Integrating a global load balancer to an APGAS distributed collections library

Finnerty, Patrick

Kamada, Tomio

Ohta, Chikara

Integrating a Global Load Balancer to an APGAS Distributed Collections Library

Patrick Finnerty
Graduate School of System Informatics
Kobe University
Kobe, Japan
finnerty.patrick@fine.cs.kobe-u.ac.jp

Tomio Kamada
Graduate School of System Informatics
Kobe University
Kobe, Japan
kamada@fine.cs.kobe-u.ac.jp

Chikara Ohta
Graduate School of Science Technology and Innovation
Kobe University
Kobe, Japan
ohta@port.kobe-u.ac.jp

Abstract

In this paper, we introduce the global load balancer integrated into our distributed collections library for APGAS for Java model. Inspired by the lifeline Global Load Balancer scheme, our load balancer makes it possible for programmers to perform actions on the objects recorded into a distributed collection while allowing the library to relocate some entries between hosts if load imbalances occur. The programming model we adopt introduces minimal impact on the legibility of programs, with the regions of the program where entries of a distributed collection may be relocated at the library's initiative are clearly identified. Internally, our integrated global load balancer implements an hybrid scheme which balancer the load both between the threads on a host and between the hosts participating in the computation. It allows for multiple concurrent computations on multiple collections with individual termination detection. We evaluate the performance of our integrated Global Load Balancer and its ability to handle various situations on a many-core supercomputer.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; *Concurrent algorithms*; Distributed algorithms.

Keywords: distributed collection, load balancing, distributed computation

ACM Reference Format:
Patrick Finnerty, Tomio Kamada, and Chikara Ohta. 2021. Integrating a Global Load Balancer to an APGAS Distributed Collections Library. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 a distributed collections library to supplement the APGAS for Java [12] library. 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.

However, this poses a challenge for programmers as the performance and computing resources available on each host will 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 themselves to redistributed the load between processes, this comes as a significant burden and can significantly obfuscate the program. Instead, we believe such analysis and load-balancing measures should be taken by the library itself.

What we would like to achieve is to detect such cases where load imbalance appears during the distributed computation and take measures so that entries are dynamically relocated from overloaded processes to under-loaded ones.

In this article, we claim the implementation of an Global Load Balancer integrated into our distributed collection library. Inspired by the lifeline-based Global Load Balancer scheme first implemented in X10 [11], our integrated load-balancer is capable of automatically relocating work along with the entries of distributed collection to maintain performance in an environment where the performance of hosts evolves over time. The programming model we propose makes it easy for users to recognize which parts of the distributed program are conducted under this load balancer

The remainder of this article is organized in the following sections. We start by recalling some useful background in Section 2. In Section 3, we present the main contribution of this article, detailing the abstractions provided to the programmer and how load balanced parts integrate in a distributed program written with the help of our library. In Section 4, we discuss the internal implementation, the termination detection scheme, as well as the progress tracking system we developed. We present the results of our evaluation in Section 5 and open a broader discussion in Section 6. We discuss related work in Section 7 before concluding in Section 8.

2 Background

In this section, we recall some work useful for the good comprehension of our contribution.

2.1 Distributed Collection Library for APGAS

We focus on the Asynchronous Partitioned Global Address Space (APGAS) model introduced in X10 [13]. 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 the dedicated keywords `async` and `at`. Termination detection is implemented through an elegant construct called `finish` which waits until all transitively spawned asynchronous activities have completed. This model was later ported to Java [12], with the X10 keywords being converted to static methods taking lambda-expressions as parameter.

To further ease the burden of programmers, we created a Java distributed collection library for APGAS. While the details of this library and the features is supports fall outside the scope of this paper, we recall here its main characteristics with the help of a short example.

Conference'17, July 2017, Washington, DC, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
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 1. Distributed map creation and record insertion

In this library, we provide a number of distributed collections (distributed map `DistMap`, distributed array `DistChunkedList`) 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 and read entries into the local handle of a distributed collection by using the usual AGPAS asynchronous activities. As per the usual APGAS semantics, asynchronous activities interact only with the contents of the collection held by the handle they are operating on.

The most significant innovation of our distributed collections library is that it allows programmers to relocate entries of a collection between its local handles. 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 derivatives.

The program presented in Listing 1 and Figure 1 illustrate these characteristics. The `TeamedPlaceGroup` object obtained on the first line of Listing 1 represents the group of all APGAS places. It is used on the second line to create the distributed map `dMap`, specifying that this distributed collection will have a handle on every process participating in the execution. One 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.

From lines 4 to 11, the short-hand `broadcastFlat(Runnable)` is used to spawn the same asynchronous activity on all hosts participating in the computation. The `broadcastFlat` method will return when the asynchronous activity running on each host has completed. 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 `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 decides to relocate its `main:running` entry to place 1. The transfer is actually 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 the `main:running` entry has been deleted from the handle on Place 0 and inserted into the handle of Place 1.

Programmers can also invoke methods on every element recorded in the collection, irrespective of the process on which they are located through a specific member, i.e.

```
myDistCol.GLOBAL.forEach(e->{/*action on e*/});
```

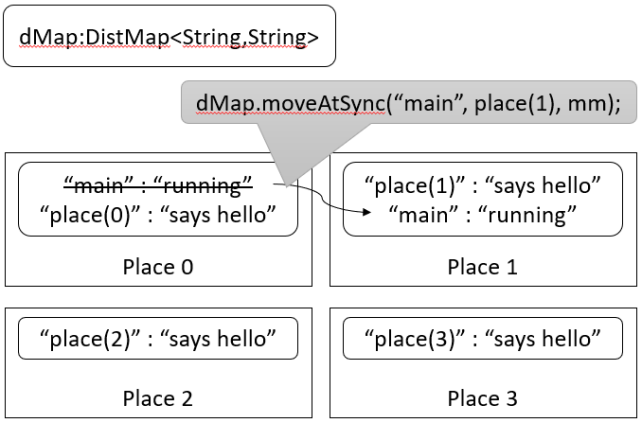


Figure 1. State of the distributed map dMap after the Listing 1 program has run with 4 processes

with the action to perform on each element of the collection specified as a lambda expression (`e->{/*action on e*/}`). Our library applies this lambda expression on each element "e" contained in the distributed collection. The method `forEach` returns when the provided lambda expression has been applied on every element located of every Place on which the collection is defined. Parallel variants of such `GLOBAL` methods are also provided by our library, i.e. `parallelForEach`, in which case the provided lambda expression is applied by multiple threads on each process participating in the computation.

In addition to `GLOBAL` methods, some computation patterns require some communication between hosts. We call such methods "teamed". This is the case for instance of *reduction* computation. Our library provides the facilities for programmers to implement user-defined reductions on objects contained by our distributed collections. In the case of a "teamed" reduction, the computation will take place in two phases: (1) the "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. Internally, specific communication patterns involving multiple processes are supported by MPI through the MPJ-Express library [2] while matters related to termination detection are handled by the APGAS `async/finish` constructs.

2.2 Lifeline-based Global Load Balancer

The lifeline-based Global Load Balancer is a work-stealing scheme first implemented in X10 [11, 16]. 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-implemented tasks queues. The main worker process consists in processing a certain number of these tasks and then answering received steal requests by giving away some (generally half) of its tasks to the thief.

When a node runs out of work and fails to steal some from a randomly selected host, it signals itself as "dormant" to its lifeline neighbors and remains idle until one of its lifeline neighbors sends him some work using an asynchronous activity.

This scheme elegantly resolves 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, 15]. 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.

3 Programming Model

The main contribution of this article consists in the introduction of a new scheme which operates on all the elements of the distributed collection. These functionalities are accessible to programmers through a specific `GLB` member of the distributed collection. Contrary to previously introduced `GLBOAL` methods, this allows the library to relocate entries of the underlying distributed collection from a handle to another as it detects load imbalances.

In this section, we introduce main contribution of this article in Section 3.1. We then detail how these new features are used in our applications in Sections 3.2 and 3.3. Implementation-related topics are discussed in Section 4.

All the source code of our distributed collections library[1] and our applications are publicly available[2].

3.1 GLB semantics

Load-balanced computations on our distributed collections are accessible through a special `GLB` handle. Analogous to the `GLOBAL` member which provides control over the entirety of a distributed collection from a single place, the methods proposed through this special handle act on the entire distributed collection. It also allows the library to relocate entries between handles if it notices some load imbalance during the computation.

Currently, only our distributed array collection `DistChunkedList` and its derivatives are fitted with this feature. A summary of the computations supported is shown in Table 1.

Inside an APGAS program written with our library, `GLB`-type methods can only be called inside a specifically designed `underGLB` method which takes a closure as parameter as shown in Figure 2. 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 block has the added benefit of defining a clear boundary for the programmer within which entries of the distributed collections manipulated inside this block may be relocated freely by the library.

Inside the `underGLB` block, the computation does not start immediately but is internally staged, with an instance of `DistFuture` 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 collection being computed at the same time. The load-balanced computation will start when one of three following cases is encountered:

1. the result of a GLB computation is called through method `DistFuture.result()`
2. the static method `GlobalLoadBalancer.start()` is called
3. the end of the `underGLB` block is reached

In all cases, every `GLB` computation "staged" up until that point is started. A short example illustrating each of these cases is presented in Listing 2.

The first case presents itself on line 9. Up until that point, only the `map` operation on collection `eleCol` was staged inside the GLB block. This single GLB computation is started. The call to `fut1.result()` of line 9 blocks until this computation completes and a newly created `DistCol<Integer>` collection resulting of the computation is returned.

On line 13, we encounter the second case. Here, both the `forEach` and the `reduce` operations staged on lines 12 and 13 are started. Method `start` is non-blocking and progress inside the GLB block

[1] https://github.com/handist/collections
[2] https://github.com/plham/plhamJ

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 DistCol 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

```
1  import static handist.collections.glb.GlobalLoadBalancer
       .*;
2
3  public static void main(String[] args) {
4    DistCol<Ele> eleCol = new DistCol<>();
5    // Population and distribution omitted
6    underGLB(()->{
7      DistFuture<DistCol<Integer>> fut1 = eleCol.GLB.map(
8        e->{return e.makeInt()});
9      DistCol<Integer> intCol = fut1.result(); // Case 1
10
11     DistFuture fut2 = eleCol.GLB.forEach(e->e.update());
12     DistFuture fut3 = intCol.GLB.reduce(new Average());
13     start(); // available through static import, Case 2
14
15     DistFuture<DistCol<Integer>> fut4 = eleCol.GLB.toBag(
         e->{return e.makeInt()});
16     // End of block, Case 3
17   });
18 }
```

Listing 2. Program with a part operating under dynamic load balance

```
1  DistChunkedList<Point> points;    // init omitted
2  double[][] initialClusterCenter; // chosen at random
3
4  world.broadcastFlat(() -> {
5    double[][] clusterCentroids = initialClusterCenter;
6    for (int iter = 0; iter < repetitions; iter++) {
7      final double[][] centroids = clusterCentroids;
8      // Assign each point to a cluster
9      points.parallelForEach(p -> p.assignCluster(centroids
       ));
10
11
12     // Calculate the average position of each cluster
13     final AveragePosition avgClusterPosition = points.
       team().parallelReduce(new AveragePosition(K,
       DIMENSION));
14
15     // Calculate the new centroid of each cluster
16     final ClosestPoint closestPoint = points.team().
       parallelReduce(new ClosestPoint(K, DIMENSION,
       avgClusterPosition.clusterCenters));
17
18     clusterCentroids = closestPoint.
       closestPointCoordinates;
19   }
20 });
```

Listing 3. K-Means non-GLB implementation

```
1  DistChunkedList<Point> points;    // init omitted
2  double[][] initialClusterCenter; // chosen at random
3
4  GlobalLoadBalancer.underGLB(() -> {
5    double[][] clusterCentroids = initialClusterCenter;
6    for (int iter = 0; iter < repetitions; iter++) {
7      final double[][] centroids = clusterCentroids;
8      // Assign each point to a cluster
9      points.GLB.forEach(p -> p.assignCluster(centroids)).
       result();
10
11     // Calculate the average position of each cluster
12     final AveragePosition avgClusterPosition = points.GLB
       .reduce(new AveragePosition(k, dimension))
13     .result();
14
15     // Calculate the new centroid of each cluster
16     final ClosestPoint closestPoint = points.GLB.reduce(
       new ClosestPoint(k, dimension, avgClusterPosition.
       clusterCenters)).result();
17     clusterCentroids = closestPoint.
       closestPointCoordinates;
18   }
19 });
```

Listing 4. K-Means GLB implementation

continues. This allows programmers to perform some other computation while the GLB operations are ongoing. If it is later needed to wait on the completion of a launched GLB computation, this can be done by calling `fut2.result()` or `fut3.result()`.

Finally, in the third and last case consisting of reaching the end of the GLB block, the `toBag` operation staged at line 15 is launched. In this case, the start of this last operation will be triggered on the library side. The `underGLB` method will return when all ongoing operations have completed. In the example shown in Listing 2, the potentially ongoing GLB computations are the `forEach`, the `reduce`, and the `toBag` computations of lines 11, 12, and 15. The `map` operation of line 7 has already completed due to the `fut1.result()` call on line 9.

3.2 K-Means

In this section we detail an actual program written with the GLB features of our distributed collections library in the form of the K-Means program.

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 him. 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.

With our implementation of the K-Means algorithm, each point is recorded into an instance of a Point class. 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 3 and 4.

You can notice that the program 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 (i.e. `points.reduce(...)`, not used here) and reductions taking place between all places in the cluster (i.e. `points.team().reduce(...)`) as is used in lines 11 and 15 of Listing 4. The instances of classes `AveragePosition` and `ClosestPoint` given as parameter to the `reduce` methods are user-defined classes which extend a reduction abstraction provided by our library.

In this application, we need to make sure that the previous step in the iteration has completed before starting the next step. Therefore, we use the blocking `result()` method immediately after staging the computation with methods `points.GLB.forEach` on line 8 and `points.GLB.reduce` on lines 11 and 15.

3.3 PlhamJ

PlhamJ is the Java version of the Plham financial market simulator first written in X10 [14]. 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 library then runs the simulation and provides deterministic results following a given seed.

Internally, there are multiple "runners" that can run the simulation. In this study, we focus on two distributed runner implementations: a manually load-balanced version, referred to as "manual" thereinafter, and the GLB version. We first quickly describe how simulations are run in general before outlining the differences between these two runner implementations.

A Plham simulation iteration consists of the following basic steps:

1. *market update*: the latest state of the markets is broadcast to the agents in the simulation
2. *order submission*: the agent place their orders on the markets

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

Listing 5. Order submission of non-GLB program

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

Listing 6. Order-submission of GLB program

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 (called the master) 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 the Trades contracted information, the details of which fall outside the scope of this article.

The difference between the "manual" and the "glb" runner lies in the order submission step shown in Listings 5 and 6. This step is an example of a "parallel producer/receiver" pattern where each Agent returns the orders it wants to submit in a List containing from 0 to several orders. If orders are returned by the Agent's `submitOrders` method, they are recorded in the `orderBag` distributed collection. This is an instance of class `DistBag<T>` which is specifically designed to accept many "T" objects coming from multiple threads concurrently. It does so by supplying a dedicated `Consumer<T>` handle to each thread. In this present case, the generic type `T` resolves to a `List<Order>`, hence the rather lengthy type of parameter `orderCollector` which appears on line 5 in both Listing 5 and 6.

The "manual" runner uses a parallel construct shown in Listing 5. Internally, this allocates an even number of Agents to each thread available on the host. This version measures the computation time needed to perform this step on each process. If disparities appear, agents are relocated between processes. We call this version because this form of load-balancing requires "manual" intervention in the program.

By opposition, the "glb" version relies on the integrated GLB `toBag` operation, as shown in Listing6. This allows agents to be relocated while the agent order submission takes place. No explicit load-balancing measures are taken from within the PlhamJ GLB runner, it is entirely left to the library.

4 Implementation

The load-balancing scheme we have currently implemented is inspired by the lifeline-based global load balancer of X10 whose key principled we recalled in Section 2.2. While some key concepts are re-used as is, there are a number of key differences between the original implementation and what we use for our specific context.

We rely on the same general global termination detection mechanism in our integrated global load balancer, with one enclosing `finish` per operation submitted. However this brings about some modification of the original scheme and some extensions of the APGAS for Java library.

In this section, we discuss into further details select implementation topics.

4.1 Progress tracking with Assignment

As laid out in Section 1, we need to accurately track the progress of each operation so that when relocating work is necessary, instances with some computation left in them are transferred from busy nodes to idle nodes. This is done through what we call an "Assignment". An assignment represents a part of the underlying distributed collection and the progress of the various operation being performed on that subset of the collection.

Currently, we have only implemented the `Assignment` class for the distributed array collection. In this case, the part of the collection are designated using a pair of `long` integers that designate a range $[a, b)$ of indices in the array. The progress of each operation is tracked using a `long` integer whose value evolves from a to b as the operation progresses through the range designated by the assignment.

Without an assignment, a worker is not allowed to access any distributed collection. Only with an assignment is a worker 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 dedicated to a collection located on each process is tracked throughout the computation using an atomic counter. The number of assignments left to complete for each operation is also tracked using an atomic counter. When a worker completes an operation for an assignment, it decrements the corresponding operation counter. When the counter reaches 0 (meaning the worker completed the last remaining assignment for this computation), it unblocks the *witness activity* which marking the presence of work for, allowing it to terminate. The nature of this "witness thread" 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. These assignments are kept in a single reserve of assignments on each host (as opposed to 2 in the multithreaded GLB [15]), and sorted into three queues depending on their status:

- available, i.e. containing work and available for a worker to take it
- in computation, i.e. containing work and currently given to a worker
- completed

In the original GLB scheme, completed tasks can generally be discarded. In our integrated load balancer this is not the case as assignments are re-used when successive computations are staged and launched on the same underlying distributed collection. We therefore need to keep an exact record of every assignment in the system.

As part of their main routine, worker threads start by obtaining an assignment from the "available" queue. They progress the computation contained within this assignment by a fixed number of objects, the so-called "grain". When a worker completes the assignment it received, it is placed back into the "completed" queue.

When the "available" 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 instances targeting contiguous ranges. In the process, they update the number of assignments for the underlying collection and the number of assignments left to complete for each operation tracked by the assignment.

4.3 De-coupling of worker threads and computation

In the original lifeline-based scheme, all the workers are asynchronous activities managed by the same enclosed finish. In our case, since we may have multiple operations on possibly multiple collections ongoing at the same time, we decided 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.

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 initiate the inter-host work stealing before returning.

When work is received from a remote host, a new witness activity will be created and remain 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 into 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 between our scheme and 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 gives us a new perspective on lifelines in the context of our integrated load balancer. In the original load-balancing scheme, 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 initially present on all hosts where entries of the distributed collection are present, this is not a concern anymore. Using a non-connected lifeline network will guarantee that the entries of the distributed collection remain located with the subset of hosts connected by lifelines. In the PlhamJ "GLB runner" discussed in Section 3.3, we rely on this property to ensure that no agents are relocated on the master host.

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. The reason for this is detailed in the following paragraphs.

To preserve the integrity of the global termination detection, the asynchronous activity used to answer the steal request needs to spawn a new "witness" activity on the thief, as mentioned in the preceding section. However it is possible that the transferred assignments contain work from multiple ongoing operations. In this 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 [12] library to allow any thread to spawn an activity registered into multiple arbitrary `finish`. 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 an other 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 an 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 case does not present itself in our situation.

This extension of the APGAS library allows us to somewhat simplify the load balancing scheme compared as worker threads (which are not registered into any finish) can now directly answer thieves as part of their main routine by spawning the appropriate asynchronous activity using our extended APGAS construct.

4.5 Restrictions imposed by the integrated load balancer

There are a number of conditions that need to be observed for programs to run successfully with our integrated global load balancer. First, every range contained in the distributed array can only be recorded into a single place. In other words, no two handles of a distributed collection can contain ranges that overlap. While this is in general possible, it is not compatible with our integrated load balancer as ranges relocated as part of inter-host load balancing may clash.

Table 2. Hardware and Software environment on the OakForest-PACS supercomputer

Property	Value
Processor	Intel Xeon Phi 7250 (68 cores)
RAM	96GB DDR4
Java version	Open JDK 1.8.0_222
MPI version	Intel MPI with MPJ-Express v0_44 Java native bindings

Table 3. Program parameters used for K-Means

parameter	value
k	2000
nb points	10m per host (weak scaling)
point dimension	5

Secondly, no new entries can be recorded or removed from the distributed array while a computation is ongoing. More specifically, any new range added into a local handle of the distributed array would be ignored by any ongoing computation as it will miss 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.

If entries need to be added or removed from the collection between parts run under GLB, it should be done outside of `underGLB` blocks. Subsequent `underGLB` methods will take into account the changes and re-generate the assignments based on the contents of the collection at that time.

5 Preliminary Evaluation

To evaluate the capabilities of our integrated global load balancer, we evaluate its performance on 2 distributed and parallel applications, *K-Means* and *PlhamJ*.

For both of these applications, we prepared two versions using our distributed collections library: one which relies on the integrated GLB presented in this article, and one which does not. In the non-glb version of our programs, the entries of the distributed collection contained on each host are split into even amounts and given to each available thread on the process in the form of an iterator. Each thread processes the entries it was assigned with no load-balance taking place either between processes or between threads within a process.

We perform our evaluation on the OakForest-PACS supercomputer. The details of the hardware and software environment used is summarized in Table 2. Evaluation with the K-Means program are conducted in weak scaling following the parameters outlined in Table 3. For Plham, we prepared a special simulation in which agents are assigned a certain amount of artificial load when submitting their orders. More detail about the configuration is given in Table 4. The large number of executions necessary for this study were managed using OACIS [8].

5.1 Overhead

First, we want to estimate the amount of overhead created by our load-balancing scheme in situations where no load-balance measures are necessary. We do this in both of our applications by using an uniform distribution and comparing the glb version of our program against the non-glb version and the glb version stripped of its inter-process load balancing. In this situation there is a priori no need for inter-host load balance measures. This experiment allows us to evaluate how much overhead the lifeline scheme contributes to the system. The results are presented in Figure 2 and 3.

There are a number of unexpected results here. First, it appears that the "non-glb" version of our K-Means program is slower due to its implementation relying on iterators. Compared to the "glb" version of the program, it delivers execution times up to 50% longer depending on the number of hosts used. We also obtain a similar situation on PlhamJ when running on 3 hosts. The "manual" version of the program which relies on the same iterators shows execution times about 8% longer than the "glb" versions. But on executions

Table 4. Program parameters used for the PlhamJ program

parameter	value
number of agents	20 thousand
artificial load	7500
number of iterations	300

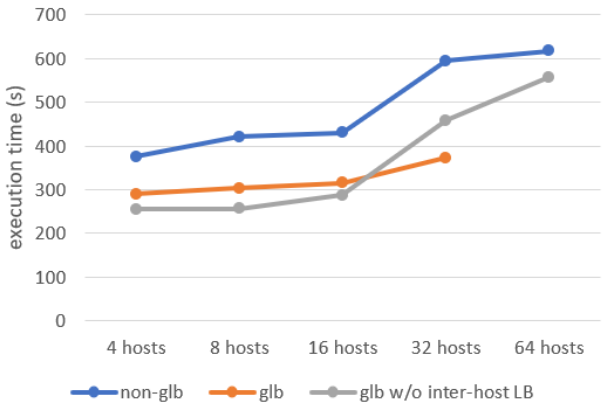


Figure 2. Weak Scaling evaluation of the distributed K-Means on up to 64 nodes of the OakForest-PACS supercomputer

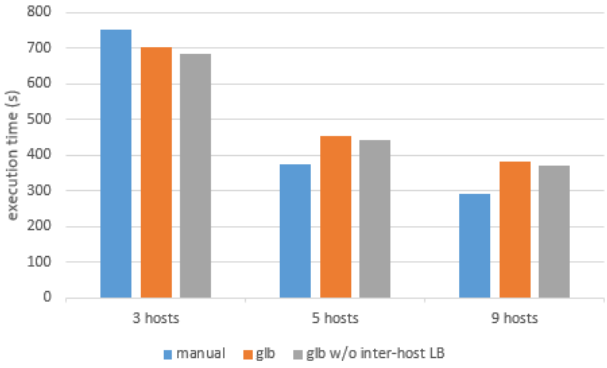


Figure 3. Strong scaling evaluation of the PlhamJ simulator on up to 9 nodes of the OakForest-PACS supercomputer

that use 5 and 9 hosts, this is no longer the case. Part of this performance gap may be linked to the number of chunks and the size of individual chunks the distributed array is split into on each host. Another cause could be the more heavy use of lambda-expressions in the "non-glb" version. Further investigation is needed.

If we focus on the two "glb" and "glb without inter-host" programs, the overhead attributable to the inter-host load balancing mechanism appears to induce execution times between 10 and 15% longer on K-Means. With PlhamJ, the difference in overall execution time is much smaller, only about 3%. However part of the relative difference is absorbed by the other computation steps of the program. When accounting for the other parts of the program, the overhead due to the inter-host load-balancing meansures accounts for an increase in computation time on the agent submission part of about 5%.

These results are encouraging, but they need to be taken with some precautions as we were unable to successfully execute the "glb" version of the K-Means program with larger number of nodes. It appears that our implementation suffers from a programming error which tends to manifest itself on larger clusters.

5.2 With a competing computation

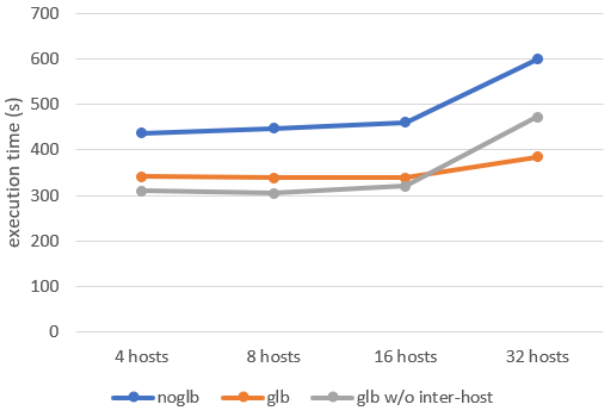
In this second experiment we want to check how our integrated GLB reacts to dynamic changes in performance on the hosts on which it is running. We use the same programs as in the previous section, but introduce a second process which steals some computing power away from the K-Means or PlhamJ program. This is meant to replicate a second computation taking place at the same time as the GLB program. 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.

We do this by introducing 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 a an initial seed to be able to replicate the same disturbance across multiple executions.

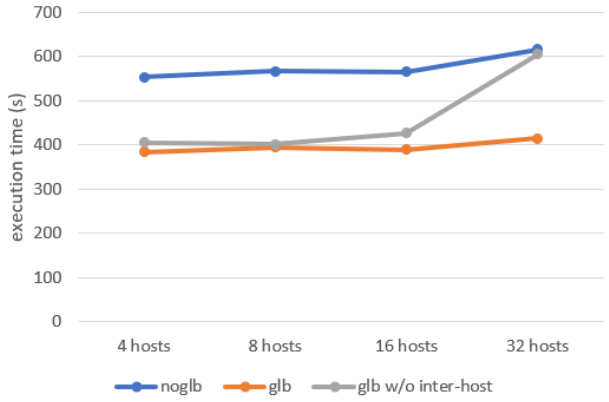
We use two levels of disturbance: 20 and 40 threads, which correspond to about 30 and 60% of the available parallelism on a node of the OakForest-PACS supercomputer. The results are presented in Figures 4 and 5 for K-Means and PlhamJ respectively.

On K-Means, the performance issue of our "no-glb" implementation that we noted in the previous section is present again. With 20 disturb threads, the performance gap between the "glb" and the "glb w/o inter-host load balancing" is reduced, but not enough for the "glb" version to show better performance. With 40 disturbing thread however, this is the case, with our "glb" program showing execution times between 2 to 5 We are not certain why the "glb" version suddenly shows much better performance on 32 hosts. It could be that the larger number of hosts increases the number of thieves capable offloading the process being disturbed.

On PlhamJ the results are substantially the same with both 20 and 40 disturbing threads. When running with 3 hosts, the "glb" version of the program shows the better performance. With larger



(a) with 20 disturb threads



(b) with 40 disturb threads

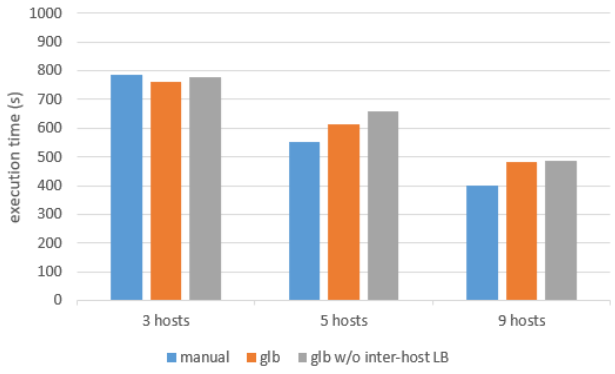
Figure 4. Execution time of the K-Means program with a competing Disturb program

clusters, the "glb" still shows better performance than the glb version deprived from its inter-host load balancing capabilities. However, the best-performing implementation on 5 and 9-hosts clusters is the manually load-balanced version. We think that this is due to our GLB mechanism being too gradual in its changes in distribution whereas the manually load-balanced version will relocate as many instances as it deems necessary based on computation time of each host over the last 5 iterations.

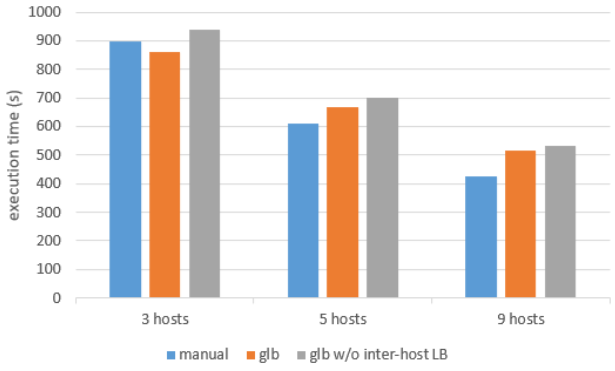
In all experiments up until that point, we have been using the "hypercube"[11] lifeline strategy for our "glb" programs. However, the present results suggests that a lifeline graph with a higher degree may yield better results.

6 Discussion

Performance evaluation. We made a number of arbitrary decisions in the design of our integrated distributed collection library. One of those was only relocate data that still has some work inside of it. Being that the applications we presented are iterative, it would still make sense to relocate entries that have already being computed in prevision of the next iteration. This may allow our GLB mechanism to react more quickly to load imbalances. We could also chose to initiate inter-host work-stealing before all the work



(a) with 20 disturb threads



(b) with 40 disturb threads

Figure 5. Execution time of the PlhamJ simulation with a competing Disturb program

from a host has disappeared. This raises the question on the victim host of whether or not to answer these "early" steal requests, but can be resolved. Both of these ideas could be implemented without modifying the termination detection scheme.

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 each one: 5 and 500 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 [6, 7].

Finally, it is clear from this preliminary evaluation that the iterator-based implementation our "non-glb" K-Means program "manually" load-balanced PlhamJ runner rely on is not efficient. This has no effect on the performance of the integrated GLB we presented in this article but we will work to resolve this clear performance issue.

Future enhancements. Currently, only the variants of our arbitrary index array distributed collection features supports load-balanced operations. The challenge in porting the same features to our other distributed collection lies in the progress tracking. In the case of our distributed arrays, we can easily track the progress of operations with pairs of `long` integers that describe a range of indices in the array. For distributed maps that may use any user-specified object as key, there is no such 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 is not be sufficient, as when work is received from remote host, inserted keys may land inside the range of an existing assignment. For these reasons, it appears to us that a hypothetical implementation of the `Assignment` class for a distributed map would have to rely on the internal representation of the map, a topic we have not explored yet.

The PlhamJ schedule we presented in Section 3.3 is not optimal for distributed computation. We are currently working on a pipelined schedule in which the order processing takes place while the agents are computing their orders for the next iteration based on slightly older market information (an acceptable compromise). This would make the program more efficient by avoiding to keep the "agent processes" idle while the order-handling is taking place.

7 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 [4]. 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.

Our closest competitor is Charm++ [1]. In terms of load-balancing capabilities, Charm++ 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. However, this surrenders all the distribution control to the Charm++ runtime. In applications with more intractable communication patterns and completion dependencies such as PlhamJ where computation and relocation of some objects may overlap, the completion and quiescence detection provided by Charm++ would make it possible to implement but with greater effort than the programming model we propose. 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.

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. As 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 [17]. However, the target for our parallel & distributed collection with integrated load balance is different. We focus on a more fine-grained level of parallelism than Hadoop, with programs that present more intractable communication patterns.

A variety of works aiming at simplifying the design of parallel and distributed programs exist, either in the from of libraries of supplementary compiler directives [3, 5, 9]. However, most of them revolve around support large numerical computations on

distributed arrays without topics related The target of our work is different in that we adopt an object-oriented programming model, with instances of various classes contained within the same collection (i.e. various `Agent` implementations in the case of Plham.)

Recent work by Posner and Fohry with the APGAS runtime [10] made it possible to dynamically start and stop processes of an APGAS runtime. This is of particular interest to us as the "pipelined" schedule for PlhamJ we mentioned in Section 6 could greatly 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.

8 Conclusion

In this article we presented the global 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. While our current scheme allowed us to gain an advantage against statically distributed programs in some dynamic situations, further analysis is needed to further comprehend the conditions that need to be met to implement an efficient scheme both when load-balance measure are and are not needed.

Acknowledgments

This work was supported by JSPS KAKENHI Grants Number JP20K11841, JP18H03232.

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 647–658. https://doi.org/10.1109/SC.2014.58

[2] Mark Baker and Bryan Carpenter. 2000. MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing. In *Parallel and Distributed Processing*, José Rolim (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 552–559.

[3] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[4] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442 arXiv:https://doi.org/10.1177/1094342007078442

[5] Javier Conejero, Sandra Corella, Rosa M Badia, and Jesus Labarta. 2018. Task-based programming in COMPSs to converge from HPC to big data. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 45–60. https://doi.org/10.1177/1094342017701278 arXiv:https://doi.org/10.1177/1094342017701278

[6] Patrick Finnerty, Tomio Kamada, and Chikara Ohta. [n.d.]. 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* n/a, n/a ([n. d.]), e6224. https://doi.org/10.1002/cpe.6224 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6224

[7] 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* (San Diego, California) *(PMAM '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3380536.3380539

[8] Y. Murase, T. Uchitane, and N. Ito. 2017. An open-source job management framework for parameter-space exploration: OACIS. *Journal of Physics: Conference Series* 921 (nov 2017), 012001. https://doi.org/10.1088/1742-6596/921/1/012001

[9] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. 2010. XcalableMP Implementation and Performance of NAS Parallel Benchmarks. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (New York, New York, USA) *(PGAS '10)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/2020373.2020384

[10] Jonas Posner and Claudia Fohry. 2021. *Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3458744.3473361

[11] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-Based Global Load Balancing. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1941553.1941582

[12] Olivier Tardieu. 2015. The APGAS Library: Resilient Parallel and Distributed Programming in Java 8. In *Proceedings of the ACM SIGPLAN Workshop on X10* (Portland, OR, USA) *(X10 2015)*. ACM, New York, NY, USA, 25–26. https://doi.org/10.1145/2771774.2771780

[13] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. 2014. X10 and APGAS at Petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2555243.2555245

[14] Takuma Torii, Tomio Kamada, Kiyoshi Izumi, and Kenta Yamada. 2017. Platform Design for Large-Scale Artificial Market Simulation and Preliminary Evaluation on the K computer. *Artif Life Robotics* 22, 3 (2017), 301–307. https://doi.org/0.1007/s10015-017-0368-z

[15] Kento Yamashita and Tomio Kamada. 2016. Introducing a Multithread and Multistage Mechanism for the Global Load Balancing Library of X10. *Journal of Information Processing* 24, 2 (2016), 416–424. https://doi.org/10.2197/ipsjjip.24.416

[16] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. 2014. GLB: Lifeline-Based Global Load Balancing Library in X10. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (Orlando, Florida, USA) *(PPAA '14)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/2567634.2567639

[17] Yunming Zhang. 2013. 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* (Indianapolis, Indiana, USA) *(SPLASH '13)*. Association for Computing Machinery, New York, NY, USA, 111–112. https://doi.org/10.1145/2508075.2514875