Automatically balancing relocatable distributed collections

Finnerty, Patrick
Kamada, Tomio
Ohta, Chikara

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE

Automatically Balancing Relocatable Distributed Collections

Patrick Finnerty*[1] | Tomio Kamada[2] | Chikara Ohta[1]

[1]Graduate School of System Informatics, Kobe University, Hyogo, Japan
[2]Department of Intelligence and Informatics, Konan University, Hyogo, Japan

Correspondence
*Patrick Finnerty, Email: finnerty.patrick@fine.cs.kobe-u.ac.jp

Present Address
657-0013
Graduate School of System Informatics S302
Rokkodai-cho 1-1,
Kobe-shi Hyogo-ken Japan

Abstract

In previous work, we introduced a distributed collections library for the APGAS for Java programming model. This library makes it possible for programmers to develop complex distributed programs thanks to the many abstractions and computation patterns supported. In particular, programmers can fully and dynamically change the distribution of data entries through high-level abstractions. However, the problem of balancing the load between processes remains, especially in cases where multiple processes may be concurrently executing on a single host, or when the performance of the hosts used differs.

To address this issue and to relieve the burden of programming a load balancing strategy for a specific application, we created a dynamic load balancer integrated into our library. This load balancer operates within a specific context in a manner which does not interfere with the program legibility. Internally, we implement a scheme inspired by the lifeline-based global load balancer scheme first introduced in X10. We evaluate the performance of our integrated load balancer on a small-scale Beowulf cluster.

KEYWORDS:
distributed collection, dynamic load balancing, distributed computation, work stealing

1 | INTRODUCTION

Creating parallel and distributed programs is difficult, with much research on programming models and libraries to ease the burden of programmers. In previous work, we created our own distributed collections library [1] to supplement the APGAS for Java library [2]. This made it possible to write complex distributed and parallel programs and to leverage the parallelism available on recent many-core processors with new and dynamic schedules. In particular, our library makes 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, or in clusters composed of computers featuring disparate performance characteristics. This poses a challenge for programmers as the performance and computing resources available on each host and the workload attributed to each process may differ from host to host (static imbalance) and vary over the course of an execution (dynamic imbalance). As under our programming model, computation is performed under an owner-compute rule approach, distributing the entries of the distributed collection uniformly will yield load imbalances. Although manually monitoring the (distributed) situation over the course of an execution could 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.

This contradicts the approach we adopted in our previous work [1]. Indeed, the distributed collections we developed were meant to be controlled entirely by programmers through high-level abstractions. For the library to also relocate entries of those distributed collections, the programming model needs to be expanded and clarified to accommodate for both user and library control. A load balancing strategy capable of handling static and dynamic imbalances is also required.

In this article, we present 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 driven under this load-balanced regime. Inspired by the lifeline-based Global Load Balancer scheme

Listing 1 Distributed Hello World in Java

```
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 " + places().size() + " places");
6      finish(() -> {
7        for (Place p : places()) {
8          asyncAt(p, () -> System.out.println("Hello from " + here()));
9        }
10     });
11     System.out.println("Bye");
12   }
13 }
```

first implemented in X10 [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 may evolve over time. This makes it possible to create programs where part of the data distribution is entirely managed by the programmer and other parts are automatically managed by the library.

This article is an extended version of our PMAM'22 contribution [4]. The internal implementation of the load balancer has been significantly modified compared to our previous publication, incurring subtle changes to the programming model we propose. Prior implementation issues have been resolved and the overall efficiency of our integrated load balancer has been improved. We expand our evaluation to include experiments on a non-uniform Beowulf cluster with various process/host allocation strategies. We show that our method can be beneficial in some situations, but that further improvements to the inter-host stealing scheme are needed to satisfactorily cover the whole spectrum of situations.

The remainder of this article is organized as follows. We start by introducing some necessary background in Section 2. In Section 3, we present the programming model and the semantics offered to programmers by our integrated load balancer. In Section 4, 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 before discussing related work in Section 6. Finally, we conclude in Section 7.

2 | BACKGROUND

2.1 | APGAS for Java

Our work focuses on the Asynchronous Partitioned Global Address Space (APGAS) programming model as implemented in X10 [5]. This model was later ported to Java [2], with the X10 keywords converted to static methods taking lambda-expressions as parameter. In this model, a process running on a host is called a `Place`. We use both "process" and "place" interchangeably in this article. Asynchronous activities can be launched on a remote host with method `asyncAt`. Termination detection is implemented through an elegant method called `finish` which waits for all transitively spawned asynchronous activities to complete.

A short *Hello World* example along with a possible program output demonstrating these principles are presented in Listing 1 and 2. In this example, the main thread running at Place 0 will not progress further than the `finish` method until all the places (including itself) have written their message to the standard output before writing "Bye."

2.2 | Lifeline-based Global Load Balancer

The lifeline-based Global Load Balancer is a work-stealing scheme that was first implemented in X10 [3,6]. The key innovation of this scheme is that it introduced preferential channels for work-stealing, the so-called "lifelines."

The computation to perform is contained into user-defined tasks queues. The main worker process consists in processing a certain number of these tasks and then answering steal requests from remote processes by giving away some (generally half) of its tasks.

Listing 2 Sample output of the Hello World program of Listing 1 running with 4 places

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

When a process runs out of work and fails to steal some from a randomly selected victim, it signals itself as "dormant" to its lifeline neighbors and remains idle rather than repeatedly probing remote processes for work. The computation process restarts when one of the lifeline neighbors sends some work using an asynchronous activity.

This scheme elegantly solves the problem of termination detection as all asynchronous activities that carry work are transitively spawned from the same `finish`. When all the activities on all the hosts terminate, the enclosing `finish` returns, guaranteeing that global termination was achieved.

In a later extension, this scheme was modified to support multiple workers per process [7,8]. In this variant, each process maintains two work queues on each host to implement work-sharing between the workers with one queue, and work-stealing between the host other hosts. The workers collectively attempt to keep some work available for stealing in both of the queues so long as their local task queues allows them to give some away.

2.3 | Distributed Collection Library for APGAS

To complement the APGAS for Java library, we created a distributed collections library [1]. While the details of this library and the features it supports fall outside the scope of this paper, we recall here its main characteristics with the help of a short example.

In this library, we provide a number of distributed collections (distributed map, distributed array) for the APGAS programming model. Each distributed collection is implemented as a group of local handles located on each process and linked together by a globally unique id. Programmers can record entries into the local handle of a distributed collection by using the usual APGAS asynchronous activities.

The most significant innovation of our distributed collections library is that it allows programmers to relocate entries of a collection between its local handles through high-level abstractions. This is done at the user's initiative, with entries to relocate being described by their keys in the case of a distributed map, or by ranges in the case of our distributed array `DistChunkedList` and its derivative `DistCol`.

The program presented in Listing 3 illustrates these characteristics. The `TeamedPlaceGroup` object obtained on the first line of Listing 3 represents the group of all APGAS places. It is used on line 2 to create the distributed map `dMap`, meaning that every process participating in the execution will have a handle of this distributed collection. On line 3, a first entry is recorded into the map by the main thread, resulting in the `main:running` mapping to be recorded on the first process, i.e. place 0.

From lines 4 to 11, the `broadcastFlat` method is used to spawn the same asynchronous activity on all hosts participating in the computation. This short-hand replaces the otherwise explicit `finish` and for loop needed to spawn the same activity on all places, enhancing the clarity of the program. The asynchronous activity given as parameter to method `broadcastFlat` makes every place add a new entry into their local handle of the distributed map `dMap` on line 5. Then, a collective relocator is created on line 6. This object is used to register various elements of distributed collections to be transferred from a handle to another. In this case, only the first place relocates its `main:running` entry to Place 1. 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 shown on Figure 1. Notice in particular that the `main:running` entry has been removed from the handle on Place 0 and inserted into the handle of Place 1.

Our library also supports a number of computation patterns that require communication between hosts. We call such methods "teamed." This is the case for instance of *reductions*. Our library provides the facilities for programmers to implement user-defined reductions on the entries contained in our distributed collections. In the case of a "teamed" reduction, the computation will take place in two phases: (1) a "local" reduction is computed on each handle, and (2) the general result of the reduction computed by merging the local results of each handle. One such example is later detailed in Section 3.2.1.

Internally, specific communication patterns involving multiple processes are supported by MPI through the Open MPI Java bindings [9]. Matters related to termination detection are handled through the APGAS `async/finish` constructs.

Listing 3 Distributed map creation and record insertion

```
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 });
```

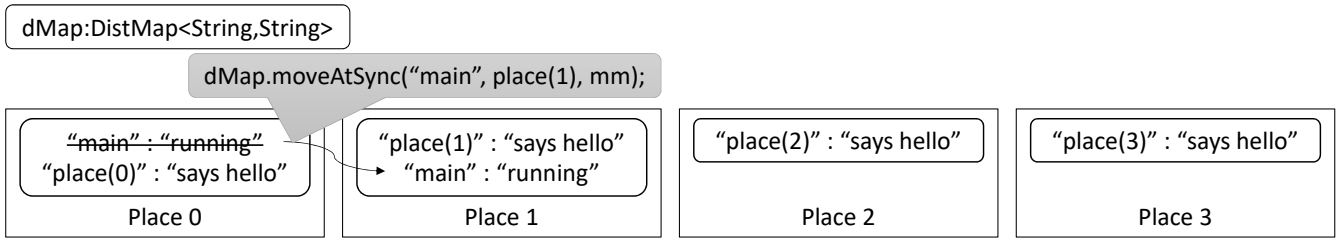


Figure 1 State of the distributed map dMap after the Listing 1 program has run with 4 processes

Table 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

3 | PROGRAMMING MODEL

In this section, we present how the integrated load balancer presents itself to users of our library. We first formally introduce the abstractions available to programmers in Section 3.1 before detailing its usage in our applications in Section 3.2. The entirety of the source code of our distributed collections library[1] and our applications[2,3] is freely available on GitHub.

Listing 4 Program with a part operating under our library's integrated dynamic load balancer

```
1  import static GlobalLoadBalancer.*;
2
3  DistCol<Ele> eleCol = new DistCol<>(); // init and population omitted
4  underGLB(()->{
5    GlbFuture<DistCol<Integer>> fut1 = eleCol.GLB.map(e->e.makeInt());
6    DistCol<Integer> intCol = fut1.result(); // Case 1
7    GlbFuture fut2 = eleCol.GLB.forEach(e->e.update());
8    GlbFuture fut3 = intCol.GLB.reduce(new Average());
9    start(); // Case 2
10   GlbFuture<DistCol<Integer>> fut4 = eleCol.GLB.toBag(e->return e.makeInt());
11 }); // End of block, Case 3
```

3.1 | GLB semantics

3.1.1 | 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 Section 2.3. Allowing the library to relocate entries of distributed collections fundamentally 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` which takes a closure as parameter as shown in Listing 4. 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 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. Outside this block, the programmer remains in full control over the entries' distribution.

3.1.2 | Staging mechanism, batch submission

Load-balanced computations on our distributed collections are accessible through a special `GLB` handle. The methods available through this handle can only be called from within the *underGLB* context. Currently, only class `DistChunkedList` (our distributed array collection) and its derivatives are fitted with this feature. A summary of the computations supported is shown in Table 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 computed at the same time.

GLB 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. Load-balanced computation may start when either one of the following three cases is encountered:

1. the result of a GLB computation is called through method `GlbFuture.result()`
2. the static method `GlobalLoadBalancer.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 operations on the same collection, all the "Ready" operations for that collection are started immediately in a new batch and moved into the "Running" state. If operations from a previous batch are still being computed at that time, 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. A short example illustrating each of the three cases triggering the start of the computation is presented in Listing 4.

The first case presents itself on line 6. Up until that point, only the `map` operation on collection `eleCol` was staged on line 5. This single GLB computation is started. The call to `fut1.result()` of line 6 blocks until this computation completes and the newly created `DistCol<Integer>` collection created as a result of the computation is returned.

[1]Distributed Collections Library https://github.com/handist/collections
[2]Benchmarks for our Distributed Collections Libraryhttps://github.com/handist/collections-benchmarks
[3]PlhamJ Financial Market Simulator https://github.com/plham/plhamJ

On line 9, we encounter the second case. Here, both the `forEach` and the `reduce` operations staged on lines 7 and 8 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 previously launched GLB computation, this can be done by calling `fut2.result()` or `fut3.result()`. These mechanisms allows programmers to perform some other computation while the GLB operations are ongoing.

Finally, the third case consisting of reaching the end of the GLB block causes the `toBag` operation staged on line 10 to start. In this case, the `forEach` operation previously staged on line 6 which operates on the same collection `eleCol` may still be running. As a result, the `toBag` operation staged on line 9 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 7 remains unaffected as it concerns a different collection.

The `underGLB` returns when all ongoing operations have completed. This guarantees that no more entries of the collections involved in GLB operations are relocated by the library beyond the `underGlb` block. User-control over the distribution of collections is therefore complete again as the program continues.

3.1.3 | Priority between operations

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

The current priority level of GLB operations can be obtained through method `GlbFuture.getPriority`. Users of the library may override this value by specifying the priority of operations while they are "Staged" by calling the `GlbFuture.setPriority()`. The priority to use for an operation cannot be modified once it switched to the "Ready" stage.

3.1.4 | 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.

3.2 | Examples

3.2.1 | K-Means

In this section we detail the implementation of a distributed K-Means implemented with our library. We compare the versions with and without the use of our integrated load balancer.

K-Means is an iterative clustering algorithm which separates points into a pre-defined k number of clusters. An iteration of K-Means consists in three steps. Starting from randomly selected cluster centroids, each point considered is assigned to a cluster based on which is closest to it. Secondly, with the points each assigned to a cluster, the average position of each cluster is computed. Lastly, the point closest to the average position of each cluster is chosen as the new centroid for the next iteration. The algorithm can either be run for a set number of iterations or until the centroids stop moving.

In both our implementations, each point is recorded in 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 user-defined reductions. The main program loop for both variants of the program are shown in Listings 5 and 6.

The actual code is sensibly the same for both implementations. The only technicality lies in the non-GLB program where the reduction methods are called through the `team()` handle of the `points` distributed collection. This handle of the `DistChunkedList` class is used to distinguish between reductions that are computed using only the entries of the local handle (e.g. `points.reduce(...)`, not used here) and reductions taking place between all places in the cluster (e.g. `points.team().reduce(...)`) as used on line 11 and 13 of Listing 5. 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 "non-glb" program, the computation starts as soon as the method is called. In the "glb" program, we rely on the blocking `result()` method immediately after staging the computation.

Listing 5 K-Means non-GLB implementation

```
// Initialization omitted
DistChunkedList<Point> points;
double[][] initialCenters;
world.broadcastFlat(() -> {
  double[][] clusterCentroids = initialCenters;
  for (int iter = 0; iter < reps; iter++) {
    double[][] centroids = clusterCentroids;
    // Assign each point to a cluster
    points.parallelForEach(p -> p.assignCluster(centroids));
    // Avg cluster position computation
    AvgPosition avgPos = points.team().parallelReduce(new AvgPosition(K, DIMENSION));
    // Compute the new centroids
    ClosestPoint closestPoint = points.team().parallelReduce(
      new ClosestPoint(K, DIMENSION, avgPos.centers));
    clusterCentroids = closestPoint.closestPointCoordinates;
  }
});
```

Listing 6 K-Means GLB implementation

```
// Initialization omitted
DistChunkedList<Point> points;
double[][] initialCenters;
GlobalLoadBalancer.underGLB(() -> {
  double[][] clusterCentroids = initialCenters;
  for (int iter = 0; iter < reps; iter++) {
    double[][] centroids = clusterCentroids;
    // Assign each point to a cluster
    points.GLB.forEach(p -> p.assignCluster(centroids)).result();
    // Avg cluster position computation
    AvgPosition avgPos = points.GLB.reduce(new AvgPosition(K, DIMENSION)).result();
    // Compute the new centroids
    ClosestPoint closestPoint = points.GLB.reduce(
      new ClosestPoint(K, DIMENSION, avgPos.centers)).result();
    clusterCentroids = closestPoint.closestPointCoordinates;
  }
});
```

3.2.2 | PlhamJ

PlhamJ is the Java version of the Plham financial market simulator first written in X10 [10]. Simulations are given to the simulator in the form of a JSON configuration file which lists the markets, agents, and events that will occur over the course of the simulation. The length of the simulation is determined by the number of iterations specified in the configuration. The simulator provides deterministic results following a given seed.

Internally, there are multiple "runners" that can run the simulation. We first quickly describe how simulations are run in general before outlining the differences between the version relying on our integrated load balancer "glb," and a load-balanced version whose strategy is directly implemented in the runner, the "manual" version.

A Plham simulation iteration consists of the following basic steps:

Listing 7 Order submission of the non-GLB version of the PlhamJ program

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

Listing 8 Order-submission of the GLB version of the PlhamJ program

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

1. *market update*: the latest state of the markets is broadcast to the agents in the simulation
2. *order submission*: the agents place their orders on the markets
3. *order handling*: the buy and sell orders of agents are matched, resulting in trades being contracted
4. *agent update*: the agents that made trades during the order-handling step are informed

In both the "manual" and "glb" distributed runners, one process is dedicated to order-handling while the other processes are dedicated to computing the agents' orders during the order submission step. Our distributed collection library allows for the transfer (relocation) of updated market information, orders, and contracted trade information, the details of which fall outside the scope of this article.

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 submit 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 in line 4 of both Listing 7 and 8. To collect the orders placed by agents, the "glb" version relies on a GLB operation while the "manually load-balanced" version internally relies on iterators with static allotment of entries to threads, as shown in Listings 7 and 8.

The second difference lies in the way load balance is performed in these two implementations. The "glb" version performs load-balancing of `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 2.3. This is the reason we call this version "manually load-balanced" as, contrary to the "glb" version, the relocation of agents is written explicitly by the programmer.

4 | IMPLEMENTATION

In this section, we detail the important implementation topics of our load balancer integrated into our distributed collections library. The load-balancing scheme we have implemented is inspired by the lifeline-based global load balancer of X10 whose key principles we recalled in Section 2.2. We rely on the same general global termination detection mechanism in our integrated global load balancer, with one enclosing `finish` per operation submitted. While key concepts are re-used as is, there are a number of differences between the original implementation and what we use in the context of our distributed collections library.

4.1 | 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 the distributed array collection. For this collection, 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 per 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 to only 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 4.3.

4.2 | 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 two in the multithreaded GLB[7] scheme, 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 their assignment into two assignments targeting contiguous ranges. In the process, they appropriately update the number of assignments left to complete for each operation tracked by their assignment.

4.3 | De-coupling of worker activities and computation

In the original lifeline-based scheme, all the workers are asynchronous activities managed by the same enclosing finish. In our context this would imply running as many kinds of worker activities as there are ongoing GLB 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 computation 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 ongoing computation 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 inter-host work stealing before returning.

When work is received from a remote host, a new witness activity is created and remains present until all the newly received assignments complete. When all witness activities of a given computation have terminated, the finish under which they were spawned returns, marking the completion of the corresponding computation.

4.4 | 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 ease the already complex implementation of the scheme and could now be implemented. One consequence of not using any random steals is that it gives us a new perspective on lifelines in the context of our integrated load balancer. In the original scheme, the use of non-connected lifeline graphs is discouraged as it prevents work from trickling down the lifelines 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. 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 "GLB runner" discussed in Section 3.2.2, 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 a 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 preserve 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 implementation.

To resolve this issue, we extended the APGAS for Java [2] 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 that host. In our case, if work pertaining to multiple computations is transferred as part of a lifeline answer, then there necessarily exists a corresponding "witness" activity for each computation transferred. 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` object which does not have any ongoing activity on the local host, something that does not occur in our situation.

This extension of the APGAS library also allows us to somewhat simplify the load balancing scheme. In previous implementations of the multi-worker lifeline load-balancing scheme [8,11], 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.

4.5 | Restrictions imposed by our integrated load balancer

There are a number of conditions that need to be observed for programs to run successfully with our integrated load balancer. First, no two handles of our distributed array collection can contain ranges that overlap. While this is in general possible, it is not compatible with our integrated load balancer as ranges relocated to the same process as part of inter-host load balancing measures 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. The next batch of GLB operations will re-generate the assignments based on the contents of the collection at that time and take these changes into account.

Table 2 Hardware characteristics of our Beowulf cluster

Machine Type	"piccolo"	"harp"
Nb of servers	8	1
Processor	Intel Xeon E3-1230 V2 (3.3GHz, 4 cores)	dual Intel Xeon E5-2680 V3 (2.5GHz, 24 cores combined)
RAM	16GB DDR4	128GB DDR4
Java version	OpenJDK v1.8.0_312	
MPI version	Open MPI v4.1.4 with Open MPI Java bindings	

Table 3 Program parameters used for K-Means

Parameter	Value
k	500
nb of points	1m per host (weak scaling)
point dimension	5
iterations	15

Table 4 Program parameters used for PlhamJ

Parameter	Value
nb of agents	1,000
artificial load	20,000
nb of iterations	300

5 | EVALUATION

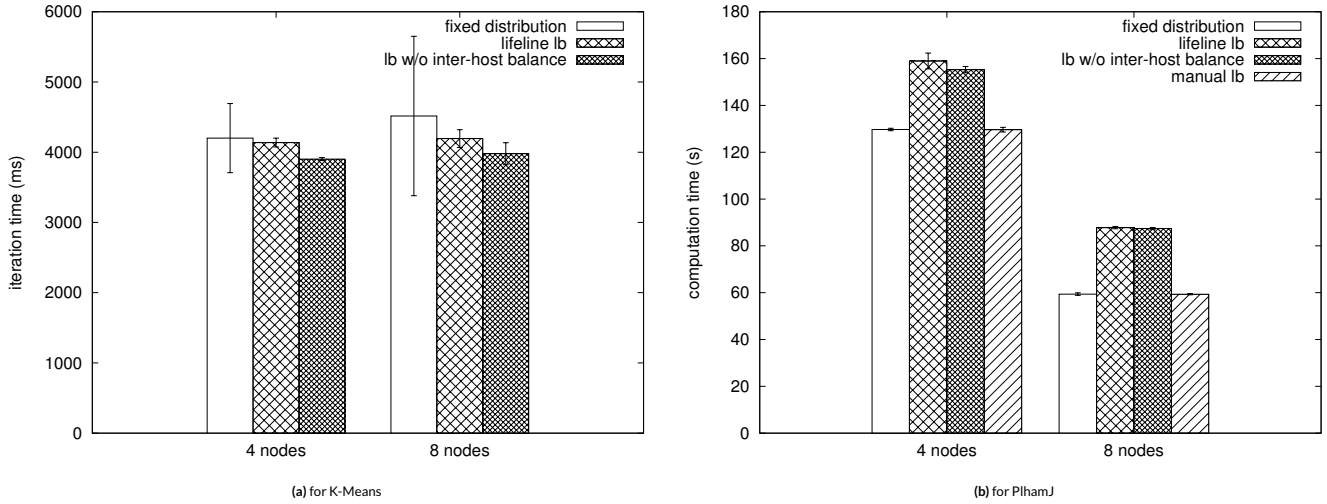
To evaluate the capabilities of our integrated load balancer, we measure its performance on the 2 applications we presented in Section 3.2, *K-Means* and *PlhamJ*. For both of these applications, we compare the version of the program which relies on the integrated load presented in this article (lifeline lb) against the one which does not (fixed distribution). We also compare against our load balanced version deprived from the inter-host load balancing features (lb w/o inter-host balance). For PlhamJ, we also compare against a load balance strategy manually implemented in the simulator itself in which the program periodically transfers agents from the most overloaded process to the most under-loaded process every 10 iterations (manual lb).

Our objective here is three-fold. 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 evaluate the capabilities of our integrated load balancer in situations of static imbalance due to discrepancies in the hosts used. Lastly, we want to evaluate the capabilities of our load balancer in situations of dynamic load imbalance. The results of these three experiments are presented in Sections 5.1, 5.2, and 5.3 respectively. We then discuss the results in Section 5.4.

We perform the evaluation on our Beowulf cluster composed of two types of hosts: "piccolo" hosts which feature a 4-core CPU, and the higher-parallelism "harp" host which features two 12-core CPUs. The details of our system is outlined in Table 2. Evaluation with the K-Means program are conducted in weak scaling. For Plham, we prepared a special simulation in which agents are assigned a certain amount of artificial load when submitting their orders. More details about the configuration of both applications are given in Table 3 and 4. Each benchmark is run 5 times in each configuration. We present the average iteration time and the average computation time for K-Means and PlhamJ respectively. The standard deviation is also indicated with the error bracket. The large number of executions necessary for this study was managed using OACIS [12].

5.1 | Overhead

To measure the overhead cost of our integrated load balancer, we run our programs on our "piccolo" servers only, in a situation where no inter-host load balance is necessary. The results are presented in Figure 2.



(a) for K-Means

(b) for PlhamJ

Figure 2 Result of the overhead experiment

We can draw two conclusions from these results. First, as can be seen in the K-Means experiment, the internal load balancing feature seem to bring some kind of stability to the performance obtained, with the standard deviation of iteration time reduced to under 100 ms compared to above 500 ms for the program which does not use our integrated load balancer. A small gain in performance is also obtained. Still on K-Means, the inter-host load balancing procedures incur a 3% penalty compared to the integrated load balancer removed from the inter-host load balancing features.

In stark contrast, our internal load balancing system seems to incur a significant penalty to the performance of the program in the case of PlhamJ. This can be explained by the small number of agents involved in this simulation, a thousand, compared to the millions of points handled in K-Means. The cost of allocating the *Assignments* and the other data structures necessary to the scheme is too large to be offset by the benefits.

Consistent with the K-Means results, no significant overhead due to the inter-host load balancing procedures is witnessed here, as there is practically no difference in computation time between between the "lifeline lb" and "lb w/o inter-host balance" executions.

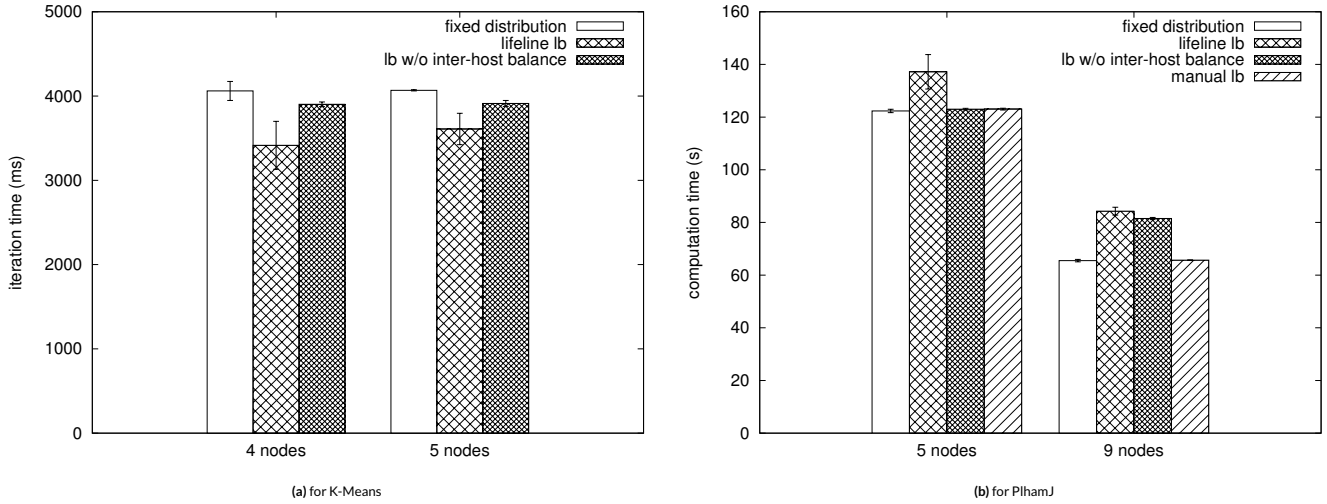
5.2 | Static imbalance

For this experiment, we run our K-Means and our PlhamJ programs with one process allocated on our "harp" server and the other processes allocated on the "piccolo" servers. This creates some imbalance as the "harp" server has a much higher parallelism than the "piccolo" servers (24 threads against 4 threads).

The results are presented in Figure 3. For K-Means, our integrated load balancer shows some benefits on both the 4 and 5 nodes executions, delivering the shortest iteration times of all versions. Greater variations in the iteration times can be witnessed and can be explained by the inter-host load balancing procedures that transfer the entries in the first iterations of the program.

For PlhamJ, the relative gap between the programs relying on the integrated load balancer and those that do not is reduced compared to the overhead results presented previously, with the version deprived of the inter-host load balancing measures just on par on the 5 node execution. This is not due to a gain brought by the internal load balancer implementation but rather by a loss of performance of the non load-balanced version. Indeed on the 9 node execution where the relative load imbalance is reduced, the fixed distribution program still performs better.

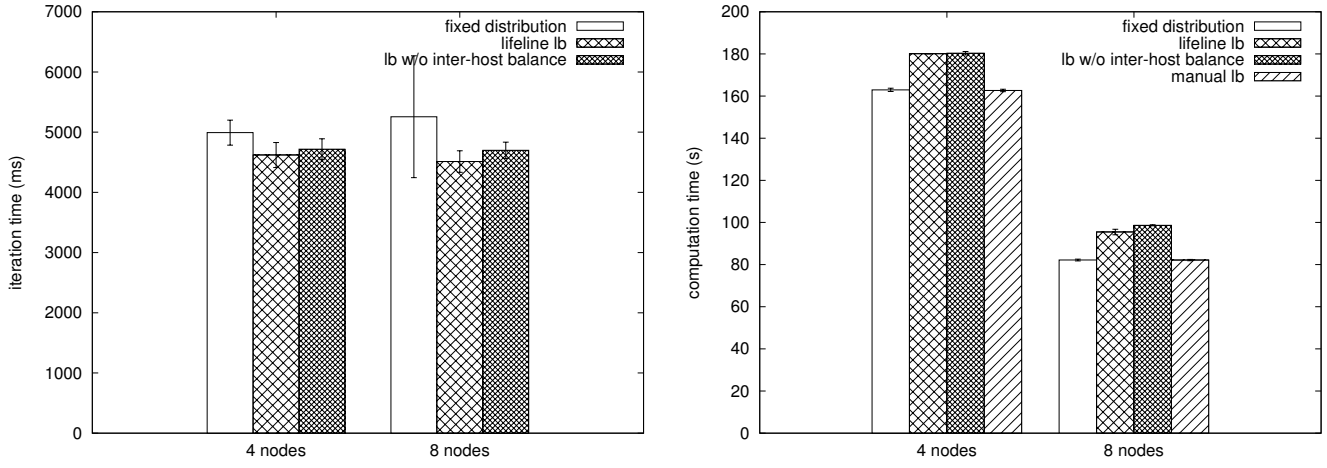
More surprisingly, the inter-host load balancing feature seem to be counterproductive in this situation, with the PlhamJ execution time on 5 nodes about 20 second longer than the others. This points to another limitation of our current implementation: the size of the assignments stolen is not taken into account when relocating work. As a result, it is possible for too much work to be transferred in a round of load balancing, causing yet another unbalanced situation in the next round. This issue is exacerbated by our PlhamJ implementation which allocates the agents in a single chunk on each process, initially resulting in a single assignment in the integrated load balancer. From there on, the size of the assignments available for inter-host work stealing depends on the nondeterministic nature of the intra-host worker-to-worker assignment splitting procedures. This was not an issue with K-Means, where the large number of points are separated in chunks of 10 thousand entries.



(a) for K-Means

(b) for PlhamJ

Figure 3 Result of the static imbalance experiment



(a) for K-Means / 4 nodes

(b) for PlhamJ / 2 disturb threads

Figure 4 Result of the dynamic imbalance experiment

5.3 | Dynamic imbalance

For this last experiment, we only use the "piccolo" servers with a single process allocated per node. To simulate situations of dynamic imbalance, we introduce a deterministic "parasite" program called `Disturb`. This program randomly chooses a victim hosts and spawns a number of threads which perform hash computations in a loop. This effectively steals some computation resources from the main program. After a set amount of time has elapsed, a new victim is chosen and the program spawns threads there. The sequence of disturbed hosts is deterministic following an initial seed to be able to replicate the same disturbance across multiple executions. When activated, our parasite program occupies two threads of one host of the hosts for 20 seconds before moving to another host. The results are presented in Figure 4.

On K-Means our integrated load balancer again delivers the best performance, with iteration times shorter by about 8% compared to the version without the integrated load balancer. The gain is smaller than with the static imbalance situation as the integrated load balancer needs to constantly adapt to the "Disturb" program, incurring entry relocation costs throughout the execution while the "fixed distribution" program remains slightly unbalanced throughout. For PlhamJ, the inter-host load balancing features delivers slightly better performance than the version deprived from the inter-host load balancing features, although similar to the overhead experiment, the management costs of the integrated load balancer remain too heavy to be advantageous.

5.4 | Discussion

As seen in the previous section, there are situations where our integrated load balancer does not deliver ideal performance. Most of the issues appear to be related to the inter-host load balancing procedures and the complexity of the scheme to be able to support them. In that regard, we made a number of arbitrary decisions in the design of our scheme. One of them was to only relocate entries that still have a matching assignment with work inside of it. Seeing as the applications we presented are iterative, 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.

Another possibility to improve the inter-host load balancing procedure could consist in adopting a profiling approach instead of the lifelines. Keeping the current abstractions unchanged, it would be possible to measure the computational load of each process over a certain period of time and determine the appropriate number of entries to relocate. This could also simplify the scheme as no load balancing would take place during the computation as is the case in our current implementation.

Another element to consider may be the size of the entries to relocate. Both applications we presented here rely on relatively "lightweight" objects. When relocating object instances that comprise more data, it may become needed to strike a balance between the amount of work stolen and the time needed to actually perform the transfer.

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 tuning mechanism to automatically adjust this setting, similar to what we did in previous work [8,11].

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.

6 | RELATED WORK

There are several runtimes and languages that aim at handling the distributed nature of program. Chapel is a PGAS programming language developed as part of the DARPA's high productivity computing systems program [13]. It allows distribution of arrays through a number of library-supplied *Block*, *Cyclic*, and *Cyclic Block* distributions. However, dynamic relocation of some arbitrary ranges in an array is not supported. It also lacks support for distributed maps, which are supported in our distributed collections library.

Our closest competitor is Charm++ [14]. It relies on problem over-decomposition into many "Chares" and is capable of dynamically relocating them on processing elements based on information obtained through profiling and selectable policies. This surrenders all the distribution control to the Charm++ runtime. In applications with more intractable communication patterns and completion dependencies such as PlhamJ, the completion and quiescence detection provided by Charm++ would make it possible to implement but with greater effort than the programming model we propose. In Charm++, the order in which messages are processed by chares is non-deterministic. The advantage of our system over Charm++ is that the completion of certain asynchronous activities can be elegantly controlled through the finish/async model. This is important for simulations where a very high level of control over the completion of asynchronous tasks is necessary to guarantee deterministic results, as is the case in the PlhamJ financial market simulator.

The K-Means benchmark we used to demonstrate the performance of our dynamic load balancer could be programmed using the Map-Reduce model of Hadoop. At its core, Hadoop involves over-decomposing a problem in a set of independent tasks which can then be scheduled on a computation cluster. Some work has shown that Habanero-Java combined with Hadoop can be more efficient both in terms of memory consumption and execution time by taking advantage of multithreading [15]. The target for our parallel & distributed collection with integrated load balancer is different as we focus on a more fine-grained level of parallelism.

A variety of work aiming at simplifying the design of parallel and distributed programs exist, either in the from of libraries or supplementary compiler directives [16,17,18]. Most of them revolve around supporting large numerical computations on distributed arrays without topics related to dynamic load balancing. The target of our work differs in that we adopt an object-oriented programming model, with instances of various classes contained within the same collection.

7 | CONCLUSION

In this article we presented the load balancer integrated into our Java-APGAS 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. We showed that our integrated load balancer can be beneficial in some situations. However, further improvements to the inter-host load balancing scheme are needed to satisfactorily handle all situations.

Recent work by Posner and Fohry with the APGAS runtime [19] made it possible to dynamically start and stop processes. This is of particular interest to us as PlhamJ could benefit from the capability to shrink and grow the number of processes it is running on as the workload of the simulation dynamically evolves over time. Our integrated load balancer should also be capable of re-balancing agents between processes following such a reconfiguration.

ACKNOWLEDGEMENT

This work was supported by the JSPS KAKENHI Grants Number JP20K11841 and JP18H03232. This work used computational resources of supercomputer Fugaku provided by the RIKEN Center for Computational Science through the HPCI System Research Project (Project ID: hp220190 and hp220334).

DATA AVAILABILITY STATEMENT

Experimental data is available upon simple request to the corresponding author.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

References

1. Finnerty P, Kawanishi Y, Kamada T, Ohta C. Supercharging the APGAS Programming Model with Relocatable Distributed Collections. *Scientific Programming* 2022; 2022(1058-9244). doi: 10.1155/2022/5092422

2. Tardieu O. The APGAS Library: Resilient Parallel and Distributed Programming in Java 8. In: X10 2015. ACM. ACM; 2015; New York, NY, USA: 25–26

3. Zhang W, Tardieu O, Grove D, et al. GLB: Lifeline-Based Global Load Balancing Library in X10. In: PPAA '14. Association for Computing Machinery; 2014; New York, NY, USA: 31-40

4. Finnerty P, Kamada T, Ohta C. Integrating a Global Load Balancer to an APGAS Distributed Collections Library. In: PMAM '22. Association for Computing Machinery; 2022; New York, NY, USA: 55–64

5. Tardieu O, Herta B, Cunningham D, et al. X10 and APGAS at Petascale. In: PPoPP '14. ACM; 2014; New York, NY, USA: 53–66

6. Saraswat VA, Kambadur P, Kodali S, Grove D, Krishnamoorthy S. Lifeline-Based Global Load Balancing. In: PPoPP '11. Association for Computing Machinery; 2011; New York, NY, USA: 201–212

7. Yamashita K, Kamada T. Introducing a Multithread and Multistage Mechanism for the Global Load Balancing Library of X10. *Journal of Information Processing* 2016; 24(2): 416-424. doi: 10.2197/ipsjjip.24.416

8. Finnerty P, Kamada T, Ohta C. Self-Adjusting Task Granularity for Global Load Balancer Library on Clusters of Many-Core Processors. In: PMAM '20. Association for Computing Machinery; 2020; New York, NY, USA

9. Vega-Gisbert O, Roman JE, Squyres JM. Design and implementation of Java bindings in Open MPI. *Parallel Computing* 2016; 59: 1–20. Theory and Practice of Irregular Applicationsdoi: 10.1016/j.parco.2016.08.004

10. Torii T, Kamada T, Izumi K, Yamada K. Platform Design for Large-Scale Artificial Market Simulation and Preliminary Evaluation on the K computer. *Artificial Life and Robotics* 2017; 22(3): 301-307. doi: 10.1007/s10015-017-0368-z

11. Finnerty P, Kamada T, Ohta C. A self-adjusting task granularity mechanism for the Java lifeline-based global load balancer library on many-core clusters. *Concurrency and Computation: Practice and Experience* 2021; 34(2): e6224. doi: https://doi.org/10.1002/cpe.6224

12. Murase Y, Uchitane T, Ito N. An open-source job management framework for parameter-space exploration: OACIS. *Journal of Physics: Conference Series* 2017; 921: 012001. doi: 10.1088/1742-6596/921/1/012001

13. Chamberlain B, Callahan D, Zima H. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 2007; 21(3): 291-312. doi: 10.1177/1094342007078442

14. Acun B, Gupta A, Jain N, et al. Parallel Programming with Migratable Objects: Charm++ in Practice. In: SC '14. IEEE Press; 2014; New York, NY, USA: 647–658

15. Zhang Y. HJ-Hadoop: An Optimized Mapreduce Runtime for Multi-Core Systems. In: SPLASH '13. Association for Computing Machinery; 2013; New York, NY, USA: 111–112

16. Carter Edwards H, Trott CR, Sunderland D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 2014; 74(12): 3202-3216. doi: https://doi.org/10.1016/j.jpdc.2014.07.003

17. Nakao M, Lee J, Boku T, Sato M. XcalableMP Implementation and Performance of NAS Parallel Benchmarks. In: PGAS '10. Association for Computing Machinery; 2010; New York, NY, USA

18. Conejero J, Corella S, Badia RM, Labarta J. Task-based programming in COMPSs to converge from HPC to big data. *The International Journal of High Performance Computing Applications* 2018; 32(1): 45-60. doi: 10.1177/1094342017701278

19. Posner J, Fohry C. *Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing*; New York, NY, USA: Association for Computing Machinery . 2021.

How to cite this article: Finnerty P., Kamada T., and Ohta C. (2023), A Better Integrated Global Load Balancer for the APGAS Distributed Collections Library, *CCPE*, *2022;00:42–42*.