International Journal of Networking and Computing – `www.ijnc.org`, ISSN 2185-2847
Volume 8, Number 1, pages 2–31, January 2018

A Java Task Pool Framework providing Fault-Tolerant Global Load Balancing

Jonas Posner and Claudia Fohry

Research Group Programming Languages / Methodologies
University of Kassel, Germany
Email: {jonas.posner | fohry}@uni-kassel.de

Received: July 24, 2017
Revised: October 30, 2017
Accepted: November 29, 2017
Communicated by Akihiro Fujiwara

Abstract

Fault tolerance is gaining importance in parallel computing, especially on large clusters. Traditional approaches handle the issue on system-level. Application-level approaches are becoming increasingly popular, since they may be more efficient. This paper presents a fault-tolerant work stealing technique on application level, and describes its implementation in a generic reusable task pool framework for Java. When using this framework, programmers can focus on writing sequential code to solve their actual problem.

The framework is written in Java and utilizes the APGAS library for parallel programming. It implements a comparatively simple algorithm that relies on a resilient data structure for storing backups of local pools and other information. Our implementation uses Hazelcast's `IMap` for this purpose, which is an automatically distributed and fault-tolerant key-value store. The number of backup copies is configurable and determines how many simultaneous failures can be tolerated. Our algorithm is shown to be correct in the sense that failures are either tolerated and the computed result is the same as in non-failure case, or the program aborts with an error message.

Experiments were conducted with the UTS, NQueens and BC benchmarks on up to 144 workers. First, we compared the performance of our framework with that of a non-fault-tolerant variant during failure-free operation. Depending on the parameter settings, the overhead was at most 46.06%. The particular value tended to increase with the number of steals. Second, we compared our framework's performance with that of a related, but less flexible fault-tolerant X10 framework. Here, we did not observe a clear winner. Finally, we measured the overheads for restoring a failed worker and for raising the number of backup copies from one to six, and found both to be negligible.[1]

Keywords: fault tolerance, resiliency, load balancing, task pool, Java, APGAS, Hazelcast

[1]This paper is an extended version of Jonas Posner and Claudia Fohry: Fault Tolerance for Cooperative Lifeline-Based Global Load Balancing in Java with APGAS and Hazelcast. IEEE Int. Parallel and Distributed Processing Symposium Workshops (APDCM), 2017.

1 Introduction

Large-scale parallel applications are typically run on multiple nodes of a cluster. Since the probability of node failures increases with the cluster size, fault tolerance is of increasing importance. Traditionally, fault tolerance is achieved through system-level checkpoint/restart [18], but this technique introduces a high overhead.

Application-level fault tolerance techniques may be more efficient, but programmers are responsible for their implementation. To reduce the programming effort, it is beneficial to implement these techniques in reusable and generic software frameworks. Ideally, users of such frameworks need not take care of fault tolerance and other parallelization details, and can thus concentrate on writing code for their actual problem.

We consider a task-based approach, i.e., the application programmer specifies a large number of tasks that can be executed in parallel. The well-known task pool pattern maps these tasks to a fixed number of workers. We assume that the tasks are internally sequential. The computation of a task may give rise to other tasks, which are usually processed by the same worker. For that, each worker maintains a local pool in which it holds open tasks. Load balancing among workers is accomplished by work stealing, in which an idle worker requests tasks from a busy co-worker.

More specifically, we consider the lifeline variant of task pools. It has been implemented in the *GLB* [42] framework, which is part of the standard library of the parallel programming language X10.

X10 is a Partitioned Global Address Space (PGAS) language [35]. Accordingly, it assumes that there is a global address space, which is logically divided into partitions. By *place*, X10 denotes a memory partition together with some computational resources. Typically, a place corresponds to a cluster node. Each place can access every memory partition, but access to the local partition is faster. X10 is based on an asynchronous variant of PGAS. Therefore, any number of software threads, called *activities*, can be created dynamically.

A fault-tolerant variant of GLB has already been developed in previous work [13]. Its basic approach is writing snapshots of each place's local pool into the main memory of another place. These snapshots are updated in the event of stealing. The overall algorithm involves a sophisticated orchestration of several asynchronous protocols for backup writing, stealing, failure detection, restore, etc. Therefore, it is hard to understand, implement and verify. Moreover, the algorithm is restricted to use of a single backup copy per data item. Thus, a simultaneous crash of two or more places in an unfavorable constellation can not be tolerated. Overall, that fault-tolerant GLB variant has the following drawbacks:

- the fault tolerance algorithm is complex and hard to maintain,
- multiple unfavorably correlated place failures lead to program abort, and
- it is implemented in X10, which is not widely used.

The present paper introduces a new variant of GLB, which overcomes these drawbacks. Its underlying algorithm is much simpler. The key idea is reliance on a resilient data structure to hold backup copies, as well as reliance on automatic and system-wide failure notification. Like the X10 algorithm, the new one regularly saves snapshots of each place's local pool. Backups are written at fixed times, as well as during stealing and recovery. During stealing, the loot is saved as well. When a place crashes, all other places are notified and can take appropriate action. In particular, a designated *backup place* takes over the failed place's tasks.

Therefore, our approach can be classified as uncoordinated application-level checkpointing with a stealing-related message logging [10]. Unlike in typical application-level checkpointing schemes, we

- do not re-start the failed place, but distribute its work to the existing places,
- store the backups in-memory, and
- use a resilient data structure instead of a checkpointing library.

Moreover, the new algorithm has been implemented with Java. Java is one of the most popular high-level programming languages and also gains increasing attention in high-performance computing [7]. With Java, we are able to address more potential users for our new framework.

Java provides a rich set of synchronization constructs for concurrency control, but does not have an integrated mechanism for parallel programming with multiple JVMs (Java Virtual Machines). Therefore, we additionally deploy the APGAS library for Java [37]. This library has been developed by a member of the X10 team and brings the parallelism and distribution features of X10 to Java.

Non-fault-tolerant versions of GLB for Java and APGAS have already been developed in own previous work [31]. This paper builds on the cooperative variant from [31], which achieves good speedups. In the rest of this paper, we denote the GLB variants as follows:

- **X_GLB**: original non-fault-tolerant X10 variant, which is part of the standard library of X10 [42].
- **XFT_GLB**: fault-tolerant X10 variant from [13].
- **J_GLB**: cooperative Java / APGAS variant from [31].
- **JFT_GLB**: new variant introduced in this paper.

The APGAS library internally uses the Java framework Hazelcast for connecting JVM's. This allows us to easily use other Hazelcast functionality as well. In particular, we deploy Hazelcast's `IMap` as the resilient data structure for our algorithm. The `IMap` is a distributed and concurrent variant of a traditional map. It is able to tolerate multiple simultaneous place failures by storing a configurable number of backup copies. Moreover, it supports transactions.

Hazelcast guarantees the consistency of all `IMap` entries even if places crash. Therefore, no backup is ever lost, which is a major advantage over the XFT_GLB approach. Altogether, JFT_GLB is easier than XFT_GLB since it requires less asynchronous protocols, and can rely on the persistence of `IMap` entries.

Like XFT_GLB, JFT_GLB is correct in the sense that it either computes the correct result, or halts with an error message. The error message appears in two situations:

- Failure of place 0: This case is not supported by the underlying APGAS library, and therefore can not be handled by JFT_GLB either.
- Loss of all backup copies: If the number of simultaneous place failures exceeds the number of backup copies, it may happen that an `IMap` entry and all of its replicas are lost.

JFT_GLB has been tested extensively by calling the Java function `System.exit()` to simulate place failures in specific situations. For performance measurements, we used the Unbalanced Tree Search (UTS), NQueens and Betweenness Centrality (BC) benchmarks. During failure-free executions on up to 144 places, we measured overheads of up to 46.06% over J_GLB. Moreover, we compared JFT_GLB to XFT_GLB, and did not observe a clear winner. Further experiments investigated the overheads for restore, and for raising the number of the backup copies to six. Both were found to be negligible.

The paper is organized as follows. Section 2 provides background information about X10, APGAS, Hazelcast, X_GLB [42], and J_GLB [31]. Afterwards, Section 3 explains the different aspects of our fault tolerance algorithm: worker main loop, writing backups of different types, and recovery from one or multiple place failures. Moreover, unrecoverable situations are discussed. Section 4 explains framework contracts and shows a sample application using JFT_GLB. Then, Section 5 is devoted to implementation. For instance, it discusses synchronization concepts. Afterwards, Section 6 discusses correctness informally, and Section 7 compares JFT_GLB to XFT_GLB. Section 8 describes our experimental setup and discusses the experimental results. The paper finishes with related work in Section 9, and conclusions in Section 10.

2 Background

This section starts with an overview of the programming language X10, as well as the APGAS and Hazelcast frameworks for Java. Then it introduces task pools, the lifeline scheme, and the X_GLB, J_GLB, and XFT_GLB implementations.

2.1 X10 and APGAS

X10 is a parallel, object-oriented and class-based programming language that has been developed by IBM. Its main target platform are clusters, where up to 10^5 hardware-threads are supported. The name X10 reflects the language's goal to raise programming efficiency by a factor of 10. The X10 syntax is inspired by Java. An X10 application can be compiled to C++ (called *Native X10*), or to Java (called *Managed X10*). For parallelization, X10 deploys the asynchronous variant of the PGAS model, which has already been described in Section 1.

In June 2015, a member of the X10 developer team released the APGAS library for Java [37]. It realizes the same parallelization and distribution concepts as X10. As formulated in [37], the APGAS library "supports resilient, elastic, parallel, distributed programming on clusters of JVMs". The X10 concepts are brought to Java by using lambdas. In the following, we describe and contrast X10 and APGAS.

X10 is a programming language, whereas APGAS is written in Java and uses Java language constructs. X10 objects are automatically serialized, whereas APGAS user classes must manually implement Java's interface `Serializable`.

In both systems, a *place* denotes a computational unit with a finite amount of shared memory, as noted in Section 1. A place typically corresponds to a cluster node, but multiple places can also be assigned to the same physical node.

Computations are accomplished by *activities*, which are lightweight threads on software-level. At program start, one activity is started on place 0 and executes the `main` method. Activities are mapped to system threads of the respective place. The number of system threads per place can be configured in both X10 and APGAS.

Both systems provide an `at` construct, which synchronously moves the current activity to another place. There, it executes a given code section and then returns. If an exception is raised on the other place, it can be caught on the origin place by a surrounding `try-catch` block. The `at` construct supports return values. Moreover, its invocation transparently copies variables from the origin place that are accessed on the remote place, and sends them along.

A new activity can be spawned at runtime with `async`. The activity is executed on the current place when a system thread becomes available and then runs to completion. In both X10 and APGAS, the scheduler assigns activities in arbitrary order. The `async` construct does not block and always returns instantly. Consequently, there is no well-defined point to catch an exception that is thrown inside an `async` block.

By combining `at` and `async`, one may start an asynchronous activity on a remote place. In this case, the parent activity does not wait for the child's termination. The APGAS library provides a combined `asyncAt` construct, X10 has no such construct. Asynchronous activities are also denoted as active messages, or briefly messages, in this paper.

Activities can be spawned within a `finish` block. In this case, the parent activity waits until all spawned activities, including recursively and/or remotely spawned activities, have been terminated. Only then, the program execution is resumed. Inside a `finish` block, exceptions are accumulated. Thrown exceptions can be caught by a `try-catch` block surrounding the `finish` on the origin place.

The X10 runtime provides the function `Runtime.probe()`, which interrupts the calling activity so that all pending activities are executed successively in any order. Neither APGAS nor Java include such a function.

Both X10 and APGAS support user-level failure handling wrt. permanent place failures. They provide the following types of failure notifications:

- A `DeadPlaceException` is thrown when a place failure occurs. APGAS combines multiple `DeadPlaceException`s into a `DeadPlacesException`, whereas X10 accumulates them in a `MultipleException`.

- Programmers can register a handler on each place which is automatically invoked when any other place crashes. In APGAS, it is called `placeFailureHandler`, and in X10 `placeRemovedHandler`. The latter is only available in Managed X10.

- X10 provides the function `boolean isDead(id)` to inquire the liveliness of place `id`. The function can be called at any time. APGAS does not natively provide such a function, but we implemented it by extending the APGAS source code.

X10 offers only limited place-internal concurrency control. For instance, the keyword `atomic` marks a critical section. APGAS programmers, in contrast, may use Java's extensive facilities for concurrency control. They include the keyword `synchronized`, which requires an object as parameter and uses this object for locking a specified code block. Synchronized blocks do not guarantee fairness wrt. the execution order of pending activities. Another Java concurrency construct is the data structure `AtomicBoolean`, which is a thread-safe implementation of a traditional `boolean` variable. A `ConcurrentLinkedQueue` manages data structure access in a thread-safe way.

Moreover, Java offers a wait-and-notify mechanism, which enables thread communication in locking situations. If a thread inside a `synchronized` block discovers that some condition is not fulfilled, it can call the function `wait()` on the lock object. Then, the lock is released and the thread waits until another thread calls the function `notify()` on this lock object. Because of the lock release, another thread can enter the `synchronized` block and apply changes that cause the condition to become fulfilled. In this case, it calls `notify()`.

2.2 Hazelcast

The networking layer of APGAS is based on the Hazelcast library. Each place is represented by a Hazelcast node, which corresponds to a single JVM. APGAS deploys Hazelcast for its ability to handle the connection between these JVMs. Nevertheless, deployment of this library brings to APGAS programmers the additional benefit that they can easily use the whole Hazelcast library.

It provides several beneficial features for parallel and distributed programming. In particular, we use the `IMap` data structure, which is an automatically distributed, concurrent, and fault-tolerant variant of a traditional key-value store. Moreover, several handlers can be registered at an `IMap` instance, which are executed automatically at corresponding events. Internally, an `IMap` is divided into partitions. These partitions are distributed evenly over all nodes of a Hazelcast cluster. If a new node joins the cluster, the partitions are automatically redistributed, so that they remain balanced. The same applies if a node leaves the cluster, intentionally or after a failure. Each key is unambiguously assigned to a single partition with a hash function, so that the entries are distributed evenly as well.

For fault tolerance, each partition is replicated to other nodes. The number of replicas is configurable between zero and six. If a node leaves the cluster, a backup of its data is still available (if the number of replicas has been set to at least one). So the lost data can be restored. A high number of replicas increases the availability of data, but also the network traffic.

If all owners of a replica leave the cluster before the restore has been finished, a partition may get lost. For handling this case, Hazelcast allows to register a `PartitionLostHandler` at each place. It is automatically triggered and executes user-defined code.

The `IMap` data structure is thread-safe. It provides a wide set of functions with different trade-offs between usability and performance. The most simple and efficient type of access functions is `get()` and `set()`, which correspond to thread-safe variants of the traditional map functions. Each call is independent and performs an own network operation. Therefore, multiple calls of those functions are carried out in any order. The `get()` and `set()` functions do not lock their `IMap` entry.

If multiple access operations refer to the same entry, they can be encapsulated into an `EntryProcessor`. It locks and unlocks the entry automatically and retains the order of operations. Such an `EntryProcessor` plus a key can be passed to the function `executeOnKey()`, which processes the `EntryProcessor` directly on the key owner. Therefore, network traffic is saved and the operations are executed atomically.

Moreover, Hazelcast allows to bundle multiple data structure accesses in a transaction. Transactions guarantee atomicity, consistency, isolation and durability of the included code. Instead of executing operations immediately on a `IMap` instance, transactions first lock all involved entries. Then, they perform the changes locally on data copies in a transaction context. Only if all operations have been successfully performed, the changes are committed to the real `IMap` instance, and afterwards the entries are unlocked. In cases of error, all previously executed operations are rolled back and a `TransactionException` is thrown. Error cases include place failure. If the transaction terminates successfully, all operations have been executed. Rollback only works for operations on supported Hazelcast structures.

2.3 Lifeline-Based Global Load Balancing

As noted in Section 1, X_GLB implements a variant of the *task pool* pattern. In this pattern, a fixed number of workers processes a large number of tasks. The X_GLB *task model* has the following characteristics:

- Tasks have no side-effects.
- Processing a task can generate new tasks.
- Processing a task produces a task result.
- All task results have the same type.
- Each worker accumulates task results into a partial result.
- The final result is computed from partial results by reduction, using a commutative and associative operator.

In X_GLB's lifeline scheme each worker maintains its own local pool. From this local pool, the worker takes out tasks and processes them. Moreover, it inserts any newly generated tasks there. The local pool is a user-defined data structure. At program start, at least one local pool contains at least one task. On each place, one worker is started.

If the local pool of a worker runs empty, the worker tries to steal tasks from another worker. Correspondingly, the involved workers are called *thief* and *victim*, respectively. First, a thief tries to steal tasks from up to w random victims. If these attempts were unsuccessful, it tries to steal tasks from up to z so-called *lifeline buddies*. The latter are preselected, such that the overall lifeline buddy relationship gives rise to a z-dimensional hypercube [36]. If a thief tries to steal task from a lifeline buddy, it activates the corresponding *lifeline*. This is a flag that indicates that there is an open request to this victim. A thief does not steal from a lifeline buddy if the lifeline is already activated. When a lifeline buddy sends tasks to the thief, the corresponding lifeline is deactivated.

X_GLB's deploys a *cooperative* style of work stealing. This means that a thief sends a steal request and waits for the answer. It is not allowed to access the victim's task pool directly. After receipt of the steal request, the victim responds by sending tasks, called *loot*, or a reject message if it has no tasks to share. If the victim is a lifeline-buddy and has no tasks, it additionally stores the request and tries to send tasks later.

If all $w + z$ steal requests were unsuccessful, the thief goes *inactive*. An inactive worker can only be reactivated by a lifeline-buddy that sends tasks later. When all workers are inactive, the overall computation has been completed. Then, the final result is computed by collecting and reducing all partial results.

2.4 X_GLB vs. J_GLB

Recall that J_GLB denotes our re-implementation of the original X_GLB framework in Java using APGAS [31]. In the following, we sketch X_GLB and J_GLB, more detailed descriptions can be found in [42] and [31], respectively.

X_GLB always starts one worker per place. Moreover, it allows only one running activity per place, i.e., it precludes any place-internal parallelism. An *outer* `finish` block encapsulates all worker activities. This `finish` block is invoked on place 0, but each worker enters its main loop on its respective place. The *worker main loop* of a X_GLB worker is shown in Listing 1 in simplified pseudocode.

```
1  while (tasks available) {
2    while (local pool is not empty) {
3      process up to n tasks;
4      Runtime.probe();
5      send loot to recorded thieves;
6    }
7    try to steal from up to w+z victims;
8  }
```

Listing 1: Worker main loop in X_GLB [31]

As shown in Listing 1, a worker alternately processes up to n tasks (line 3), and answers the steal requests that have arrived in the meantime (line 5). Listing 1 only shows the case that the worker has enough tasks to share, and therefore answers all steal requests by sending loot. Otherwise, the remaining requests are rejected in a similar way. The victim answers random requests first, and lifeline requests thereafter. Thus, random requests have a higher chance of being fulfilled than lifeline requests. If a lifeline request is fulfilled, the corresponding lifeline is deactivated. Moreover, if the thief has become inactive since sending the request, it is restarted.

If a worker runs out of tasks, it leaves the inner loop and tries to steal tasks from up to $w + z$ victims (line 7). When it has received tasks, from a current steal request or a former lifeline steal request, it enters the inner loop in line 2 again. Otherwise, the conditions in lines 1 and 2 become `false` and the worker main loop ends. After the main loop, the overall worker activity ends, which corresponds to inactive state. When an inactive worker receives loot from a lifeline buddy, the same active message that sends the loot additionally restarts the worker. For that, it invokes a new activity on the thief place, which executes the main loop from Listing 1 again.

The send and steal operations (lines 5 and 7) create new asynchronous activities on the remote place by using `at async`. Since X_GLB permits only one running activity, these activities are queued and not executed instantly. When the remote worker activity calls `Runtime.probe()` in line 4, all queued activities are executed in arbitrary order. The parameter n in line 3 controls how many tasks are processed without interruption and therefore the delay between arrival and execution of the activities. The default value is $n = 511$.

Each steal messages from line 7 checks whether the victim's task pool is empty by inspecting a flag. If the flag is `true`, the activity immediately sends a reject message to the thief. Otherwise, the activity inserts the ID of the thief into a data structure on the victim place. Such queued requests are later answered by the worker activity itself in line 5.

J_GLB and X_GLB run one worker per place. However, J_GLB allows multiple simultaneously running activities on the same place, while X_GLB does not. The J_GLB relaxation has several merits [31], but requires place-internal synchronization. For that, J_GLB utilizes Java's `synchronized` construct to realize exclusive access to the local pools. These critical sections are used sparingly. In particular, the registration of a thief can be done concurrently to task processing. Listing 2 shows the worker main loop of J_GLB, again in simplified pseudocode.

Note that all `synchronized` blocks in Listing 2 (lines 3 and 8) are parameterized with the same local object. Therefore all accesses to the local pool are protected by the same lock. On the victim place, a steal request activity from line 9 does not require a `synchronized` block, because it does not access the local pool. The accesses to the `empty` flag and the list of thieves are protected with the Java data structures `AtomicBoolean` and `ConcurrentLinkedQueue`, respectively.

```
1   while (tasks available) {
2     while (local pool is not empty) {
3       synchronized (worker object) {
4         process up to n tasks;
5         send tasks to recorded thieves;
6       }
7     }
8     synchronized (worker object) {
9       try to steal from up to w+z victims;
10    }
11  }
```

Listing 2: Worker main loop in J_GLB [31]

On the thief place, in contrast, the activity that delivers loot needs access to the thief's local pool and therefore deploys `synchronized`. Since a worker can receive tasks from its lifeline buddy at any time, all local pool accesses in the main loop are encapsulated by a `synchronized` block. Consequently, loot delivery activities may have to pend until the worker activity on the thief place leaves the `synchronized` block in line 6 or 10. To avoid blocking other activities on the victim side, loot delivery activities are started asynchronously by using `asyncAt`.

The stealing in line 9 is performed within a `synchronized` block, but whenever the thief has sent out a steal request by starting a new asynchronous activity on the victim place, the thief calls `wait()` to release the lock. Thus, the contacted victim (or a lifeline buddy) is able to start a new activity on the thief place, which can acquire the lock and merge the loot into the thief's task pool. After this merge, the thief is woken up by calling `notify()`.

3 Fault-Tolerant Algorithm

Our fault-tolerant algorithm occasionally writes backups of each local pool. Each backup includes all tasks of the corresponding local pool plus some administrative information, as will be explained later.

Backups are written to one system-wide `IMap` instance, named `iMapBackup`. There are four types of backups: Regular and final backups are written at fixed times, restore backups are written during recovery, and steal backups are written during work stealing. The transfer of tasks during work stealing requires particular attention. Even if a place crashes during this transfer, victim and thief, including their backups, must be kept consistent. For instance, if the victim crashes before the sent loot has arrived, the sent loot must not be lost.

Therefore, we deploy a second system-wide `IMap` instance to hold loot during steal operations. This instance is named `iMapOpenLoot` and stores one or several pieces of loot. A piece of loot is called *open* while being contained in `iMapOpenLoot`. Briefly stated, before sending loot, a victim enters it into `iMapOpenLoot`. Upon receipt, the thief merges the loot into its local pool, writes a backup to `iMapBackup`, and notifies the victim. Thereafter, the victim removes the loot from `iMapOpenLoot`.

When a place crashes, `iMapOpenLoot` is checked for open loot that has been sent to or from the failed place. If such loot is found, a consistent state is reconstructed with the help of both `iMapBackup` and `iMapOpenLoot`. Recovery is explained in detail in Section 3.3 for the single-failure case, and in Section 3.4 for the multiple-failure case.

A piece of loot is identified by a so-called loot identifier, or shortly *lid*. Lids have type `Integer` and are assigned consecutively to the pieces of loot sent by each victim. In combination with the victim's place ID, lids are system-wide unique.

The original GLB allows two open steal requests from the same thief to the same victim, namely a lifeline steal followed by a random steal. JFT_GLB introduces a restriction of only one open steal request from the same thief to the same victim. This guarantees that a thief receives loot from a particular victim in the order of increasing lids (since a piece of loot remains open until it is received). The restriction can be easily enforced by discarding the random request and treating the stored lifeline request as if it would have arrived just now.

Consequently, a thief only needs to store the lid of the last received piece of loot from each victim, to keep a record on all loot received in the past. The corresponding array holds one entry per victim is named `lidsReceived`. It is held in the local pool data structure. Similarly, a victim stores its last lid sent in local pool field `lidSent`.

In both `IMap` instances, each entry has a unique owner. In `iMapBackup`, it is the place that wrote the backup. In `iMapOpenLoot`, it is the victim place. To reduce the need for locking `IMap` accesses, we enforce a strict access policy: During failure-free operation, only the owner of an entry is allowed to access it. After a failure, its backup place takes over ownership. It marks the entry as `done` to avoid interference with any late messages from the original owner. Technically, `done` is a `boolean` field in the local pool data structure.

Consequently, outside failures, locking is only required for accesses by different concurrent activities on the same place. Recall that in J_GLB accesses to the local pool are protected by a critical section. JFT_GLB uses the same lock to protect accesses to the place's entries in the `IMap` instances.

Listing 3 presents simplified pseudocode for JFT_GLB's worker main loop. It extends the corresponding J_GLB code from Listing 2 and is explained below.

```
1  while (tasks available) {
2    while (local pool is not empty) {
3      synchronized (worker object) {
4        process up to n tasks;
5        for each recorded steal request {
6          write steal backup;
7          send tasks to recorded thief;
8        }
9        if (currentK++ % k == 0) {
10         write regular backup;
11       }
12     }
13   }
14   synchronized (worker object) {
15     try to steal from up to w+z victims;
16   }
17 }
18 write final backup;
```

Listing 3: Worker main loop in JFT_GLB

3.1 Backups

As noted before, there are four types of backups: Regular and final backups are invoked in lines 10 and 18 of Listing 3, respectively. Steal backups are invoked in line 6 and, indirectly, in line 15. Restore backups are written when tasks are merged during restore, they are not shown in Listing 3.

Backups of any type include the same data from the local pool: all tasks, the partial result, the `lidsReceived` array, `lidSent`, `done`, and possibly other information that a user may have accommodated in the task pool class (see Section 5).

Backup writing is uncoordinated among places. For each place it is realized by a call of `executeOnKey()` on `iMapBackup`, to which the matching key and the new backup are passed. If a place fails during backup writing, it may happen that the backup arrives at `iMapBackup` after the backup place has taken over ownership. To avoid that the late backup from the original owner is written, backup writing is always performed conditionally, i.e., only when `done` has not yet been set.

Regular backups (line 10) are written independently by each worker every k main loop iterations, i.e., after having processed $n \cdot k$ tasks (line 9). Like n, parameter k can be set by the user. The default value $k = 2048$ should be adapted to the task size to avoid, for instance, the overhead of too frequent backups.

Final backups (line 18) are written just before a worker goes inactive. They do not contain tasks, but all other aforementioned information, in particular the worker's partial result.

Steal backups (line 6 and 15) are required to keep the backups of victim and thief consistent. They are written at every steal on both the victim and thief sides. Details are defined in the steal protocol, which is introduced next.

Restore backups accompany the restore of a place. They are written after the corresponding tasks are merged into the local pool.

3.2 Steal Protocol

Figure 1 depicts our steal protocol. It shows the case of a successful random steal with one thief (on the right side) and one victim (on the left side). There may be several thieves at the same time, which are not depicted for clarity. Arrows in the figure denote active messages, i.e., activities invoked with `asyncAt`. If the wavy line is interrupted, the corresponding operation(s) are performed in a critical section (e.g. `delete loot`), so that no tasks are processed concurrently. Otherwise, the

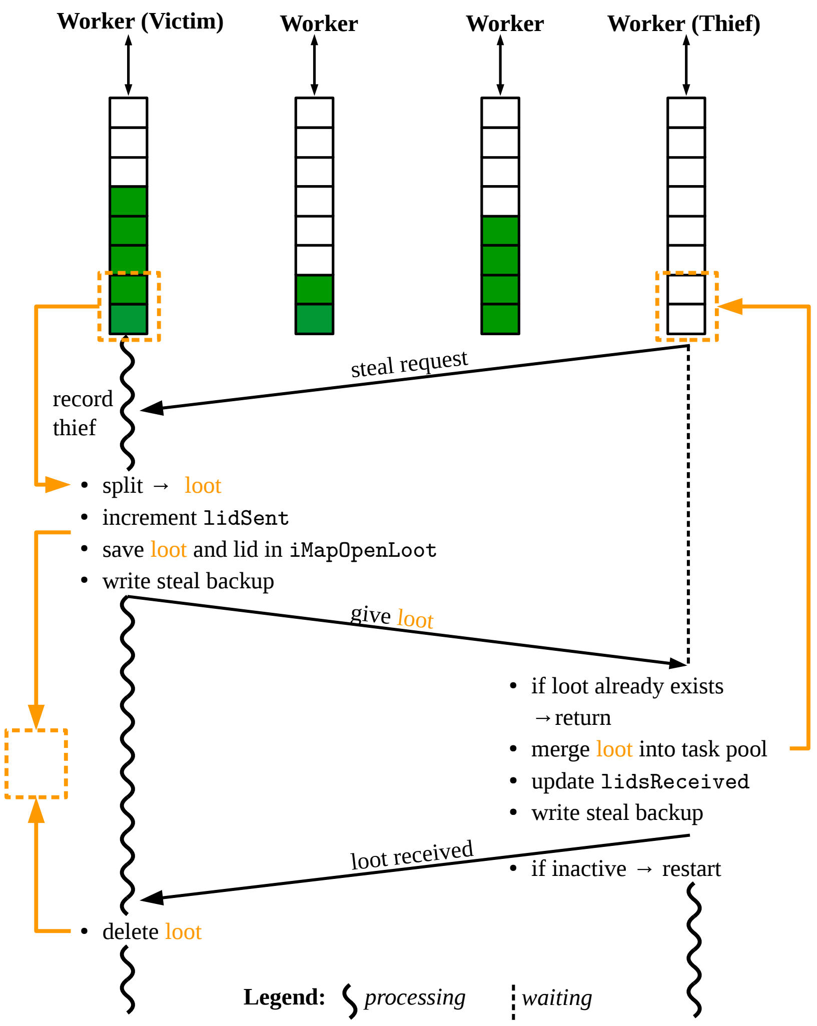

Figure 1: Steal Protocol

operation is executed concurrently to other local activities, which is the case for `record thief`.

The protocol starts when the thief runs out of tasks or receives a reject message. The steal request shown in Figure 1 corresponds to one of the activities started by line 15 of Listing 3. Recall that line 15 starts one or several activities on up to `w+z` victim places and after each steal attempt waits for an answer by calling `wait()`.

The steal request activity determines whether the victim has tasks to share, by inspecting the `empty` flag. The activity either answers by sending a reject message (not shown in the figure), or it records the thief as shown.

In the depicted case, the victim worker activity notices the request in line 5 of Listing 3. It extracts loot, increments `lidSent`, stores a (loot, lid) pair in `iMapOpenLoot` and writes a backup to `iMapBackup`, as shown in Figure 1. Only then, loot and lid are sent to the thief, which is called `give loot` in Figure 1.

When the thief receives the loot, it checks whether it has already received the same piece of loot before. This may be the case if a piece of loot is sent twice during our recovery protocol (see Section 3.3). In that case, the second delivery is ignored. The check can be performed easily by comparing the received lid with `lidsReceived[victim]`.

Usually, `lidsReceived[victim]` is less than the received lid, and the thief merges the received loot into its local pool. Thereafter, the thief updates `lidsReceived[victim]`, writes a steal backup, and notifies the victim, called `loot received` in Figure 1. On the victim place, the `loot received` activity removes the loot from `iMapOpenLoot`. Finally, if the thief worker was inactive, it is restarted to process the received tasks.

3.3 Recovery after one Place Failure

For simplicity, we assume in this section that only a single failure occurs, and that each place can be sure of that. Of course, this assumption is unrealistic. Therefore, the algorithm is actually more complex and involves precautions for multiple-failure cases. Section 3.4 adds the missing details and introduces the complete algorithm.

JFT_GLB registers a `placeFailureHandler` on each place. The APGAS runtime invokes these handlers automatically when a place `x` fails and passes `x`'s ID to each handler. In the following, we describe the actions that are performed by the handler on place `p`:

1. It discards any recorded steal request from thief `x` to victim `p`.

2. It treats any outstanding steal request to `x` as if it was rejected.

3. If `x` is `p`'s lifeline buddy, it activates the lifeline. Therefore, it will not send any lifeline steal requests to `x` in the future.

4. For any open loot that has been sent from victim `p` to thief `x`, the handler checks whether this piece of loot is already contained in `x`'s backup, by inspecting the corresponding `lidsReceived` entry. If `x`'s backup contains the loot, `p` deletes the loot from `iMapOpenLoot`. Otherwise, the handler re-merges the loot into `p`'s local pool, deletes it from `iMapOpenLoot`, writes a new backup, and marks `x`'s backup as `done`. Operation 4 is carried out in a transaction. Otherwise, it could happen that `p` inspects the backup and decides to re-merge the loot, but the loot arrives late in `x`'s backup before `p` has set `done`.

5. If `p` is `x`'s backup place, the handler merges all tasks from `x`'s `iMapBackup` entry into `p`'s local pool, removes them from the `IMap` entry, and marks this entry as `done`. Afterwards, `p` writes a new backup of its own local pool.

6. If `p` is `x`'s backup place, the handler additionally checks whether `x` has open loot. If so, `p` essentially re-sends the loot to the respective thieves, because it is unknown whether `x` has actually sent the loot. Re-sending is done synchronously by using `at` (see 3.4). A return of the `at` indicates that the respective thief has taken over the loot, either in reaction to the original sending or to the re-sending. Therefore, `p` can now safely remove the loot from `iMapOpenLoot`.

In particular, before re-sending loot, p checks whether the loot is still contained in x's backup (which is already marked done at this point). This may occur, if x has crashed right after saving the loot to iMapOpenLoot, but before writing the backup and sending the loot (see Figure 1). In this case, the loot is deleted from iMapOpenLoot and not re-sent.

3.4 Recovery after multiple Place Failures

For multiple failures, let us first extend the definition of a backup place: We consider places as being arranged in a ring, according to their numbers and with wraparound. For any place p, its *backup place* is the closest predecessor of p in this ring that is alive. A place p can easily find out whether it is the backup place of place x, e.g. by inspecting x's ID and the liveliness of all places from its right neighbor up to x.

With that definition, independent failures, i.e., failures that occur at different times and/or regard disjoint subsets of workers can be handled like a sequence of single failures.

In the following, we examine dependent failures. First of all, we discuss which place is responsible for restoring a dead place x, i.e., for performing operations 5 and 6 from Section 3.3. In contrast to the previous section, it can not always be the backup place, since that place may likewise be affected by failure.

Let us consider the following situation as an example:

$$\mathtt{p} \quad \mathtt{x}_1 \quad \mathtt{x}_3 \quad \mathtt{x}_2 \quad \mathtt{x}_4 \quad \mathtt{x}_0 \quad \mathtt{x}_5 \quad \mathtt{q}$$

This example shows a sequence of places in ring order, i.e., p comes first in the ring, then $\mathtt{x}_1$, $\mathtt{x}_3$, and so on. Places named $\mathtt{x}_i$ fail, and the numbers indicate the order of failure, i.e., $\mathtt{x}_0$ fails first, then $\mathtt{x}_1$, $\mathtt{x}_2$, and so on.

When $\mathtt{x}_0$ and $\mathtt{x}_1$ fail, they are restored by the handlers on $\mathtt{x}_4$ and p, respectively. The failures are independent of each other and both are handled as described in Section 3.3.

When $\mathtt{x}_2$ fails, the handler on $\mathtt{x}_3$ is responsible for the restore of $\mathtt{x}_2$. However, if $\mathtt{x}_3$ fails before or during restoring $\mathtt{x}_2$, p and all others are notified about $\mathtt{x}_3$'s failure as usual. Then, the handler on p takes over responsibility for restoring *both* $\mathtt{x}_3$ and $\mathtt{x}_2$. On the assumption that $\mathtt{x}_4$ crashes shortly after $\mathtt{x}_3$, the same handler on p is even responsible for restoring $\mathtt{x}_4$.

In general, a handler that is invoked on the backup place of the failed place iterates over all places to its right in the ring, until the next place alive. The current place in this loop is named iterPlace. For each iterPlace covered, the handler checks whether restore is needed, i.e., whether the place's backup contains tasks and/or the place has open loot.

If restore is needed, the handler performs the restore. In our example, upon failure notification for $\mathtt{x}_3$, iterPlace takes on values $\mathtt{x}_1$, $\mathtt{x}_3$, $\mathtt{x}_2$ and $\mathtt{x}_4$. Assuming that $\mathtt{x}_5$ fails *after* the iteration, it is restored by another handler on p later, when p has become $\mathtt{x}_5$'s backup place. Eventually, p becomes the backup place of q.

Overall, the failure handler on each place *p* first performs operations 1 to 4 from Section 3.3 and then carries out the steps of the flow diagram in Figure 2.

The flow diagram in Figure 2 starts with a loop over all failed places to the right. If no iterPlace candidate is found anymore, the recovery has finished ⑩. If an iterPlace is found ⓪, the failure handler first performs a transaction that carries out operation 5 from Section 3.3. Use of a transaction avoids, e.g., that the tasks from iterPlace's backup are merged into p's local pool, but remain in iterPlace's backup. If the transaction fails, which is the case if p dies during its execution, the backup remains unchanged and can be restored later by another place.

Afterwards, the handler carries out operation 6 from Section 3.3 which, again, iterates over all loot contained in iMapOpenLoot(iterPlace) ①. The current loot in this loop is called iterLoot in Figure 2. As in the single-failure case, the handler makes sure that the loot is no longer contained in iterPlace's backup, and otherwise just deletes it ②. If no more iterLoot is found for the current iterPlace, the next iterPlace is dealt with ⑨.

Normally, the handler tries to re-send the loot to the respective thief, called iterThief ③. While not strictly necessary, we first check whether iterThief is alive, before starting the at for re-sending. Use of synchronous communication for the re-sending allows to react immediately to a potential place failure.

If `iterThief` is alive ⑤, successful return of the `at` ⑥ indicates that the loot has been delivered for sure and can be deleted. This has already been discussed in Section 3.3. If `iterThief` is dead ④ or fails during the re-send ⑦, our handler checks whether `iterThief`'s backup contains the loot ⑧. Otherwise, `p` takes over the loot by merging the tasks into its own local pool, deleting it from `iMapOpenLoot`, and writing a new backup ⑧.

The lookup of `iterThief`'s backup and the respective actions in ⑧ need to be performed within a transaction. Otherwise, it may happen, e.g., that `iterLoot` arrives at `iterThief`'s backup after the lookup, but `done` has been set, or that `iterLoot` is deleted but not inserted into `p`'s backup. Again, if `p` crashes during the transaction, `p`'s backup place will find and handle `iterLoot`.

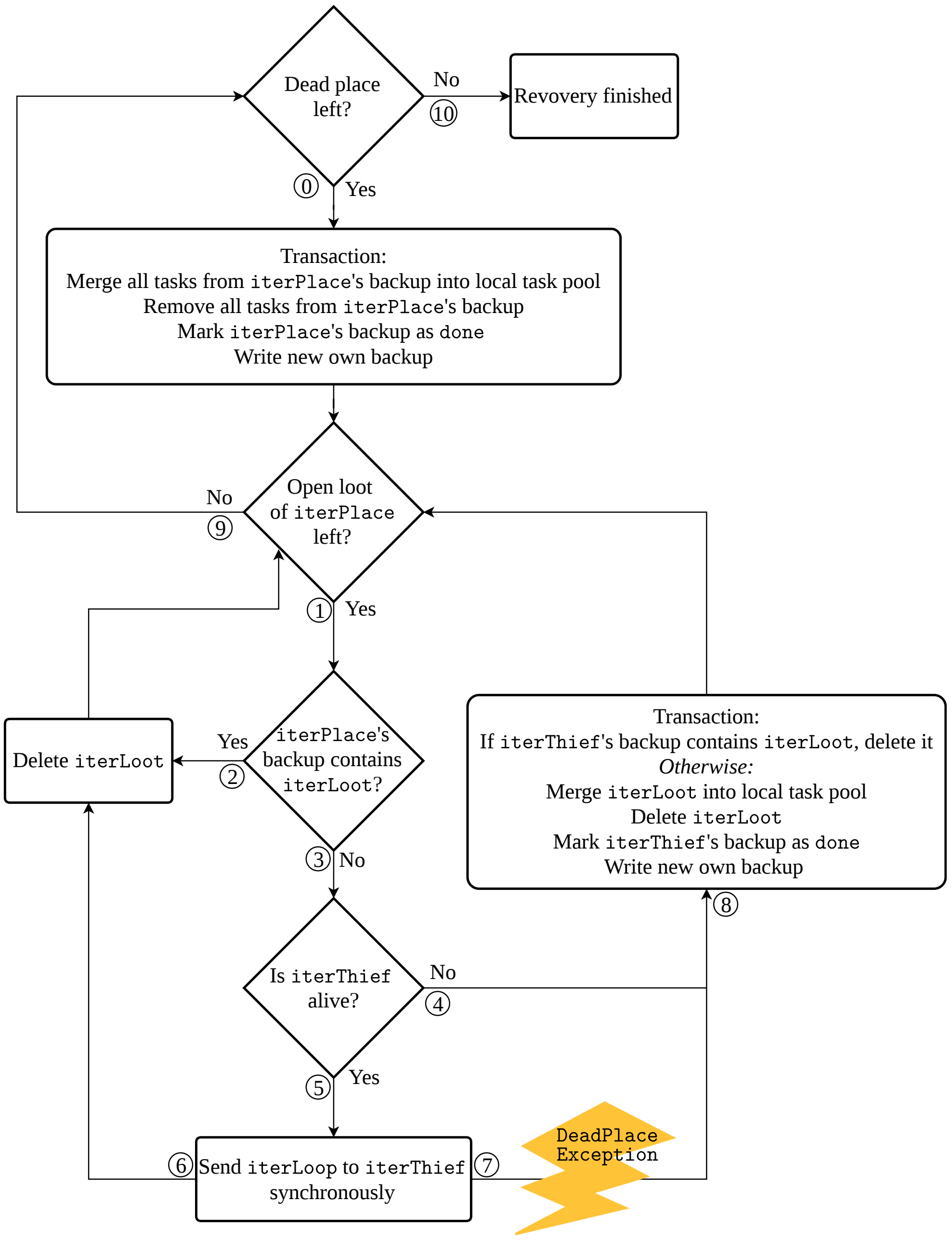
Dead place left?
No
⑩
Revovery finished
⓪
Yes
Transaction:
Merge all tasks from iterPlace's backup into local task pool
Remove all tasks from iterPlace's backup
Mark iterPlace's backup as done
Write new own backup
No
⑨
Open loot of iterPlace left?
①
Yes
Yes
②
iterPlace's backup contains iterLoot?
Delete iterLoot
③
No
Transaction:
If iterThief's backup contains iterLoot, delete it
Otherwise:
Merge iterLoot into local task pool
Delete iterLoot
Mark iterThief's backup as done
Write new own backup
⑧
Is iterThief alive?
No
④
⑤
Yes
⑥
Send iterLoop to iterThief synchronously
⑦
DeadPlace Exception

Figure 2: Flow diagram of a multiple place failure recovery

3.5 Unrecoverable Situations

Our fault tolerance algorithm has no inherent limitations on the number of failures that can be tolerated. Nevertheless, failure of place 0 or loss of an `IMap` partition lead to an unrecoverable situation. The first case is detected by the `placeFailureHandler`s, and the second by the `partitionLostListener` which is triggered automatically by Hazelcast. A place 0 crash can not be tolerated by the APGAS library, since APGAS is not able to migrate the outer `finish` from place 0 to another place. An `IMap` partition gets lost if all backup copies are gone.

4 Usage of JFT_GLB

4.1 Framework Contracts

As noted in Section 2.3, GLB users must implement a class for the local pool. This class must implement the following methods:

- `boolean process(n)` takes out n tasks from the local pool, processes them, and inserts any newly generated tasks into the local pool again. The function returns `true` if and only if the local pool contains at least one task afterwards.
- `TaskBag split()` extracts tasks from the local pool and returns them as a `TaskBag` object. `TaskBag` is a user class, which represents a simple container for tasks. The `split()` function is called by victims, and the return value corresponds to a piece of loot.
- `void merge(TaskBag)` merges the `TaskBag` object into the local pool. This function is called by thieves when receiving loot.

In JFT_GLB, users must implement additional methods:

- `TaskBag getAllTasks()` returns all tasks from the local pool in a `TaskBag` object. The function is called during restore, for example by the backup place. Note that the tasks are not deleted from the local pool.
- `void clearTasks()` deletes all tasks from the local pool.
- Getter/Setter methods for `receivedLids`, `sentLid` and `done`.

Users may also add application-specific fields and functions to the pool class. These fields may be excluded from backup by marking them with the Java keyword `transient`.

4.2 Pi

This section illustrates JFT_GLB usage with a simple example: the calculation of π from integrals. For this application, we provide the complete, compilable code. The example is naturally coded with a static initial work distribution, since all tasks are known at program start. For demonstration purposes however, we provide implementations for both static and dynamic initial work distributions. The code is depicted in Listings 4 to 7, and explained in the following.

Bag (Listing 4) represents a piece of loot, which consists of a set of tasks. Here, a task is represented by a single `Integer` value. This class has to implement the JFT_GLB interface `TaskBag`.

Queue (Listing 5) contains the sequential computation of the actual problem (line 32), the local pool (line 4) and the partial worker result (line 5). It has to implement the JFT_GLB interface `TaskQueue`. The generic parameter `Double` defines the type of the result. The generic parameter `Queue` defines the return type of some generic functions and is set the type of the class.

Workers process tasks by invoking the function `process()` (line 32), which pops and processes up to n tasks successively. In the function `split()` (line 22), the application may decide how many tasks are stolen (line 23).

Since the function `getResult()` (line 47) returns an instance of `GLBResult`, the inner class `PiResult` (line 67) extends the JFT_GLB class `GLBResult`.

The function `init()` (line 19), which initializes the local pool, is only invoked when using dynamic initial work distribution.

StartDynamically (Listing 6) starts this application dynamically so that all initial tasks are located on place 0. Therefore, the `Queue` constructor (line 4) and the `GLB` constructor (line 6) is invoked with `true` as last parameter. The `result` (line 8) is an array of size 1, in which index 0 contains pi's value.

StartStatically (Listing 7) starts this example statically by invoking the `Queue` constructor (line 4) and the `GLB` constructor (line 6) with `false` as last parameter. The result has the same form as above.

```java
1  public class Bag implements TaskBag {
2    public Deque<Integer> list = new LinkedList<>();
3
4    @Override
5    public int size() { return list.size(); }
6  }
```

Listing 4: Class Bag

```java
1  public class Queue implements TaskQueue<Queue, Double> {
2    int N;
3    double deltaX;
4    Deque<Integer> list = new LinkedList<>();
5    double result = 0;
6
7    public Queue(int n, boolean dynamicDistribution) {
8      N = n;
9      deltaX = 1.0 / N;
10     if (dynamicDistribution == false) {
11       int step = N / places().size();
12       int start = here().id * step;
13       int end = Math.min(start + step, N);
14       for (int i = start; i < end; i++)
15         list.add(i);
16     }
17   }
18
19   public void init() { for (int i = 0; i < N; i++) list.add(i); }
20
21   @Override
22   public TaskBag split() {
23     int size = size() / 2;
24     if (size <= 0) return null;
25     Bag bag = new Bag();
26     for (int i = 0; i < size; i++)
27       bag.list.add(list.poll());
28     return bag;
29   }
30
```

```java
  @Override
  public boolean process(int n) {
    double r = 0;
    for (int i = 0; i < n; i++) {
      double x = (list.pop() + 0.5) * deltaX;
      r += 4.0 / (1 + x * x);
      result += r * deltaX;
      if (size() <= 0) break;
    }
    return (size() > 0);
  }

  @Override
  public void merge(TaskBag taskBag) { list.addAll(((Bag) taskBag).list); }

  @Override
  public GLBResult<Double> getResult() { return new PiResult(); }

  @Override
  public void mergeResult(TaskQueue<Queue, Double> that) {
    result += that.getResult().getResult()[0];
  }

  @Override
  public int size() { return list.size(); }

  @Override
  public void clearTasks() { list = new LinkedList<>(); }

  @Override
  public TaskBag getAllTasks() {
    Bag bag = new Bag();
    bag.list.addAll(list);
    return bag;
  }

  public class PiResult extends GLBResult<Double> {
    @Override
    public Double[] getResult() { return new Double[]{result}; }
}
```

Listing 5: Class Queue

```java
public class StartDynamically {
  public static void main(String... args) {
    int N = 1000000;
    SerializableCallable<Queue> init = () -> new Queue(N, true);
    GLBParameter para = new GLBParameter();
    GLB<Queue, Double> glb = new GLB<>(init, para, true);
    Runnable start = () -> glb.getTaskQueue().init();
    Double[] result = glb.run(start);
  }
}
```

Listing 6: Class StartDynamically

```java
public class StartStatically {
  public static void main(String... args) {
    int N = 1000000;
    SerializableCallable<Queue> init = () -> new Queue(N, false);
    GLBParameter para = new GLBParameter();
    GLB<Queue, Double> glb = new GLB<>(init, para, false);
    Double[] result = glb.runParallel();
  }
}
```

Listing 7: Class StartStatically

5 Implementation

The implementation of JFT_GLB is based on the source code of J_GLB from our previous work [31]. The code will be published on the first author's homepage upon acceptance of the paper. In the following paragraphs, we discuss several interesting details of our implementation.

5.1 Synchronization

We protect each access to the local pool and to the `IMap` instances with the same lock. For that, we use `synchronized()` blocks, parametrized with the variable `waiting`. This variable is a worker attribute of type `AtomicBoolean`. If it has value `true`, the worker is waiting for a steal answer, and if it has value `false`, the worker is carrying out local operations such as processing tasks. Our locking scheme could be slightly improved by deleting the loot in parallel to task processing. For that, we would need multiple locks. To keep the program easier to understand and implement, we stayed with the one-lock approach.

J_GLB utilizes Java's `ConcurrentLinkedQueue` for storing steal requests from thieves and lifeline thieves. This data structure is a thread-safe variant of a traditional linked queue. However, a sequence of accesses is not protected against interruption. JFT_GLB uses such sequences to avoid recording both a lifeline steal request and a random steal request from the same victim. To state it in more detail, a new random steal request should not be stored while a previous lifeline steal request from the same thief is being answered. So, we utilize `synchronized()` blocks with a `thieves` parameter to protect each access to the two thief lists, which store steal requests from thieves and lifeline thieves. Consequently, we do not need concurrent lists and utilize traditional non-concurrent lists as type.

5.2 DeadPlaceExceptions

A place change can be performed with `at` and `asyncAt`. If the remote place is dead when calling such a construct, APGAS throws a `DeadPlaceException`. To timely catch these exceptions, we surround each place change by a `try-catch` block. Moreover, there is a `try-catch` block around the outer `finish`, to catch exceptions that are raised during the `asyncAt` blocks. Since our algorithm handles place failures in the `placeFailureHandler`s, all `catch` blocks are left empty, except for the `catch` block that handles the `DeadPlaceException` in Figure 2 (marked ⑦).

5.3 PlaceFailureHandler

We register one `placeFailureHandler` at each place by passing a method reference to the worker constructor. As noted before, the `placeFailureHandler`s are automatically invoked by the APGAS runtime when a place crashes. Our implementation of the handlers performs the recovery actions described in Sections 3.3 and 3.4. Moreover, they remove dead places from local lists to avoid sending messages that are already known to be dead. Note that the handler is invoked, no matter whether the worker is active. At an inactive worker, the `placeFailureHandler` may merge tasks into the local pool. In this case, it restarts the worker activity and binds it to the outer `finish` as explained in Section 4.6.

5.4 PartitionLostListener

We register one `partitionLostListener` on each `IMap` instance within the worker constructor. If a partition of an `IMap` instance gets lost, Hazelcast automatically executes the handler code. In our implementation, it prints an error message. Moreover, it terminates the program by starting an asynchronous activity on each place alive that terminates the respective JVM.

5.5 Thread-Safe Hazelcast Operations

We use two kinds of thread-safe Hazelcast operations to access the two `IMap` instances. Writing backups is performed with the function `executeOnKey()`. This function transfers code to the owner of the respective entry. According to this code, the owner only updates the backup when `done` is `false`. Otherwise, no action is performed.

We implemented our transactions by calling Hazelcast's function `executeTransaction()`. Transactions perform either all or none of the operations passed in a lambda parameter, as has been explained in Section 2.2. Each transaction is surrounded by a `try-catch` block to catch a potentially thrown `TransactionException`.

5.6 Restart-Daemon

When a place crashes, the APGAS runtime invokes all registered `placeFailureHandler`s by starting a new asynchronous activity on each place which executes the handler code. Unfortunately, there is no option for APGAS users to bind this activity to a user-defined `finish`. This may result in several difficulties. In JFT_GLB, a `placeFailureHandler` may have to restart an inactive worker and bind the new worker activity to the outer `finish` for correct termination detection.

Therefore, we implemented a workaround. It deploys a so-called *restart-daemon*, which is executed by an additional asynchronous activity on place 0. We start the daemon in the first line of the outer `finish` block. It runs until the system-wide task pool is empty. The main purpose of the daemon is to restart inactive workers, which got new tasks by a `placeFailureHandler`.

We first added the attribute `restartPlaces` of type `ConcurrentLinkedQueue` to the worker on place 0. If an inactive worker has to be re-started, its ID is inserted into `restartPlaces` by the corresponding `placeFailureHandler`. The restart-daemon cyclically checks `restartPlaces` and, if needed, restarts a worker activity on the corresponding place.

APGAS provides no guarantees when exactly a `placeFailureHandler` is executed, because the execution is subject to Java scheduling. Therefore, it may happen that the outer `finish` is already terminated when a delayed `placeFailureHandler` starts. This case may result in wrong results because several tasks may not have been processed. To avoid such situations, the daemon finishes only when all handlers were run. Therefore, we added an attribute `countHandler` of type `HashMap<Integer, HashMap<Integer, Boolean>>` to the worker on place 0. The first argument, `Integer`, represents the ID of the failed places. The second argument, `HashMap<Integer, Boolean>>`, indicates whether a place has already executed its handler.

For each failure, the first `placeFailureHandler` invoked creates the `countHandler` entry and initializes all HashMap entries with `false`. When a `placeFailureHandler` ends, it writes `true` to its entry as last operation. The daemon runs until all entries are `true`.

This technique still does not avoid the situation that all `placeFailureHandler`s are delayed and all workers have gone inactive. In this situation, the program could finish and the result would be wrong. To solve this problem, the daemon finishes only if the size of `countHandler` plus the number of places alive is equal to the initial number of places at program start.

Finally, the daemon terminates only when `iMapOpenLoot` is empty. It never handles any open loot itself, however, because delayed `placeRemovedHandler`s will do that. So the daemon just waits.

Checking all entries in `iMapOpenLoot` causes much network traffic, but this operation is only executed in the rare case that all workers are inactive and the last place alive crashes.

5.7 Extensions of APGAS

Beyond the implementation of JFT_GLB, we extended the APGAS framework. That framework is open-source, and we submitted our modifications to the official APGAS repository [21]. We added the following three functions, which are well-known from X10 and have facilitated the JFT_GLB implementation:

- `Place nextPlace(id)` returns the next place alive in the ring of places that comes after place `id`.
- `Place prevPlace(id)` returns the previous place alive in the ring of places that comes before place `id`.
- `boolean isDead(id)` returns `true` if the place is alive, and `false` otherwise.

Moreover, during JFT_GLB development, we found a bug in the runtime class which handles the `placeFailureHandler`s and implemented a preliminary fix [32].

6 Correctness

Recall that correctness requires the program to output the correct result or terminate with an error message. At its core, JFT_GLB correctness is established by the correctness guarantees of Hazelcast and APGAS:

- `IMap` entries are safe despite failures. If needed, the `partitionLostHandler` is triggered automatically by Hazelcast, and the program aborts.
- APGAS guarantees that all place failures are recognized and the `placeFailureHandler`s are invoked. A place 0 failure leads to program abort.

In our fault-tolerant algorithm, every `IMap` entry exactly captures the subset of tasks that are assigned to this place at the time of backup writing. This includes:

- *finished tasks*, which are captured by the partial result
- *open tasks*, which are contained in the local pool, and
- *future tasks*, which have not yet been generated but are encoded in the parent task descriptor.

Each task belongs to one of these groups, since backups are written outside task processing.

Tasks are moved between the subsets of different places only during stealing and restore. These moves change multiple `IMap` entries simultaneously, but transactions ensure data integrity despite possible failures.

For stealing, the steal protocol with its handshaking and backup writing on both sides guarantees that the task subsets of victim and thief remain consistent. In particular, the case that failures occur while the loot is in transit is unraveled with the help of the corresponding `iMapOpenLoot` entry.

In restore, exactly one place takes over the failed place's open and future tasks. The partial result is not touched, and therefore the finished tasks stay in the failed place's subset. Moreover, the restore protocol from Section 3.4 guarantees that the loot is taken over by exactly one place. Altogether, in both stealing and restore, each task remains in exactly one place's subset.

The algorithm makes no assumptions on message ordering on system level, but takes care that late messages do no harm. In particular:

- Successive backups of a place are written one after the other, since backup writing is a synchronous Hazelcast operation.

- Late backups from a failed place are refused by inspecting the `done` attribute beforehand.
- A victim will only send out further loot to the same thief, if the previous loot was acknowledged.

Termination of the algorithm follows from the continuity of task processing. Workers only interrupt task processing when they perform protocol operations such as answering a steal request, or restoring a place. All of these operations perform a finite number of actions. When all tasks have been processed, termination is detected by the outer `finish`, according to the lifeline scheme [36].

Beyond theoretical establishment of correctness, we tested our implementation experimentally, by provoking critical situations with `System.exit()` calls, see Section 8.1.

7 Comparison between JFT_GLB and XFT_GLB

As mentioned in the introduction, JFT_GLB and XFT_GLB [12–15] provide similar functionalities. We compare the two frameworks in this section. For *XFT_GLB*, we refer to the most recent version from [13].

A major difference between XFT_GLB and JFT_GLB is use of programming languages X10 and Java, respectively. JFT_GLB's use of Java allows to utilize Hazelcast and especially its `IMap` data structure. Java has the advantage of being widely used. Moreover, JFT_GLB users can write their applications with other JVM-based languages such as Scala [9]. X10 applications can be compiled to Java and C++. However, XFT_GLB only compiles to C++, otherwise unexpected errors occur.

Another important difference between XFT_GLB and JFT_GLB regards the fault-tolerant algorithm. While JFT_GLB relies on a resilient data structure (the `IMap`), XFT_GLB handles all aspects of data backup manually. In particular, the XFT_GLB algorithm must explicitly deal with the case that a backup gets lost after a place crash. JFT_GLB, in contrast, deploys Hazelcast to manage those cases.

Moreover, XFT_GLB must manually monitor the liveness of places, whereas JFT_GLB simply utilizes the APGAS `placeFailureHandler`, which is automatically invoked after a failure.

On the positive side, XFT_GLB has no dependencies on foreign frameworks. Therefore, the algorithm could be designed such that it consistently adopts a clear actor-like structure. In particular, backup-related communication was implemented the same way as stealing-related communication. Nevertheless, the XFT_GLB algorithm is more complex than the JFT_GLB algorithm, which delegates many responsibilities to the underlying Hazelcast layer. Consequently, XFT_GLB is more difficult to maintain and extend than JFT_GLB.

The XFT_GLB algorithm consistently adopts asynchronous communication, which keeps workers responsive. JFT_GLB achieves responsiveness by running multiple concurrent activities on each place. Still, most communication is asynchronous to improve the performance via parallelism between communication and task processing. JFT_GLB uses synchronous communication only for `IMap` accesses, and for re-sending loot after failure.

JFT_GLB has the advantage that the number of backup copies is easily configurable between zero and six, whereas XFT_GLB always uses one copy. The XFT_GLB setting could only be changed with a major redesign of the algorithm. Consequently, JFT_GLB can tolerate more cases of simultaneous place failures than XFT_GLB. As another advantage, Hazelcast automatically re-writes a backup after loss of a copy for which the original entry is still available. In XFT_GLB, that re-writing has to be initiated manually.

The two systems handle partial results in different ways. In XFT_GLB, backup places adopt them after failures. In JFT_GLB, they remain in the failed place's `IMap` entry. Consequently, the final result is computed from the partial results of live places in XFT_GLB, and from all `IMap` entries in JFT_GLB. The JFT_GLB approach is enabled by `IMap` persistence.

Another difference between the two systems regards backup handling during stealings. While JFT_GLB writes backups on both the victim and thief sides, XFT_GLB only writes them at the victim side. On the thief side, instead, it stores the victim's identity. Finally, XFT_GLB writes a single steal backup for multiple steals from the same victim. The JFT_GLB approach is simpler, but has a higher communication volume.

For XFT_GLB, there is a variant with incremental backups that is less flexible, but sometimes more efficient [15]. The integration of incremental backups into JFT_GLB is a subject for future work.

8 Experiments

Experiments were run on a cluster with 12 homogeneous nodes. Each node has 256 GB of main memory and two six-core Intel Xeon E5-2643-v4E5 CPUs [39]. We started up to 144 places, which were assigned to the 12 nodes, starting up to 12 places per node. For instance, 12 places were run on 1 node, 24 places were run on 2 nodes etc. We used X10 release 2.6.1 and APGAS in their latest available revisions from the official repositories (July 6, 2017 [20] and July 6, 2017 [21]). Resilience mode was only switched on for runs with the fault-tolerant framework versions. X10 programs were compiled with Native X10. Java version 1.8.0_131 and gcc version 4.9.0 were used.

As benchmarks, we used Unbalanced Tree Search (UTS) [28], Betweenness Centrality (BC) [16], and NQueens [17]. UTS and BC are included in the standard library of X10 [20] as samples for GLB usage. For BC, we deployed a slightly modified version, which avoids overlong tasks by writing shadow results [14]. NQueens was taken from the HabaneroUPC++ repository [34]. We ported all benchmarks to Java, and NQueens to X10.

UTS dynamically generates a highly irregular tree from node descriptors and counts the number of nodes. BC calculates a centrality score for each node in a graph. NQueens calculates the number of placements of N queens on a $N \times N$ chessboard, so that no two queens threaten each other.

Regarding GLB Usage, the benchmarks have the following characteristics:

- UTS and NQueens start with a single task, from which the others are spawned dynamically. BC creates all tasks statically at the beginning and distributes them evenly, so that each worker is assigned about same number of tasks.
- The result of UTS and NQueens is a single `long` value. The result of BC is a `long` array with one entry for each graph node.
- The sum is used as the reduction operator for all three benchmarks.

Parameters were set as follows:

- UTS: geometric tree shape, branching factor $b = 4$, random seed $s = 19$, tree depth $d = 17$.
- BC: random seed $s = 2$, number of graph nodes $N = 2^{17}$.
- NQueens: number of queens and chessboard size $N = 16$, threshold $t = 10$.
- GLB: $n = 511$ (default).
- XFT_GLB and JFT_GLB: $k = 2048$ (UTS and NQueens) or
  $k = 32768$ (BC), backup count $= 1$.

The different `k` values were adjusted to the steal rates. Since XFT_GLB has a hard-coded backup count of 1, we used the same value in JFT_GLB, for comparability.

8.1 Fault Tolerance Tests

We tested JFT_GLB by provoking place failures with `System.exit()` calls. After each test, we made sure that all places involved performed the correct actions, by inspecting log files. The following situations were considered with each 144 places:

1. Place 2 crashes after processing its first n tasks, but before answering any recorded steal requests.
2. Place 2 crashes before it goes inactive.

3. Place 2 is the victim of a random steal request crashes right after it has extracted loot, saved it in `iMapOpenLoot`, and wrote a steal backup to `iMapBackup`.
4. Like situation 3, but place 2 crashes after sending the loot to the thief.
5. Like situation 3, but for the case of a lifeline steal request.
6. Like situation 3, but additionally, the thief crashes when it receives the loot from the backup place (place 1) during recovery.
7. Like situation 6, but place 1 (backup place of crashed place 2) crashes during handling the thief crash from situation 6.
8. Place 2 crashes during task processing, approximately in the middle of the overall computation.
9. Place 2 crashes after merging received loot into its queue and writing the corresponding steal backup to `iMapBackup`, but before sending the `loot received` message to the victim.
10. Like situation 1, but we modified, the program so that the backup place waits at the beginning of its `placeFailureHandler()` until the rest of the computation has finished. This way, we test the restart-daemon.
11. Like situation 5, but additionally, place 1 (the backup place) crashes inside the `merge()` call.
12. Like situation 1, but 90% of the places crash. This case provokes program abort and tests the detection of unrecoverable situations.

8.2 Performance Measurements

We evaluated the performance of the fault tolerance schemes chiefly by measuring their overhead in comparison to non-fault-tolerant program versions. Figures 3 (UTS), 4 (NQueens) and 5 (BC) depict the overhead. The overhead is expressed as a percentage and corresponds to the ratio of wall-clock times (decremented by one). In all figures, the horizontal axis ends at 144.

For UTS on up to 144 places, the overhead of JFT_GLB is at most 12.87%. From 12 places (maximum of one node) upwards, it slightly increases with the number of places. The overhead of XFT_GLB is higher than that of JFT_GLB, in most cases. Exceptions occur at 4, 132 and 144 places. For higher place counts, the XFT_GLB is quite constant. However, on 12 places, XFT_GLB has a surprisingly high peak with 77.54% overhead. In other experiments, we discovered that the source of this peak is fully loading each 12 core node by starting 12 places. When we start 12 places on two nodes, the peak disappears. This behavior is currently under investigation.

For NQueens, the overhead of JFT_GLB is at most 46.06% on 144 places. It is always lower than the XFT_GLB overhead, except on 144 places. The overhead of XFT_GLB is at most 48.34% on 120 places. Both overheads slightly increase with the number of places: for XFT_GLB from 4 places upwards, and for JFT_GLB from 12 places upwards.

For BC, the overhead of JFT_GLB is at most 25.40% on 144 places. It is mostly smaller than the overhead of XFT_GLB, which is at most 53.00% on 144 places. The overhead of XFT_GLB has a high peak on 12 places, which apparently has the same reason as for UTS. Moreover, on 2 places, there is an additional peak. Both overheads increase with the number of places, the increase is higher for XFT_GLB.

Overall, there is no clear winner between XFT_GLB and JFT_GLB in direct comparison. However, on average JFT_GLB has less overhead than XFT_GLB. So, the simpler JFT_GLB approach of relying on a resilient data structure appears to be preferable.

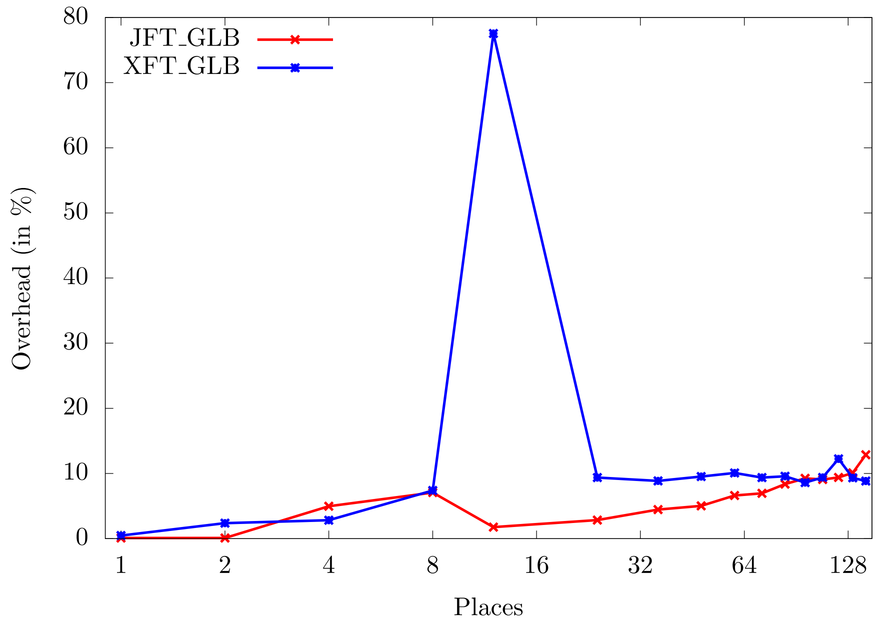

Figure 3: UTS: Fault Tolerance overhead of XFT_GLB and JFT_GLB

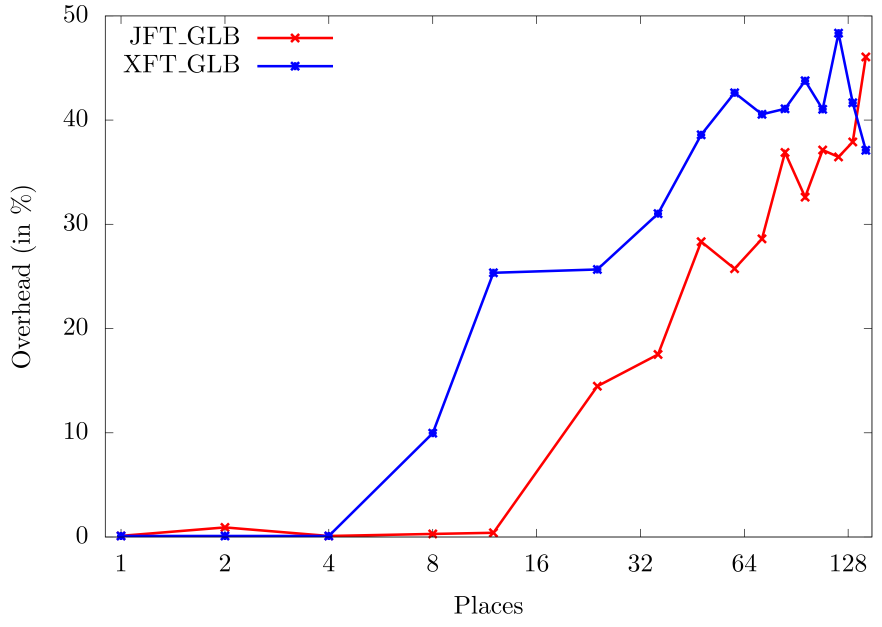

Figure 4: NQueens: Fault Tolerance overhead of XFT_GLB and JFT_GLB

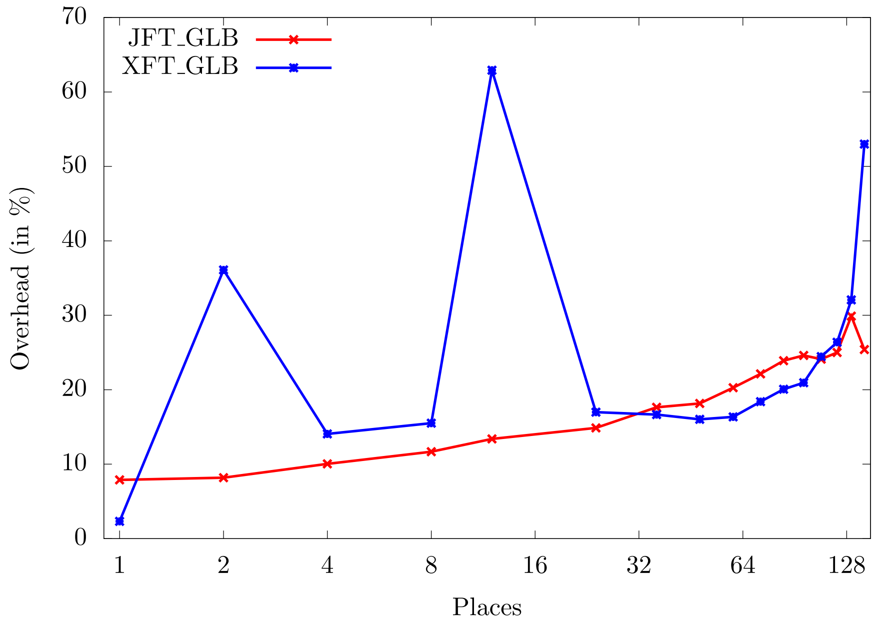

Figure 5: BC: Fault Tolerance overhead of XFT_GLB and JFT_GLB

8.3 Performance Analysis

To analyze the measured performance, we enhanced our previous logging feature from [31]. The logged data are plotted as phase diagrams in Figures 6 (UTS), 7 (NQueens) and 8 (BC). Parameters are the same as before. The phase diagrams refer to experiments with 144 places on 12 nodes. J_GLB is shown on the left side and JFT_GLB on the right side. Each phase diagram depicts the percentage of workers that are in the following states:

- *processing*: worker processes tasks.
- *idling*: worker is inactive.
- *communicating*: worker communicates with another worker by writing backups, sending steal requests, answering steal requests, or receiving tasks.
- *waiting*: worker waits for an answer to a steal request.

The UTS phase diagrams in Figure 6 reveal a longer startup phase for JFT_GLB than for J_GLB. Here, the idling and communicating states are more prominent because the steal backups during initial work stealing consume time. Similarly, JFT_GLB takes longer towards the end of the computation, where the steal rate increases. In the main phase, the communicating and waiting states take somewhat more time in JFT_GLB .

The NQueens phase diagram in Figure 7 has similar characteristics as that of UTS, except that the communicating and waiting states in the main phase are less prominent.

The BC phase diagrams in Figure 8 does not show a longer startup phase, which is due to the static initial task distribution. Since BC's result is an array, backup writing takes longer and therefore more time is spent in state communicating. The higher communication expense can be observed especially in the end phase, and also leads to periodic peaks during the main phase.

Overall, the phase diagrams depict the expected characteristics of each case.

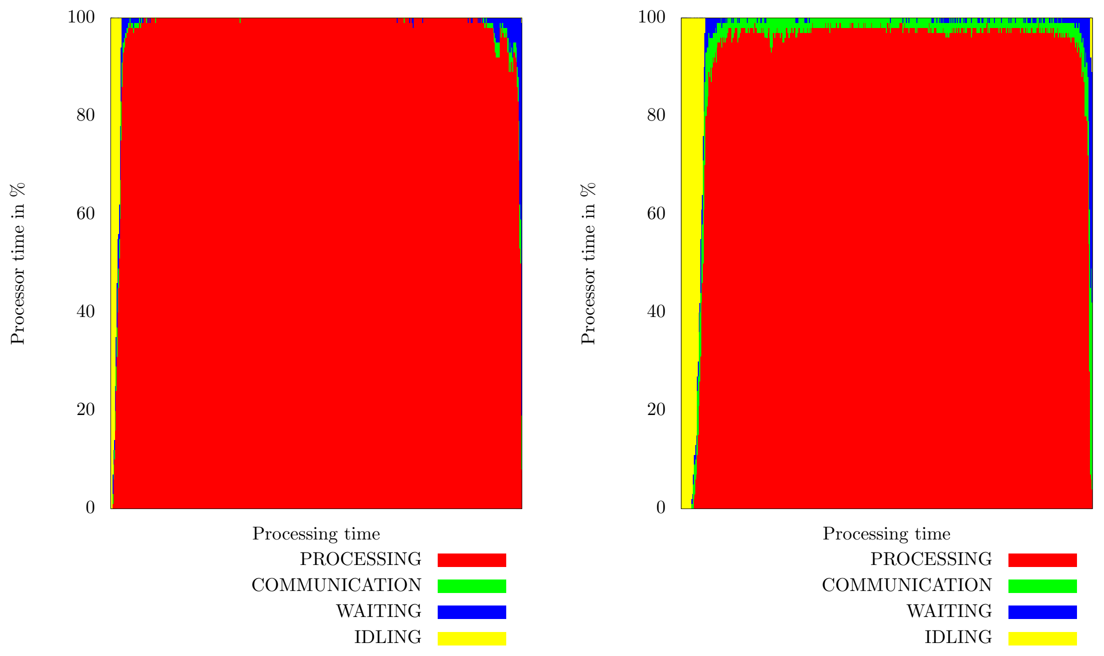

Figure 6: Phase Diagram for UTS with J_GLB (left) and JFT_GLB (right)

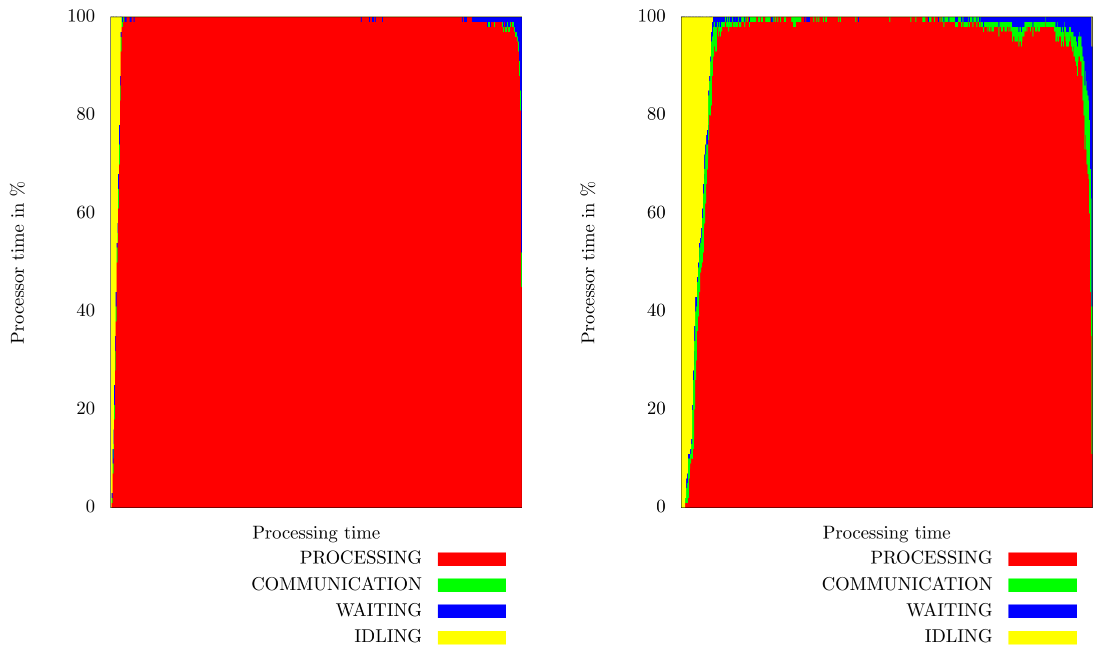

Figure 7: Phase Diagram for NQueens with J_GLB (left) and JFT_GLB (right)

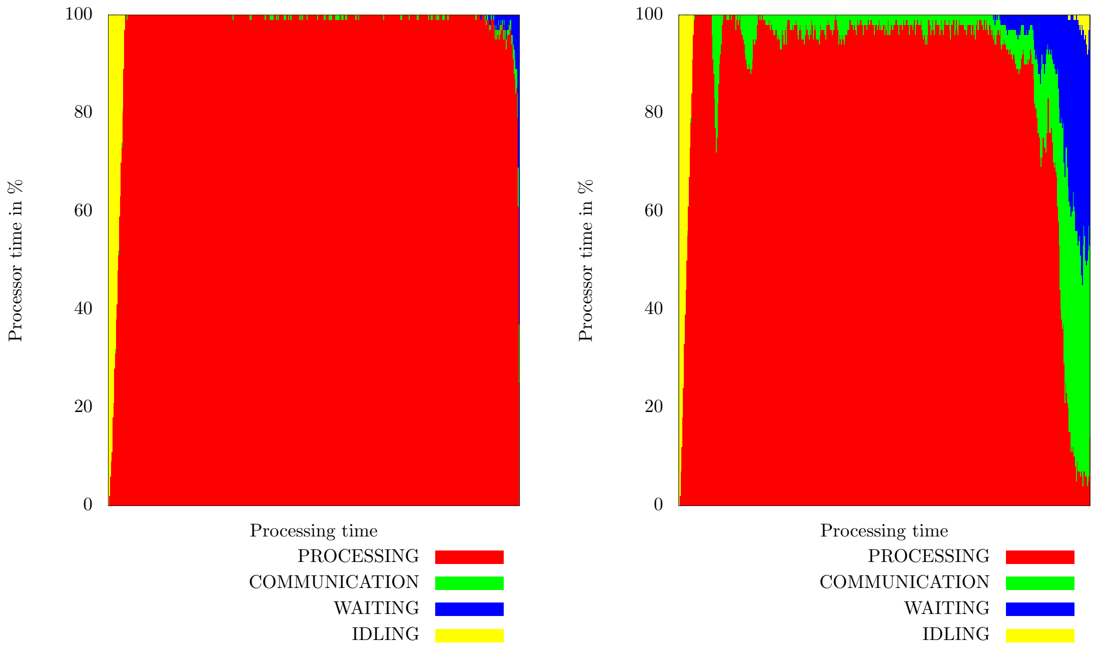

Figure 8: Phase Diagram for BC with J_GLB (left) and JFT_GLB (right)

8.4 Extra Experiments

In further experiments with JFT_GLB, we increased the backup count from one to six, which is currently the largest value supported by Hazelcast. For all three benchmarks on 144 places, there was no measurable increase in the execution time.

Moreover, we measured the time for handling a place failure. We let 14 out of 144 places crash by calling `System.exit()` after all tasks have been processed, but before the global reduction began. For all three benchmarks, we found the average time for handling a single place failure to be negligible.

9 Related Work

Fault tolerance research receives more and more attention, see e.g. [18]. As mentioned in Section 1, an established technique on system-level is checkpoint/restart. This technique is gradually improved and may achieve an overhead of 8% [2]. An extensive analysis of this technique is given in [3], but it is expected that checkpoint/restart will not be efficient enough for next generation clusters [11].

Fault tolerance on application-level is supported by an increasing number of programming systems, e.g.ULFM [4] and Resilient X10 [6]. Programs may, for instance, exploit redundancy in matrix computations, called algorithm-based fault tolerance (ABFT) [1].

The official APGAS repository contains a fault-tolerant UTS variant. Like our algorithm, it utilizes an `IMap` instance. However, it introduces a restricted stealing scheme [21] in which the lifeline graph has dimension of one. We are not aware of any other previous work on resilient APGAS applications.

There are several resilient data structures around. For instance, the Terracotta framework [38] includes a `ConcurrentDistributedMap`, which has similar feature as the Hazelcast `IMap`.

The `IMap` of Hazelcast does not appear to be widely used or at least the usage is not documented. Instead, Hazelcast has been mostly utilized to connect JVMs and distribute storage across them [8, 29]. Kathiravelu *et al.* [23] introduce a framework for detecting and deleting duplicates in the big data context, which utilizes Hazelcast's MapReduce support. The distributed simulator Cloud2Sim [22, 24] deploys Hazelcast for simulating cloud and MapReduce algorithms.

Load balancing and traditional task pools without fault tolerance have been researched intensely, see e.g. [19, 26, 30, 33]. Previous work on fault-tolerant task pools includes recovery from silent errors [40], coping with side effects [27], and fault-tolerant work stealing semantics [41].

Our work shares some similarity with fault tolerance for nested fork-join programs [5,25]. These papers consider strict computations, in which children return a result to their parent, whereas our task pool computes the final result by reduction. In our approach, parent tasks are discarded after having generated their children, and therefore they can not help in recovery. Like ours, the algorithm in [25] is able to recover from multiple simultaneous failures, except failure of place 0.

The related fault-tolerant task pool framework for X10 has already been discussed in Section 7.

10 Conclusions

This paper has introduced a fault-tolerant task pool algorithm and its implementation in a generic reusable framework. The algorithm presupposes the existence of a resilient data structure and is therefore simpler than a previous fault-tolerant algorithm for X10 [13]. Partly due to the new algorithm, our framework has several advantages over the corresponding X10 framework. In particular, it is less complex, easier to maintain, configurable, robust against multiple simultaneous failures, and available for a mainstream language.

A core idea of our algorithm is writing backups of local pools into the resilient data structure in an uncoordinated way. The framework uses Hazelcast's automatically distributed and fault-tolerant `IMap` for this purpose. Backups are written regularly, and during work stealing.

Our algorithm computes the correct result despite one or multiple place failures. In rare cases, it aborts with an error message. In experiments, we observed an overhead of at most 46.06% in comparison to a non-fault-tolerant base variant. The overhead is comparable to that of the X10 framework.

Our framework is probably among the first programs ever written that implements application-level fault tolerance with a combination of APGAS resilience and the Hazelcast `IMap`. We found this environment to be effective and user-friendly.

Acknowledgments

This work is supported by the Deutsche Forschungsgemeinschaft, under grant FO 1035/5-1. Experiments were carried out on the High Performance Computer of University of Kassel, Germany.

References

[1] Nawab Ali, Sriram Krishnamoorthy, Mahantesh Halappanavar, et al. Multi-fault tolerance for cartesian data distributions. *Int. Journal of Parallel Programming*, 41(3):469–493, 2013.

[2] Leonardo Bautista-Gomez, Dimitri Komatitsch, Naoya Maruyama, et al. FTI: high performance fault tolerance interface for hybrid systems. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[3] Anne Benoit, Loc Pottier, and Yves Robert. Resilient application co-scheduling with processor redistribution. In *Proc. Int. Conf. on Parallel Processing*, pages 123–132, 2016.

[4] Wesley Bland, George Bosilca, Aurelien Bouteiller, et al. A proposal for user-level failure mitigation in the MPI-3 standard. `http://icl.cs.utk.edu/news_pub/submissions/mpi3ft.pdf`.

[5] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proc. USENIX Annual Technical Symp.*, 1997.

[6] David Cunningham, David Grove, Benjamin Herta, et al. Resilient X10: Efficient failure-aware programming. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 67–80, 2014.

[7] J. Diaz, C. Muoz-Caro, and A. Nio. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Trans. on Parallel and Distributed Systems*, 23(8):1369–1386, Aug 2012.

[8] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Elasticon: An elastic distributed sdn controller. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 17–28. ACM, 2014.

[9] École Polytechnique Fédérale Lausanne (EPFL). The scala programming language. `https://www.scala-lang.org/`, 2017.

[10] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[11] E.N. Elnozahy, Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godf, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, James Plank, Partha Ranganathan, and Josh Simons. System resilience at extreme scale. Technical report, DARPA, 2008.

[12] Claudia Fohry and Marco Bungart. A robust fault tolerance scheme for lifeline-based taskpools. In *Int. Conf. on Parallel Processing Workshops (P2S2)*, pages 200–209, 2016.

[13] Claudia Fohry, Marco Bungart, and Paul Plock. Fault tolerance for lifeline-based global load balancing. *Parallel Computing*, 2017. To appear.

[14] Claudia Fohry, Marco Bungart, and Jonas Posner. Fault tolerance schemes for global load balancing in X10. In *Scalable Computing: Practice and Experience*, volume 16, pages 169–185, 2015.

[15] Claudia Fohry, Marco Bungart, and Jonas Posner. Towards an efficient fault-tolerance scheme for GLB. In *Proc. ACM SIGPLAN X10 Workshop*, pages 27–32, 2015.

[16] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[17] J. Gik. *Schach und Mathematik*. Deutsch Harri GmbH, Frankfurt a. M., 1987.

[18] Thomas Herault and Yves Robert, editors. *Fault-tolerance techniques for high-performance computing*. Springer, 2015.

[19] Ralf Hoffmann and Thomas Rauber. Adaptive task pools: Efficiently balancing large number of tasks on shared-address spaces. *Int. Journal of Parallel Programming*, 39(5):553–581, 2011.

[20] IBM. Core implementation of X10 programming language including compiler, runtime, class libraries, sample programs and test suite. `https://github.com/x10-lang/x10`, 2017.

[21] IBM. The APGAS library for fault-tolerant distributed programming in Java 8. `https://github.com/x10-lang/apgas`, 2017.

[22] Kathiravelu, Pradeeban, and Luis Veiga. Concurrent and distributed cloudsim simulations. In *Proceedings of the 2014 IEEE 22Nd Int. Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 490–493, Washington, DC, USA, 2014. IEEE Computer Society.

[23] Pradeeban Kathiravelu, Helena Galhardas, and Luís Veiga. $\partial u \partial u$ multi-tenanted framework: Distributed near duplicate detection for big data. In *Proceedings of the Confederated Int. Conferences on On the Move to Meaningful Internet Systems: OTM 2015 Conferences*, volume 9415, pages 237–256. Springer-Verlag New York, Inc., 2015.

[24] Pradeeban Kathiravelu and Luis Veiga. An adaptive distributed simulator for cloud and mapreduce algorithms and architectures. In *Proceedings of the 2014 IEEE/ACM 7th Int. Conference on Utility and Cloud Computing*, UCC '14, pages 79–88. IEEE Computer Society, 2014.

[25] Gokcen Kestor, Sriram Krishnamoorthy, and Wenjing Ma. Localized fault recovery for nested fork-join programs. In *Proc. Int. Parallel and Distributed Processing Symposium*, pages 397–408, May 2017.

[26] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[27] Wenjing Ma and Sriram Krishnamoorthy. Data-driven fault tolerance for work stealing computations. In *Proc. ACM Int. Conf. on Supercomputing*, pages 79–90, 2012.

[28] Stephen Olivier, Jun Huan, Jinze Liu, et al. UTS: An Unbalanced Tree Search benchmark. In *Proc. Workshop on Languages and Compilers for High-Performance Computing*, pages 235–250. Springer LNCS 4382, 2006.

[29] Vishal Pachori, Gunjan Ansari, and Neha Chaudhary. Improved performance of advance encryption standard using parallel computing. In *Int. Journal of Engineering Research and Applications (IJERA)*, volume 2, pages 967–971. Springer-Verlag New York, Inc., 2012.

[30] Swann Perarnau and Mitsuhisa Sato. Victim selection and distributed work stealing performance: A case study. In *Proc. IEEE Int. Parallel and Distributed Processing Symp.*, pages 659–668, 2014.

[31] Jonas Posner and Claudia Fohry. Cooperation vs. coordination for lifeline-based global load balancing in APGAS. In *Proc. ACM SIGPLAN X10 Workshop*, pages 13–17, 2016.

[32] Jonas Posner and Olivier Tardieu. Unexpected NullPointerException from an at in an placeFailureHandler. `https://sourceforge.net/p/x10/mailman/message/35413843/`, 2016.

[33] Kaushik Ravicandran, Sangho Lee, and Santosh Pande. Work stealing for multi-core HPC clusters. In *Proc. Euro-Par*, pages 205–217. Springer LNCS 6852, 2011.

[34] Rice University. HabaneroUPC++: a Compiler-free PGAS Library. `https://github.com/habanero-rice/habanero-upc`, 2017.

[35] Vijay Saraswat, George Almasi, Ganesh Bikshandi, et al. The asynchronous partitioned global address space model. Technical report, IBM, Toronto, Canada, 2010.

[36] Vijay Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, et al. Lifeline-based global load balancing. In *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, pages 201–212, 2011.

[37] Olivier Tardieu. The APGAS library: Resilient parallel and distributed programming in Java 8. In *Proc. ACM SIGPLAN X10 Workshop*, pages 25–26, 2015.

[38] Terracotta. In-memory data management for the enterprise. `http://www.terracotta.org/`, 2017.

[39] University of Kassel. Scientific data processing. `https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html`, 2017.

[40] Yizhuo Wang, Weixing Ji, Feng Shi, and Qi Zuo. A work-stealing scheduling framework supporting fault tolerance. In *Proc. Design, Automation and Test in Europe*. EDA Consortium / ACM DL, 2013.

[41] Mustafa Zengin and Viktor Vafeiadis. A programming language approach to fault tolerance for fork-join parallelism. In *Int. Symp. on Theoretical Aspects of Software Engineering*, pages 105–112, 2013.

[42] Wei Zhang, Olivier Tardieu, David Grove, et al. GLB: Lifeline-based global load balancing library in X10. In *Proc. ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, pages 31–40, 2014.