression to allow other activities submitted to the process to execute before resuming execution.



(a) problematic scheduling **(b)** satisfactory scheduling

Figure 3.2: Scheduling of steal activities in the multi-GLB work stealing scheme

Internally, this is implemented using the `ManagedBlocker` feature provided by the `ForkJoinPool` class [40]. Usually, a `ManagedBlocker` is used when a task in the pool needs to perform some kind of long-waiting blocking operation or synchronization. Using the `ManagedBlocker` mechanism to perform such blocking operations allows the `ForJoinPool` to put temporarily suspend this thread and schedule a new thread in its place.

We exploit this mechanism by adding a step in the main routine of workers, consisting of checking if there are any tasks submitted to the pool waiting to be scheduled. If the maximum number of workers is currently running and tasks are pending, the worker voluntarily blocks on a semaphore within a managed blocker, allowing the thread pool to schedule the pending tasks.

Looking at a single worker activity, this yielding mechanism appears to go against the "work first" principle generally observed in work-stealing schemes, but considering the scheme as a whole draws a different picture. First, the operation of this yielding mechanism is only done when the maximum number of worker activities are running, i.e. monopolizing the available cores of the processor. Also, a maximum of one worker activity is allowed to yield at any time. Finally, all activities used in the load balancing scheme proceed to unblock the potentially waiting worker activity before terminating. This guarantees that as soon as any activity completes, be it a steal activity or another worker activity, the yielding worker is immediately allowed to resume execution, as illustrated in Figure 3.2b.

The result is that worker activities and other activities are capable of sharing the resources on the process and intertwine their execution. Steal activities coming from remote hosts are addressed in a timely manner and the maximum available parallelism on the host is used for computation.

3.3 Related work about granularity tuning

Parallel processing generally involves breaking down the computation into smaller parts which can be independently and/or concurrently computed. However, breaking up the computation at hand into the smallest unit possible is usually not the most efficient solution as there is a limit beyond which further exposing the inherent parallelism of the computation does not provide any advantage. On the contrary, it may cause increased management cost or memory footprint. To counter the undesirable effects of over-decomposing computation units, these irreducible independent tasks get packed into larger computation unit. This is referred to as grain coarsening. The number of individual tasks aggregated into these individual units is called the *grain size* or *grain*. One difficulty in precisely defining what the "grain" is stems from the fact that it depends on the context considered. Different programming paradigms introduce different nuances to this notion.

The most popular programming model for parallel computation relies on the Fork/Join model. Typical implementations in shared memory processor rely on a pool of threads, with each thread having its own queue of tasks to process. Tasks can generate new tasks which are added to the worker's queue. When a worker runs out of work, it attempts to steal tasks from a neighboring thread to resume its computation. Several works elaborate on this scheme to reduce the overhead due to the task creation, such as Wang et al [41], or influence the tasks stolen to favor cache consistency, as in LAWS[42] and Constrained Work Stealing [43]. Min, Iancu and Yelick also present their own implementation of a distributed task library over UPC in HotSLAW [44]. They define a hierarchy that matches the characteristics of the (distributed) hardware at hand (cache, socket, and node level). Workers that run out of work try to steal on workers that are close to them first, only stealing from workers further away in the hierarchy if failing to obtain some from close workers. Moreover, they adapt the number of tasks stolen at each level, with closest level steals stealing only 1 or 2 tasks and remote steals stealing half of the tasks contained in the queue. This is a characteristic not supported under the lifeline-based global load balancer as the grain is not related to the amount of work transferred when a bag is `split`.

Our tuning mechanism bears some resemblance with the adaptive grain mechanism presented by Cong et al. in XWS [45]. They reuse the task-parallelism model of Cilk [46] and enhance it with the capabilities of the X10 language to target graph algorithms. In their target applications, each node of the graph at hand represents one task. Coarsening

is achieved by grouping the nodes in the workers' queues in batches. Working threads always process and steal entire batches of tasks. The appropriate batch size for each worker is dynamically adjusted following a heuristic on the current size of its queue. The abstractions offered by the lifeline-based global load balancer are different. First, while programmers may choose to implement their bag as double-ended queue of tasks, there is not obligation for them to do so. The load balancing routines remain oblivious to the internal implementation of the `Bag` abstraction. Secondly, our library guarantees that bags are only manipulated by a single thread at a time. In our scheme, load balance operations using either of the work reserves are done in mutual exclusion whereas in XWS, a worker is allowed to steal from a second worker while this second worker is processing a batch. Again, in our case there is no relationship between the size of the grain used and the amount of work which can be stolen from a bag.

3.4 Grain tuning mechanism

In the context of both the lifeline-based global load balancer and its multithreaded variant, the task granularity corresponds to the number of individual tasks processed during each iteration of the workers' main routine. This is a sensitive setting since this integer parameter does not have any meaning outside the context of a specific application. Setting an arbitrary value on the library side is not satisfactory. Also, we cannot expect users of the library to be able to predict what a good value would be for their application. The ideal grain size will vary depending on the problem at hand, as well as the size of the cluster used. Changes to the implementation of a problem may also change the performance characteristics of a problem.

We aim at eliminating the need for users to guesstimate this value by integrating a tuning mechanism into the load balancer library that will automatically adjust the grain size to achieve good performance. In Section 3.4.1 we detail how the grain size influences the behavior of the load balancer. We then discuss the assumptions and heuristics on which our tuning mechanism relies in Section 3.4.2. Implementation details are briefly discussed in Section 3.4.3 before the evaluation in Section 3.4.4.

3.4.1 Influence of the granularity on the worker activity

The multithreaded global load-balancing scheme relies on several kinds of asynchronous activities to handle the distributed computation [6]. In this section, we will discuss the main routine of the "worker activity" along with some of the load-balancing mechanisms within a host as they are directly relevant to how our tuning mechanism operates. The main routine of the worker activity is presented in Listing 3.1. Note that some elements pertaining to synchronization were removed for clarity.

When an idle host receives work, the computation received is merged into one of the workers' bag and a first worker activity is spawned with that bag given as parameter.

```
1  void workerActivity (Bag workerBag) {
2    do {
3      do {
4        // Step 1 - if able, spawn a worker activity
5        if (runningWorkers < maxWorkers && workerBag.isSplittable()) {
6          Bag fragment = workerBag.split(false);
7          idleWorkerBag.merge(fragment);
8          asyncAt(here(), ()-> workerTask(idleWorkerBag));
9        }
10
11       // Step 2 - if the intra-bag is empty and the worker's bag can
12       //          be split, feed the intra-bag.
13       if (intraBagEmpty) { // volatile boolean flag
14         if (workerBag.isSplittable()) {
15           // Workers will block here in case of extreme contention
16           synchronized (intraBag) {
17             Bag b = workerBag.split(false);
18             intraBag.merge(b);
19             intraBagEmpty = false; // flag update
20           }
21         }
22       }
23
24       // Step 3 - if feeding the inter-bag is needed and the
25       //          workerBag can be split, feed the inter-bag
26       // Step 4 - Check if there are remote thieves that can be
27       //          answered
28       // Step 5 - Yield to load-balancing activities if needed
29
30       // Step 6 - Do some work
31       workerBag.process(n, sharedResult);
32
33       // Repeat from step 1 until the workerBag is empty
34     } while (!workerBag.isEmpty());
35
36     // Step 7 attempt to steal from the intra-bag
37     if (the intra-bag is not empty) {
38       workerBag.merge(intraBag.split(true));
39       intraBagEmpty = intraBag.isEmpty(); // Update the flag
40     }
41     // Step 7-bis if unable to steal from the intra-bag attempt to
42     //          steal from the inter-bag
43     else if (the inter-bag is not empty) {
44       workerBag.merge(interQueue.split(true));
45     }
46     // If work could be obtained, repeat from step 1.
47   } while (!bag.isEmpty());
48   // The worker could not get work from either bag, it stops.
49   // It may be spawned again by another worker performing step 1.
50 }
```

Listing 3.1: Worker activity main procedure

While the worker has some work in its bag, they will loop through steps 1 to 6 of their main procedure (lines 3 to 34 in Listing 3.1), with step 6 consisting in performing the computation. In its first step, the worker checks if it is possible to spawn an additional worker activity in the first step of its main routine. Provided this first worker's bag can be split, another worker activity will be spawned, which will in turn (transitively) spawn other workers until the maximum number of concurrent workers on the host is reached.

In steps 2 and 3, the workers try to maintain work in both shared bags on the place. If a worker performing step 2 finds that the intra-bag is empty and that it is capable of splitting its bag, the worker splits a fragment from its bag and merges it in the intra-bag. The inter-bag involved in step 3 follows a similar scheme. We do not detail steps 4 and 5 which are involved in guaranteeing the scheduling of work stealing activities as discussed in Section 3.2

When a worker runs out of work after performing step 6 and exits the inner do-while loop of its procedure, it will attempt to take some computation from the intra-bag in step 7, or as a last resort from the inter-bag in step 7-bis, to immediately resume its computation (lines 36 to 45 in Listing 3.1). If the intra-bag gets emptied as a result, another worker performing step 2 will place some computation back into it. The next worker to run out of work will therefore be able to take some computation from the intra-bag again.

If a worker runs out of work when neither the intra-bag nor the inter-bag contain any work, it will escape the outer `do while` loop (line 47 in Listing 3.1) and terminate. A new asynchronous worker activity may be spawned back again by a worker performing step 1 of its main routine.

The attentive reader will have noticed the parameter "`n`" of the `process` method in step 6 of the worker's main procedure (line 31 in Listing 3.1). This integer determines the grain size. In general, it should be seen by programmers as the number of indivisible computation units to be performed in a call to method `process` before returning. As a consequence, the grain size is correlated to how much time workers spend in step 6, influencing how often they go through the inner do-while loop. If this parameter is low, the worker activities will go through their loop more frequently. Conversely if the chosen grain is large, workers will spend more time in step 6 and go through the loop less frequently.

The purpose of the two shared bags on each host is for workers that run out of work to steal from them and continue to participate in the computation. An issue that arises when the grain is too large is that when these queues become empty, there is a delay until a worker checks the queues status in steps 2 and 3 and puts some computation back into them. As a result, workers that run out of work are more likely not to be able to steal any work in step 7 and terminate, reducing throughput. These workers will eventually be spawned back by other workers performing step 1, but for the same reason this will also happen after a delay. A situation where the grain size is too large on a

host will therefore be characterized by intervals of time where fewer than the maximum number of concurrent workers are running.

As workers go through the inner loop of their procedure, various checks are made. These consists of reading some volatile boolean flags and calling methods of atomic data structures. These are quite lightweight, but they will still generate some overhead if they are made too often. Moreover, with many workers running in parallel, there is also an increased risk of contention on the bags used for load balancing when load-balancing operations are actually needed. Since the accesses to the bags are made in mutual exclusion using synchronized blocks, we risk creating a bottleneck by using a grain size too low.

3.4.2 Heuristics

The tuning mechanism we integrated into the multi-GLB relies on a pair of heuristics to construct a basic feedback mechanism.

3.4.2.1 Diagnosis of a grain too large

As explained above, executions with a grain too large will cause workers that run out of work to remain idle for prolonged periods of time. As an indicator that the task granularity is too large, we use the proportion of the time spent with the maximum number of workers. If this proportion drops below a certain trigger level, it is inferred that the grain currently in use is too large. Based on the empiric characteristics of static grain executions of the Unbalanced Tree Search benchmark, we deem the grain size to be too large if less than 90% of the elapsed time is spent with the maximum number of workers.

3.4.2.2 Diagnosis of a grain too small

We developed two heuristics to detect cases where the grain is too low. Both rely on a subtle effect low grain executions have on the handling of the intra-bag.

When a worker empties the intra-bag, it sets the boolean `intraBagEmpty` flag to `true` in line 39 of Listing 3.1. The next worker to perform the second step of its main routine will read the value of the flag as `true` in line 13 and (if able), place some work back into the intra-bag before setting the `intraBagEmpty` flag back to `false` on line 19. Any subsequent worker to perform the check in line 13 will read the updated value of the boolean flag and move on to the next step of its routine without placing work into the intra-bag.

However, it is possible for multiple workers to place work back into the intra-queue as a result of it being emptied once due to a data race between the "read" on the `intraBagEmpty` flag on line 13 and the "write" of the first worker placing work back into the queue on line 19. Usually, data races are best avoided in concurrent programs.

However in this particular case, it does not adversely affect the correctness of the program thanks to the synchronized block protecting the access to the intra-queue spanning lines 16 to 20.

This situation where the intra-bag is likely to be fed several times after getting emptied only once is more likely to occur in situations where the grain is low. Our tuning mechanism leverages that fact to detect this situation.

We could eliminate the redundant feeding of the intra-bag by changing our volatile boolean flag for an atomic integer. However, the redundant feeding isn't a performance problem in itself. Rather, the fact that it occurs beyond a reasonable level is the sign that workers go through their loop too often, creating overhead. In such a situation, the performance suffers more from the overhead created through excessive checking than from the contention on the shared queue when it becomes empty. As our tuning mechanism detects this situation and increases the grain size, any contention on the queues will naturally disappears.

Early "split/merge" design In an early design we introduced, we used the ratio between the number of times work is `split` from the intra-bag and the number of times work is `merge`d back into the bag to determine if multiple workers were able to redundantly feed the intra-bag. We call the tuner that relies on this criterion "split/merge tuner." This indicator, however, relies on the assumption that the programmer implements the `split` method of its bag such that successive calls to this method recursively give away half of the contents of the bag. Under this assumption, if the intra-bag is fed by multiple workers as a result of being emptied once, it will take comparatively fewer `split` calls to empty it again than it would have if a single worker placed work into the intra-bag each time it got emptied.

Using empiric data, we set the split/merge "trigger" level to 2, meaning that if fewer than two workers are able to take work from the intra-queue for each time work is placed back into the queue, the grain is deemed too small.

"merge/empty" design We have since departed from this criterion and designed a new version of our tuning mechanism. We now directly measure the number of redundant feedings of the intra-bag by comparing the number of times the intra-bag was emptied in lines 39–41, with the number of times a worker puts work back into the intra-bag in lines 17–19 of Listing 3.1.

We call this new criterion "merge/empty" because in the context of our global load balancer library, it corresponds to the number of times workers `merge` work into the intra-bag divided by the number of times the intra-bag is emptied. From a runtime perspective, it is the ratio between the number of workers that go through the `if` block in lines 13–22, and the number of times the boolean flag which guards this `if` block is set to `true`, allowing workers to enter it.

3.4.3 Integration with the GLB runtime

The tuning mechanism is implemented as an extra asynchronous activity, the *tuning activity*, on each place of the global load balancer's runtime. The tuner is called periodically and remains inactive the rest of the time.

When the tuner is called, it directly reads the information accumulated in the load balancer's local logger. By comparing the current values with the ones from the previous time the tuner was called, the tuner is able to determine what took place during the last interval. It can then evaluate the heuristics mentioned above, draw its conclusions, and modify the grain size if necessary.

Initial experiments showed that the two indicators we use to detect if the grain is too low or too high are not infallible. Throughout the execution, there are times when the indicators draw the "wrong" conclusion or contradict themselves. As a moderation mechanism, we choose to only modify the grain size if the same conclusion is drawn by the tuner twice in a row.

When changing the value, we double (or divide) the current value by a factor 2. Combined with a tuning interval of 1 millisecond, this allows us to cover the very large range of values that the grain can take over a short period of time. In all our experiments, we purposely set the initial grain size at a very low value of 10. The tuner activities of each place are free to adjust the grain as soon as the computation starts, and do so independently from one another. As a result, the chosen grain on two different compute nodes of the distributed computation may differ.

We did not witness any overhead imputable to this extra activity. This was checked by running distributed computations with the tuner activity active but keeping the chosen grain fixed. These executions produced the same execution time as regular fixed grain executions without this additional activity. This can be explained by the fact that the decision making takes an insignificant amount of computing power.

3.4.4 Evaluation

3.4.4.1 Benchmarks used

To evaluate the performance of our tuning mechanism, we use four backtrack-search applications: N-Queens, Pentomino, the Traveling Salesman Problem (TSP), and the Unbalanced Tree Search (UTS).

We implemented these problems in a similar manner, using a pair of arrays to describe ranges of branches at each level of the exploration tree. Splitting the exploration is reduced to dividing this interval into two, matching the lowerbound of the thief to the upper bound of the victim for each layer of the exploration. This operation is therefore bounded in time and space by the depth of the exploration. As the branches are implicitly described, the size of the data transfered from host to host during load balance operations is independent from the actual amount of work transferred.

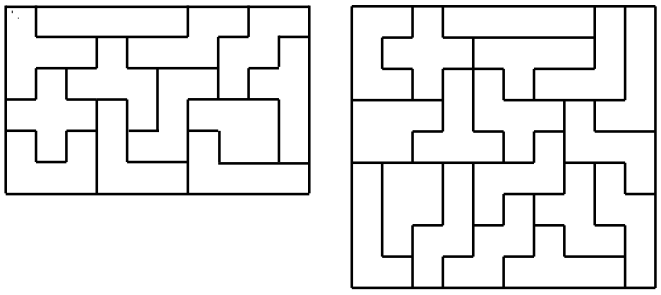


Figure 3.3: A particular solution to the Pentomino and the One-sided Pentomino problems

In this section, we briefly introduce each application and discuss some selected details about our implementations.

N-Queens The N-Queens problem consists in finding all possible arrangements such that a maximum number of Queens are placed on a chessboard of width N without any two Queens threatening one another. We model the problem as an exact cover problem, which consists in finding all the different subsets of rows of a matrix of 0s and 1s such that for each column of the matrix, exactly one row has a 1 in that column. In the case of the N-Queens problem, the columns of the cover matrix correspond to the files, ranks, and diagonals of the chessboard. Each row in the matrix corresponds to a possible queen placement on the board and contains four 1s: one for the rank, the file, the diagonal, and the anti-diagonal that the piece occupies.

We use Knuth's "Dancing Links" data structure [47] to represent the sparse matrix of the exact cover problem. This data structure exploits doubly linked lists to hide and restore parts of the matrix as the backtrack exploration progresses. Exploration is made in a depth-first manner. At each step in the exploration, the first column of the cover matrix that remains to be covered is arbitrarily chosen. The various rows that can cover this column represent the options available in the exploration and can be explored in parallel.

Pentomino The pentominoes are the 12 different shapes that can be formed by stitching 5 square tiles edge-to-edge. The Pentomino problem consists in enumerating all the possible ways to arrange these 12 shapes to cover a 10x6 rectangle. The One-sided Pentomino is an analogous problem but treats the face-down variations of the chiral pentominoes as pieces of their own. As a result, the problem is significantly larger, consisting of arranging 18 pieces on a 10x9 rectangular board. A solution to the Pentomino and the One-sided Pentomino problems are presented in Figure 3.3.

In our implementation, we use a single array with sentinels to represent the rectangular board. We recursively attempt to place every rotation of every piece on the top-most and left-most unoccupied tile of the board. If the piece can be placed, the exploration

proceeds and we attempt to place the remaining pieces of the board. If the chosen piece cannot fit on the board, the next orientation and/or piece is selected as a candidate. When all the candidates at a certain stage of the exploration have been attempted, the exploration backtracks by removing the last piece that was placed and selecting a new candidate. Some restrictions on piece placement and orientation can be made to eliminate the symmetries of the problem and only enumerate the fundamentally different solutions to the problem. This also reduces the size of the exploration tree. In terms of scaling potential, this problem shows a wide exploration tree and a low computation cost for exploring individual nodes. Of all our applications, the Pentomino has the greatest potential for strong scaling on larger clusters.

Traveling Salesman Problem The Traveling Salesman Problem (TSP) is an optimization problem which consists in finding the shortest round-trip through a number of cities. In our implementation, we use an exact Branch & Bound algorithm [48]. The nearest neighbor heuristic is used to favor exploration of cities that are close to the current city first. The length of the best round-trip found so far is kept in a shared object on every place in the computation. Our library periodically checks on each place if a better bound has been found and, if so, recursively propagates newly found bounds to remote places using the same network as the lifeline graph.

We should note that the non-deterministic nature of the load balancer can introduce great variations in the execution times of this problem as it influences how early better solutions are found and how much of the exploration tree is trimmed. This problem is also prone to poor scalability when tested in strong scaling due to the parallel exploration of branches that would be trimmed out if a better bound had been found earlier.

UTS The Unbalanced Tree Search [39] consists in a depth-first traversal of a randomly-generated tree. We use geometric trees in which the number of children of each tree node follows a geometric distribution of average 4. The resulting tree is unbalanced by construction as two nodes on the same level are unlikely to spawn similar size sub-trees. The size of the exploration is adjusted by setting a maximum depth to the tree. The shape and the size of the tree are entirely deterministic following an initial seed and the maximum depth. Parallel traversal is done by exploring nodes that have not been traversed yet and whose sub-trees have yet to be generated.

We conduct the evaluation of this problem in weak-scaling, that is, we use increasingly larger trees for increasingly larger clusters.

3.4.4.2 On many-core clusters

We first perform an evaluation of the global load balancer library without the tuning mechanism on the OakForest-PACS supercomputer to determine the best performance achievable on each of our problems. As a sample, we show the relationship between the

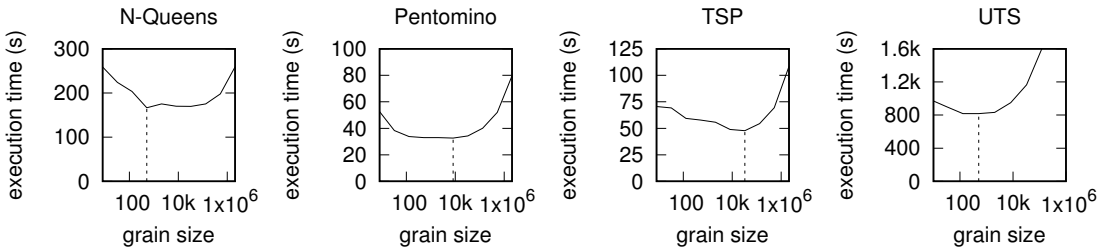


Figure 3.4: Relationship between the grain size and the execution time of our four benchmarks when running on 16 hosts of the OakForest-PACS supercomputer

Table 3.1: Problem settings used for the experiments involving varying number of workers on the Oakforest-PACS supercomputer and our Beowulf server

Problem	Oakforest-PACS	"harp" server
UTS	Branching factor: 4, Depth: 18 on 4 hosts, 19 on 8 and 16 hosts, 20 on 32 hosts, 21 on 64 hosts	Branching factor: 4, Depth: 17
Pentomino	One-sided, Board width: 9, Board height: 10, with symmetry removal	One-sided, Board width: 9, Board height: 10, with symmetry removal
N-Queens	$N = 19$	$N = 17$
TSP	35 cities	24 cities

grain size and the execution times on our 4 application problems in Figure 3.4. On each problem, we have a range of acceptable values grain values, which can be wide (such as Pentomino) or more narrow (like TSP and UTS). Although the range of acceptable grain sizes tend to overlap between different clusters, they can shift or get narrower as we increase the cluster size. We therefore conducted a thorough evaluation to identify the best performing grain for each of our problem on every cluster configuration. We compare how both our tuning mechanisms fares against the best "fixed grain" executions.

The main characteristics of the OakForest-PACS supercomputer are summarized in Table B.2. The problem parameters we used are shown in Table 3.1. The results are summarized in Figure 3.5.

In general, both our tuning mechanisms operate as intended on NQueens, Pentomino, the Traveling Salesman problems, and UTS, delivering close to the best fixed grain executions recorded. We even achieve slightly better performance on the TSP when running on 4 to 16 hosts with our "merge/empty" tuner. We also note that our new "merge/empty" tuner shows identical performance (on Pentomino and UTS) or better performance (on N-Queens and TSP) than the "split/merge" tuner. This is a net improvement over our initial design and is not the only advantage brought about by the new criterion, as we

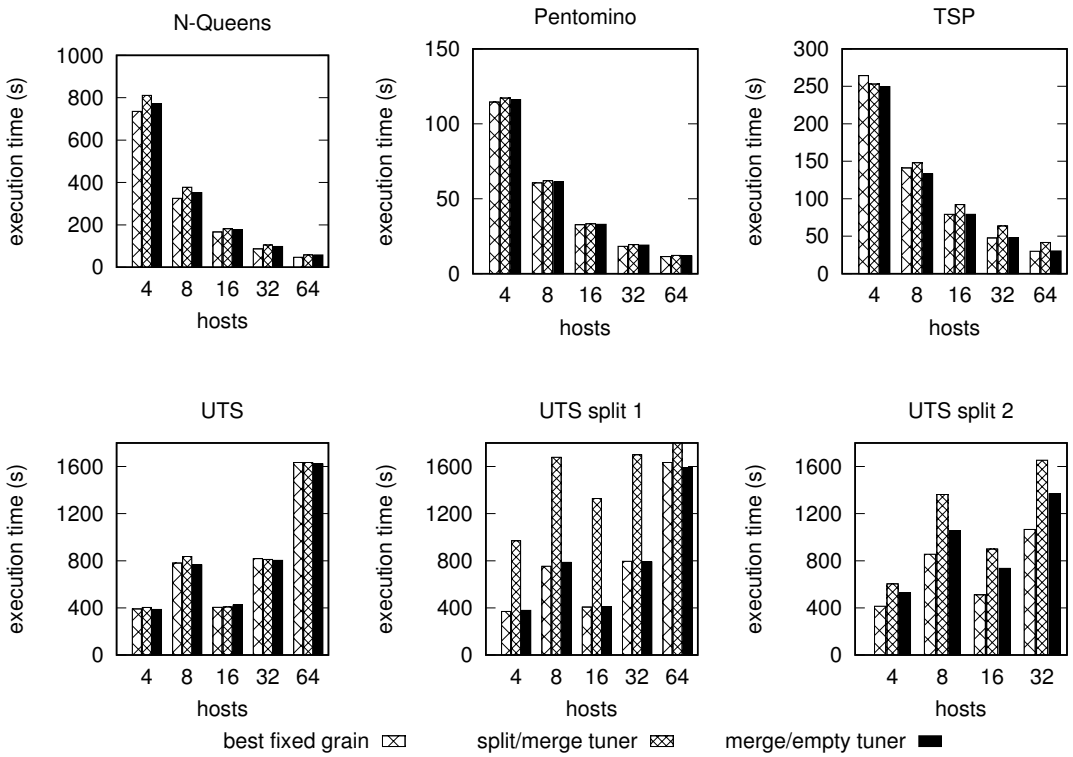


Figure 3.5: Execution time comparison between the best fixed grain and our "split/merge" and "merge/empty" tuning mechanisms on the OakForest-PACS supercomputer

will discuss in the following section.

3.4.4.3 Robustness

One concern with the early "split/merge" criterion used to detect cases where the grain is too small is its reliance on the implementation of a reasonable "split-half" strategy by the `Bag` implementation of the user program. In cases where the program received by the multi-GLB does not follow this recommendation, this criterion may fail to correctly recognize the situation. To confirm this weakness and to demonstrate the robustness of our "merge/empty" criterion, we implemented two variants of the UTS benchmark which differ in the implementation of the `split` method.

- *UTS split 1*: The intra-bag gives away its entire contents when the split method is called on it. As a result, the maximum split/merge ratio is 1, and lower if the intra-bag is redundantly fed as is the case at lower grain sizes.
- *UTS split 2*: The intra-bag successively gives away half of fragment received. The number of split operation needed to empty the intra-bag is therefore proportional to the number of times work was merged into it. As a result, the split/merge ratio remains largely the same, regardless of if there were redundant merges made.

The relationship between the grain size, the execution time, and the two ratios that we used in our tuning mechanisms to detect when the grain size is too low are presented in Figure 3.6. We can see that the execution time depends on the proper balance of the grain chosen for the execution. We can also recognize just how much the intra-bag is redundantly fed at lower grain sizes by looking at the "merge/empty ratio" plot. We note that the merge/empty curves of the regular UTS implementation and the two variants are almost identical. We therefore expect our new tuning mechanism which relies on this criterion to be able to accommodate for these vastly different splitting implementations.

On the contrary, the split/merge ratio our previous design relies on shows great disparity depending on the implementation. We had chosen to describe programs whose split/merge ratio was below 2 as having a grain size too low. This worked for the original splitting implementation (labeled "UTS" in Figure 3.6) but will not for the other two implementations. With the *split 1* and *split 2* UTS variants, the "split/merge" ratio remains respectively below or above 2 regardless of the grain size used. As a result, our original "split/merge" tuner will fail to recognize the situation correctly for both of these problems and yield poor performance.

Looking at the results on the two UTS variants in Figure 3.5, the limits of our "split/merge" tuner become evident, sometimes presenting execution times more than double the best fixed grain achieved. By contrast, our new design yields execution times almost identical to the best fixed grain executions on the "split 1" variant. The largest gap occurred on the 16 hosts configuration with an execution time longer by just 30 seconds.

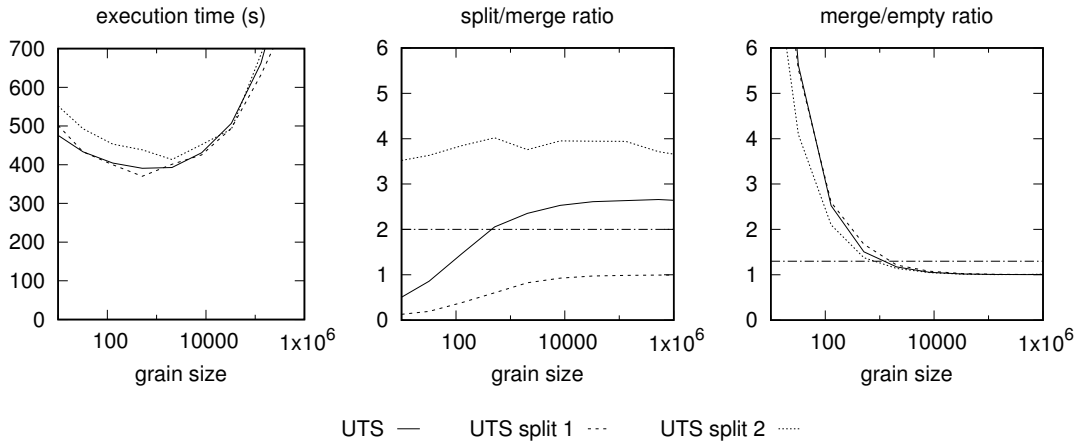


Figure 3.6: Execution time, split/merge, and merge/empty ratios of 3 implementations of the UTS benchmark depending on the grain size chosen
The trigger value for both criterion is indicated as a horizontal dotted line.

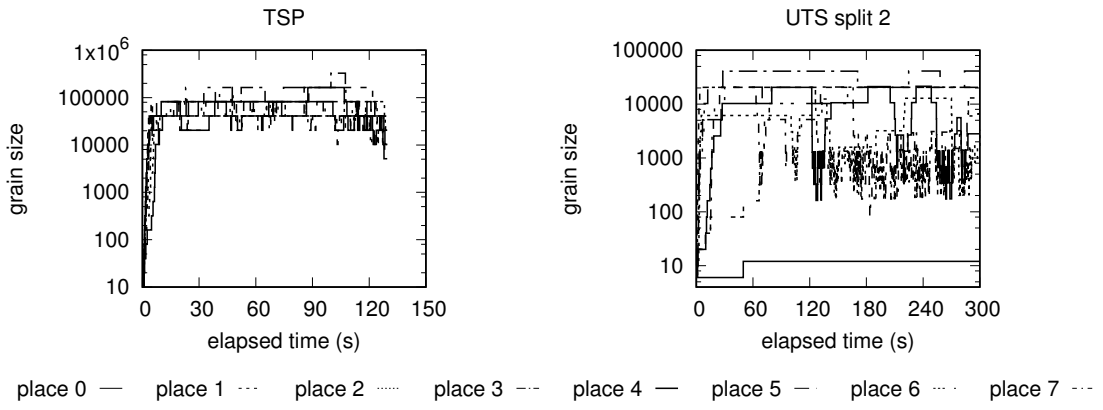


Figure 3.7: Evolution of the grain size on a 8 host execution of the Traveling Salesman Problem and the UTS split 2 benchmark on the OakForest-PACS supercomputer

On the "split 2" variant, our new tuner design clearly outperforms our first "split/merge" design. However, we observe a certain performance gap to the best fixed grain executions with run times about 20% longer (the largest gap being 40% on the 16 nodes execution). This can be explained by the grain value chosen by our tuning mechanism. Figure 3.7 presents the evolution of the grain size as chosen my our "merge/empty" tuner on an eight host execution of the TSP and the UTS "split 2" variant. On the TSP execution, our tuner mechanism behaves as intended, increasing the grain size from its initial value on all the hosts. However, on the UTS split 2 variant, place 0 keeps a very small value for the first 5 minutes presented here. It is only in the last five seconds of this execution (not shown in Figure 3.7) that the grain size on host 0 is finally increased. We are able to account for this phenomenon, which we discuss in Section 3.4.4.4.

3.4.4.4 Limitations

So far, we focused our analysis on clusters of many-core processors. However, nothing is preventing us from using our library on clusters of ordinary multi-core processors. In fact, this load-balancing scheme was first designed for such systems [6].

We were concerned that our tuning mechanism would not work with fewer workers per place. In particular, the criterion that determines if the grain size is too low relies on multiple worker threads entering the same branch of their main routine before one of them completes it. As we arbitrarily chose to launch the computation with a small grain value, our tuning mechanism may fail to raise the grain size to a satisfactory level by lack of contention in the workers' main procedure.

We evaluated the performance of our tuning mechanisms on the "harp" server of our Beowulf cluster. This server is fitted with two 12-core Intel Xeon processors. The hardware details of this server are presented in Table B.1. We evaluate the performance of our four benchmarks on this host in three different configurations:

- `24x1`: 24 workers per process, 1 process
- `12x2`: 12 workers per process, 2 processes
- `6x4`: 6 workers per process, 4 processes

The results are presented in Figure 3.8. With the exception of the UTS benchmark, executions with either of our tuning mechanisms show a performance gap compared to the best performance obtained with fixed grain executions. However, the cause of these gaps was not what we anticipated.

A detailed look at the grain chosen by our tuning mechanisms allows us to obtain more insights. Representative grain evolutions over time for each cluster configuration are presented in Figure 3.9. On the execution with a single process, the tuner keeps the grain size at its initial value of 10 for a long time. On the particular execution shown

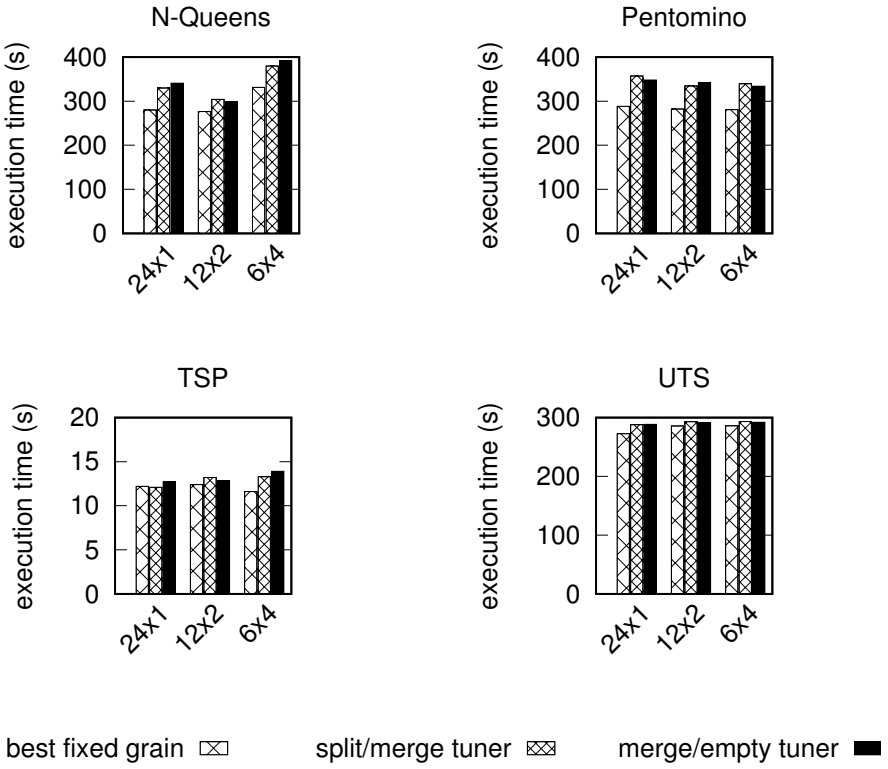


Figure 3.8: Execution times of our four benchmarks in the three configurations on our "harp" server

in Figure 3.9, the grain size is suddenly increased to the ideal range after 2 and a half minute have elapsed.

We can explain this phenomenon by the nature of the criteria used to detect that the grain is too low. In its current design, the intra-bag needs to be emptied for the tuner to obtain data to be able to make its decision. In an execution where all the work is concentrated on the single process participating in the computation, the workers all obtain a large amount of work to begin with. They will therefore be able to keep going through their loop without running out of work for a long time. Moreover, a large amount of computation is likely to have been placed in the intra-bag. This means that even when the workers start running out of work, it may also take a while to empty the intra-bag for the first time. Only when load imbalance actually occurs on the single process involved in the computation will the tuning mechanism be able to recognize that the grain is too low and finally raise it.

We see a similar pattern on the two- and four-process executions, with place 0 keeping its grain value at its initial value of 10 up to the very end of the computation when it finally jumps. On the other places however, the grain size is increased by the tuning mechanism from the start. This can be explained by the fact that these places obtain a fragment of the computation held by place 0 through their lifeline steals. This fragment being smaller, they need to perform load-balancing tasks operations sooner.

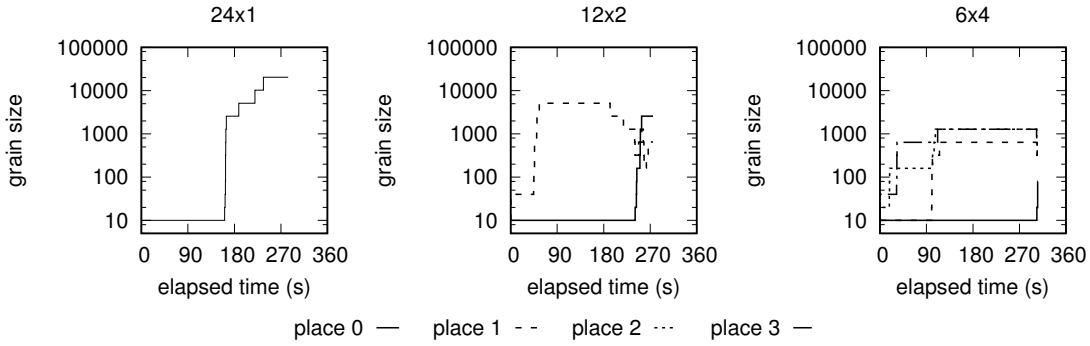


Figure 3.9: Evolution of the grain size over time depending on the cluster configuration of our Beowulf server using our "merge/empty" tuner on the TSP problem

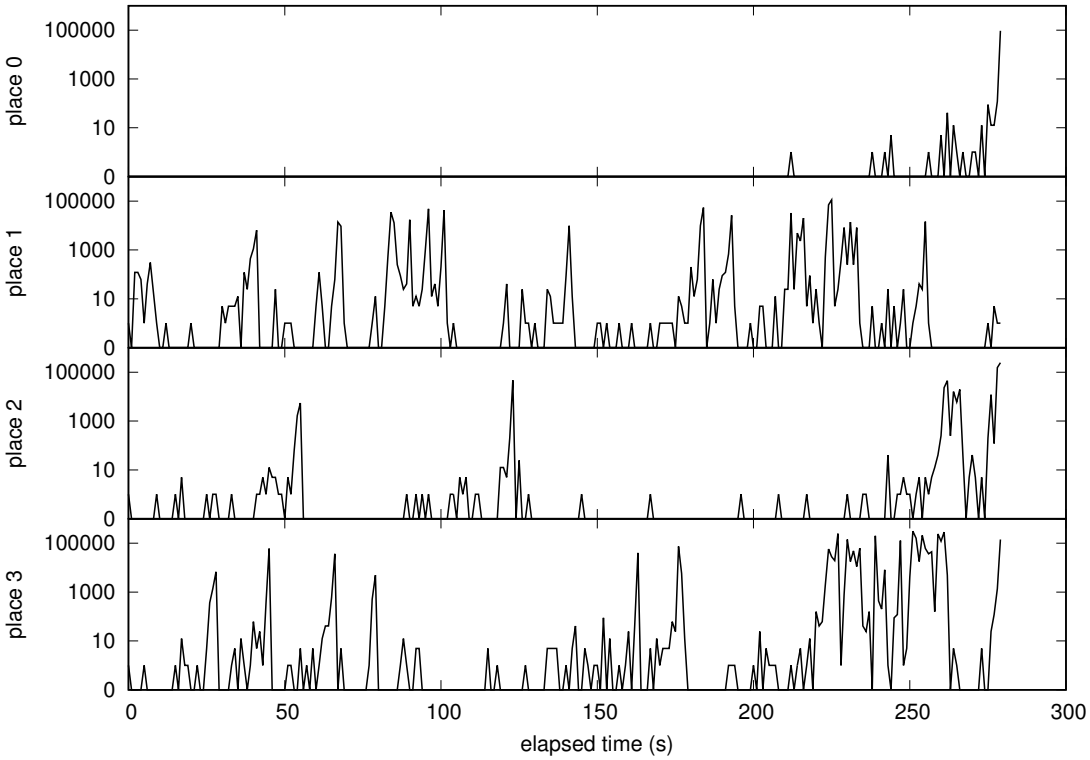


Figure 3.10: Evolution of the number of times the intra-bag is emptied on each place of a four-place execution of the Pentomino problem on our Beowulf server

This is confirmed by how frequently the intra-bag is emptied throughout the computation. If we focus on Figure 3.10, we can see that the places 1, 2, and 3 face situations where the intra-bag is emptied from the start of the computation. By contrast, the intra-bag is emptied on place 0 after 240 seconds have elapsed in the 270 second execution. This reassures us in the capability of our tuning mechanism to operate on systems with fewer concurrent workers as it appears that the reduced number of worker per place is not what causes the tuner to fail. Rather, it is the lack of load imbalance that prevents it from recognizing situations where the grain is too low.

We believe this also explains the performance gap we saw with our "merge/empty" tuner design on the UTS split 2 variant on the OFP supercomputer. The particular implementation of the UTS split 2 variant yields little work when the `split` method is called. This causes the workload to have greater difficulty trickling down from the bags used for load balancing on place 0. As a result, it will take longer for the intra-bag of place 0 to get emptied and for the tuning mechanism to detect the overhead on this process.

We have attempted to resolve this problem by spuriously setting the `intraBagEmpty` flag to `true`, making workers feed the intra-bag as if it had being emptied. However, these efforts have not to come to fruition. Another approach for multi-process executions could consist in homogenizing the grain size used on the entire cluster. This would cause the higher values of the remote hosts to raise the value used on the first process from the start of the computation.

3.5 Conclusion and future work

Our tuning mechanism for the multi-threaded lifeline-based global load balancer is capable of correctly adjusting the grain size of the computation on both cluster of many-core processors and on more common multi-core environments. It does so by sampling the runtime of the load balancer, relying on a pair of heuristics to determine if the current grain is either two small or too large. Cases of grain too small are diagnosed through the occurrence of a data race in the worker's main loop, while the cases of grain too large are diagnosed through the apparition of starvation between the workers. The mechanism is robust against changes in the implementation of the computation being load-balanced and incurs no noticeable overhead.

However, to be able to successfully achieve this, it requires some load balance operations to take place. In cases where the computation remains concentrated on a single host, either through an inadvisable work splitting implementation or by running on a small (maybe single-process) environment, it is possible for our current design to miss the presence of overhead and keep grain sizes that are too low to deliver good performance.

We believe the mechanism relying on a data race has the potential to be re-used in other contexts to help estimate the state of contention of a resource in cases where no

obvious performance issue is detected. Some kind of probabilistic model may provide further insights into this phenomenon such as the relationship between the number of threads and the likelihood of a data race occurring.

The objective of the scheme presented here is to keep all the processes participating in the computation busy for as long as possible. In Palirria [49], Varisteas and Brorsson tackle the reverse problem consisting of matching the computing resources to the (varying) degree of parallelism of an application. On single applications, they are able to maintain ideal performance with higher efficiency by adjusting the number of allocated cores to the computation based on its potential for parallelization. Similar to our approach, they are able to make a decision on whether to change the number of cores allocated to their program based on the measurement of some selected runtime metrics over the most recent elapsed interval. Their approach makes it possible to envision several programs running concurrently and making the most of the available computing power of a shared memory processor. More recently, Posner and Fohry transposed this idea to our implementation of the multi-worker Lifeline-Based Global Load Balancer [50]. With their modifications to the APGAS for Java library and their adaptations to the load balancing scheme, the lifeline-based global load balancer becomes an elastic application capable of dynamically reducing and increasing the number of processes it is running on as the needs of the computation evolves.

Chapter 4

Distributed Relocatable Collections

4.1 Introduction

Writing parallel and distributed programs is inherently difficult, with many dedicated languages, runtime libraries, and programming models attempting to reduce the difficulty by introducing abstractions to programmers. In judging what makes a good language and runtime for object-oriented distributed computation, we are interested in 3 criteria:

- support for local parallelism
- support for communication across processes
- dynamic data/work relocation

In the first aspect, X10 and APGAS for Java are quite successful. Indeed, the asynchronous activity concept and the use of the `finish`/`async` constructs to manage these activities allows programmers to launch as many independent tasks as they want. Matching the number of threads to the available parallelism on a host is therefore trivial. The introduction of the `Place` abstraction allows programmers to clearly indicate the process on which computation is done.

In the latter two aspects, X10 and APGAS for Java are lacking. There is no official communication layer provided with the language, forcing programmers to rely on external libraries. When it comes to load balancing capabilities, the lifeline-based global load balancer discussed in the preceding chapter only applies to self-contained tasks. For objects that persist across iterations, there are no obvious facilities provided with the language.

To address these shortcomings, we introduce *relocatable distributed collections* in the form of a library to complement APGAS for Java. Relying on a combination of the APGAS for Java programming model [36] and MPI, our library makes it possible to write complex distributed and parallel programs with ease. We introduce the notion of "teamed operation" to describe computation or communication patterns that involve multiple

processes. We also introduce a number of intra-node parallel patterns operating on these collections, such as reductions and producer/receiver. Our distributed collections come with an API close to that of the Java standard library, providing a sense of familiarity to programmers who are then capable of reusing any prior knowledge.

The remainder of this chapter is organized as follows. We start by detailing how we combined the runtimes of APGAS for Java with MPI in Section 4.2. We then formally introduce key concepts of our relocatable distributed collections in Section 4.3. In Section 4.4, we showcase the main features of our library using distributed programs taken from well-known Java benchmark suites [51, 52] and a complex simulator [53, 54].We then discuss specific design choices and select implementation details in Section 4.5. We evaluate the performance of our library in Section 4.6. Finally, we conclude and discuss future work in Section 4.7.

4.2 Combining APGAS for Java with MPI

The APGAS for Java and MPI runtimes are quite compatible. In fact, the X10 programming language has a runtime implementation built on top of MPI. Each process becomes the combination of an APGAS "Place" and an MPI "rank" and the terms "process," "Place," and "rank" can therefore be used interchangeably in this context. One difference when combining APGAS for Java and MPI is that unlike pure MPI programs, only rank/place 0 runs the program "main". Code is executed on other ranks through asynchronous activities managed by AGPAS.

As part of our library, we introduced some classes that supplement the existing APGAS constructs. While these additions have little technical merit on their own, they bring some convenience to the programming model of APGAS and are use throughout our library. The most significant addition for the purposes of our library consists in class `TeamedPlaceGroup`.

Class `TeamedPlaceGroup` represents a group of APGAS places. This class proposes a `broadcastFlat` method taking a closure as parameter which will spawn the provided closure in an asynchronous activity on each place within the group and returns when the provided closure has completed on all places. A "world" group which contains all the places participating in the computation is initialized by our library and can be obtained through the `TeamedPlaceGroup.getWorld()` method. Other groups containing a subset of the "world" can be created at will.

We introduce Listing 4.1 to illustrate the benefit of using class `TeamedPlaceGroup`. Notice that the `broadcastFlat` method call on line 3 replaces the finish/asyncAt loop used in Listing 2.1. Overall it is a practical shorthand which simplifies programs by mimicking the MPI programming style within a clearly identified block. We use it extensively when writing programs with our library.

Internally, the `TeamedPlaceGroup` carries an MPI communicator which is used by our

```
1 TeamedPlaceGroup world = TeamedPlaceGroup.getWorld();
2 world.broadcastFlat(()-> {
3   System.out.println("Hello from " + here());
4 });
5 System.out.println("Bye");
```

Listing 4.1: Equivalent program to Listing 2.1 using class TeamedPlaceGroup

library to communicate information between the places participating in the group. A number of convenience methods that translate APGAS places into MPI ranks and vice-versa are also provided. While the runtime we rely on combines APGAS for Java and MPI, we cannot consider it to be an hybrid technique on the same level as MPI+OpenMP. We rely primarily on APGAS to manage code execution locality and termination detection, with MPI taking up the secondary role of supporting specific communication patterns. Our library is structured in a way that would allow us to substitute MPI for a different communication layer with some adaptations.

4.3 Relocatable distributed collections

The concept of distributed collections is not new. The work we present here bears resemblance with earlier work from Lee & Gannon [55] in which they define the *Distributed Collection Model* for the pC++ programming language. Under this model, a distributed collection contains elements that can be referenced through a globally unique handle. A distribution describes how the elements are assigned to the virtual processors used at runtime. Parallelism is supported by sending a message to the collection which will in turn invokes the specified method on all elements of the collection. It is also possible to send such a message to a subset of the virtual processors. One peculiarity of this model is the capability for individual elements to obtain information from the structure of the collection (i.e. their position in a 1D array or 2D grid). One limitation of this programming model is that there is a single main control thread for the program resulting in calls on an entire collection to be synchronous. Under the APGAS programming model this constraint is relaxed, with the progress of asynchronous activities on various hosts being only halted if some communication between hosts is needed as part of the activity. And while multiple distribution strategies are available in this language, there is also no obvious mechanism that would allow to modify the distribution of a collection.

In the context of the APGAS programming model, a *distributed collection* consists in a group of *local handles* linked by a globally unique identifier. We say that a collection is *defined* on a group of places to represent the fact that a collection has a handle on each place belonging to this group. When creating a new distributed collection, the `TeamedPlaceGroup` on which the collection will be defined is given to the constructor as a parameter. Object entries can be recorded into the local handles of collections and

be either kept in these handles, or a distributed collection may be used to temporarily record information produced on a host before transfering this information to a single process where it will processed. We also provide support for parallel computation taking place on every instance recorded into a collection, taking the parallelism avaialble on the underlying host. We also support common distributed computation patterns such as reductions using the collections as the provider of input data and its definition the subset of processes involved in the distribution computation. We also foresee situations where some information updated by a process needs to be duplicated to the other hosts participating in the computation.

4.3.1 Proposed collections

The main collections we provide with our distributed collections library are summarized in Table 4.1.

Table 4.1: Collection classes proposed by our library

Collection	Description
`Bag<T>`	Iterable set
`DistBag<T>`	Distributed variant of class `Bag`
`CachableArray<T>`	Array used to share and replicate information across processes
`ChunkedList<T>`	Arbitrary long-index array
`DistChunkedList<T>`	Distributed variant of `ChunkedList`
`DistCol<T>`	Variant of `DistChunkedList` whose distribution is tracked
`CachableChunkedList<T>`	Variant of `DistCol` whose entries can be replicated on multiple hosts
`DistMap<K,V>`	Distributed map from `K` to `V` objects
`DistConcurrentMap<K,V>`	Variant of `DistMap` with additional protections for concurrency
`DistIdMap<V>`	Distributed map from `long` indices to `V` objects, its distribution is tracked
`DistMultiMap<K,V>`	Distributed map from `K` objects to multiple `V` objects

Bag<T>

The `Bag` collection (and its distributed variant `DistBag`) consist in a (distributed) iterable set. Duplicated entries are allowed. Special care was taken to its internal structure for it to efficiently receive elements from multiple concurrent threads.

```
1  TeamedPlaceGroup world = TeamedPlaceGroup.getWorld();
2  DistMap<String, String> dMap = new DistMap<>(world);
3  dMap.put("main", "running");
4  world.broadcastFlat(()->{
5    dMap.put(here(), "says hello");
6    CollectiveMoveManager mm = new CollectiveMoveManager(world);
7    if (here() == place(0)) {
8      dMap.moveAtSync("main", place(1), mm);
9    }
10   mm.sync();
11 });
```

Listing 4.2: Distributed map creation, record insertion, and relocation example

CachableArray<T>

A cachable array takes the form of an array containing objects that need to be replicated on each host and may be periodically updated. Custom serialization and deserialization methods can be specified to use a user-chosen object to transport the updates to replicas.

ChunkedList<T>

Class `ChunkedList` and its distributed variants propose a collection which handles elements in multiple one dimension arrays mapped from ranges of `long` indices. We call each of these arrays mapped from a range of indices a "chunk". Individual elements can be accessed and set through their `long` index. Some computations and/or manipulations on the distributed collections can be applied on ranges of entries.

We developed variants based on this class allow for more specific behaviors such as guaranteeing that chunks are unique across all hosts, or for chunks to be replicated on other hosts. This enables support of various distributed applications in which replication of entries, entry distribution tracking, or other features are desired.

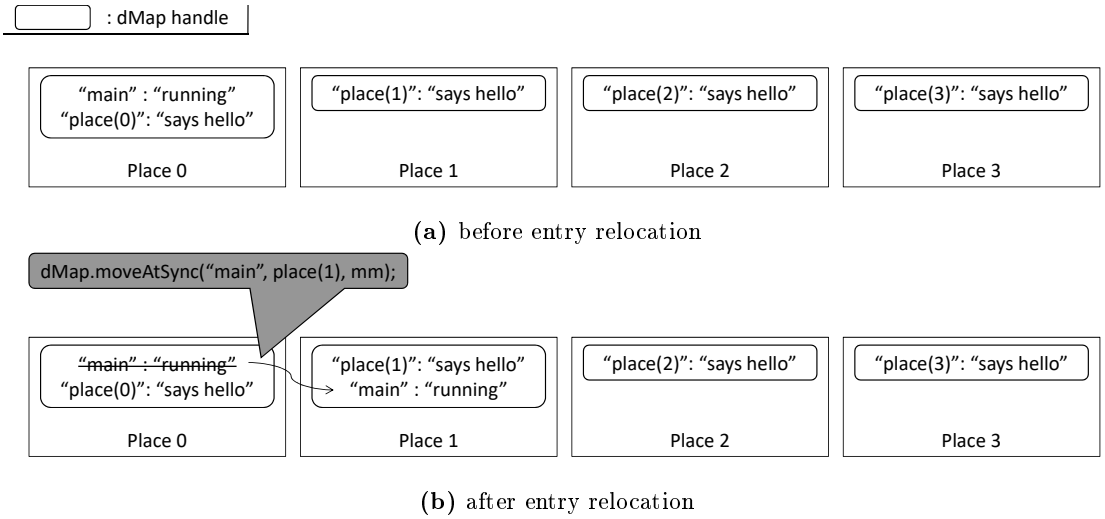
DistMap<K,V>

The distributed map `DistMap` is a generic distributed map taking `K` objects as keys and `V` objects as values. `DistMultiMap` is similar, but allows for multiple values to be mapped to a single key.

4.3.2 Local handle of a distributed collection

Before diving into specific features of our library let us first illustrate the notion of *local handle* and *teamed operation* with the sample program of Listing 4.2 and the accompanying Figure 4.1.

In Listing 4.2, a distributed map using `String` for both keys and values is created on line 2. This distributed collection is defined on the entire "world," i.e. it will have a local handle on every process participating in the computation. Then, a first entry is



(a) before entry relocation

(b) after entry relocation

Figure 4.1: State of the distributed map "dMap" in a 4 processes execution of the Listing 4.2 program

inserted on the running process on line 3. The call to method `put` only acts on the local handle registered on this place. As such, the "main":"running" entry is only registered on the *place 0* handle.

On line 5, a second call to method `put` registers new entries into the distributed map. In this case however, since the call is contained in a `broadcastFlat` method call, every place adds a new entry to their local handle. Contrary to ordinary objects, the distributed collections used inside a closure are not copied to the remote processes but instead allocated on the fly. As a result, the `dmap` handles on place 1 to 3 do not contain the entry previously placed in the handle of place 0. We provide more details about this topic in Section 4.5.1.

Note that the key used to place new entries on line 7 differs on each host due to the APGAS method call `here()` which returns the `Place` object representing the currently running process. Each local handle therefore contains a different key ("place(0)," "place(1)" etc.) mapped to the `String "says hello"`, as is reflected in the contents of each local handle of `dmap` in Figure 4.1a.

This illustrates the fact that conforming to the AGPAS programming model, all accesses to our distributed collections are "local" in the sense that APGAS asynchronous activities only ever interact with the handle of the distributed collection located on the process they are running.

4.3.3 Teamed operations

Teamed operation is a generic term we use to describe operations or computations which involve some form of coordination or communication between the processes participating in the computation. They are clearly identified as such in the documentation of our collections.

In principle, teamed operations operate on the set of processes on which the collection (or the other object otherwise supporting the operation) is defined on. The method needs to be called by an asynchronous activity on each of the processes for it to complete. As such, teamed operations form natural synchronization points in the distributed program. We demonstrate these characteristics in the following example.

Example

In Listing 4.2 from line 6 onwards, we present one of the teamed operation supported by our library in the form of an entry relocation between the handles of the `distMap` distributed collection.

A "collective move manager" is first created on line 6. This object is used to register entries of our distributed collections to be transferred from a handle to another. In this case, only the first place decides to relocate the `main:running` entry to place 1, with all other places keeping their current entries. The transfer is performed on line 10 when the `mm.sync()` method is called by all the places participating in the computation. The final state of the distributed map `dmap` is what is presented in Figure 4.1b. In particular, note that the `main:running` entry has been removed from the handle on place 0 and inserted into the handle of place 1.

There are a variety of "teamed operations" implemented in our library supporting various features, including reductions, entry relocation, replication etc. We will introduce the most significant of them in the next section.

In the example presented above, the group of processes participating in a teamed relocation is determined by the `TeamedPlaceGroup` object passed given to the constructor of the *collective move manager* on line 6. Here, the `world` place group is used, meaning that every place in the computation needs an asynchronous activity to call `mm.sync()` before they can respectively resume their progress, even if that place does not send or receive any entry as part of the collective relocation.

Teamed operations pair nicely with the `broadcastFlat` method provided by class `TeamedPlaceGroup`, whose purpose is precisely to launch an asynchronous activity on each place of an identified group. There is however no requirement to call "teamed operations" from within a matching `broadcastFlat`. This gives more experienced programmers the freedom to implement more complex synchronization patterns by combining the "normal" finish/asyncAt programming style of APGAS for Java with the teamed operations proposed by our library. For instance, if we wanted to allow place 2 and place 3 to continue their progress while place 0 and place 1 exchange entries, a different `TeamedPlaceGroup` containing only the first two places could be used when creating the collective relocator on line 6, with only place 0 and place 1 calling the `mm.sync()` method of that relocator on line 10.

4.3.4 Support for intra-node parallelism

As we will demonstrate in the next section, all our distributed collections feature typical `forEach`, `reduce` and other such methods that take a closure as argument. This closure is then applied to the entries contained in the local handle of the distributed collection. Parallel variations of these methods are also implemented, allowing programmers to benefit from a multithreaded runtime without having to manually schedule the required threads.

Internally, we rely on the APGAS finish/async pair of constructs to spawn and control the threads needed for the parallel variants of these methods. For `ChunkedList` and its variants, we allocate entries evenly between the threads available on the local host. This is made trivial by the nature of this collection whose entries are recorded by ranges.

Spawning explicit activities on the library side also helps when objects dedicated to a single thread are needed by the computation pattern. This is the case for instance of the parallel producer/receiver pattern, reductions, and "accumulators," presented in Section 4.4.1.2, Section 4.4.2.1, and Section 4.4.3.3 respectively.

In each of these computation patterns, our library handles to allocation of the necessary objects for the threads to work in isolation from one-another. This lightens the burden on programmers, re-focusing the program on the computation at hand rather than the schedule needed to support intra-node parallelism.

4.4 Motivating cases

In this section, we develop the abstractions available to programmers using examples taken from distributed programs written with our library. We rely on the distributed implementation of the *PlhamJ* financial market simulator, a distributed *K-Means* implementation, and the N-body simulation *MolDyn*.

Following a brief presentation of each program, we illustrate the abstractions and features they rely on in dedicated subsections. The features are presented in order of appearance in their respective applications, but the reader may choose to forego this order and browse by feature category: *intra-node parallelism*, *teamed relocation*, and *replication*.

4.4.1 PlhamJ

Plham is a financial market simulator first implemented in X10 [53, 54]. Simulations are prepared using a JSON configuration file which details the agents, markets, sessions, and events that will occur during the simulation. Users of this program can prepare trader implementations by extending the included `Agent` class. The simulator produces configurable outputs based on the information available over the course of the simulation, with the produced results deterministic following an initial seed. Internally several "runners"

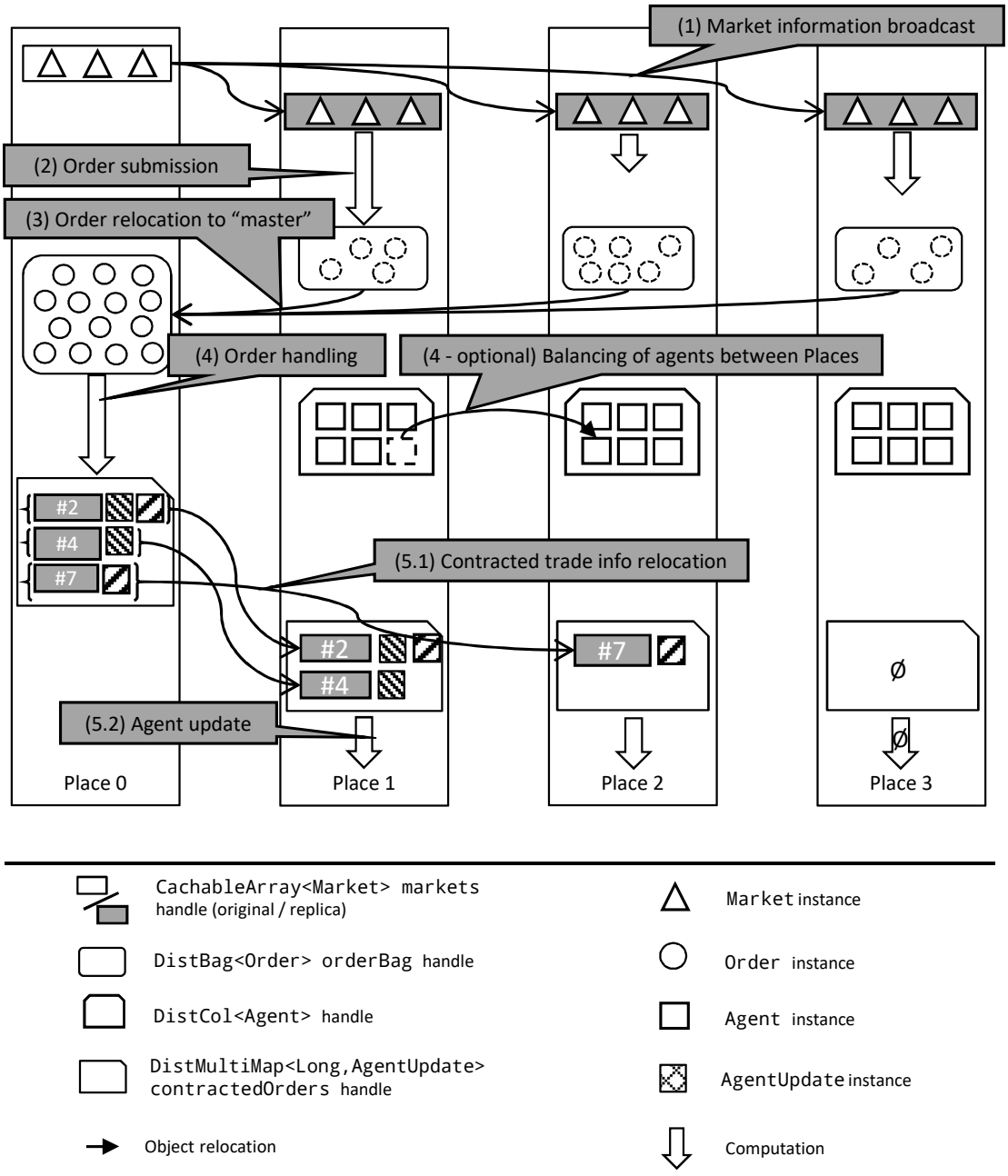


Figure 4.2: Figurative representation of the communications and computations processes that take place during a round of the Plham simulation

implementations are available (sequential, parallel ...).

A round of the Plham simulator comprises the following steps. First, agents place orders based on the current market information. Secondly, buy and sell orders placed by agents are matched to contract trades, updating the state of the market. Lastly, agents that have contracted a trade during this round are informed. These steps then repeat using the updated state of the markets for as many rounds as specified in the simulation configuration.

To make use of larger-scale computer clusters, a distributed version of the Plham simulator is available. In this implementation traders are distributed over multiple processes to leverage the greater parallelism of the underlying distributed runtime. However, this poses a number of challenges as the computation in charge of matching buy and sell orders needs to remain centralized on a single process (arbitrarily the first process, place 0) to provide the opportunity for high-frequency traders to place orders based on the most up-to-date market information.

As a consequence, we need to:

- propagate the updated state of the market to all the hosts participating in the simulation
- relocate the `Order` objects placed by agents to the centralized order-processing host
- dispatch the contracted trade notifications to the processes that hold the intended Agent recipient

To further complicate matters, if one of the processes takes longer than the others to compute the orders of the agents it was assigned, the progress of the entire program is delayed. In non-dedicated clusters, such a load unbalance can be caused by disparities in the hardware used to support the distributed computation (different CPUs, different frequencies or number of cores), or by other processes competing for resources. This poses a challenge as it is not reasonable to create a specific initial distribution for each cluster and/or simulation. Moreover, even "ideal" distributions would not be able to react to dynamic changes in the cluster' performance. While we could implement dynamic load-balancing of agent across hosts to resolve these situations as they occur, this poses a problem when sending contracted trade information to agents as their location will evolve dynamically over time.

PlhamJ is the Java implementation of Plham and was re-written using the features of our distributed collection library. This gave us the opportunity to revisit the implementation of some communication patterns as well as integrating a simple dynamic load balancer within the simulator. Under the distributed implementation of this simulator, a round takes place in 5 main computation and communication steps represented in Figure 4.2:

```
1 CachableArray<Market> markets;
2 world.broadcastFlat(() -> {
3   // (1) Broadcast the updated state of markets
4   markets.broadcast(MarketUpdate::pack, MarketUpdate::unpack);
5 });
```

Listing 4.3: Replication of Market objects in the PlhamJ simulator

1. the updated state of the markets is broadcast to its replicas on the agent-handling processes

2. the agent-handling processes collect the orders of the agents they hold

3. these orders are gathered on the order-handling process

4. the order-handling process tries to match sell orders with buy orders, creating an `AgentUpdate` object for each agent involved in a trade. Meanwhile, the agents are balanced between the other processes so that they all take roughly the same time during the order submission step. In our load-balanced version, this is done every few rounds.

5. the agent updates are dispatched to their respective Agent location (step 5.1) where the targeted agents are informed are then informed of the trades they made (step 5.2)

In the following subsections, we detail with the accompanying code the various features of our library that support this implementation.

In an effort not to overwhelm the reader, we chose to introduce the relevant code piece-by-piece in each subsection. Listing A.1 in the appendix consolidates all of them into a single Listing.

4.4.1.1 Replication: CachableArray

In the Plham simulator, the most up-to-date market information is located in `Market` objects located on the order-processing place. To replicate the updated state of the market information to the other processes in the computation, we rely on class `CachableArray` as shown in Listing 4.3.

The replicas on the other processes are updated using the teamed operation `broadcast` of line 4. This method is called by all hosts participating in the computation and also serves as a synchronizing mechanism between the asynchronous activities running the simulation on each host.

The two methods given as parameter to this function, `pack` and `unpack`, are respectively used to extract information from the market objects and record it into a

```
1  DistCol<Agent> agents;
2  DistBag<List<Order>> orderBag;
3  world.broadcastFlat(() -> {
4    // (2) Submit agent orders
5    if (!isMaster) {
6      agents.parallelToBag((agent, orderCollector) -> {
7        List<Order> orders = agent.submitOrders(markets);
8        if (orders != null && !orders.isEmpty()) {
9          orderCollector.accept(orders);
10       }
11     }, orderBag);
12   }
13
14   // (3) Collect all orders on the ''master''
15   orderBag.team().gather(place(0));
16 });
```

Listing 4.4: Parallel Order collection and relocation in the Plham simulator

`MarketUpdate` object, and to update the market replica based on the information contained in the `MarketUpdate` object. This allows the user to choose any object to carry the data necessary to update the objects.

4.4.1.2 Intra-node parallelism: producer / receiver

In the second step of a PlhamJ iteration, every agent is asked to submit its orders based on the current Market information. This consists in calling method `submitOrders` on every `Agent` object participating in the computation. This method returns a list of orders, with agents able to place a single, multiple, or no orders at all. In Figure 4.2, we represented a total of 14 orders submitted by the agents during step (2). The order are collected into the `DistBag` "orderBag".

The corresponding code is shown in Listing 4.4. The method `parallelToBag` called on line 6 relies on the internal features of class `DistBag` to allow multiple threads to concurrently place the orders into the local handle of this collection. This method takes two parameters. The first one is a closure taking an Agent and an "orderCollector" as parameter. This closure will be applied to every agent in the local `agents` handle in parallel, with the "orderCollector" taking the value `Bag` instance being used to collect the orders. The second parameter to method `parallelToBag` is the `Bag` into which all collected objects will be placed.

In this particular case, empty or `null` lists returned by agents that choose not to place any new order for this round are discarded using the condition on line 8. In cases where every entry in the collection produces an object to record in the specified bag, more simple signatures of method `parallelToBag` can be used.

4.4.1.3 Teamed relocation: gather

After each place has gathered the orders placed by its agents, all the orders are relocated to the order-processing place where they will matched to create trades.

This is performed when each host calls the `gather` method of class `DistBag`, as shown on line 15 of Listing 4.4. This method is a teamed operation which needs to be called on every handle of the distributed collection `orderBag` for the calling activities to progress. As such, it is used as a synchronization point between place 0 (which does not produce orders during the second step) and the other agent-processing places.

When the relocation has completed and all the orders produced during this rounds have been relocated to place 0, the order-matching computation of step (4) begins on place 0.

4.4.1.4 Teamed relocation: dispatch

During the order-handling process, each trade contracted results in two `AgentUpdate` objects to be created, one for each Agent involved in the trade.

In Figure 4.2 we show 2 trades to be contracted: one trade between agent #2 and #4, and a second trade between agent #2 and #7. These agent updates are placed into the `contractedOrders` distributed multi-map at the index matching their intended agent recipient.In other words, if agent #2 (contained in collection `agents`) contracts a trade, the "agent update" containing this information is placed at index #2 in the `contractedOrders` handle of place 0.

To inform the agents of the trades they contracted during this round, the entries of `contractedOrders` first need to be relocated to the location of their intended recipient. This is done as part of step (5.1) where the current distribution of collection `agents` is used to determine the new location of each entry in the multi-map.

The corresponding code is shown in Listing 4.5. First, the current distribution of `agents` is retrieved on line 6. This is possible thanks to the distribution tracking mechanism integrated in class `DistCol` which contains the agents participating in the simulation. Then, the entries of collection `contractedOrders` are relocated at the place where the corresponding agent is located by calling method `relocate` on line 7.

This method is a teamed operation which relocates the entries it contains to match the distribution given as parameter. In this particular example, the location of `Agent` is recorded in a mapping from *ranges of indices* to `Place` objects. This distribution is assimilated as a distribution from `long` indices to `Place` object by class `DistMultiMap` to determine the new location of each individual key recorded in `contractedOrders`.

For the illustration purposes of Figure 4.2, we assume that both agent #2 and #4 are located on place 1, while place 2 holds agent #7. The entries of the contracted trade information are therefore relocated according to this distribution; `contractedOrders` entries with key #2 and #4 are relocated to place 1, and the entry with key #7 is

```
1  DistCol<Agent> agents;
2  DistMultiMap<Long, AgentUpdate> contractedOrders;
3  world.broadcastFlat(() -> {
4    // (5) Inform the agents of the trades they made
5    // (5.1) Relocate contracted trade information
6    LongRangeDistribution agentDistribution = agents.getDistribution();
7    contractedOrders.relocate(agentDistribution);
8    // (5.2) Update the agents
9    if (!isMaster) {
10     contractedOrders.parallelForEach((idx, updates) -> {
11       // Retrieve the agent targeted by the update
12       Agent a = agents.get(idx);
13       // Apply each update to this agent
14       for (AgentUpdate u : updates) {
15         a.executeUpdate(u);
16       }
17     });
18   }
19 });
```

Listing 4.5: Dispatch of contracted order updates and agent update

relocated to place 2. place 3 holds no agents that were able to make a trade in this round.

In step (5.2), each agent which contracted trades during the previous round receives its updates in parallel using a typical `parallelForEach` shown from line 10 to 17 in Listing 4.5. The signature used here takes both the index (`idx`) and the list of updates (`updates`) contained in collection `contractedOrders` as parameter. This allows retrieval of the targeted agent instance on line 12 by calling `agents.get(idx)`.

4.4.1.5 Teamed relocation: load-balancing

In the situation presented in Section 4.4.1.2, the order submission of agents takes place in parallel on several processes. Initially, agents are distributed evenly across processes. However this may not be ideal as disparities in the hosts used or the presence of competing processes on said host may introduce imbalance in the cluster.

As a result, some hosts take longer than other to process the agents they hold, delaying progress of the entire simulation. We represented this in Figure 4.2 by different arrow lengths in step (2), with place 1 taking longer than all the other hosts to complete the order submission.

Fortunately, our relocatable distributed collection library allows us to take measures when such a case occurs. In the PlhamJ simulator, we introduced a load balancer mechanism shown in Listing 4.6. On each host, the amount of time dedicated to computing the orders is accumulated into the local `accumulatedOrderComputeTime` variable (not shown in previous listings, refer to line 16 and 23 of Listing A.1). After a chosen number of iterations have elapsed, the optional load-balancing step is triggered on line 9. The

```
1  DistCol<Agent> agents;
2  long accumulatedOrderComputeTime = 0;
3  int lbPeriod = 10; // load-balance period
4  int iter;          // current iteration number
5
6  world.broadcastFlat(() -> {
7    finish(()->{
8      // (4 - optional) balance the agents between places 1..n
9      if (iter % lbPeriod == 0) {
10       async(()->{
11         // Exchange time information between hosts
12         CollectiveMoveManager mm = new CollectiveMoveManager(world);
13         long [] computationTime =
14           world.allGather1(accumulatedOrderComputeTime);
15         performLoadBalance(computationTimes, mm);
16         mm.sync();
17         accumulatedOrderComputeTime = 0;
18         agents.updateDist();
19       });
20     }
21
22     if (isMaster) {
23       handleOrders();
24     }
25   });
26 });
```

Listing 4.6: Load Balance step in PlhamJ simulator

load-balancing is performed in a dedicated asynchronous activity spawned using APGAS' `async` method. This means the load balancing (lines 11 to 18 in Listing 4.6) is done concurrently to the order-handling on place 0 (method `handleOrders` on line 23). The program progresses to the following step only when both the order handling and the load-balancing have completed thanks to the `finish` opened on line 7 which contains both of these operations.

The transfer of agents is made using a *collective relocator*, as was previously introduced in Section 4.3.3. To determine the number of agents to transfer, the processes first exchange the amount of time they each spent on the order-submission part of the main loop using a `allGather1` call on lines 14. This information serves as the basis for each host to decide if it gives agents away inside the `performLoadBalance` method called on line 15. In this method, the agent instances to relocate are registered into the collective relocator previously created on line 12.

As a first approach, we chose to relocate agents from the most overloaded process to the most underloaded process. We call this simplistic load-balancing strategy "level-extremes". We will be able to revisit this part in later work to implement more sophisticated strategies.

The agents are then transferred between the handle of collection `agents` when the teamed method `sync` is called on line 16. In Figure 4.2, we represented this by one agent held by place 1 being relocated to place 2 to reflect the load-balancing decision based on previous iterations. In reality, entire ranges of agents will be relocated, depending on how severely unbalanced the situation is. The counter which tracks the time spent computing the agents' orders is then reset on line 17 so that the next load-balancing round takes information relevant to this new distribution.

We offer more details about the ways programmers can use to relocate entries of our distributed collections in Section 4.5.2.

4.4.1.6 Distribution tracking

In the absence of an integrated entry location record, managing a distribution record manually comes with tremendous effort. In essence, tracking the location of entries of a distributed collection requires the active maintenance of a second distributed collection, with each insertion, removal, and transfer of an entry in the first collection requiring an update into the second. This would greatly obfuscate the code and increase the chances of introducing bugs into the program.

In our library, we have implemented the facilities that allow for tracking of entry location and relocation in two of our distributed collections, the distributed arbitrary index array `DistCol`, and the distributed map `DistIdMap`. The premise of tracking the location of a distributed collection's entries implies that there exists some way to uniquely identify each entry. In both of these collections, individual entries can be identified by their unique `long` index.

Our distribution tracking system associates each index with the location (`Place`) of the associated record. However, in a concern for efficiency, we do not keep a location record for each individual index in the case of class `DistCol`. Instead, we rely on range descriptions of locations to reduce the number of key/value pairs necessary to record of the location of each entry in these distributed collection.

The information concerning entries relocated between handles, or entries added/removed from a handle is not eagerly propagated to the other handles of the distributed collection. Instead our distribution management proposes a teamed `updateDist` method through which the local distribution records of a collection are reconciled to reflect the actual distribution at the moment of the call. We took care in the implementation of this process to only communicate the distribution changes that occurred since the previous `updateDist` call in order to minimize the amount of information exchanged.

In the PlhamJ simulator, we use class `DistCol` to contain the agents participating in the computation. It is the distribution tracking facilities of this class that allow us to dynamically relocate agents over the course of the simulation without compromising the dispatch of contracted trades update as was laid out in Section 4.4.1.4. After agents have been relocated, method `updateDist` is called on line 18 of Listing 4.6 to refresh the distribution information contained in each handle. As a result, the distribution of agents obtained on line 6 of Listing 4.5 during the subsequent contracted trade dispatch will be up-to-date, guaranteeing that each agent involved in a trade receive their intended updates in step (5) of the PlhamJ round.

4.4.2 K-Means

K-Means is an iterative clustering algorithm which separates points into a pre-defined "k" number of clusters. There are three steps in a K-Means iteration. Starting with randomly selected initial centroids, each point is assigned to the cluster of its closest centroid. Then, the average position of each cluster is computed. Finally, the point closest to each average position is chosen as the new centroid for the next iteration.

We chose to adapt the K-Means algorithm from the Java Renaissance benchmark suite [51]. We rely on class `DistChunkedList` to contain the points subject to the algorithm. In this distributed version, each place participating in the computation takes care of the points it contains in its local handle. Listing 4.7 presents the main computation loop of our distributed K-Means implementation. The assignment of each point to a cluster is done in parallel using a `parallelForEach` method call on line 12. On the other hand, the average cluster position and the selection of the next centroid are implemented as *teamed reductions* on lines 17 and 22 respectively. We will discuss the implementation of a reducer and its embedded support for parallelism in Section 4.4.2.1 first. Then, we will discuss the difference between a "local" reduction and the "teamed" reduction used in K-Means in Section 4.4.2.2.

```
1  TeamedPlaceGroup world =
2  TeamedPlaceGroup.getWorld();
3  DistChunkedList<Point> points;  // init' omitted
4  double[][] initialClusterCenter;// randomly chosen
5
6  world.broadcastFlat(() -> {
7    double[][] clusterCentroids = initialClusterCenter;
8    for (int iter = 0; iter < repetitions; iter++) {
9      final double[][] centroids = clusterCentroids;
10     // Assign each point to a cluster
11     points.parallelForEach(p -> p.assignCluster(centroids));
12
13     // Compute the avg position of each cluster
14     AveragePosition avgClusterPosition =
15       points.team().parallelReduce(new AveragePosition(K, DIM));
16
17     // Compute the new centroid of each cluster
18     ClosestPoint newCentroids = points.team().parallelReduce(
19       new ClosestPoint(K, DIM, avgClusterPosition));
20
21     // Update the centroids for the next iteration
22     clusterCentroids = newCentroids.closestPointCoordinates;
23   }
24 });
```

Listing 4.7: Distributed K-Means implementation with our collection library

4.4.2.1 Intra-node parallelism: reduction

To compute a reduction on the objects of one of our collection, a "reducer" object needs to be prepared. This is the nature of classes `AveragePosition` and `ClosestPoint` which are used on lines 15 and 19 of Listing 4.7. These classes are in charge of computing the average cluster positions and the new centroids respectively. Both of these classes are user-defined and extend the generic abstract class `Reducer` provided by our library.

As part of a `Reducer` implementation, programmers need to provide 3 methods:

- the `R newReducer()` method which creates a new instance of the reducer
- the `void reduce(T)` method which reduces the given `T` object into this reducer instance
- the `void merge(R)` method which merges the contents of the reducer given as parameter into this instance

When creating a custom reduction object, the programmer need not care about concurrency. Our library ensures that no reducer object is used concurrently by multiple threads.

When computing a parallel reduction, each thread participating in the computation is given its own dedicated reducer instance obtained through the `newReducer` method of

the reducer object supplied as parameter. Each thread then calls method `reduce(T)` on the entries of the collection it was allocated with its dedicated reducer instance. When all threads have reduced their attributed entries, the reducer objects are merged back into a single instance using method `merge(R)` to obtain the final result.

4.4.2.2 Teamed reduction

A *local reduction* consists in a reduction computed on the entries contained in a single local handle. A *teamed reduction* on the other hand, is a reduction which is computed on all the entries contained in all the local handles of a distributed collection. They are accessible through a special `team()` method to distinguish them from the reduction which operates on the local handle only. In other words, method `parallelReduce(R)` operates on the contents of the local handle of a distributed collection, while method `team().parallelReduce(R)` used in the K-Means implementation shown in Listing 4.7 computes the reduction on the contents of the entire distributed collection.

A teamed reduction takes place in two stages. First, a "local" reduction is computed following the process detailed in the preceding section. Then, the local results of each handle are merged together into a single instance which is then returned as the result by each of the calling activities. Internally, an MPI `allReduce` call is made to communicate and compute the global result of the reduction across all running processes. The MPI communicator used to make this call is the one of the `TeamedPlaceGroup` on which the collection is defined. The registration of the user-supplied reducer object necessary to use MPI object reductions is made automatically by our library.

The underlying use of MPI routines remains hidden from the user. The only practical consequence is that the teamed reduction call is blocking until all handles of the distributed collection complete their local reduction and exchange their results, after which each thread resumes its progress.

4.4.3 MolDyn

MolDyn is a molecular simulation part of the Java Grande benchmark suite [52] implemented with the MPI/Java compatibility layer MPJ [10]. It consists in a N-body simulation with all the force interaction between all the particles computed. The particles are replicated on every host, with each host responsible for computing a subset of the force interactions. This information is then communicated between all hosts before updating the position and velocity of each particle.

An iteration of the distributed MolDyn program takes place in three stages. First, a subset of the force interactions between the particles is computed on each host. Then, the force subjected to each particle are summed across hosts using an MPI allreduce call. Finally, the position and velocity vectors of particles are updated.

We ported this benchmark using our distributed collections library to a hybrid im-

```
1  TeamedPlaceGroup world = TeamedPlaceGroup.getWorld();
2  LongRange particleRange = new LongRange(0, nbParticles);
3  CachableChunkedList<Particle> particles;
4
5  world.broadcastFlat(() -> {
6    if (world.rank() == 0) {
7      particles.share(particleRange);
8    } else {
9      particles.share();
10   }
11 });
```

Listing 4.8: Particule replication in MolDyn

plementation taking advantage of the multithreaded capabilities available within each process. The arrays of `double` used in the original implementation were converted to `Particle` objects managed by a `CachableChunkedList`.

Contrary to the previous examples we showed in this section, the computation pattern brought by MolDyn is no longer strictly "owner-based". Instead of a particle operating based on its own information, it is the interaction between each pair of particles that serves as the basis for the computation. To support such patterns, we introduced class `RangedListProduct`. This class is used to represent combination pairs between the entries of two `ChunkedList`s as depicted in Figure 4.3 and provides a number of iterators and `forEach` methods that act on the pairs it contains.

As we did for PlhamJ, we will introduce the features needed to support this program piece by piece in the following subsections. The consolidated MolDyn program can be found in the appendix in Listing A.2.

4.4.3.1 Replication: CachableChunkedList

We use the `CachableChunkedList` distributed collection to contain the particles of the simulation. Similar to the `CachableArray` previously discussed in Section 4.4.1.1, this collections allows for entries to be replicated on multiple hosts. However, unlike the `CachableArray`, `CachableChunkedList` allows for multiple handles to be the primary owners of certain ranges of entries where the former only allows a single source to update the replicas.

In the case of the MolDyn simulator, the particles are initialized on the first process in the distributed system. At the start of the computation, these entries are replicated on the other hosts by calling the `share` method on lines 7 and 9 in Listing 4.8. This teamed method takes one or multiple ranges as parameter and replicates the matching ranges of entries on the other hosts. In this particular case, only the first process shares the range of initialized particles on line 7, while the other processes (that do not contain any entries) merely receive the ranges shared by the other processes by calling the `share` method without arguments on line 9.

4.4.3.2 Product of ranged lists

Creating a product between two ranged lists is done by calling a factory methods provided by class `RangedListProduct`. In listing 4.9, this is done on line 10 where the `newProductTriangle` method is called. The ranged list containing the particles is given as argument to this method as it takes the role of both operands. We note that this method eliminates the mirrored pairs as depicted in Figure 4.3: only the pairs residing in the upper triangle are included in the `product` object.

In a second stage, the pairs of entries to process by each host are determined by calling the `teamedSplit` method on line 11. This method performs two operations. First, it splits the pairs contained in the product into tiles, creating as many columns and lines as was specified as parameter. If we assume that there are 100 particles in the simulation and that 5 columns and 5 rows are created, each tile will cover and area of 20x20 pairs, as depicted in Figure 4.3.

Then, a new instance of `RangedListProduct` containing a subset of the created tiles is returned. The `TeamedPlaceGroup` given as parameter is used to determine the number of hosts involved in the "split". The running process' position inside the group and the seed are used to select the assignments returned by this method call.

Although not communication takes place, we still consider this operation to be "teamed" as it needs to be called with the same parameters on all processes participating in the computation to operate correctly. This guarantees that every tile gets processed by at least one host as depicted in the lower part of Figure 4.3.

We note that the use of tiles in our implementation differs from the original MolDyn implementation where the rows of the upper triangle are allocated to each host in a cyclic manner.

4.4.3.3 Intra-node parallelism: Accumulator

The conversion to an hybrid implementation which uses local parallelism to compute the force interaction between the particles brings about an additional challenge compared to the single-worker-per-host implementation of the Java Grande benchmark. In the original implementation, the force sum can be written directly to the particles. However in a hybrid implementation this is no longer possible as there would a risk that two threads concurrently write the contribution of interactions involving the same particle. To address this issue, we introduced what we call *accumulators* to our library.

This mechanism (no relation to the `LongAccumulator` or the `DoubleAccumulator` classes from the standard `atomic` package) is used by threads participating in a parallel computation to store information independently from one-another. The `Accumulator` object serves as a factory for multiple "thread-local accumulators" which are objects dedicated to an individual thread during a parallel computation. In turn, each of these "thread-local accumulators" will contain individual objects of any user-chosen type into

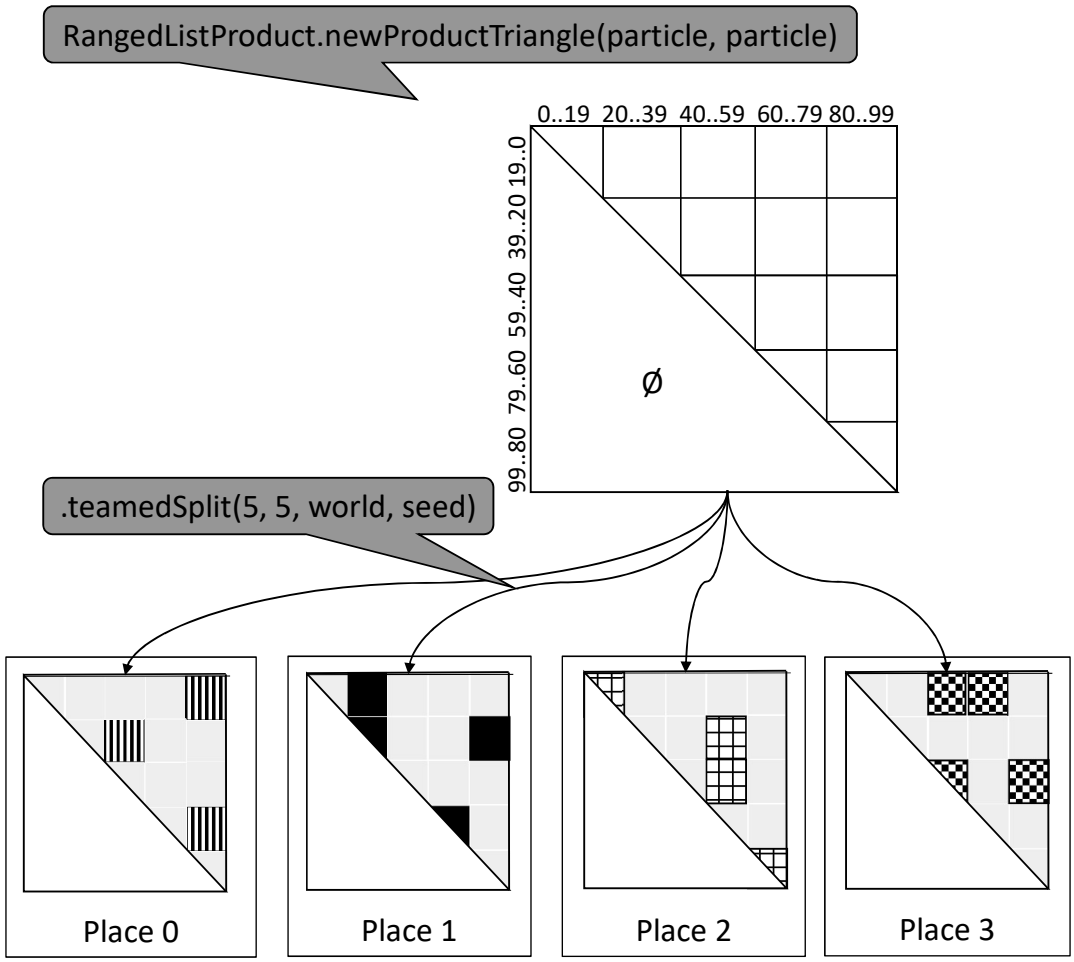


Figure 4.3: Illustration of the teamed split product used to allocate the particle interaction pairs between processes in our MolDyn implementation

```
1  TeamedPlaceGroup world = TeamedPlaceGroup.getWorld();
2  LongRange particleRange = new LongRange(0, nbParticles);
3  CachableChunkedList<Particle> particles;
4  int Ndivide = 5;
5  long seed = 0;
6
7  world.broadcastFlat(() -> {
8    RangedList<Particle> prl = particles.getChunk(particleRange);
9    RangedListProduct<Particle, Particle> prod =
10     RangedListProduct.newProductTriangle(prl,prl);
11   prod = prod.teamedSplit(Ndivide, Ndivide, world, seed);
12
13   Accumulator<Sp> acc =
14     new AccumulatorCompleteRange<>(particleRange, Sp::newSp);
15   prod.parallelForEachRow(acc,
16     (Particle p, RangedList<Particle> pairs, tla) -> {
17       force(p, pairs, tla);
18     });
19
20   particles.parallelAccept(acc, (Particle p, Sp a) -> p.addForce(a));
21 });
```

Listing 4.9: MolDyn force interaction computation using RangedListProduct and Accumulators

which information can be stored at a specified index. These individual objects are initialized using the function given as parameter at the time of the `Accumulator` creation.

An accumulator's lifecycle takes place in 3 phases: (1) creation, (2) accumulation of information into the accumulator, and (3) acceptance of the accumulated information by an existing collection. In the case of MolDyn, the accumulator used during the force computation is created on line 14 of Listing 4.9. The type used to store information in regards to each particle is class `Sp`, which contains 3 `double` members to represent the "x,y,z" force components.

The force computation takes place on lines 15 through 18. Let us briefly detail what method `parallelForEachRow` does. The closure it takes as parameter will be applied to each row of the tiles contained in the underlying `RangedListProduct`. The first parameter of the closure `Particle p` consist in the first half of the particle pairs to compute within this method, while the second half of the pairs are provided by the second `RangedList<Particle> pairs` argument. Inside method `force`, the force resulting of each interaction is stored into the `Sp` instance dedicated to the involved particles. The thread-dedicated `Sp` instances are available through the third parameter of the closure: `tla`. This parameter is populated by our library using the `acc` accumulator given as the first parameter to the `parallelForEachRow` method on line 15.

Finally, the information stored in the various `Sp` objects is used to apply changes to the particles using the `parallelAccept` method as demonstrated on line 20 of Listing 4.9. The closure given as parameter to the `parallelAccept` method sums the force

vectors contained in the various `Sp` instances into the dedicated member of the particles. Internally, this closure is applied to each `Sp` instances prepared for each thread that participated during in the "accumulation" phase.

Here, we demonstrated the use of the accumulator for a single computation before using it to modify a collection. It is also possible to perform multiple accumulations on various collections before "accepting" the accumulator.

4.4.3.4 Replication: reduction

After the force contribution computed on each host has been completed and integrated into the local replicas of each particule, the replicas all bear different force components due to the different subset of interactions that was computed on each host. To reconcile the force subjected to each particle, a reduction is made on each particle shared by the local handles of the `CachableChunkedList` used to support the program.

This is done on lines 6 to 14 in Listing 4.10 using method `allreduce`. This is a specific feature of class `CachableChunkedList` operating on the entries shared across hosts. Unlike the teamed reduction discussed in the context of K-Means in Section 4.4.2.2, in this situation each particle replica of matching indices are reduced into a single instance and stored back into the local handle of the `particles` collection.

We also demonstrate here the capability for our library to support primitive-type communication patterns. In this case, the force information is converted from each particle into three `double` numbers using the first closure running from line 6 to line 9. Then the MPI operation `MPI.SUM` is used to reduce these raw types. Finally, the reduced values are written back into the particle entries in each host using the second closure running from line 10 to 13.

Internally, buffer arrays of the appropriate length are automatically allocated based on the number of entries shared between hosts and the number of raw types used to describe each entry. This allows for more efficient use of MPI functionalities as serializing the entire particle object and implementing a custom reduction on this object is not necessary here.

After the force subjected to each particle has been consolidated across all hosts, each particle "moves" (i.e. updates its position and velocity vector) on line 16 of Listing 4.10, concluding an iteration of the program.

4.5 Design & implementation

In this section, we detail select design elements and implementation topics of our distributed collection library that were not detailed in the preceding section. We also briefly demonstrate how to compile and execute programs with our library.

```
1  TeamedPlaceGroup world =
2  TeamedPlaceGroup.getWorld();
3  CachableChunkedList<Particle> particles;
4
5  world.broadcastFlat(() -> {
6    particles.allreduce((out, Particle p) -> {
7      out.writeDouble(p.xforce);
8      out.writeDouble(p.yforce);
9      out.writeDouble(p.zforce);
10   }, (in, Particle p) -> {
11     p.xforce = in.readDouble();
12     p.yforce = in.readDouble();
13     p.zforce = in.readDouble();
14   }, MPI.SUM);
15
16   particles.parallelForEach(p -> move());
17 });
```

Listing 4.10: Force reduction on each particule in the MolDyn simulation

4.5.1 Lazy allocation of local handles

For every distributed collection whose classes we presented in Table 4.1, there is in reality one instance of the corresponding class on each process on which the collection is defined. These instances implement what we refer to as the "local" handles of distributed collections.

When a distributed collection is created, a local handle bearing a globally unique identifier is created on the process on which the constructor was called. Handles on the other processes are not created immediately. Instead, we implemented a "lazy" allocation mechanism to create the handles of distributed collections on the other processes. Under this mechanism, the local handle of a collection is allocated on remote hosts the first time a distributed collection is used in an asynchronous activity executed on a remote host.

This is implemented by customizing the serialization of our distributed collections such that the table of global ids is checked upon deserialization. If there are no bindings for the global id of the distributed collection being deserialized, the constructor is called to create the local handle and bind it to this global id on this place. If there was already an object bound to this global id (meaning this is not the first time a closure with this distributed collection is called on this host), then the deserialization resolves to the existing handle.

In the example presented in Listing 4.2, the local handle for the `dmap` collection on place 0 is allocated during the construction on line 2. The handles on the other hosts are created as part of the deserialization of the lambda-expression running from line 4 to 10, prior to its execution on these hosts.

Using this mechanism has the advantage of removing synchronizations over the entire

cluster each time a collection is created. Instead, the local handles of every distributed collection are created little by little as they become necessary. There is no risk of executing an asynchronous activity on a collection whose local handle is not initialized, as the mere fact that a collection is used in the activity guarantees that the local handle will be created (if it doesn't already exist) as part of this activity deserialization process.

4.5.2 Registering entries for relocation

One of the key features of our distributed collections library lies in its ability to relocate entries of a distributed collection between its handles. Our library builds on and expands a scheme first developed in X10 [56].

As briefly introduced as in example in Section 4.3.3, the `CollectiveMoveManager` can be used to transfer entries belonging to one or multiple collections between all or a subset of the processes participating in the computation, this group being specified at construction using a `TeamedPlaceGroup` instance. The transfer is initiated when the `sync` method is called on all the places of the group it operates on. This call is blocking until it is called on all places involved in the relocation. As such, the collective relocator mechanism is a synchronization point between asynchronous activities participating in the computation.

The novelty with our library compared to the original scheme lies in the variety of ways programmers can register entries for relocation. These methods are defined through modular interfaces implemented by our various collections, improving consistency and reducing future development effort. They allow programmers to specify what entries need to be relocated by specifying relevant arguments and the "move manager" used to perform the transfer.

Let us introduce the program of Listing 4.11 to illustrate the various ways entries of our distrubted collections can be marked for relocation. This program demonstrates a single collective relocation used to relocate objects belonging to multiple collections. For the sake of simplicity, we chose to make each process send entries to its neighboring $(rank + 1)\%n$ process, but this is in no case a limitation of the relocation system as entries originating from a process can be relocated to multiple other processes.

4.5.2.1 Relocation in bulk

Relocatin in bulk is available to all of our distributed collections. They feature a method called `moveAtSyncCount` which is used to relocate the specified number of entries. The library decides which entries are relocated without the input of the programmer. In Listing 4.11, this method is used to transfer 20 entries contained in each `bag` handle to their neighbor on line 14. This is the only available relocation method for the distributed set `DistBag<T>` as individual entries in this collection are devoid from any "identity."

```
1  final DistBag<Integer> bag;
2  final DistChunkedList<Element> cl;
3  final DistMap<String, String> map;
4
5  TeamedPlaceGroup world =
6  TeamedPlaceGroup.getWorld();
7  final int n = world.size();
8  world.broadcastFlat(() -> {
9    // Prepare the collective relocator
10   CollectiveMoveManager mm = new CollectiveMoveManager(world);
11   Place destination = place((here().id + 1)%n);
12
13   // Relocation in bulk
14   bag.moveAtSyncCount(20, destination, mm);
15
16   // Relocation by range
17   for (LongRange range : cl.ranges()) {
18     cl.moveRangeAtSync(range, destination, mm);
19   }
20
21   // Relocation by key->destination function
22   Function<String, Place> relocationRule =
23     (String key) -> destination;
24   map.moveAtSync(relocationRule, mm);
25   mm.sync(); // Perform the transfer
26 });
```

Listing 4.11: Rotation of entries between processes using a collective relocator

4.5.2.2 Relocation by range or by key

Relocation by range of by key is possible for distributed collection in which entries are identified by a unique identifier. We distinguish between collections where entries can be identified by a key, such as `DistMap` and `DistMultiMap`, and collections where entries can be designated through an entire range, such as `DistChunkedList` and its derivatives. In our library, this is enforced using two generic interfaces `RangeRelocatable<R>` and `KeyRelocatable<K>` which define a number of signatures for methods `moveRangeAtSync` and `moveAtSync` respectively.

We demonstrate the relocation using a range on line 18 of Listing 4.11. Using the loop of lines 17-19, all the ranges contained in collection `cl` are marked for relocation to the neighboring host. It is not an obligation to specify a range which corresponds exactly to a "chunk" contained by the local handle. Programmers can specify a range which either spans several of the "chunks" contained in the local handle or is a sub-range of a single chunk. In this case, the existing chunks will be split as necessary before relocation.

On line 24, the entries of the distributed map `map` are all marked for relocation using the `relocationRule` function defined just above. Internally, the `relocationRule` function is applied to each key contained in the local handle to determine their respective destination. In this example, the "key" parameter is not used in `relocationRule` which always return the same `Place` object as the destination, but more sophisticated implementations are entirely possible.

4.5.3 Communication patterns for entry relocation

When registering some entries for relocation into a *move manager*, our library actually registers a pair of serializer and deserializer into the move manager instance provided as argument. When the `sync` method of the collective relocator is called, the serializer is called to convert the targeted objects into bytes. The deserializers are also written to the byte array.

In a collective relocation, each place therefore obtains an array of bytes (possibly empty) to send to every other place participating in the computation. The transfer of objects is then performed in two steps. First, the number of bytes to be sent by each process participating in the transfer is exchanged with an MPI `Alltoall` call using the underlying communicator of the `TeamedPlaceGroup` specified with the constructor of the `CollectiveMoveManager`. This allows each process to know how many total bytes to expect and prepare buffer arrays of the appropriate size. Then, the byte arrays are exchanged between the processes using an MPI `Alltoallv` call. Each host then proceeds to deserialize the bytes it received and place the entries into their respective collection handle. Due to the blocking MPI calls used to perform the relocation, the `sync` method of the `CollectiveMoveManager` is a synchronizing call between asynchronous activities

running on different processes.

The same general process is used to implement other features of the library. In the case of the market replication in PlhamJ shown in Listing 4.3, the closures provided as argument to the `broadcast` method are used to produce the objects being transferred (in this case instances of class `MarketUpdate`) and to update the `Market` replicas located on the remote host. Our library takes care of serializing and deserializing the objects used as intermediary vessels. Then, MPI `Bcast` calls are used instead of `Alltoall` as the order-handling process is the sole source of information. Similarly, the order relocation performed in Listing 4.4 relocates all the entries of collection `orderBag` to the first process of the distributed program. After the serialization of the entries to transfer, `Gather` and `Gatherv` calls are used as there is only one "recipient" in this communication pattern.

4.6 Evaluation

The goal of our evaluation is threefold. First, we want to establish the greater programmability of our library when writing distributed programs. Secondly, we want to establish the performance of programs written with our library against equivalent ones. Lastly, we want to verify that load-balancing techniques made possible by our library are capable of adapting distributions to match uneven or evolving cluster performance.

We use the three applications presented in Section 4.4: the K-Means benchmark adapted from the Java Renaissance benchmark suite [51], the N-Body molecular simulation *MolDyn* adapted from the Java Grande benchmark suite [52], and our financial market simulator *PlhamJ*.

We first discuss matters related to programmability in Section 4.6.1. We then compare the performance of the original K-Means and MolDyn implementation against the versions we implemented with our library in Section 4.6.2. Finally, we establish the capabilities of our high-level load-balancing features using PlhamJ in Section 4.6.3.

4.6.1 Programmability

Programmability is a difficult criteria to judge. Comparing programs using quantitative criteria such as *lines of code* (loc) can be done, but such criteria alone cannot be used to determine whether some model or library is beneficial. An abstraction supporting a particular pattern may reduce the amount of code necessary, but if it is too specific or convoluted to be used in other applications the claim of better programmability is weak. On the other hand, qualitative criteria may be controversial or present a certain level of subjectivity.

We believe our library brings significant gains in programmability thanks to three key characteristics: (1) its support for local parallelism, (2) the notion of "teamed operation," and (3) the high-level support for distribution management.

Concerning the support for local parallelism, as we demonstrated in Section 4.4, our distributed collections provide multiple parallel methods taking closures as arguments. This approach re-centers programs on the actual computation at hand rather than how the parallelism is supported. On this matter, the comparison between the K-Means implementation with our library and the original Java Renaissance suite [51] is particularly interesting.

In the Renaissance K-Means implementation, the points are managed by range explicitly in order to implement the "recursive task" implementation required by the Java `ForkJoinPool`. By comparison, the range management remains totally internal to the `ChunkedList` class we use in our implementation. The range management remains also absent from the classes used to support the reductions needed by the algorithm. As a result, the total size of the distributed K-Means written with our library (excluding argument parsing and initialization) amounts to just over 200 lines of code compared to over 400 lines of code for the Renaissance implementation. Moreover, the legibility of the program is entirely preserved despite its distributed nature, as made evident by Listing 4.7.

Moreover, the management by the library of thread-dedicated data structures greatly simplifies programs for what would otherwise become a cumbersome implementation. Our library allocates just the necessary data structures to support the number of available threads on the system. This remains entirely transparent to the programmer who may use the various parallel methods knowing that the appropriate number of threads will be spawned even if the number of threads available varies from a host to another.

The second gain brought by our library comes by the introduction of teamed operations on our distributed collections. These methods define the scope of their intervention by using either the group of processes on which the supporting collection is defined, or by specifying the group explicitly in a constructor (as is the case for the collective relocator). This contributes to the clear identification of both the hosts that are involved in said teamed operation and the synchronizing point between the asynchronous activities running on different host.

This is particularly evident in the case of PlhamJ, where the first host performs different tasks than the others. In this application, the teamed methods both serve as the necessary communication support to implement the program but also as the synchronization point used to determine completion of a remote procedure. For instance, the computation of the agents' orders cannot start until the market information broadcast is completed. Similarly, the order-handling on the first process cannot start until the orders submitted by each agent for this round are received through the teamed gather operation.

Finally, programmers have complete and dynamic control over the entry distribution of the distributed collections. The high-level relocation abstractions we provide makes this management easy, with supporting features such as the distribution tracking

countering the challenging nature of a dynamic distribution where necessary. The most prominent example of this lies in PlhamJ where agents are relocated from hosts to hosts to balance the computational load while the distribution tracking ensures that information meant for a specific agent reaches its destination. Internally, the management of entries by range makes this both elegant and efficient.

4.6.2 Performance comparison against original benchmark implementation

To verify that our distributed collections library provides reasonable performance, we compare the performance of two programs written with our library against the reference benchmark implementations of K-Means and MolDyn. We conduct our performance evaluation on the OakForest-PACS supercomputer using up to 64 compute nodes. The characteristics of the OakForest-PACS supercomputer are summarized in Table B.2.

4.6.2.1 K-Means

The original Renaissance benchmark operates on a single process. We compare it against two implementations of K-Means prepared with our library: a "single-host" version, and the distributed "teamed" version discussed in Section 4.4.

We perform our evaluation in weak scaling from 1 to 64 hosts, increasing the number of points proportionally to the number of hosts involved in the computation. We run the K-Means algorithm for 30 iterations and compare the iteration time between the implementations. The details of the program parameters we used are shown in Table 4.2.

The results are presented in Figure 4.4 where we plot the minumum, first quartile, third quartile, maximum, and average iteration time obtained with each program version.

With the "small" parameter configuration, the average iteration time is kept just below 500ms with the Renaissance benchmark. Our implementation on a single host is 20% faster. This higher performance is maintained on 4 hosts (12% faster than Renaissance) despite the communication needed by the reductions. However, our implementation is not capable to scale further with such short iteration times. Over the course of the "teamed" program executions we witnessed a few particularly long iterations, the longest of which occurred on a 16 host execution and lasted just under 4 seconds. This drives the average iteration time upwards despite the overwhelming majority of iterations completing within 500ms.

We are not certain as to what causes this phenomenon. We believe it could be explained by some of the processes in the cluster performing garbage collection with unfortunate timing and delaying the communication with the other processes during the teamed reductions of the program. This would in turn delay the progress of the entire program as the other processes are stuck waiting on the result of the reduction.

In the second "large" K-Means, the number of clusters is dramatically increased

Table 4.2: K-Means benchmark parameters

Configuration	"small"	"large"
Nb of points/host	10 million	
Point dimension	3	5
Number of clusters	50	2000
Iterations	30	

compared to the "small" parameter. As a result the computational load consisting of assigning each point to a cluster becomes predominant over the two reductions and the iteration times increase for the Renaissance version to an average of 28.3 seconds. Our single host implementation however, maintains an average iteration time far below, just under 11 seconds.

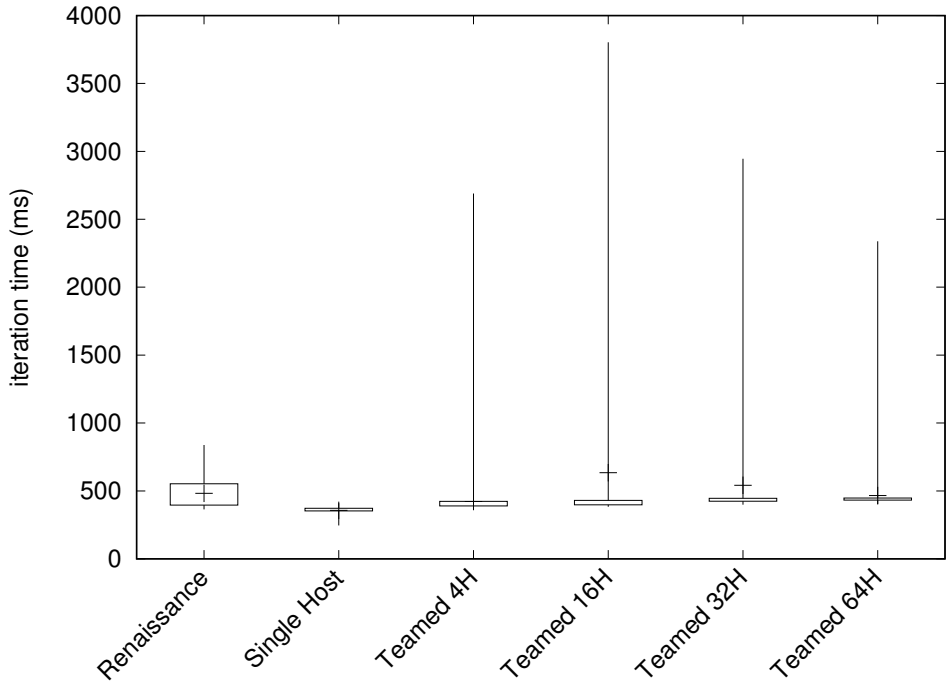
This performance gap between the Renaissance implementation and our Single Host implementation can be explained by the higher memory consumption (and more frequent garbage collection) of the Fork/Join implementation. While the Renaissance version is capable of delivering short iteration times, as is made clear by the minimum iteration time of 13.2s, it is not capable of sustaining them over the entire course of the execution. Our distributed implementation also shows better performance than the original implementation, with the average iteration times kept below 15s up to 64 hosts. Under this higher computational load, the performance trouble witnessed under the "small" parameter configuration is absorbed, with the iteration times of obtained in the 64 hosts configuration only 30% longer than our "single host" version, but still half that of the Renaissance implementation.

4.6.2.2 MolDyn

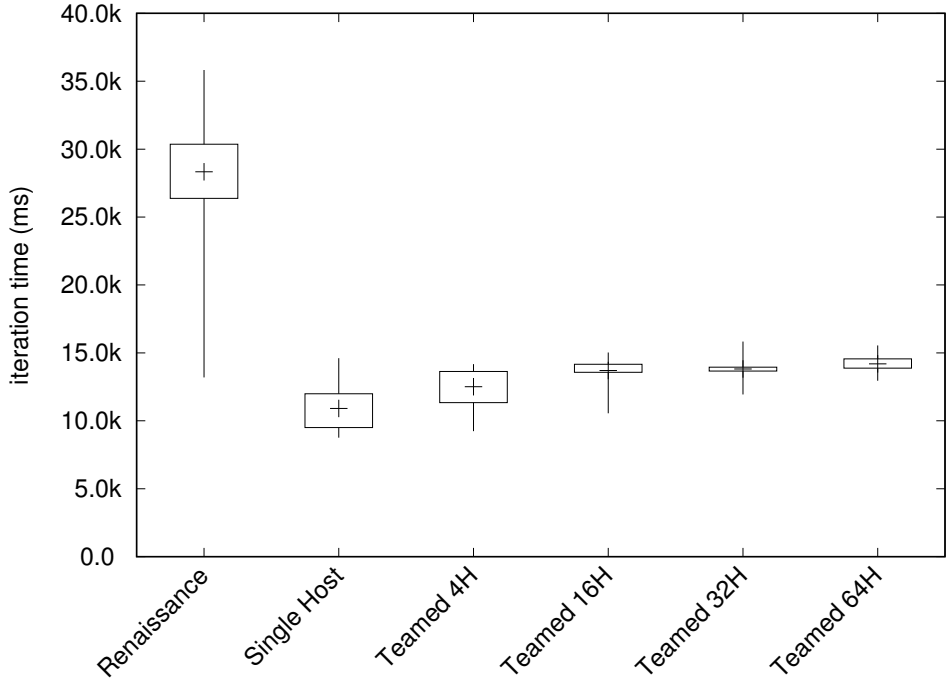
The original Java Grande version of MolDyn built on MPI uses 1 thread per host. We compare two versions implemented with our library against the original version: a single-threaded version (Handist ST) similar to the original implementation, and a multithreaded version (Handist MT) which uses multiple threads on each process.

We run the MolDyn benchmark in strong scaling (same problem size for increasing cluster size) on the OakForest-PACS supercomputer from 1 to 64 hosts with 32,000 particles. We use 68 threads per process for our MT version, resulting in its parallelism level with a single process to be slightly higher than the Java Grande and the ST version running on 64 hosts. We measure the total computation time of the simulation after a short warmup. The computation times and the efficiency of each program version are are presented in Figure 4.5. An ideal efficiency of 100%, i.e. perfect scaling, would mean that increasing the computational resources by a factor n yields execution times n times shorter.

First, comparing the Java Grande version against our single-thread (ST) implementation, we note a 20% increase in computation time.We believe this is a reasonable amount



(a) small configuration



(b) large configuration

Figure 4.4: K-Means iteration times

The brackets and boxes represent the minimum, 1st quartile, 3rd quartile, and maximum values while the cross corresponds to the average value of 5 sample runs of 30 iterations each.

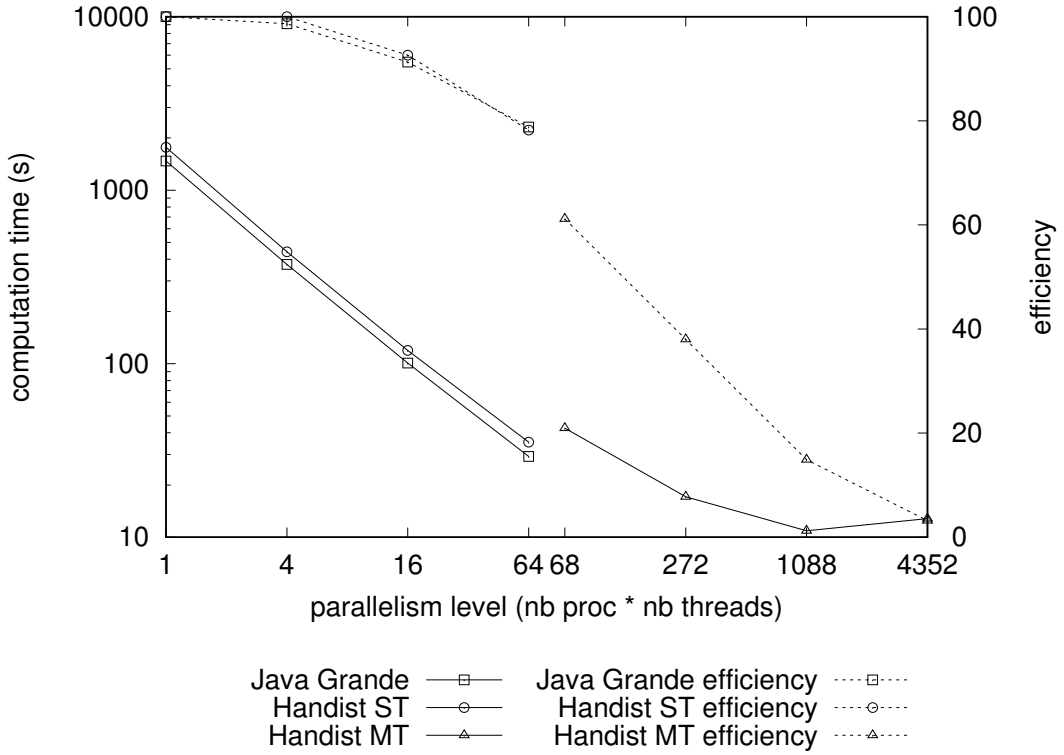


Figure 4.5: Computation time and efficiency of the MolDyn benchmark on the OakForest-PACS supercomputer

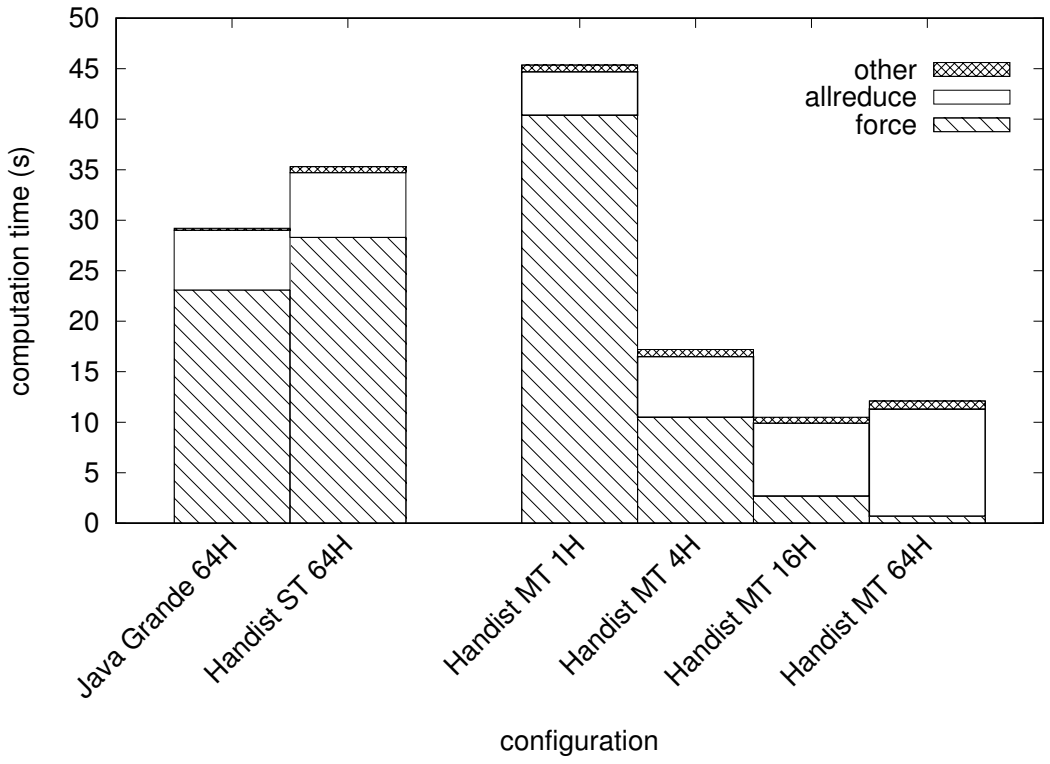


Figure 4.6: Computation time breakdown of the higher-parallelism executions of the MolDyn benchmark

of overhead considering the fact we moved away from the primitive type arrays to use objects to store the particles in our ST and MT implementations. The efficiency for both the ST and Java Grande versions follow the same pattern, decreasing down to 78% on 64 hosts.

This can be explained by the nature of the computation at hand. In all three versions of MolDyn studied here, the time taken by the "allreduce" sum of the forces across hosts takes a total of about 5 seconds of the total computation time, irrespective of the number of hosts or threads used. On the Java Grande and Handist ST executions from 1 to 16 hosts, the computation time was dominated by the force computation. As can be seen in Figure 4.6, this is no longer the case on 64 hosts where the "allreduce" part represents about 15% of the computation time. As the parallelism increases and the force interaction computation time decreases, this incompressible part of the program takes up a relatively larger part of the total computation time, decreasing efficiency.

Secondly, our MT implementation shows a slightly different efficiency pattern compared to the other implementations. Its efficiency for the 1 host/68 threads configuration loses an additional 17 percentage points of efficiency compared to the similar level of parallelism of the ST version running on 64 hosts. This is mostly imputable to the overhead brought about by the use of the accumulator mechanism in the MT version. Also, the fact that we used an entire host for each single-threaded Java Grande and Handist ST gives those versions a certain advantage. In future work, we hope to be able to reduce the overhead brought by the use of the accumulator mechanism by introducing alternative implementations that would only allocate ranges on a per-need basis rather than allocating the complete range from the start.

We are able to further reduce the execution time down to just over 10 seconds with the MT version running on 16 hosts (1088 total threads), albeit with decreasing efficiency. The execution on 64 hosts shows it is counterproductive to stretch the program any further, with the total computation time increasing from 10 to 12 seconds. As can be seen in Figure 4.6, the computation time is dominated by the "allreduce" part of the computation on executions with larger parallelism.

4.6.3 Dynamic load balancing in PlhamJ

The objectives of the evaluation conducted with our PlhamJ distributed financial simulator is twofold. First, we want to demonstrate the capability of a distributed program to adapt itself to the uneven performance of the cluster on which it is runs thanks to the features of our library. Second, we want to verify that the load-balancing measures we implemented in PlhamJ are able to react to dynamic changes in the cluster performance.

We perform the evaluation on our Beowulf cluster composed of two types of hosts: "piccolo" hosts which are fitted with a 4-core CPU, and the higher-parallelism "harp" host which contains two 12-core CPUs. The detailed hardware characteristics are outline in Table B.1. We use up to 5 hosts in three different cluster configurations summarized in

Table 4.3: PlhamJ execution configuration summary

Configuration	Description
Config A	4 processes on 4 piccolos (no load unbalance expected)
Config B	5 processes on 4 piccolos (piccolo 0 hosts the order-handling process and one agent-handling process)
Config C	6 processes on 4 piccolos and 1 harp (piccolo 0 hosts the order-handling process and one agent-handling process)

Table 4.3.

In *Config A*, we use a typical approach consisting of allocating one process on each "piccolo". The order-processing process is allocated on one host, while the three other hosts are dedicated to agents' order submission.

In *Config B*, we allocate an additional agent-processing process on the host holding the order-processing host (5 processes on 4 piccolos). This choice of allocation can be justified by the fact that the process which handles the orders remains idle while the agents are making their submission. There is therefore some amount of computational resources left untapped on the server hosting the handling of orders which we can tap into with this second configuration.

Finally, in *Config C*, we add the "harp" host as an order-handling process compared to Config B. The challenge of Config C lies in the nature of this additional server which brings more parallelism than the identical "piccolo" hosts used so far. It is therefore difficult to predict a priori what a good distribution of agents should be with such a cluster configuration.

To simulate dynamic changes in performance, we introduced a parasite program called "Disturb". This program runs concurrently to our simulator and computes an artificial 20 seconds load on one of the hosts. When the 20 seconds have elapsed, another host is chosen as the victim. The sequence of hosts "disturbed" by this program is deterministic following an initial seed to allow us to reproduce its effects over multiple executions.

We compare the performance of our "level extremes" load-balancing strategy previously discussed in Section 4.4.1.5 against the fixed uniform distribution without load balance "no lb". The results are presented in Figure 4.2.

We are able to draw two conclusions from the PlhamJ executions without the `Disturb` program. First, our basic load-balancing incurs no overhead in our distributed PlhamJ simulator as demonstrated under the "Config A" results. Execution times for the static and the load-balanced version are almost identical at 75.3 and 76.0 seconds in this configuration where no load-balancing is required. This can be explained by the fact that the (hypothetical) transfer of Agents between hosts takes place concurrently to the order-handling on the first process. In our experience, the transfer of Agents completes before the order-handling and thus does not negatively impact performance.

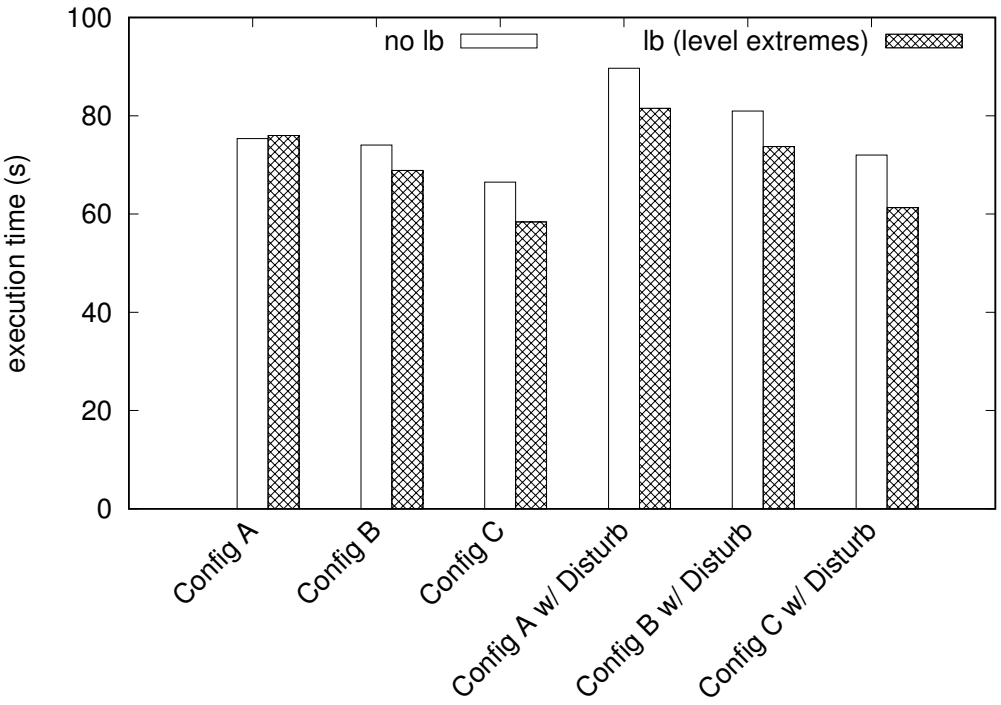
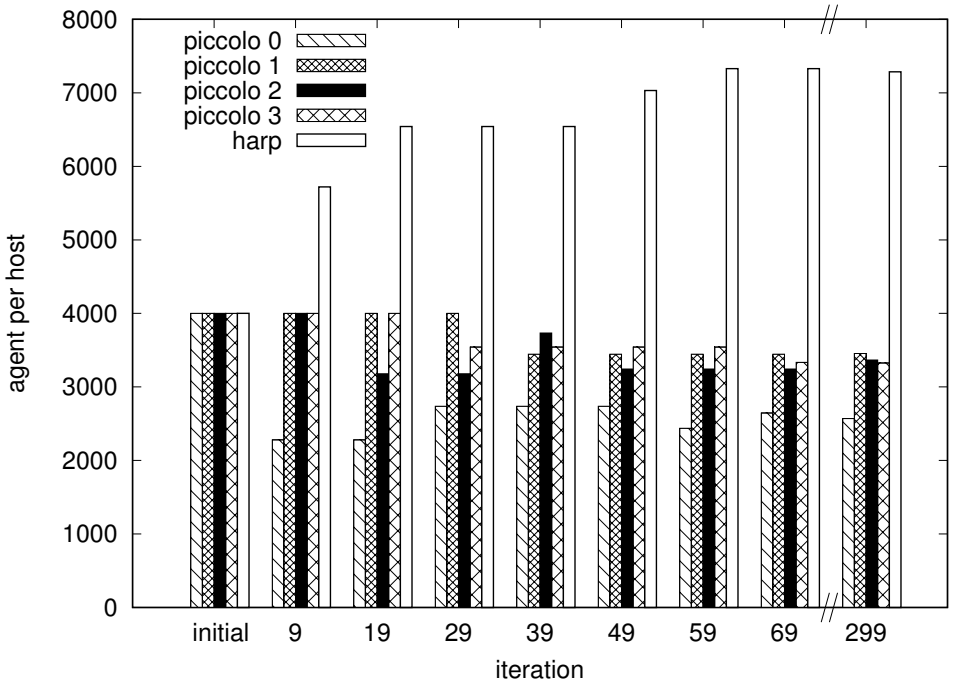


Figure 4.7: Execution time of the PlhamJ simulation depending on the cluster configuration

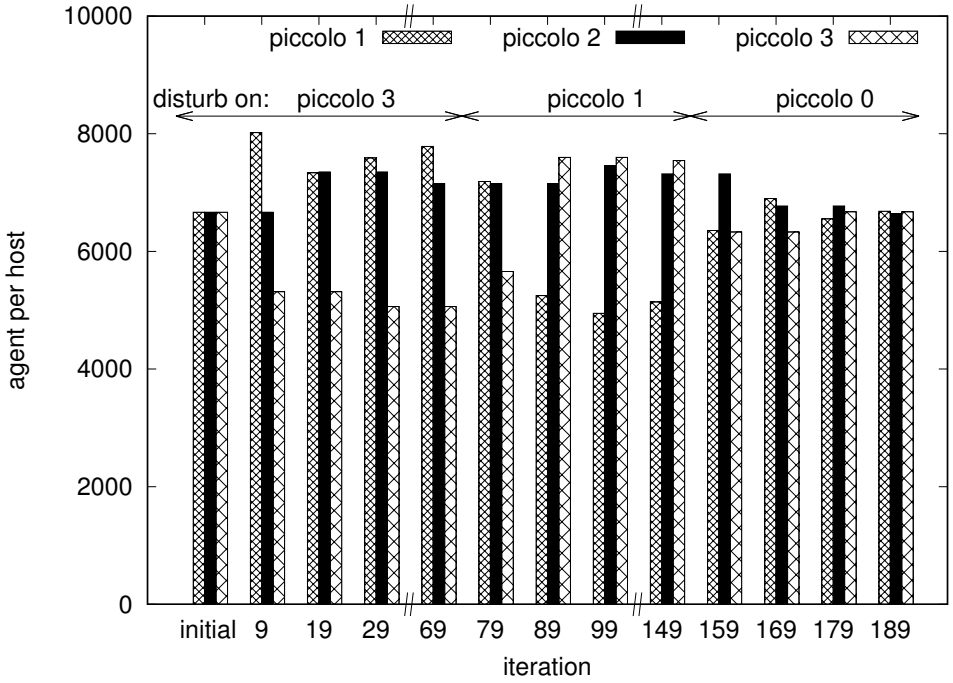
Secondly, this basic load-balancing technique is capable of handling an uneven cluster configuration, as can be seen in the execution time of PlhamJ under Config B and Config C. Depending on the configuration, our load balancing strategy delivers execution times between 7 and 15% shorter than the fixed uniform agent distribution.

The distribution of agents over time during an execution under Config C is presented in Figure 4.8a. The distribution becomes stable after only 30 iterations (4 seconds into the simualtion). Seeing as piccolo 0 hosts both the order-handling process and an agent-handling process, it ends up containing fewer agents than its piccolo 1-3 counterparts. Also, the higher parallelism available to the process allocated on our "harp" server is made evident by the fact it obtains over a third of the total agents in the simulation.

The experiments conducted in the presence of the parasite program presented show that our basic load balancing strategy is capable of handling dynamic changes in the cluster performance, with execution times between 8 and 15% shorter depending on the configuration. In Figure 4.8b, we show the evolution of the agent distribution under Config A w/ Disturb. Under this configuration, the only source of disparities between the hosts performance is the presence of the parasite program on one of the hosts. At the beginning of the simulation, the server hosting process piccolo 3 is being disturbed, resulting in some of its agents to be offloaded to the other processes. Then, starting between the 70th and 80th iteration of the simulation, the disturb process moves to piccolo 1. As a result, agent are moved away from piccolo 1 and the previously disturbed



(a) under Config C w/o disturb configuration



(b) under Config A w/ disturb configuration

Figure 4.8: PlhamJ agent distribution over time

piccolo 3 is assigned more agents. In the last part shown on this graph, the disturb program moves to piccolo 0 which hosts the process dedicated to processing the orders. As a result, there is no longer a discrepancy in the available processing power between the piccolo 1, 2, and 3. Our load balancer therefore gradually returns the agents to an even distribution between hosts starting from the 160th iteration onwards.

4.7 Conclusion and future work

In this chapter, we introduced our relocatable distributed collections library for the AGPAS for Java programming model. Our library allows users to write complex parallel and distributed programs by providing clear abstractions to handle both parallelism and distribution. We established the programmability gains and the performance of our system using two well-known Java benchmarks. Using the PlhamJ financial market simulator, we demonstrated the capability of programmers to balance the computational load between hosts using the integrated relocation mechanisms of our library.

A key difference between the PGAS languages discussed in Section 2.1 and our distributed collections lies on how access to data located on a remote host is handled. With our library, only asynchronous activities executing on the same process can access the data, whereas in UPC, Coarray Fortran, and Chapel, remote data can be accessed somewhat implicitly through global pointers or the indices of the distributed array. XMP also introduces the notion of "shadowing" where, in a nested for loop, if the computation needs to access neighboring data the compiler directives will generating code to access the data points which are on remote hosts. We can work around this limitation with our library using "owner/replica" schemes, but not in a manner quite as elegant as the other PGAS langages.

We did not cover topics related to resilience. This is a matter of importance because large-scale systems are more prone to failure. We do plan to implement features that will allow programmers to easily backup the (distributed) state of their collections into checkpoints, making it possible to recover after a failure. For simulations, this would also make it possible to restart a simulation from an intermediary stage to explore a different evolution of the system, rather than restarting the entire simulation from the beginning, saving much computational resources.

A choice that we made early on in the development of our library was to use MPI as the communication layer. One advantage of PCJ [28] over our approach lies in the fact that it supports its collective communications using a pure Java-based implementation [29]. This means that PCJ is easily portable to non-traditional HPC infrastructures such as the cloud [57]. In our case, users of our library need to install MPI and compile the Java compatibility layer before they can start executing programs. This is not a trivial operation and may discourage prospective users. In future work, we may try to make the choice of the communication layer modular to allow programmers to switch

between a pure Java implementation for easy development and program validation on the workstation, and MPI for cluster environments that support it.

Finally, we are considering introducing support for elasticity to our library. We believe our library would be a great help to programmers in such situations where the number of running processes increases and decreases over time.Indeed, the bulk relocation features would allow programmers to relocate the persistent objects of their program away from a process before releasing it, or, on the contrary, to offload some computation to a newly launched process with ease.

Chapter 5

Integrated Global Load Balancer

5.1 Introduction

In Chapter 4, we introduced our distributed collections library which supplements APGAS for Java [36] with the features necessary to write complex parallel and distributed programs with new and dynamic schedules. In particular, the features we introduced make it possible to dynamically relocate entries between processes using high-level abstractions.

We envision cases where programs run on non-dedicated hardware with potentially multiple processes competing for resources. This poses a challenge for programmers as the performance and computing resources available on each host, and the workload attributed to each process may vary over the course of an execution. Although manually monitoring the (distributed) situation over the course of an execution can allow programmers to take measures of their own to balance the load, this comes as a significant burden and has to potential to greatly obfuscate programs. Instead, we believe such load-balancing measures should be left to the library itself.

However, this poses a direct challenge to the principles implemented in our distributed collections. Indeed, the distribution management was up until now entirely left up to programmers. We therefore require a clear way for programmers to indicate the collections whose distribution is to be managed by the library and those whose are not.

Furthermore, even if a collection's distribution is managed by the library, there are applications which require that the location of each entry is known and remains fixed to ensure correctness. The PlhamJ financial market simulator [53, 54] is one such case. As we showed in the previous chapters, the agents may be relocated across processes to balance the computational load. The difficulty comes when the notifications of contracted trades need to be sent from the first process to the agents that have contracted that trade. To be able to correctly implement this, the location of each agent must be known and remain unchanged while this phase of the simulation takes place.

As a result, simply leaving the distribution of a collection up to the library throughout

the program execution is not an option. Intervals within which entries of the collection may be relocated by the library, and intervals during which the distribution of the collection is wholly under the programmer's control need to be enforced and clearly identifiable.

In this chapter, we introduce the load balancer integrated in our distributed collections library. The programming model we propose makes it easy for users to choose which parts of their program are conducted under this load balancer and which are not. Inspired by the lifeline-based global load balancer scheme discussed in Chapter 3, our integrated load-balancer is capable of automatically relocating work along with the entries of distributed collections to maintain the balance in an environment where the performance of hosts evolves over time.

The remainder of this chapter is organized as follows. In Section 5.2, we present the programming model and the semantics offered to programmers by our integrated load balancer. In Section 5.3, we discuss the internal implementation, the termination detection scheme, as well as the progress tracking system we developed. We present our evaluation in Section 5.4 before concluding in Section 5.5.

5.2 Programming model

In this section, we present how the global load balancer presents itself to users of our library. We first formal introduce the abstractions available to programmers in Section 5.2.1 before detailing its usage in our applications in Section 5.2.2.

5.2.1 Integrated load balancer semantics

5.2.1.1 Load-balanced context

In programs written with the help of our distributed collections library, the (re-)distribution of entries of distributed collections is normally entirely left up to programmers using the facilities demonstrated in Chapter 4. Suddenly allowing the library to relocate entries of distributed collections contradicts this principle.

To resolve this dichotomy, we introduce a specifically designed context within which our integrated load balancer is allowed to operate. This takes the form of the static method `underGLB` method which takes a closure as parameter as shown on line 5 of Listing 5.1. This choice of a static method taking a closure as parameter was made to minimize the impact on program legibility while allowing for some necessary internal preparations.

This also has the added benefit of defining a clear boundary within which entries of the distributed collections manipulated inside this block may be relocated by the library. Any computation which takes place outside of this context is ensured that the entries of distributed collections will not be arbitrarily relocated by the library.

Table 5.1: GLB operations currently implemented for class DistChunkedList<T>

Op	Parameter	Description
forEach	`Consumer<T>`	Applies the provided consumer on each `T` element in the collection
map	`Function<T,U>`	Creates a new chunked list which contains the result of the function given as parameter applied on this collection at the matching index
reduce	`Reducer<T>`	Makes a global reduction, applying the reduction on each element and merging the various reducer instances created in the process back into a single instance
toBag	`Function<T,U>`	Produces in parallel U instances from every T instance in the collection and collects them into a parallel receiver given as parameter

5.2.1.2 Staging mechanism, batch submission

Having defined a context within which load-balance will occur, there remains the question of with regards to which computation should the library be relocating distributed collection entries. Here, we make a number of arbitrary design choices. We consider load-balanced computation to be computation which applies in parallel to every entry recorded into a distributed collection. Our load balancer attempts to relocate entries of the targeted collection away from the hosts which take longer to process all their entries and towards processes which complete the entries they hold early.

The load-balanced computations are accessible through a special `GLB` handle of the targeted distributed collection class. They are analogous to the typical `forEach` and teamed `parallelReduce` methods discussed in the preceding chapter. However, the methods available through this handle can only be called from within the *underGLB* context; calling them outside of this context will result in exceptions to be thrown. Currently, only class `DistChunkedList` (our distributed array collection) and its derivatives are fitted with this feature. A summary of the computations supported is presented in Table 5.1.

Inside the `underGLB` block, GLB computations do not start as soon as they are called but are internally staged, with an instance of class `GlbFuture` representing that computation returned to the user. This allows for multiple computations to be "staged" before they start together. Our mechanism supports multiple computations on a single collection and multiple collections being processed at the same time.

Load-balanced operations go through a four-stage lifecyle: *Staged*, *Ready*, *Running*, and *Terminated*. All newly created operations initially start in the "Staged" state when a "GLB" method of a collection is called. The computation will actually start when either

```
1  import static handist.collections.glb.GlobalLoadBalancer.*;
2
3  DistCol<Ele> eleCol = new DistCol<>(); // Population omitted
4
5  underGLB(()->{
6    GlbFuture<DistCol<Integer>> fut1 = eleCol.GLB.map(e->e.makeInt());
7    DistCol<Integer> intCol = fut1.result(); // Case 1
8    GlbFuture fut2 = eleCol.GLB.forEach(e->e.update());
9    GlbFuture fut3 = intCol.GLB.reduce(new Average());
10   start(); // static import, Case 2
11   GlbFuture<DistCol<Integer>> fut4 =
12   eleCol.GLB.toBag(e->return e.makeInt());
13 }); // End of underGLB block, Case 3
14
15 // Programmer is back in full control
```

Listing 5.1: Program with a part operating under our library's integrated dynamic load balancer

one of the following three cases is encountered:

1. the result of a load-balanced computation is called through method `result` of that computation's `GlbFuture` object
2. the static method `start()` is called
3. the end of the `underGLB` block is reached

In all three cases, every "GLB" computation staged up until that point is moved to the "Ready" state. If there are no already running operation on the same collection, all the "Ready" operations for that collection are started immediately in a new batch and move into the "Running" state. If operations from a previous batch are still being computed at the time "Staged" operations transition into the "Ready" stage, the operations remain in that stage until the last running operation on that collection completes. When an operation has been computed on all the entries contained in the underlying distributed collection, it reaches its final "Terminated" stage.

Each of these three cases are represented in Listing 5.1. The first case presents itself on line 7. Up until that point, only the `map` operation on collection `eleCol` was staged on line 6. This single computation is started. The call to `fut1.result()` of line 7 blocks until this computation completes and the newly created `DistCol<Integer>` collection created as a result of the computation is returned.

On line 10, we encounter the second case. Here, both the `forEach` and the `reduce` operations staged on lines 8 and 9 are started. Note that in this case, two operations operating on two different collections are submitted to the load balancer. Method `start` is non-blocking and progress inside the GLB block continues while the computation takes place in the background. If it is later needed to wait on the completion of a running operation, this can be done by calling `fut2.result()` or `fut3.result()`. This

mechanisms allows programmers to perform other computation while the load-balanced operations are ongoing in the background.

Finally, the third case consisting of reaching the end of the GLB block causes the `toBag` operation staged on line 12 to start. In this case, the `forEach` operation previously staged on line 8 which operates on the same collection `eleCol` may still be running. As a result, the `toBag` operation staged on line 12 will be kept in the "Ready" state until the operation operating on the same collection terminates. When the previous `forEach` operation completes, it will trigger the start of the `toBag` operation. Progress of the `reduce` operation staged on line 9 remains unaffected as it concerns a different collection.

The `underGLB` method returns when all ongoing GLB operations (and any other non-GLB code present in the block) has completed. This guarantees that no more entries of the collections involved in a load-balanced operations are relocated by the library beyond the `underGlb` block. User-control over the distribution of collections is therefore whole again in the next part of the program.

There can be successive `underGlb` blocks opened in the same program. Programmers may choose to place the load balanced part of the computation within a loop, or place the main loop of their program inside the `underGlb` block. It is not allowed to transitively open another `underGlb` block from inside the block, nor is it possible to concurrently open two blocks from two concurrent asynchronous activities. Attempting to do so will result in exceptions to be thrown.

5.2.1.3 Priority between operations

A priority mechanism is embedded into the load balancer. By default, the order into which operations were staged is used to determine the relative priority between operations. If multiple operations are computed concurrently, workers will tend to take on the operation with the highest priority first, taking on other computations if no fragment of the highest priority operation can be obtained.

The current priority level of operations can be obtained through a method of the `GlbFuture` object representing this operation. Users of the library may override this value by specifying the priority of operations while they are "Staged" by calling the `GlbFuture.setPriority(int)` method. This can be useful if some operation staged later then others due to pending dependencies or other constraints needs to complete with higher priority to ensure program progress. In our current implementation, the priority to use for an operation can be modified during the "Staged" phase. It can no longer be modified once the operation transitions into the "Ready" phase.

5.2.1.4 Completion dependencies

We also prepared a completion dependency mechanism. This allows programmers to indicate that an operation cannot start before some other operation has completed. Unlike

the staging mechanism which concerns operation that operate on the same collection, this mechanism can be used on any operation regardless of the underlying operation.

Additional completion dependencies can be added to any "Staged" operation. Precautions need to be taken by the programmer to avoid inadvertently creating a circular dependencies. Failing to do so will prevent all the operations involved to progress into the "Ready" stage, resulting in none of the operations to be computed.

5.2.2 Examples

In this section, we demonstrate how our integrated load balancer is used on two applications, K-Means and the PlhamJ financial market simulator, and compare the version implemented with out integrated load balancer against the usual "teamed" version introduced in the previous chapter.

5.2.2.1 K-Means

The "teamed" implementation of the K-Means algorithm implemented with our library was previously presented in Section 4.4.2 We compare the versions with and without the use of our integrated global load balancer.

In both our implementations, each point is recorded into an instance of class `Point`. The cluster assignment step is implemented using a parallel "for each" method, while the average cluster location and the new centroid location are implemented using a user-defined reduction. The main program loop for both variants of the program are shown in Listings 5.2 and 5.3.

The actual code is sensibly the same for both implementations. In the "teamed" version, the `broadcastflat` method is used to launch the computation on all the hosts, whereas in the version using our integrated load balancer, the load balanced context `underGLB` is used. Similarly, the teamed reductions in Listing 5.2 are swapped for their "GLB" counterpart in Listing 5.3. As discussed in Section 4.4.2.1, the instances of `AvgPosition` and `ClosestPoint` given as parameter to the `reduce` methods are user-defined classes which extend an abstract `Reducer` class provided by our library.

In this application, the previous step in the iteration needs to complete before starting the next step. In the "glb" program, calling the blocking `result` to obtain the result in preparation for the next step causes the computation to start immediately after staging.

5.2.2.2 PlhamJ

The general routine of the PlhamJ financial market simulator was previously introduced in Section 4.4.1. Here, we compare the two different runner implementations: the manually load-balanced version discussed in the preceding chapter, and the version which relies on the integrated global load balancer to balance the agents between processes.

```
1  DistChunkedList<Point> points; // Initialization omitted
2  double[][][] initialCenters;
3
4  world.broadcastFlat(() -> {
5    double[][][] clusterCentroids = initialCenters;
6    for (int iter = 0; iter < reps; iter++) {
7      double[][][] centroids = clusterCentroids;
8      // Assign each point to a cluster
9      points.parallelForEach(p -> p.assignCluster(centroids));
10     // Avg cluster position computation
11     AvgPosition avgPos = points.team().parallelReduce(
12       new AvgPosition(K, DIMENSION));
13     // Compute the new centroids
14     ClosestPoint closestPoint = points.team().parallelReduce(
15       new ClosestPoint(K, DIMENSION, avgPos.centers));
16     clusterCentroids = closestPoint.closestPointCoordinates;
17   }
18 });
```

Listing 5.2: K-Means non-GLB implementation

```
1  DistChunkedList<Point> points; // Initialization omitted
2  double[][][] initialCenters;
3
4  GlobalLoadBalancer.underGLB(() -> {
5    double[][][] clusterCentroids = initialCenters;
6    for (int iter = 0; iter < reps; iter++) {
7      double[][][] centroids = clusterCentroids;
8      // Assign each point to a cluster
9      points.GLB.forEach(p -> p.assignCluster(centroids)).result();
10     // Avg cluster position computation
11     AvgPosition avgPos = points.GLB.reduce(
12       new AvgPosition(K, DIMENSION)).result();
13     // Compute the new centroids
14     ClosestPoint closestPoint = points.GLB.reduce(
15       new ClosestPoint(K, DIMENSION, avgPos.centers)).result();
16     clusterCentroids = closestPoint.closestPointCoordinates;
17   }
18 });
```

Listing 5.3: K-Means GLB implementation

```
1  DistCol<Agent> agents; // Initialization omitted
2
3  world.broadcastFlat(()->{
4    // ...
5    // previous steps omitted
6    agents.parallelToBag((Agent a, Consumer<List<Order>> collector)->{
7      List<Order> orders = a.submitOrders(markets);
8      // some output-related part omitted
9      if (orders != null && !orders.isEmpty())
10     collector.accept(orders);
11   }, orderBag);
12   // following steps omitted
13   // ...
14 });
```

Listing 5.4: Order submission of non-GLB program

```
1  DistCol<Agent> agents; // Initialization omitted
2
3  GlobalLoadBalancer.underGLB(()->{
4    // ...
5    // previous steps omitted
6    agents.GLB.toBag((Agent a, Consumer<List<Order>> collector)->{
7      List<Order> orders = a.submitOrders(markets);
8      // some output-related part omitted
9      if (orders != null && !orders.isEmpty())
10     collector.accept(orders);
11   }, orderBag);
12   // following steps omitted
13   // ...
14 });
```

Listing 5.5: Order-submission of GLB program

The "manually load-balanced" and the "glb" version of PlhamJ differ in two points. The first difference between the "manually load-balanced" and the "glb" version lies in the order submission step. This step is an example of a "parallel producer/receiver" pattern where each Agent returns the orders it wants to place in a list. The orders object returned by the Agent's `submitOrders` method are then recorded in the `orderBag` distributed collection. This collection is an instance of class `DistBag<T>`, which is specifically designed to accept many "T" objects coming from multiple threads concurrently by supplying a dedicated `Consumer<T>` handle to each thread. In this present case, the generic type `T` accepted by the `DistBag<T>` resolves to a `List<Order>`, hence the rather lengthy type of parameter `collector` which appears on line 6 in both Listing 5.4 and 5.5. To collect the orders placed by agents, the integrated load balancer version relies on a "GLB" operation while the "manually load-balanced" version calls a "local parallelism" implementation on each process using the `broadcastFlat` method of the `world TeamedPlaceGroup`.

The second difference lies in the way load balance is performed in these two implementations. The version with our integrated load balancer transfer `Agents` while the order submission is taking place. On the other hand, the "manually load-balanced" version measures the time taken by this step on each host over the course of a few iterations. If disparities appear, agents are relocated between processes using the features presented in Section 4.5.2 in a separate dedicated step. This is the reason we call this version "manually load-balanced," as the relocation of agents is written explicitly by the programmer.

5.3 Implementation

The load-balancing scheme we have currently implemented is inspired by the lifeline-based global load balancer of X10 whose key principles we recalled in Section 3.1.3. We rely on the same general global termination detection mechanism in our integrated global load balancer, with one enclosing `finish` per operation submitted. While this concept is re-used as is, there are a number of key differences between the original implementation and what we use in the context of our distributed collections library. We discuss the internal implementation of the scheme and point out the differences with the original scheme in this section.

5.3.1 Progress tracking with Assignment

In our scheme, accurately tracking the progress of each operation is necessary to guarantee that when relocating work becomes necessary, instances with some computation left in them are transferred between processes. This is done through what we call an "Assignment". An assignment represents a subset of the underlying distributed collection and the progress of the various operations being performed on that subset of the collection.

Currently, we have only implemented the `Assignment` class for our distributed array collections. For these collections, a pair of `long` integers is used to designate a range $[a, b)$ of entries in the array. The progress of each operation is tracked using a dedicated `long` integer for each operation whose value evolves from a to b as the computation progresses through the range designated by the assignment.

Only with an assignment is a worker of the integrated load balancer authorized to access the underlying collection, and even then, restricted only to the entries targeted by the assignment it holds. This guarantees that no concurrent accesses are made to individual objects in the collection.

The number of assignments left to complete for each GLB operation is tracked on each host using atomic counters. When a worker completes an operation on an assignment, it decrements the corresponding operation counter. When this counter reaches 0, meaning the last remaining assignment for this operation was completed, the worker unblocks the *witness activity* of the corresponding operation, allowing it to terminate. The nature of this "witness activity" and its purpose are discussed in Section 5.3.3.

5.3.2 Intra-host load-balancing

As part of the initialization process of the integrated load balancer, an initial assignment is prepared for each range of the distributed array held by the local handle. We keep these assignments in a single reserve on each host, as opposed to 2 in the multithreaded lifeline-based global load balancer [6], and sort them according to their priority.

As part of their main routine, worker threads start by obtaining an assignment from this centralized queue. They then progress the computation contained within this assignment by a fixed number of objects, the so-called "grain", before checking if some load-balancing measures need to be taken. When a worker completes the operation with the higher priority in its assignment, the assignment is either discarded if no other operations are present in the assignment, or placed back into the queue where it will be sorted according to the priority of the remaining operations.

When the queue gets depleted (either by a worker or through a lifeline steal), all the workers are asked to place some work back into it. In this case, workers that have enough computation left (determined by a minimum assignment size) will split the assignment they hold into 2 assignments targeting contiguous ranges. In the process, they appropriately update the number of assignments left to complete for each operation tracked by their assignment.

5.3.3 De-coupling of worker activities and computation

In the original lifeline-based scheme, all the workers are asynchronous activities managed by the same superseding finish. In our context this would imply running as many kinds of worker activities as there are ongoing operations, significantly obfuscating the scheme.

We decided instead to de-couple the worker threads from the termination detection. This allows us to spawn independent workers that can process any and all available assignments on the host regardless of the computation undertaken. However, this requires a number of changes to guarantee the proper global termination detection.

Termination detection of each ongoing operation is still achieved using the original scheme by using what we call a *witness activity*. In our load-balancing scheme, there is one such "witness" activity on each process for each operation ongoing on the host. This activity does not perform any computation and remains blocked on a semaphore throughout. When the last assignment of its corresponding computation has been completed by a worker, it gets unblocked and initiates the inter-host work stealing before completing.

When work is received from a remote host, a new witness activity is spawned and remain present until all the newly received assignments complete. When all witness activities of a given computation have terminated, their superseding finish returns, marking the completion of the corresponding computation.

5.3.4 Inter-host load balancing and termination detection

In the original lifeline-based global load balancer scheme, the computation at hand is self-contained within asynchronous activities. In the context of our distributed collections library, the assignments contain the information about the computation to perform, but they are not self-contained anymore. When inter-host load-balancing is performed, the entries of the distributed collection targeted by the assignments also need to be relocated. This is done using the relocation features of our distributed collection library.

One difference with the original lifeline-based load balancer is that we chose not to implement the random victim selection. This was initially done to simplify the implementation of the scheme. One consequence of not using any random steals is that it gives us a new perspective on the lifelines in the context of our integrated load balancer. As discussed in Section 3.1.2, the use of non-connected lifeline graphs is discouraged in the original scheme as it prevents work from trickling down to would-be idle hosts. In our situation where work is present on all hosts where entries of the distributed collection are present, this is not a concern.

Instead, using a non-connected lifeline network will guarantee that the entries of the distributed collection remain located with the subset of hosts connected by the lifelines. In the PlhamJ version implemented with our integrated load balancer discussed in Section 5.2.2.2, we rely on this property to ensure that no agents are relocated to the order-handling process. The lifeline strategy used in this application still consists in an hypercube, but with the first process excluded from the lifeline network.

Another subtle difference lies in the lifelines' nature. While the network of lifelines remains configurable as was the case for the original scheme, lifelines are established on a "per-collection" basis rather than a "per-computation" basis. This is necessary to pre-

serve the integrity of the global termination detection. As mentioned in the preceding section, the asynchronous activity used to answer the steal request needs to spawn a new "witness" activity on the thief. However it is possible that transferred assignments contain work pertaining to multiple GLB operations. In such a case, the activity transferring the assignments will have to spawn multiple "witness" activities registered into different "finish" constructs, something which is not possible under the normal APGAS `finish`/`async` semantics.

To resolve this issue, we extended the APGAS for Java library to allow any thread to spawn an activity registered into one or multiple arbitrary `finish`. The additional constructs we introduced have the potential to disrupt the termination detection mechanisms of the original library. Let us detail under which conditions this is agreeable and why it is possible in our particular situation.

Arbitrarily registering an asynchronous activity into multiple "finish" does not compromise the finish/async termination detection of APGAS if for every finish into which the answer activity is registered, there exists another running activity on the host. In our case, if work coming from multiple computation is transferred as part of an answer, then there necessarily exists a corresponding "witness" activity for each computation. It was therefore "possible" that this asynchronous activity was spawned by this witness activity. A problematic case would consist in registering an asynchronous activity into a finish which does not contain any ongoing activity on the local host, but this does not occur in our situation.

This extension of the APGAS library also allows us to somewhat simplify the load balancing scheme. In previous implementation of the multi-worker lifeline load-balancing scheme discussed in Chapter 3, a dedicated "lifeline answer" activity was blocked for most of the time and unblocked when lifeline answers became needed. In this present scheme, worker activities (which are not registered into any finish) can now directly answer thieves by spawning the appropriate asynchronous activity using our extended APGAS construct. Similarly, on the thief, a new witness activity is spawned for each `finish` the steal answer is registered into, re-instating a witness for each operation in the process.

5.3.5 Chunk splitting

One of the challenges of the inter-host work relocation is the potential for relocated assignments to target a portion of a chunk. In such cases, the "original" chunk registered into the distributed collection is split into two (or three) chunks before the targeted chunk is removed to be transferred over to the thief.

This poses a number of problems for our load balancing scheme, as other workers may be processing assignments targeting another portion of the original chunk and attempt to read portions of the original array while the splitting is taking place. Moreover, there may be multiple remote thieves being answered at the same time, and therefore multiple

chunks being split concurrently.

To preserve performance, we would like to avoid blocking the progress of any task as much as possible. We achieve this through the implementation of an algorithm which guarantees that all entries of a chunk undergoing a split remains readable. Multiple chunk splitting procedures may occur concurrently as long as the same underlying range is not already being split by another thread.

5.3.5.1 Range ordering

Inside of the arbitrary-rage distributed arrays, the chunks are kept in a concurrent and sorted map provided by the Java standard library, `ConcurrentSkipListMap`. In this map, index ranges $[a, b)$ represented by instances by object `LongRange` are mapped to `Object` arrays. The keys of the map are sorted in a total order through increasing a and decreasing b, i.e.

$$[0, 100) < [0, 50) < [0, 0) < [1, 100) < [50, 70)$$

This method of sorting the chunks simplifies the common operation consisting of accessing a single index, or a sub-range of entries in the arbitrary range array. Indeed, if the targeted range is contained in the `ChunkedList`, then it will necessarily be registered in the mapping to the left of the range looked for it. Using the perhaps more intuitive ordering consisting of sorting ranges through increasing a and increasing b actually requires additional checks to determine whether the targeted range is located to the left or to the right.

5.3.5.2 Splitting procedure

The algorithm used to split chunks is presented in Listing 5.6. The key idea is to add new mappings to the map before removing the original ones. As a result, the map transitively contains multiple arrays mapped from intersecting ranges (something which is normally not allowed). As the split entries are added, the original mapping sees its contents progressively shadowed by the mappings, until it becomes entirely hidden. Due to the key ordering in the map, inserting the newly created mappings from the right is critical as it guarantees that all object entries remain visible throughout this process, either through the newly created mappings or the original mapping. Figure 5.1 presents the gradual shadowing of the original mapping as the new chunks are inserted.

5.3.6 Restrictions

There are a number of conditions that need to be observed for programs to run successfully with our integrated global load balancer. First, no two handles of our distributed array collection can contain ranges that overlap. While this is in general possible, it is

```
1 Acquire a lock on the key of the range to split
2   Check if the key is still registered in the distributed array
3     If not, release the lock and restart the procedure
4   Prepare 2 (or 3) arrays to replace the existing mapping
5   Insert the newly created arrays into the map
6   Remove the original mapping
7 Release the lock
```

Listing 5.6: Chunk-splitting procedure

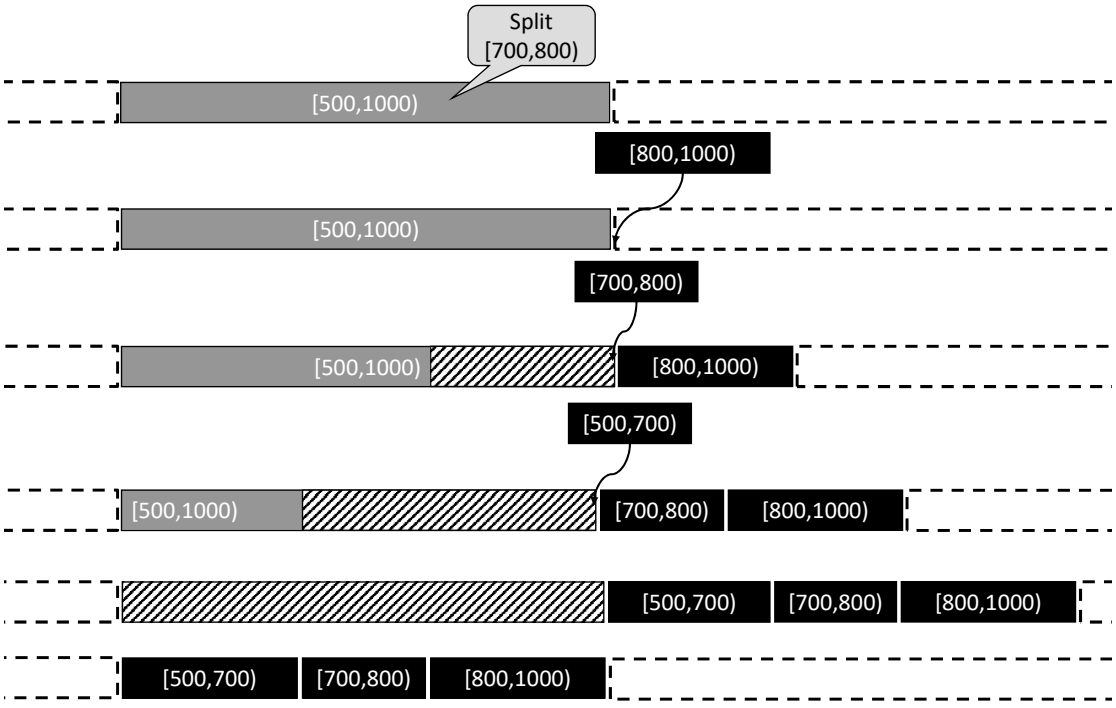


Figure 5.1: Chunk splitting guaranteeing concurrent read access

not compatible with our integrated load balancer as ranges relocated to the same process as part of inter-host load balancing may clash.

Secondly, no new entries can be recorded or removed from the distributed array while a computation is ongoing. More precisely, any new range added into a local handle of the distributed array would be ignored by any ongoing GLB computation as it will lack the corresponding assignment. Removing ranges from the collection while the computation is ongoing would result in unpredictable behavior as the assignments on which the removed ranges may or may not have been processed prior to removal. If entries need to be added or removed from a collection, it should be done outside of any ongoing GLB computation. Subsequently staged operation batches will re-generate the assignments based on the contents of the collection at that time and take into account the changes.

5.4 Evaluation

To evaluate the capabilities of our integrated global load balancer, we evaluate its performance on the 2 applications we presented in Section 5.2.2, *K-Means* and *PlhamJ*. For both of these applications, we compare the version of the program which relies on the integrated GLB presented in this article against the one which does not.

We have two objectives here. First, we want to estimate the amount of overhead created by our integrated load-balancing scheme in situations where no load-balance measures are necessary. Secondly, we want to check the capability of our scheme to react to dynamic changes in performance on the hosts on which it is running. For this, we repeat the experiment we performed in Section 4.6.3, in which we introduced the "Disturb" program to randomly steal away some computational resources of the hosts on which our program is running.

In these conditions we want to verify that the GLB program is able to run despite the presence of the competing computation and if it is capable of relocating work away from the hosts being disturbed.

The results are presented in Section 5.2.2.2. The characteristics of our Beowulf cluster on which we perform this evaluation are summarized in Table B.1.

5.4.1 K-Means

For K-Means, we perform this evaluation on our beowulf cluster on up to 8 "piccolo" servers. We conduct the evaluation in weak scaling. The details of the parameters used for this benchmark are presented in Table 5.2. We compare the glb version of our program against the non-glb version and the glb version stripped of its inter-process load balancing features.

There are two main takeaways from this experiment. First, the intra-host load balancing routines used in our integrated load balancer does not generate any noticeable

Table 5.2: Program parameters used for the K-Means evaluation

Parameter	Value
nb of points	10m per host (weak scaling)
nb of iterations	30
k	2000
point dimension	5

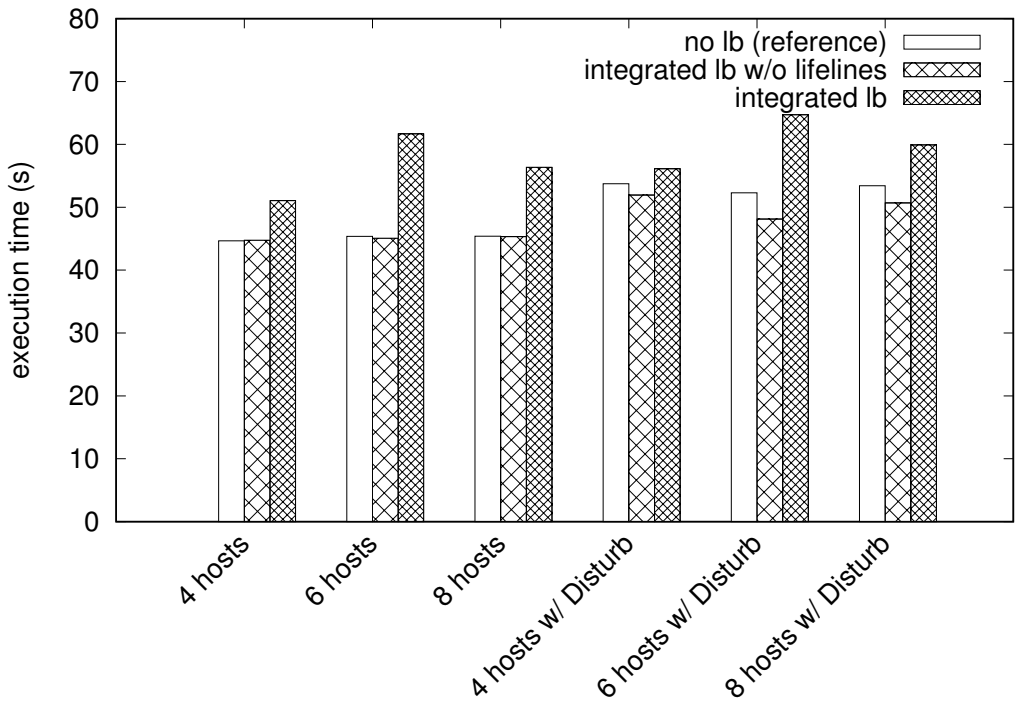


Figure 5.2: Execution time of the K-Means benchmark on up to 8 "piccolo" servers of our Beowulf cluster

overhead compared to the static allotment to each thread on our Beowulf cluster.This is established by comparing the execution time of the "no lb" and the "integrated lb w/o lifelines" in Figure 5.2.

Secondly, the inter-host load balancing strategy can be a significant source of performance issues. On the cluster configurations where no load balance is expected, the total execution time of the "integrated lb" version is up to 40% longer on the 6 hosts configuration. This can be explained by the manner in which lifeline steals are handled. Currently, there is no control over the amount of work transmitted to a remote host. A lifeline steal can take up to an arbitrary 10 assignments if there are available on the victim host. In cases where no load unbalance is expected, work can be needlessly relocated as no consideration as to the cost/merit is given.

In the presence of the Disturb program, the intra-host load balancing measures have a positive impact as the computation time is reduced compared to the "no lb" implementation. However, the complete load balancer does not improve performance compared to the reference version. This can be explained in part by the amount of computation which remains relatively small compared to the amount of data needed to transfer when relocating work a more appropriate strategy is clearly desired. Also, as only ever one host sees some of its computational resources stolen, redundant work relocation between unaffected processes can still occur, which would explain why the performance gap compared to the reference version is greater when using 6 or 8 hosts compared to 4 hosts.

5.4.2 PlhamJ

For PlhamJ, we repeat the evalaution we performed in Section 4.6.3 in which we introduced various cluster configuration. The details of every cluster and process allotment configuration was presented in Table 4.3. We add the execution time of the integrated load balancer to the comparison previously drawn in Figure 4.7 in Figure 5.3.

First, on Config A where no load-balance is needed, our PlhamJ version relying on our integrated load balancer presents some overhead compared to the reference version. We believe the main cause of this performance gap is the same as the case in K-Means in that agents are redundantly relocated between processes.

In all other configurations that either present static performance disparities or dynamic changes in performance or both, the version using our integrated load balancer shows performance on par or better than the reference version. The manually crafted load-balanced version remains the best performing in all situations.

Overall, this shows that our integrated load balancer is capable of handling the variety of situation with no effort required from programmers, while more experienced users willing to put in the effort are still capable of implementing their own load balancing strategies.

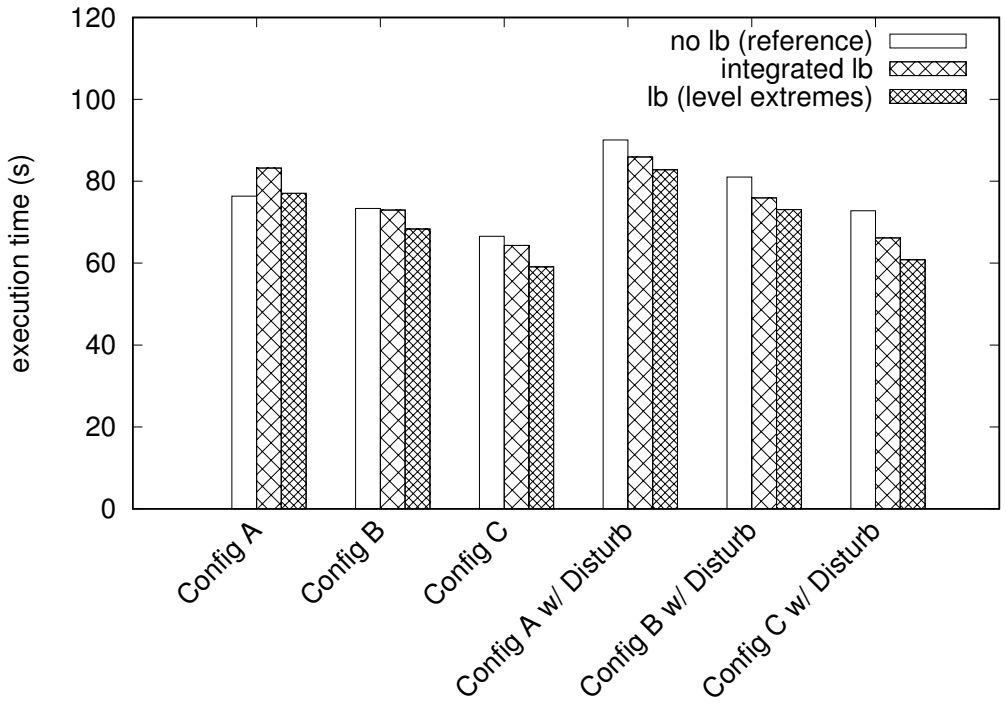


Figure 5.3: Execution time of the PlhamJ simulation depending on the cluster configuration

5.5 Conclusion and future work

In this chapter, we presented the global load balancer integrated into our distributed collection library. The programming interface we propose is simple to use and allows programmers to clearly identify the parts of their program that operate under this regime. We re-visited the global load balancer scheme of X10 and gained new insights into the "lifelines" used to implement this work-stealing scheme. While our current scheme allowed us to gain an advantage against statically distributed programs in some dynamic situations, further work is needed to implement an efficient scheme both when load-balance measures are and are not needed.

Further investigation into situations where multiple GLB computations are taking place at the same time are desired. In such cases, the load balancing strategy may have to take into account the fact that multiple operations are ongoing at the same time. PlhamJ is one such application. In its full version, PlhamJ supports 3 classes of trading agents: high-frequency, short-term, and long-term traders. In the evaluation presented here, no long-term traders were used, resulting in a simplified runtime. A number of non-trivial modifications to the simulator implementation are necessary to be able to compute both short-term and long-term agents' submissions concurrently. However, the programming model offered by the integrated load balancer which relies on "glb futures"

to manage the submission and the retrieval of computation of background computation will be instrumental in successfully implementing the full-fledged simulator. Indeed, this model allows use to overcome a limitation of the X10-style `finish`/`async` model in that it allows us manage the termination of 2 different group of transitive activities whose start and completion intertwine.

We made a number of arbitrary decisions in the design of our integrated distributed collection library. One of them was to only relocate data that still has some work inside of it. In the case of iterative applications, it would still make sense to relocate entries that have already being computed in anticipation of the next iteration. We could also choose to initiate inter-host work-stealing before all the work from a host has disappeared. Both of these ideas could be implemented without modifying the termination detection scheme. We also anticipate that depending on the application, the cost/benefit of performing the entry transfer will have to be taken into account when answering lifeline steal requests. Consider a parallel computation which is rather light in terms of computational cost but is based on large objects in terms of memory cost. If the cost of transferring the data entries takes longer than expected gain thanks to the newly gained balanced situation, transferring data entries with the computation would be counterproductive. In our current implementation, the size of the transferred assignments is not taken into account when answering lifeline thieves. We believe we are already seeing the adverse effects of this simplistic implementation in the both the K-Means and the PlhamJ results presented in Section 5.4. We will have to be revisit this issue in future work.

Another choice that we made in the design of our integrated load balancer was to only allow one such context to be opened ata time. Under certain condition, this restriction could be relaxed. If we consider collections that are defined on entirely disjoint subsets of the "world," then it should be possible to have two disjoint load balancers each operating on the subset. The issue with relaxing this constraint is that it opens the problem of how to handle cases where collections have definitions that overlap. In the absence of a clear and simple answer to this problem, we chose to restrict our design to the manner we did to preserve the clarity of the abstraction we propose.

One setting which has a consequential influence on the performance of the GLB mechanism is the granularity, i.e. the number of entries of the collection that are computed by a worker thread before the runtime is checked. The results presented here correspond to the "best-case" scenario for both K-Means and PlhamJ, with vastly different values for either application: 500 and 5 respectively. Choosing other values yielded significantly poorer results. It would make the integrated GLB significantly easier to use if it could be fitted with a similar grain tuning mechanism as was introduced in Chapter 3.

Currently, only the variants of our arbitrary index array distributed collection supports load-balanced operations. The challenge in porting the same features to our other distributed collections lies in the progress tracking through the `Assignment` class. For distributed maps that may use any user-specified object as key, there is no trivial progress

description. Even if a total order exists between the keys contained in the map, describing sets of entries with a pair of keys may not be sufficient, as when work is received from a remote host, inserted keys may land inside the range of an existing assignment. For these reasons,a hypothetical implementation of the `Assignment` class for a distributed map may have to rely on the internal representation of the map.

Chapter 6

Conclusion

In this thesis, we laid out techniques to help non-expert programmers and application developers to create distributed programs. We integrated a tuning mechanism to the multi-worker global load balancer scheme. Our tuner adjusts a critical parameter for the performance of the scheme and is capable of handling the variety of problems it may receive and is robust against variations in implementation. This contributes to making this scheme more user-friendly as it removes the responsibility of setting an arbitrary parameter from the user.

We also introduced a distributed collections library as a complement to the APGAS for Java library. This library brings distributed collections with an interface similar to the standard library of the Java programming language and fitted with features dedicated to handling the distributed nature of the computation. In particular, it is possible for programmers to dynamically relocate entries of distributed collections between the processes used to run the program through high-level abstractions. This makes distributed application development easier, as demonstrated by the successful implementation of the PlhamJ financial market simulator and its evolution to support dynamic load balancing, something not possible in previous implementations.

Finally, we introduced a dynamic load balancer integrated into the distributed collection library. Using the proposed programming model, programmers may temporarily surrender the distribution management of a collection to the load balancer. In turn, the entries of the distributed collections may be relocated between processes by the library. This system allows programmers who may not want to implement a dedicated load balancing strategy into their program to instead leave this responsibility to the library.

This work was also an opportunity to reach the limits of the `finish`/`async` model of X10. Indeed this feature had to be hacked to allow the implementation of the integrated load balancer termination detection mechanism. Another limit we identified is that it is not possible to interleave the launch and termination of multiple `finish` blocks. This was recognized by the creators of Habanero-Java who added a "future" variant of

`finish` to their language [58]. Moreover, not all the features of X10 have trickled down into APGAS for Java. More advanced functionalities such as *clocks* [59], other features related to resilience and elasticity [60, 61] have been integrated in X10 but do not have an equivalent in AGPAS for Java.

An area where the abstractions provided by our distributed collection library will be *elastic programs*. Contrary to "normal" distributed programs, malleable and evolving programs [62] are capable of dynamically changing the number of processes used to run the program during execution. The benefits of such programs have long been theoretically studied [63], with support in distributed programming language runtime appearing decades ago [8]. More recently, actual implementation of malleable programs and the necessary job scheduler and resource management system needed to support it have been published [64, 65, 66]. One major hurdle for the development of malleable programs is the need to relocate information between processes when either processes used in the running computation are released or added to the program. In that regard, our distributed collection library will be capable of helping programmers relocate important object instances. Further work is necessary to integrated the abstractions that will allow programmers to dynamically adjust the number of processes of their APGAS for Java programs.

Bibliography

[1] Riken. *Japan's Fugaku gains title as world's fastest supercomputer*. June 2020. URL: https://www.riken.jp/en/news_pubs/news/2020/20200623_1/index.html (visited on 12/07/2022).

[2] Marten van Steen and Andrew S. Tanenbaum. *Distributed Sytems, 3rd ed., Version 03*. www.distributed-systems.net, Dec. 2020. ISBN: 978-90-815406-2-9.

[3] Wikipedia, the free encycplopedia. *Eight queens puzzle*. 2022. URL: https://en.wikipedia.org/wiki/Eight_queens_puzzle (visited on 07/14/2022).

[4] Vijay A. Saraswat et al. "Lifeline-Based Global Load Balancing". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: Association for Computing Machinery, 2011, pp. 201–212. ISBN: 9781450301190. DOI: 10.1145/1941553.1941582.

[5] Wei Zhang et al. "GLB: Lifeline-Based Global Load Balancing Library in X10". In: *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*. PPAA '14. Orlando, Florida, USA: Association for Computing Machinery, 2014, pp. 31–40. ISBN: 9781450326544. DOI: 10.1145/2567634.2567639.

[6] Kento Yamashita and Tomio Kamada. "Introducing a Multithread and Multistage Mechanism for the Global Load Balancing Library of X10". In: *Journal of Information Processing* 24.2 (2016), pp. 416–424. DOI: 10.2197/ipsjjip.24.416.

[7] The MPI Forum. "MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855.

[8] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Nov. 1999. ISBN: 9780262288040. DOI: 10.7551/mitpress/7055.001.0001.

[9] Mark Baker et al. "mpiJava: An object-oriented java interface to MPI". In: *Parallel and Distributed Processing* (1999). Ed. by José Rolim et al., pp. 748–762.

[10] Mark Baker and Bryan Carpenter. "MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing". In: *Parallel and Distributed Processing*. Ed. by José Rolim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 552–559. ISBN: 978-3-540-45591-2.

[11] Oscar Vega-Gisbert, Jose E. Roman, and Jeffrey M. Squyres. "Design and implementation of Java bindings in Open MPI". In: *Parallel Computing* 59 (2016). Theory and Practice of Irregular Applications, pp. 1–20. ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.08.004.

[12] Kalé Laxmikant V. and Krishnan Sanjeev. "CHARM++". In: *Parallel Programming Using C++*. London, England: The MIT Press, July 1996. ISBN: 9780262287654. DOI: 10.7551/mitpress/5241.003.0009. eprint: https://direct.mit.edu/book/chapter-pdf/192174/9780262287654_cae.pdf.

[13] Bilge Acun 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: IEEE Press, 2014, pp. 647–658. ISBN: 9781479955008. DOI: 10.1109/SC.2014.58.

[14] Tom White. *Hadoop: The Definitive Guide, Fourth Edition*. Ed. by Mike Loukides and Meghan Blanchette. O'Reilly Media Inc., 2015. ISBN: 9781491901632.

[15] Jules S. Damji et al. *Learning Spark*. Ed. by Jonathan Hassell, Michele Cronin, and Deborah Baker. O'Reilly Media, Inc., 2020. ISBN: 9781492050049.

[16] Yunming Zhang. "HJ-Hadoop: An Optimized Mapreduce Runtime for Multi-Core Systems". In: *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*. SPLASH '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 111–112. ISBN: 9781450319959. DOI: 10.1145/2508075.2514875.

[17] M. Nithya and N. Uma Maheshwari. "Load rebalancing for Hadoop Distributed File System using distributed hash table". In: *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. 2017, pp. 939–943. DOI: 10.1109/ISS1.2017.8389317.

[18] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1383–1394. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797.

[19] Dili Wu and Aniruddha Gokhale. "A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration". In: *20th Annual International Conference on High Performance Computing*. 2013, pp. 89–98. DOI: 10.1109/HiPC.2013.6799133.

[20] Mattias De Wael et al. "Partitioned Global Address Space Languages". In: *ACM Comput. Surv.* 47.4 (May 2015). ISSN: 0360-0300. DOI: 10.1145/2716320.

[21] Robert W. Numrich and John Reid. "Co-Array Fortran for Parallel Programming". In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. ISSN: 1061-7264. DOI: 10.1145/289918.289920.

[22] John Mellor-Crummey et al. "A New Vision for Coarray Fortran". In: *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. PGAS '09. Ashburn, Virginia, USA: Association for Computing Machinery, 2009. ISBN: 9781605588360. DOI: 10.1145/1809961.1809969.

[23] Dan Bonachea and Gary Funck. *UPC Language and Library Specifications (Version 1.3)*. Tech. rep. Lawrence Berkeley National Lab., Nov. 2013. DOI: 10.2172/1134233.

[24] Cristian Coarfa et al. "An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 36–47. ISBN: 1595930809. DOI: 10.1145/1065944.1065950.

[25] John Bachan et al. "UPC++: A High-Performance Communication Framework for Asynchronous Computation". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 963–973. DOI: 10.1109/IPDPS.2019.00104.

[26] Masahiro Nakao et al. "XcalableMP Implementation and Performance of NAS Parallel Benchmarks". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. PGAS '10. New York, New York, USA: Association for Computing Machinery, 2010. ISBN: 9781450304610. DOI: 10.1145/2020373.2020384.

[27] Hitoshi Murai, Masahiro Nakao, and Mitsuhisa Sato. "XcalableMP Programming Model and Language". In: *XcalableMP PGAS Programming Language, From Programming Model to Applications*. Ed. by Mitsuhisa Sato. Singapore: Springer, 2021, pp. 1–71. DOI: 10.1007/978-981-15-7683-6.

[28] Marek Nowicki, Łukasz Gorski, and Piotr Bała. "PCJ – Java Library for Highly Scalable HPC and Big Data Processing". In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 12–20. DOI: 10.1109/HPCS.2018.00017.

[29] Marek Nowicki, Łukasz Górski, and Piotr Bała. "Scalable computing in Java with PCJ Library. Improved collective operations". In: *PoS* ISGC2021 (2021), p. 007. DOI: 10.22323/1.378.0007.

[30] Olivier Tardieu et al. "X10 and APGAS at Petascale". In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014, pp. 53–66. ISBN: 978-1-4503-2656-8. DOI: 10.1145/2555243.2555245.

[31] Vincent Cavé et al. "Habanero-Java: The New Adventures of Old X10". In: PPPJ '11. Kongens Lyngby, Denmark: Association for Computing Machinery, 2011, pp. 51–61. ISBN: 9781450309356. DOI: 10.1145/2093157.2093165.

[32] Shams Imam and Vivek Sarkar. "Habanero-Java Library: A Java 8 Framework for Multicore Programming". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: Association for Computing Machinery, 2014, pp. 75–86. ISBN: 9781450329262. DOI: 10.1145/2647508.2647514.

[33] B.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: 10.1177/1094342007078442.

[34] Steven J. Deitz et al. "Global-View Abstractions for User-Defined Reductions and Scans". In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '06. New York, New York, USA: Association for Computing Machinery, 2006, pp. 40–47. ISBN: 1595931899. DOI: 10.1145/1122971.1122980.

[35] *X10 language documentation, Class DistArray*. URL: http://x10.sourceforge.net/x10doc/latest/ (visited on 07/15/2022).

[36] Olivier Tardieu. "The APGAS Library: Resilient Parallel and Distributed Programming in Java 8". In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. Portland, OR, USA: ACM, 2015, pp. 25–26. ISBN: 978-1-4503-3586-7. DOI: 10.1145/2771774.2771780.

[37] Apache Maven Project. *What is Maven?* 2022. URL: https://maven.apache.org/what-is-maven.html (visited on 08/07/2022).

[38] James Gosling et al. "Threads and Locks". In: *The Java Language Specification, Java SE 8 Edition*. Oracle, 2015. Chap. 17. URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html.

[39] Stephen Olivier et al. "UTS: An Unbalanced Tree Search Benchmark". In: *Languages and Compilers for Parallel Computing*. Ed. by George Almási, Călin Caşcaval, and Peng Wu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 235–250. ISBN: 978-3-540-72521-3.

[40] Oracle. *Java™ Platform, Standard Edition 8 API Specification, Class ForkJoinPool*. 2014. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html (visited on 05/20/2022).

[41] Lei Wang et al. "An Adaptive Task Creation Strategy for Work-stealing Scheduling". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. Toronto, Ontario, Canada: ACM, 2010, pp. 266–277. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772992.

[42] 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: ACM, 2014, pp. 3–12. ISBN: 978-1-4503-2642-1. DOI: 10.1145/2597652.2597665.

[43] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. "Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2014, pp. 857–868. DOI: 10.1109/SC.2014.75.

[44] Min Seung-Jai, Iancu Costin, and Yelick Katherine. "Hierarchical Work Stealing on Manycore Clusters". In: *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*. PGAS '11. Galveston Island, TX, USA, 2011. URL: http://pgas11.rice.edu/papers/MinEtAl-HotSLAW-Work-Stealing-PGAS11.pdf.

[45] G. Cong et al. "Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing". In: *2008 37th International Conference on Parallel Processing*. Sept. 2008, pp. 536–545. DOI: 10.1109/ICPP.2008.88.

[46] Robert D. Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System". In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: 10.1145/209937.209958.

[47] Donald E. Knuth. *Dancing links*. 2000. DOI: 10.48550/ARXIV.CS/0011047.

[48] Jens Clausen. *Branch and Bound Algorithms – Principles and Examples*. Tech. rep. Mar. 1999. URL: https://janders.eecg.utoronto.ca/1387/readings/b_and_b.pdf (visited on 07/14/2022).

[49] Georgios Varisteas and Mats Brorsson. "Palirria: Accurate On-line Parallelism Estimation for Adaptive Work-Stealing". In: *Proceedings of Programming Models and Applications on Multicores and Manycores*. PMAM'14. Orlando, FL, USA: ACM, 2007, 120:120–120:131. ISBN: 978-1-4503-2657-5. DOI: 10.1145/2578948.2560687.

[50] Jonas Posner and Claudia Fohry. "Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing". In: *50th International Conference on Parallel Processing Workshop*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450384414. DOI: 10.1145/3458744.3473361.

[51] Aleksandar Prokopec et al. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 31–47. ISBN: 9781450367127. DOI: 10.1145/3314221.3314637.

[52] L. A. Smith, J. M. Bull, and J. Obdrzálek. "A Parallel Java Grande Benchmark Suite". In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC '01. Denver, Colorado: Association for Computing Machinery, 2001, p. 8. ISBN: 158113293X. DOI: 10.1145/582034.582042.

[53] Takuma Torii et al. "Platform Design for Large-Scale Artificial Market Simulation and Preliminary Evaluation on the K computer". In: *Artificial Life and Robotics* 22.3 (2017), pp. 301–307. DOI: 10.1007/s10015-017-0368-z.

[54] Daisuke Fujishima et al. "Overlapping Communication and Computation for Large-Scale Artificial Market Simulation". In: *Proc. of 22nd International Symposium on Artificial Life and Robotics (AROB 2017)*. Beppu, Japan, Jan. 2017, pp. 708–713.

[55] J. K. Lee and D. Gannon. "Object oriented parallel programming: experiments and results". In: *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1991, pp. 273–282. DOI: 10.1145/125826.105186.

[56] Daisuke Fujishima and Tomio Kamada. "Collective Relocation for Associative Distributed Collections of Objects". In: *Int. J. Softw. Innov.* 5.2 (Apr. 2017), pp. 55–69. ISSN: 2166-7160. DOI: 10.4018/IJSI.2017040104.

[57] Marek Nowicki, Łukasz Górski, and Piotr Bała. "Performance evaluation of Java/PCJ implementation of parallel algorithms on the cloud (extended version)". In: *Concurrency and Computation: Practice and Experience* (2021), p. 15. DOI: 10.1002/cpe.6536.

[58] Vincent Cave, Shams Mahmood, and Vivek Sarkar. *Habanero-Java*. 2014. URL: https://wiki.rice.edu/confluence/display/HABANERO/Habanero-Java (visited on 07/15/2022).

[59] Tomofumi Yuki. "Revisiting Loop Transformations with X10 Clocks". In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 1–6. ISBN: 9781450335867. DOI: 10.1145/2771774.2771778.

[60] David Cunningham et al. "Resilient X10: Efficient Failure-Aware Programming". In: *SIGPLAN Not.* 49.8 (Feb. 2014), pp. 67–80. ISSN: 0362-1340. DOI: 10.1145/2692916.2555248. URL: https://doi.org/10.1145/2692916.2555248.

[61] David Grove et al. "Failure Recovery in Resilient X10". In: 41.3 (July 2019). ISSN: 0164-0925. DOI: 10.1145/3332372. URL: https://doi.org/10.1145/3332372.

[62] Dror G. Feitelson and Larry Rudolph. "Toward convergence in job schedulers for parallel supercomputers". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–26. ISBN: 978-3-540-70710-3.

[63] Gregory Mounie, Christophe Rapine, and Dennis Trystram. "Efficient Approximation Algorithms for Scheduling Malleable Tasks". In: *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '99. Saint Malo, France: Association for Computing Machinery, 1999, pp. 23–32. ISBN: 1581131240. DOI: 10.1145/305619.305622.

[64] Abhishek Gupta et al. "Towards realizing the potential of malleable jobs". In: *2014 21st International Conference on High Performance Computing (HiPC)*. 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116905.

[65] Suraj Prabhakaran et al. "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 429–438. DOI: 10.1109/IPDPS.2015.34.

[66] Marco D'Amico, Ana Jokanovic, and Julita Corbalan. "Holistic Slowdown Driven Scheduling and Resource Management for Malleable Jobs". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337909.

[67] Y. Murase, T. Uchitane, and N. Ito. "An open-source job management framework for parameter-space exploration: OACIS". In: *Journal of Physics: Conference Series* 921 (Nov. 2017), p. 012001. DOI: 10.1088/1742-6596/921/1/012001.

Appendix A

Source Code

A.1 Aggregated program routines

In Listings A.1 and A.2 are presented the aggregated listings of the PlhamJ and K-Means applications discussed in Section 4.4.1 and 4.4.2 respectively. The reader familiar with the MolDyn benchmark will notice that the temperature scaling and the performance tracking are absent from the code we present here. These are included in our actual program, but we chose to omit them for brevity and to focus on the computation core of the program.

A.2 Published software

The entirety of the source code discussed in this thesis has been released under open source license on GitHub.

Source code for the topics discussed in Chapter 3 can be found in the following repository:

- Multi-worker Lifeline-based Global Load Balancer `https://github.com/handist/JavaGLB`

Source code for the topics discussed Chapter 4 & 5 can be found in the following repositories:

- Distributed Collections library `https://github.com/handist/collections`
- Benchmarks for the Distributed Collections library `https://github.com/handist/collections-benchmarks`
- PlhamJ Financial Market Simulator `https://github.com/plham/plhamJ`

```
1  CachableArray<Market> markets;    // market information
2  DistCol<Agent> agents; // agents
3  DistBag<List<Order>> orderBag; // orders submitted by agents
4  DistMultiMap<Long,AgentUpdate> contractedOrders; // trades contracted
5  // Runtime variables
6  TeamedPlaceGroup world = TeamedPlaceGroup.getWorld();
7  boolean isMaster = here() == place(0); // place(0) handles orders
8  long accumulatedOrderComputeTime = 0l; // time on order-submission
9  int lbPeriod = 10; // load-balance period (configurable)
10 int iter; // current iteration
11
12 world.broadcastFlat(() -> {
13   // (1) Broadcast the updated state of markets
14   markets.broadcast(MarketUpdate::pack, MarketUpdate::unpack);
15   // (2) Submit agent orders
16   long startOrder = System.nanoTime();
17   if (!isMaster) agents.parallelToBag((agent, orderCollector) -> {
18     List<Order> orders = agent.submitOrders(markets);
19     if (orders != null && !orders.isEmpty()) {
20       orderCollector.accept(orders);
21     }
22   }, orderBag);
23   accumulatedOrderComputeTime = System.nanoTime() - localSubmitTime;
24   // (3) Collect all orders on the ''master'' place(0)
25   orderBag.team().gather(place(0));
26   // (4) Match buy and sell orders, populating `contractedOrders`
27   finish(()->{
28     // (4 - optional) balance the agents between places 1..n
29     if (iter % lbPeriod == 0) {  async(()->{
30         // Exchange time information between hosts
31         long[] computationTime =
32         world.allGather1(accumulatedOrderComputeTime);
33         // prepare a relocator
34         CollectiveMoveManager mm = new CollectiveMoveManager(world);
35         performLoadBalance(computationTimes, mm);
36         mm.sync(); // perform the relocation
37         // reset accumulated order-submission time
38         accumulatedOrderComputeTime = 0l;
39         agents.updateDist(); // update the agents' distribution
40       });
41     }
42     if (isMaster) handleOrders(); // procedure details omitted
43   });
44   // (5) Inform the agents of the trades they made
45   // (5.1) Relocate contracted trade information to agents' location
46   LongRangeDistribution agentDistribution = agents.getDistribution();
47   contractedOrders.relocate(agentDistribution);
48   // (5.2) Update the agents that contracted a trade
49   if (!isMaster) contractedOrders.parallelForEach((idx, updates) -> {
50     // Retrieve the agent targeted by the update
51     Agent a = agents.get(idx);
52     // Apply each update for this agent
53     for (AgentUpdate u : updates) { a.executeUpdate(u);}
54   });
55 }); // end of broadcast flat block
```

Listing A.1: Main procedure of the PlhamJ distributed simulator

```
1  TeamedPlaceGroup world = TeamedPlaceGroup.getWorld();
2  LongRange particleRange = new LongRange(0, nbParticles);
3  CachableChunkedList<Particle> particles; // Init omitted
4  int Ndivide = 5; // Number of columns/lines to split the product into
5  long seed = 0; // Seed used to assign the tiles to hosts
6
7  world.broadcastFlat(() -> {
8    // Replicate the particles across process
9    if (world.rank() == 0) {
10     particles.share(particleRange);
11   } else {
12     particles.share();
13   }
14
15   // Prepare the interaction pairs
16   RangedList<Particle> prl = particles.getChunk(particleRange);
17   RangedListProduct<Particle, Particle> product =
18   RangedListProduct.newProductTriangle(prl, prl);
19   // Split interactions into tiles and assign them to hosts
20   product = product.teamedSplit(Ndivide, Ndivide, world, seed);
21
22   // Prepare an accumulator for the force computation
23   Accumulator<Sp> acc =
24   new AccumulatorCompleteRange<>(particleRange, Sp::newSp);
25
26   for (i = 0; i < iter; i++) {
27     // Compute the force contribution of each pair
28     product.parallelForEachRow(acc,
29     (Particle p, RangedList<Particle> pairs, tla) -> {
30       force(p, pairs, tla);
31     });
32
33     // Merge all the force contributions in the accumulators back
34     // into the designated particles
35     particles.parallelAccept(acc, (Particle p, Sp a) -> {
36       p.addForce(a);
37     });
38
39     // Sum the force contributions accross all hosts
40     particles.allreduce((out, Particle p) -> {
41       out.writeDouble(p.xforce);
42       out.writeDouble(p.yforce);
43       out.writeDouble(p.zforce);
44     }, (in, Particle p) -> {
45       p.xforce = in.readDouble();
46       p.yforce = in.readDouble();
47       p.zforce = in.readDouble();
48     }, MPI.SUM);
49
50     // Move the particles based on the computed force
51     particles.parallelForEach(p -> move());
52   }
53 });
```

Listing A.2: Hybrid MolDyn implementation

Appendix B

Evaluation environment

The hardware and software configuration of the cluster and supercomputer used in our evaluations are summarized in the tables below. The large number of executions necessary for this work was managed using OACIS [67].

Table B.1: Characteristics of our Beowulf cluster

Server Type	"piccolo"	"harp"
Nb of nodes	8	1
Processor	Intel Xeon E3-1230 V2 (3.3GHz, 4 cores)	2 Intel Xeon E5-2680 V3 (2.5GHz, 24 cores combined)
RAM	16GB DDR4	128GB DDR4
Interconnection	Gigabit Ethernet	
Java version	OpenJDK v1.8.0_312	
MPI version	Open MPI v3.1.6 with MPJ-Express v0_44 Java native bindings	

Table B.2: Characteristics of the OakForest-PACS supercomputer

Server Type	Fujitsu PRIMERGY CX1640 M1
Nb of nodes	8208
Processor	Intel Xeon Phi 7250 (1.4 GHz, 68 cores)
RAM	96GB DDR4
Interconnection	Intel Omni-Path (100 Gbps)
Java version	Open JDK 1.8.0_222
MPI version	Intel MPI with MPJ-Express v0_44 Java native bindings

Publications

Journals

- Patrick Finnerty, Tomio Kamada, and Chikara Ohta. A self-adjusting task granularity mechanism for the Java lifeline-based global load balancer library on many-core clusters. Concurrency Computat Pract Exper. 2021; 34(2):e6224. DOI: 10.1002/cpe.6224

- Patrick Finnerty, Yoshiki Kawanishi, Tomio Kamada, and Chikara Ohta. Supercharging the APGAS Programming Model with Relocatable Distributed Collections. Scientific Programming (to appear). Preprint: arXiv:2207.05452

International Conference Proceedings

- Patrick Finnerty, Tomio Kamada, and Chikara Ohta. 2020. Self-adjusting task granularity for Global load balancer library on clusters of many-core processors. In Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '20). Association for Computing Machinery, New York, NY, USA, Article 4, 1–10. DOI: 10.1145/3380536.3380539

- Patrick Finnerty, Tomio Kamada, and Chikara Ohta. 2022. Integrating a global load balancer to an APGAS distributed collections library. In Proceedings of the Thirteenth International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '22). Association for Computing Machinery, New York, NY, USA, 55–64. DOI: 10.1145/3528425.3529102

Domestic Presentations

- Patrick Finnerty, Yoshiki Kawanishi, Tomio Kamada, Chikara Ohta. 2021. Experience in testing MPI+Java parallel and distributed programs with JUnit. Information Processing Society of Japan, 135th Programming Symposium.

Doctor Thesis, Kobe University
"*High-productivity Abstractions and Efficient Runtime for Dynamically Load-balanced Distributed Programs on Multi/Many-core Clusters*", 138 pages
Submitted July 15, 2022.
When published on the Kobe University institutional repository *Kernel*, the publication date shall be indicated on the cover of the repository version.