UNI KASSEL
VERSITÄT

Universität Kassel

Master Thesis

Design and Evaluation of a Work Stealing-Based Fault Tolerance Scheme for Task Pools

presented to
Department of Electrical Engineering and Computer Science
Research Group Programming Languages/Methodologies

Lukas Reitz
33211934

Kassel, November 12, 2019

Examiners:
Prof. Dr. Claudia Fohry
Prof. Dr. Albert Zündorf

Contents

Statutory Declaration

I declare on oath that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been published or presented to an examination committee.

Kassel, November 12, 2019
Lukas Reitz

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABFT	Application-based fault-tolerance
MTBF	Mean Time Between Failures
GLB	Global Load Balancing framework [38]
AllFT	Application-level fault-tolerance scheme using checkpointing [28]
AllFTGLB	The implementation of AllFT for GLB [25]
DistLogFT	The new scheme introduced in this thesis
DistLogFTGLB	The implementation of DistLogFT for GLB
PGAS	Partitioned Global Address Space model
APGAS	Asynchronous Partitioned Global Address Space library [34]
UTS	Unbalanced Tree Search benchmark [23]
N-Queens	N-Queens benchmark [12]
MSB	Most Significant Bit

1 Introduction

Computations on parallel systems can experience failures. If non-fault-tolerant computations take significantly longer than the Mean Time Between Failures (**MTBF**) of the parallel system, they will most likely not complete. This wastes energy and makes such computations unviable. With the increasing size of parallel systems, the necessity of fault-tolerance is becoming more apparent.

In this thesis only fail-stop failures are considered, i.e., a failed process quits the computation completely. We assume that communication to a failed process returns an error and all processes are eventually notified of all failures. Also, we assume a *shrinking recovery*, where no new processes are started after failures. The computation continues with fewer processes than initially started.

The most common approach to fault-tolerance is *checkpoint/restart*, where a valid state of the computation is saved periodically in memory or on disc [10]. Then, upon failure, the computation is restarted from the saved checkpoint. There are two major variants of checkpointing. In **system-level checkpointing**, the checkpoint contains all data of the respective process. Thus, the application is restarted by loading this last saved state. The application usually does not need to know about the checkpointing, thus making this an easy way of adding fault-tolerance to existing programs. In **application-level checkpointing**, the checkpoint contains only selected data, which are sufficient to restore the computation [16]. This variant has a smaller backup volume and a potentially lower fault-tolerance overhead than system-level checkpointing, because less data has to be written and read. The downside is an increased burden on the application programmer.

Another distinction on checkpointing can be made between coordinated and uncoordinated checkpointing. In **coordinated** checkpointing, processes write the checkpoint in a coordinated way. This requires communication between the processes to organize the writing of the checkpoint. In **uncoordinated** checkpointing, every process writes its own checkpoint independently from other processes.

A popular approach to parallelization is to divide a computation into subcomputations, called *tasks*, which are processed by *workers*. This is called the *task pool pattern*. Each worker maintains a local pool, which contains tasks to be processed. The load-balancing of the local pools is realized by utilizing either work sharing or work stealing [5]. In **work sharing**, workers with full pools send tasks to workers with empty pools. In **work stealing**, workers with empty pools steal tasks from other workers. This thesis focuses on *cooperative work stealing*, where a worker, called *thief*, asks a co-worker, called *victim*, for tasks. The victim can either reject the request or answer with tasks.

Task pool variants can be divided by the way the final result is calculated. In this thesis, we focus on reduction-based task pools. Here, the final result is calculated by an associative and commutative operator, called the *reduction operator*. Each task has a result and the overall result of the computation is calculated by reduction from all task results.

One implementation of a reduction-based task pool is the Global Load Balancing (**GLB**) framework [38]. GLB operates in a Partitioned Global Address Space (**PGAS**) setting. In PGAS, the parallel system is divided into units, called *places*. Each place consists of a part of system memory and processing units. Usually a place corresponds to one cluster node. In all GLB variants considered in this thesis one worker is executed on each place.

The original GLB framework is implemented in the parallel programming language X10 [17]. Later work ported the GLB framework to Java by using the "APGAS for Java" parallel programming library (**APGAS**) [26, 34]. This library implements the Asynchronous Partitioned Global Address Space model [31]. The model extends the PGAS model by introducing tasks, called *activities*. Activities can be spawned locally or remotely and in a synchronous or asynchronous way. The execution of an APGAS program begins on place 0 from which new activities can be spawned on other places to spread the computation across the parallel system. For the experiments a fork of the APGAS library was used. It contains bug fixes and additional features. The fork is available as an online repository [24]. GLB was ported in two variants. The variants differ from each other by their load-balancing mechanism. In this thesis, *GLB* refers to the variant utilizing cooperative work stealing as the load-balancing mechanism.

GLB deploys the lifeline scheme [30] for termination detection. Workers with empty pools contact up to w random victims. If the random victims do not have tasks to share, the worker contacts up to z *lifeline buddies*. Lifeline buddies are neighbours of the worker in a connected graph. If neither of them have tasks to share, the worker becomes inactive. If all workers are inactive, the final result is computed from all worker results by reduction and the computation is completed.

An application-level fault-tolerance scheme for GLB was introduced in Reference [25], and is called *AllFT*. Each worker periodically saves the local pool contents with the current worker result in a resilient store. A backup is also written in the event of work stealing on the thief's side and the victim's side, during recovery, and before and after the computation. The current pool contents and the worker result are sufficient to save the state of the computation. The saved state has to include past tasks, current tasks and future tasks. The current tasks are the tasks in the local pool. Past tasks are reflected in the worker result. Future tasks will be generated from the current tasks. Upon failure of a worker, a specified recovery worker combines its own current result with the failed worker's result using the reduction operator. The recovery worker also inserts the tasks of the most recent backup into its local pool. The scheme was first described as a fault-tolerance scheme for GLB, but was later refined to be task pool variant-independent in Reference [28]. The implementation of AllFT from this refined variant is called AllFTGLB and is used in the experiments of this thesis.

Another popular parallelization pattern is fork-join. In this pattern, the computation is split into subcomputations recursively (*fork*). The parent computations wait for their children (*join*), which return their results to their parent.

Several fault-tolerance schemes for fork-join programs were proposed [6, 19, 37]. In particular, Reference [19] presents a scheme which adds fault-tolerance to work stealing based programs and does not involve checkpoint/restart. GLB programs differ from fork-join programs in the following aspects. In fork-join, task results influence results of other tasks due to the nature of fork-join. In GLB, task results do not influence results of other tasks and the final result is calculated by reduction. Also, in fork-join, tasks usually follow the *work-first* policy. When a worker executes the fork operation, the current task is pushed to the task queue and can be stolen. The worker proceeds to process the newly forked child task. In GLB, in contrast, tasks follow the *help-first* policy. Newly generated tasks are pushed to the task queue, from which they can be stolen [14]. Furthermore, in distributed fork-join programs with work stealing, a worker usually steals the oldest task of another worker, as it is likely to fork a high number of child tasks. This is different to GLB, where a block of tasks is stolen.

This thesis presents the design and evaluation of an adaption of a fault-tolerance scheme from Reference [19]. The original scheme, called **ForkJoinFT**, adds fault-tolerance to distributed fork-join based programs. The adaption, called **DistLogFT**, applies the idea of ForkJoinFT to GLB. ForkJoinFT does not send any additional messages during failure-free execution, which is a major advantage over schemes based on in-memory checkpointing. Upon a failure, only tasks stolen by the failed worker are restored. Tasks of alive workers which were stolen from the failed worker are not reprocessed, avoiding unnecessary work. ForkJoinFT **tracks work stealing** to decide which tasks have to be restored. In-memory checkpoint/restart algorithms can typically only tolerate up to a certain number of simultaneous crashes due to a limited number of backup replications. In contrast, DistLogFT and ForkJoinFT can recover from simultaneous failure of all workers except one. The implementation of DistLogFT for GLB is called DistLogFTGLB.

We compare two techniques for achieving fault-tolerance. The first technique, application-level checkpointing, benefits of the simplicity of checkpointing and a fast recovery. On the backside, the writing of checkpoints costs time and involves slow communication. The second technique, tracking of work stealing, has the upside of not needing additional communication during failure-free execution. A downside is the complicated recovery. For the comparison, execution times of two benchmarks were measured for GLB, AllFTGLB and DistLogFTGLB, and the relative runtime increase to GLB was denoted as the fault-tolerance *overhead*.

The experiments confirmed fault-tolerance overheads of AllFTGLB from earlier experiments of Reference [28]. The fault-tolerance and recovery overhead of AllFTGLB is small. DistLogFTGLB showed an even lower fault-tolerance overhead on average than AllFTGLB, but at the cost of a higher recovery overhead.

This thesis is organized as follows. Chapter 2 describes the idea of the original scheme. Then, Chapter 3 explains the adaption. Next, Chapter 4 provides implementation details. Then, Chapter 5 describes the experiments and discusses results. In Chapter 6, related work is presented. The thesis finishes with conclusions in Chapter 7.

2 Original Fault-Tolerance Scheme

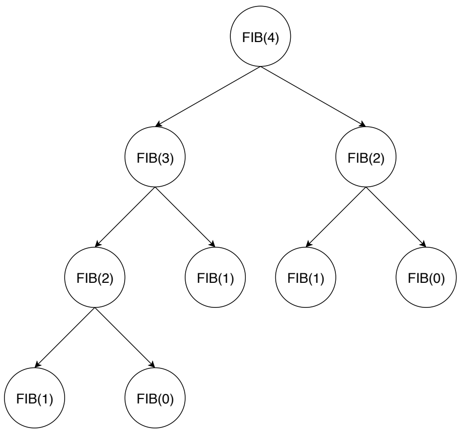
This chapter outlines the idea of ForkJoinFT from Reference [19]. First, Section 2.1 presents the general idea of the scheme. Then, Section 2.2 shows the application of the idea to distributed fork-join programs. Finally, Section 2.3 sketches the recovery of failures.

2.1 Idea

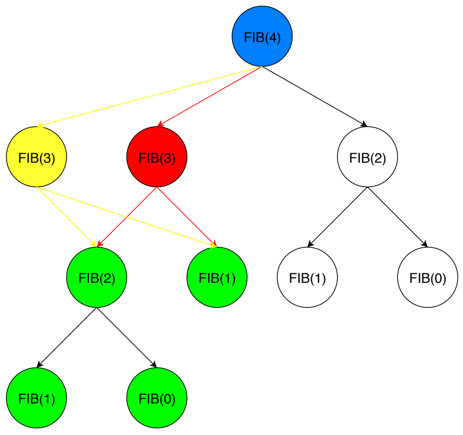
This section presents the basic idea of ForkJoinFT from a computation tree perspective. A simple fork-join program for illustration purposes is the recursive computation of Fibonacci numbers as shown in Figure 2.1.

```
function FIB(n)
  if n < 2
    RETURN n
  a = fork FIB(n - 1)
  b = fork FIB(n - 2)
  join
  RETURN a + b
```

Listing 2.1: Naive recursive Fibonacci program (adopted from Reference [19])



(a) Normal execution



(b) Execution with a lost subcomputation

Figure 2.1: Computation tree of FIB(4)

At each `fork` statement, a new subcomputation is created. When a subcomputation is finished, its result is returned to the parent subcomputation. Thus, a call to FIB(4) creates two subcomputations, which calculate FIB(3) and FIB(2), respectively. Subcomputations can be visualized by the computation tree in Figure 2.1a. Each vertex in the tree corresponds to one subcomputation and edges denote parent-child relations. Each subcomputation returns its result upwards in the tree. The result of the root subcomputation is the final result.

Each subcomputation has a location, e.g., it is saved as a task in the main memory of a computer or it is temporarily saved on a disc. The location of a subcomputation can be found out at any time. A subcomputation whose location is not accessible anymore will be called *lost subcomputation*.

Figure 2.1b shows the same tree as Figure 2.1a but the red subcomputation is lost. To complete the overall computation, a subcomputation identical to the lost subcomputation is created on another location. At first, this results in an unconnected graph. The direct children of the lost subcomputation have to be located and edges in the computation tree have to be added to connect the new subcomputation to the children of the lost subcomputation. The parent of the lost subcomputation has to be connected to the new subcomputation, too. Old edges have to be removed. Figure 2.1b shows the new edges.

A naive fault-tolerance scheme could just reprocess the red subcomputation including the green subtrees. Such a scheme would be correct, but it would not be efficient, as the green subtrees would be processed twice. ForkJoinFT only reprocesses lost subcomputations and reconnects alive subcomputations (yellow edges). The lost subcomputation is reprocessed (yellow subcomputation), but the green subtrees are not. The basic idea of the scheme is to only reprocess the lost subcomputations and to not reprocess alive child subcomputations.

2.2 Extension to Distributed Fork-Join Programs

The above idea was applied to distributed fork-join programs using work stealing for load balancing in Reference [19]. We will call such programs *ForkJoin*. In ForkJoin, program executions are divided into steal and work phases. A steal phase begins for a worker when its local pool becomes empty. A work phase begins after a successful steal. Instead of a computation tree perspective, the scheme is based on a steal tree perspective. A steal-tree is a tree with subcomputations as vertices and steal relations as edges [22]. In the steal phase, the worker steals the oldest subcomputation from a random victim. In the work phase, the worker processes subcomputations until its local pool becomes empty.

Each time a subcomputation is stolen, the subcomputation is assigned a unique identifier, called *ID*. The ID contains the level and number of preceding children in the computation tree. If the computation is repeated, the ID corresponds to the same subcomputation.

Each stolen subcomputation has a *history* of steals of parent subcomputations. The history is a path in the corresponding steal-tree. The path is used in recovery to determine if the subcomputation belongs to the subtree below the failed subcomputation. This history is saved for each stolen subcomputation and is used during recovery to identify which parts of the subcomputation need or need not be reprocessed. Recovery is described in Section 2.3.

In order to identify the steals that preceded a steal from the same victim in the same work phase, a list is maintained by each worker which contains the stolen steps of all preceding steals from the same parent. From this list it is possible to identify for the i-th child that there were $i-1$ other children who stole from the same parent even though only the i-th child is alive. This list is used to reconstruct missing parts of the steal tree (see the last step of Recovery in Section 2.3).

Additionally, each worker maintains a datastructure which contains stolen subcomputations whose results were not received yet. The datastructure is used to determine which subcomputations are lost due to the failure.

2.3 Recovery

The recovery of the scheme can be sketched as follows:

1. Determine the lost subcomputations.

   Each worker searches the lost subcomputations in the aforementioned datastructure containing the subcomputations which were stolen by other workers.

2. Determine the subtree of the lost subcomputation.

   The subtrees are collected by a distributed binary-tree-based reduction. Since they all contain the lost subcomputation, they can be merged to a so-called *replay tree* using the lost subcomputation as the root.

3. Determine a new worker to process the lost subcomputation.

   Each worker who has a stolen subcomputation which is lost due to the failure, constructs a replay unit containing the replay tree, the frame of the lost subcomputation and the subcomputation's history.

4. Distribute the replay unit.

The replay units will be sent to the next thieves in work stealing.

5. Reconnect the new worker to the parent of the lost subcomputation.

   The thief of the replay unit sends a message to the parent worker which informs the parent about the changed child.

6. Reconnect the subtree to the new worker.

   The thief of the replay unit sends a message to each direct child to inform it about the changed parent.

7. Reprocess the lost subcomputation.

   The thief reprocesses the lost subcomputation. The live parts of the subcomputation are omitted, if corresponding steals exist in the replay tree.

   If a child of the subcomputation was stolen by a worker, but the steal does not exist in the replay tree, a new replay unit is constructed to start another recovery for this child. This makes sure that several lost subcomputations that are directly connected get recovered, too. Steals that are not existing in the replay tree are still known, because of the aforementioned list of preceding steals. Since every worker, who stole a subcomputation from the same parent, knows about preceding steals of parts of the same parent subcomputation, missing parts of the steal tree can be constructed from the longest list of the worker who stole the last child subcomputation.

   If a child was stolen by a worker and is found in the replay tree, the child subcomputation can be discarded, as it is still available on another worker.

   During recovery, the lost subcomputation can not be stolen by any worker, even if the recovering worker processes a forked subcomputation of the lost subcomputation.

This results in a recovery of lost subcomputations.

3 Adapted Fault-Tolerance Scheme

This chapter explains the adapted scheme in detail. Chapter 2 described the fault-tolerance scheme for distributed fork-join programs. In this chapter, the scheme is first described in a generic way for task pools. Then, in Section 3.2, the necessary changes to apply the scheme to GLB are discussed. Thereafter, in Section 3.3, further details about the scheme's tracking of work stealing events are provided. Next, Section 3.4 describes the recovery. The chapter finishes with a discussion of special cases in Section 3.5.

3.1 Task Pool Constraints

DistLogFT imposes the following constraints on the task pool variant:

(C1) Workers periodically enter a communication phase after processing a fixed number of tasks. Communication is prohibited while not in this phase. No tasks are processed during the communication phase.

(C2) For each worker, all tasks in its local pool must originate from the same *task bag*, which is loot with additional information (see Chapter 3.3).

(C3) All operations that modifiy the local pool contents must be deterministic. This includes the order of tasks processed and inserted into the task queue. Repeated execution of a sequence of operations on a local pool must always yield the same resulting pool.

Constraint C1 can be fulfilled by a restriction of the work stealing variant. Any task pool variant that uses cooperative work stealing can fulfill this restriction, because the time when to answer steal requests is flexible. Constraint C2 is easily fulfilled by a steal protocol, where a steal request is only sent if the local pool is empty, and the next steal request is only sent, if the preceding request was rejected. Constraint C3 can be fulfilled by any task pool variant, but can lower efficiency. We focus on a task model, where tasks do not have side effects. Tasks are always taken for processing from the end of the task queue and tasks are inserted at the end. If a task generates several new tasks, the tasks are always inserted in the same order.

3.2 Changes to GLB

The GLB variant considered in this thesis already uses cooperative work stealing and only communicates before and after task processing. Therefore, constraint C1 is already fulfilled. Also, the order of task insertions and task processing is already defined in GLB, such that it does not collide with constraint C3. Tasks do not have side effects and the insertion of new tasks always happens in the same order.

To fulfill C2, a worker must only steal tasks when the local pool is empty. Since lifeline steals can be answered at any point in time, tasks can arrive in GLB even though the pool is not empty. For DistLogFT, the lifeline steal protocol was changed to prevent such task arrivals. The new protocol (depicted in Figure 3.1) is as follows:

1. The thief sends a lifeline steal request to the victim.
2. Eventually, the victim answers the steal request with a lifeline steal delivery request if the victim has tasks to share.
3. If the thief still has an empty pool, the thief will decline any other lifeline steal delivery requests until the tasks are received. The thief answers the request, after waiting for any ongoing random steals, with a message indicating that the thief is ready to receive the tasks, called lifeline delivery request answer.
4. The victim sends the tasks to the thief, called lifeline steal answer.

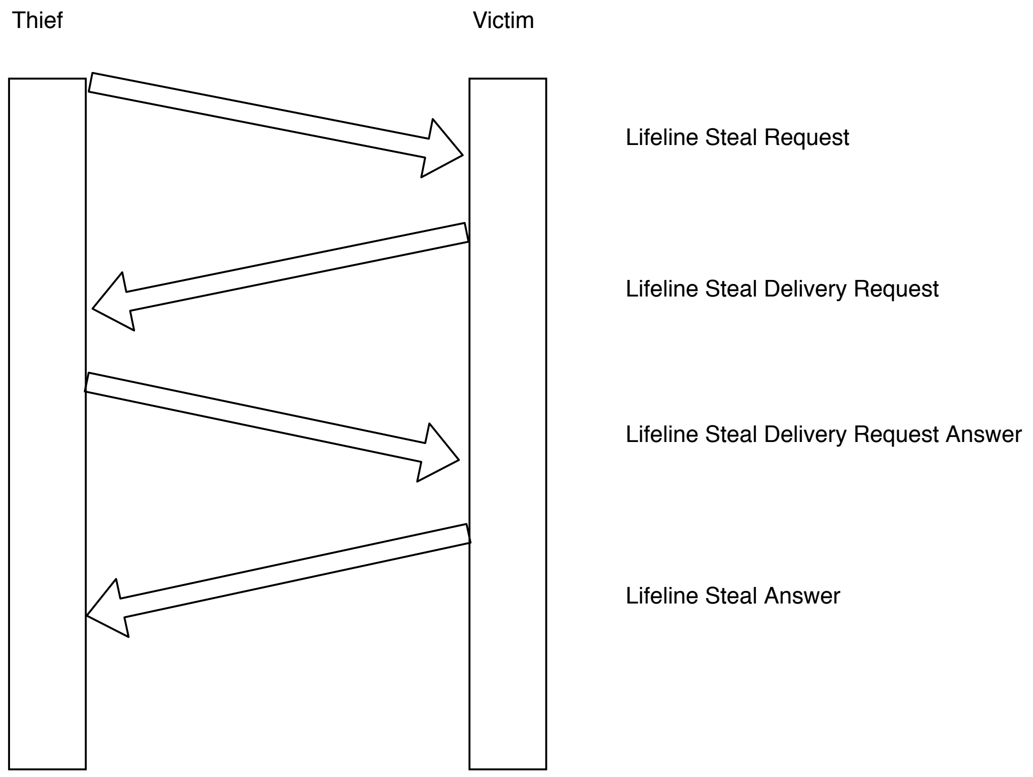

Thief
Victim
Lifeline Steal Request
Lifeline Steal Delivery Request
Lifeline Steal Delivery Request Answer
Lifeline Steal Answer

Figure 3.1: Lifeline steal protocol

DistLogFT is based on a task bag perspective. Every computation can be structured as a tree of task bags. Each edge is a steal relation from the source vertex to the target vertex. The root is an implicit task bag, called *master bag*, which contains all of the initial tasks of the computation. In the next layer, there is one task bag for each worker containing the initial tasks of that worker. These task bags have steal relations to the root. The next layers are filled with task bags which are created due to work stealing. Constraint C2 allows the creation of a tree of task bags with each task bag having exactly one parent task bag (except the master bag). Due to the tree structure, the task results of each task bag are propagated upwards, i.e., until the results reach the master bag, which will contain the final result. At each step of the propagation, the results are combined by reduction. When the result of the master bag is available, the computation is finished. This replaces the final reduction of GLB and is the main change to GLB. Without the change in the reduction, progress made by failed workers would get lost, since results would stay on the failed workers.

The change from a final reduction to a steal-tree based reduction changes the complexity of the reduction operation. A distributed binary-tree reduction results in $O(log\ w)$ steps and $O(w)$ applications of the binary reduction operator (with

w being the number of workers). Assuming a balanced steal tree, the changes in DistLogFT then result in $O(log\ s)$ steps and $O(s)$ applications of the operator (with s being the number of successful steals).

3.3 Tracking of Work Stealing

As in ForkJoinFT, for each steal, the task bag gets assigned a unique identifier (*ID*). Each task bag T contains the path in the steal tree from the master bag to T. Vertices in the path are annotated with the number of tasks processed until the loot was extracted. Since steals in GLB always steal a constant percentage of the available tasks, the number of tasks stolen (loot size) does not need to be saved. Alternative task pool variants might need to include the loot size in the annotations to be able to deterministically extract the same sized loot during recovery. A difference to ForkJoinFT is, that in DistLogFT more than one steal request can be answered at once. To distinguish between steals that happen during the same communication phase, the number of past steals from the parent bag has to be included in the annotations. The history can be generated on each steal, by copying the parent bag's history and adding the annotated vertex of the newly generated task bag.

Similar to ForkJoinFT, a list containing the number of processed tasks for each child task bag of the same parent generated in the past. The list is used to perform the loot extractions for lost child task bags with no children of a lost task bag, as they will not be contained in the history of another task bag. The list is necessary for keeping the pool state the same as it was when the original loot was extracted.

We will call the ID, annotations and the aforementioned list of preceding steals of the same parent *steal information* of a task bag. If tasks are removed from a task bag, only steal information remain. The steal information will be used in recovery.

3.4 Recovery

Recovery is realized similar to ForkJoinFT. We refer to task bags that were owned by a failed worker as *failed* task bags. Each failed task bag is recovered separately. After a failure of a worker, all workers eventually detect the failure and begin to search for failed task bags that they have sent to the failed worker and are still waiting for their results. For each failed task bag, a replay tree is constructed by a global reduction over all workers. We recall, that the history of a task bag is a path in the steal tree. The reduction retrieves the histories of the task bags and merges them to a replay tree. The replay tree contains the steal information of the failed task bag as the root, from which the corresponding parts of the histories descend. Task bags with the same ID in different histories collapse to one vertex in the tree. The replay tree and the failed task bag are combined into a replay unit. The replay unit is either given to another worker by work stealing or is processed by the worker who created the replay unit, if the worker has an empty pool.

Patching is the act of notifying the workers owning the child task bags of the failed task bag of the new worker for result propagation. The worker processing the replay unit initiates patching on place 0. Different patchings can not overlap. Also, patching is canceled if the initiating worker failed. Because we initally assumed that communication to a failed worker returns an error, this can easily be tested by sending a message to the initiating worker right before commencing patching. This prevents partial patchings and makes sure that no subtree of the steal tree becomes disconnected from the tree due to a failure during recovery and a late patching. The parent task bag and the direct child task bags of the failed task bag are reconnected to the recovered task bag. The worker then begins to process the tasks of the recovered task bag. For each direct child in the replay tree, loot is extracted from the task queue to simulate the occured steal. The extractions happen directly after the annotated number of tasks has been processed. The loot is discarded if the owner of the child task bag is still alive. Otherwise a new replay unit is created for the child task bag and is distributed by work stealing or is processed by the recovering worker. Answering of steal requests is not allowed during recovery, because loot extractions modify the pool state. The pool state must remain the same as it was when the failed worker processed the tasks, as recovery depends on it. Recovery ends after all child task bags of the failed task bag have been extracted from the pool. After recovery, the worker returns to normal operation.

3.5 Special Cases

Since the scheme is based on a task-bag perspective, failure scenarios can be described as failed task bags.

3.5.1 Failure of a Single Task Bag

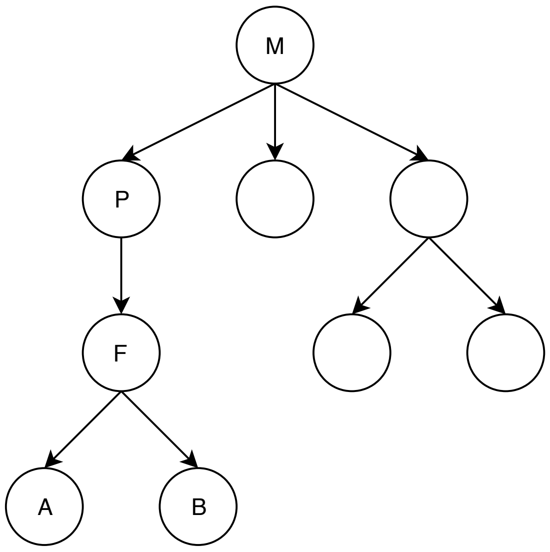


Figure 3.2: Steal tree before failure

In this scenario a single task bag F fails. Figure 3.2 shows the steal tree right before failure. M denotes the master bag. A and B are the direct children of the failed task bag. The worker, who owns the parent task bag of F eventually detects the failure and begins recovery. The worker constructs the steal tree and saves a replay unit. For this scenario, we assume the unit to be stolen by another worker, even though the unit could be immediately processed if the local pool is empty. The thief initiates patching from place 0. During and after patching, results from child task bags arrive at the thief. After patching, the steal tree looks as in Figure 3.3. The failed task bag is denoted as F and the recovered task bag is denoted as F'.

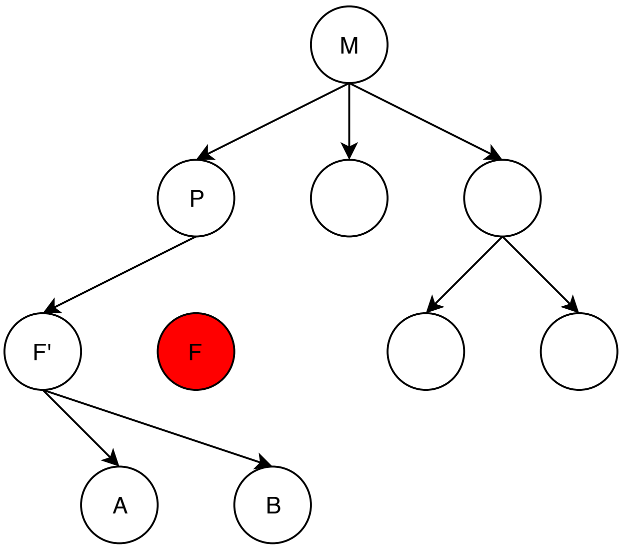


Figure 3.3: Steal tree after patching

The thief begins to reprocess the lost parts of the task bag. The thief is done but is still waiting for a result from a child task bag. After a while, the result is received and the thief sends the result of F to the parent task bag's worker. The steal tree after result propagation of the failed task bag is shown in Figure 3.4.

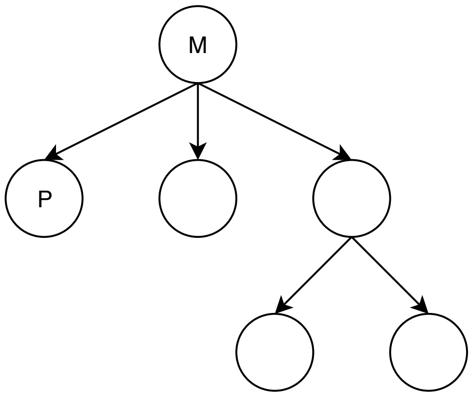


Figure 3.4: Steal tree after scenario

3.5.2 Simultaneous Failure of Two Task Bags

The failure of task bags with non-overlapping steal paths is handled as two separate failures of a single task bag. We assume that the failure of the workers happens at the same time, i.e., the workers fail before any other worker detects any of the failures.

The failure of task bags with overlapping steal paths is essentially handled the same as two separate failures of single task bags. If a failed task bag B is a direct child of a failed task bag A, recovery is begun after recovery of task A as a failure of a single task bag. During recovery of task bag B a replay unit for task bag A is created and given to another worker by work stealing. If the task bags are not direct neighbours in the steal path, they are recovered at the same time, but they are non-overlapping, so they are handled as multiple separate single task bag failures.

3.5.3 Failure During Recovery

This scenario is handled in the same way as a simultaneous failure of two task bags. When the replay unit is stolen by a worker W, the victim, who assembled the replay unit, updates the steal information for the failed task bag on the victim side. This scenario is handled as simultaneous failure of two task bags, because the victim is still the owner of the parent task bag and gives the replay unit to another worker. The recovery of the thief is handled in the same way as a failure of a single task bag.

3.5.4 Task Arrival from a Failed Worker

Task bags from failed workers are declined, because construction of the steal tree might have already happened and this steal might have happened too late to be included. The declined task bag is not included in the history of any other task bag, resulting in no change to any part of recovery due to declining.

If a worker receives a task bag from a worker whose failure was not detected yet, the loot is inserted into the pool when the worker does not participate in a recovery, such as the collection of the steal tree. If the task bag is received before the steal tree is collected, the task bag is included in the steal tree. Thus, the tasks of this task bag are not reprocessed. If the task bag is received after the steal tree collection, the task bag will be declined, because the failure of the victim is known at this point.

3.5.5 Only Few Steals

In an application with only few steals, only few result propagations occur. For every vertex in the steal tree, one result is propagated upwards. Each time a result is propagated, the task bags of the whole subtree are finished. If there are only few result propagations, progress is only made in long time intervals. A failure would cause long reprocessing. In such applications additional checkpointing can be applied to further reduce the number of tasks to be reprocessed after a failure.

4 Implementation

This chapter discusses implementation details. First, some implementation background of APGAS, GLB and AllFTGLB is given. Then details of DistLogFTGLB are discussed.

4.1 Background

4.1.1 APGAS

APGAS programs follow an *async-finish* structure with the following constructs:

- `async(SerializableJob f)`

  Spawns an activity locally. SerializableJob is a functional interface, containing the activity's code.

- `asyncAt(Place p, SerializableJob f)`

  Spawns an activity on place p.

- `finish(Callable<T> f)`

  Waits for all spawned activities in f and their successors.

- `uncountedAsyncAt(Place p, SerializableJob f)`

  Spawns an *uncounted* activity on place p. Uncounted means, that a surrounding `finish` construct does not wait for this activity.

Moreover, `places()` returns a list of all alive places.

The structure of a typical APGAS program is shown in Listing 4.1. An activity printing a text is spawned on each place by using `asyncAt`. The outer `finish` waits for all activities. The program ends when the `finish` returns.

The APGAS library implements a place as a single JVM. The places are connected by using the Hazelcast Library [15] for the network layer. Local activities, meaning activities spawned on the same place, are scheduled by using the Java ForkJoinPool [20]. The ForkJoinPool provides intra-place parallelization by keeping a number of threads and load-balancing the work between them by using work stealing.

```java
import static apgas.Constructs.*;
import apgas.Place;

class HelloWorld {
  public static void main(String[] args) {
    finish(() -> {
      for (Place p : places()) {
        asyncAt(p, () -> {
          System.out.println("Hello␣World␣from␣place␣" + p.id);
        });
      }
    });
  }
}
```

Listing 4.1: Typical APGAS program

APGAS supports resilience by throwing exceptions when calling methods that interact with failed places. The runtime system also calls a callback method, called `placeFailureHandler`, eventually after a place fails. The `finish` construct does not wait for activities that were spawned on failed places.

4.1.2 GLB

GLB employs the following task model (cited from [25]):

- Tasks have no side-effects.
- Processing a task can generate new tasks.
- Processing a task produces a 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.

The computation in GLB uses a typical async-finish structure. When the outer `finish` of the computation returns, the computation is over. The main loop of GLB is shown in Listing 4.2. When a worker exits the main loop, it enters the inactive state. When all workers are inactive, the `finish` returns, as there are no activities running anymore. The final result is calculated by reduction over all workers. The reduction also uses an async-finish structure.

```
while (tasks are available) {
  while (local pool is not empty) {
    synchronized (worker lock object) {
      process up to n tasks;
      send tasks to recorded thieves;
    }
  }
  synchronized (worker lock object) {
    try to steal from up to w+z victims;
  }
}
```

Listing 4.2: Main loop of GLB [26]

In GLB, each place executes one worker. When using GLB on a single multi-core machine, multiple places have to be executed to utilize all available resources. The outer `finish` and the final reduction is initiated by place 0.

Application programmers have to implement the task queue interface `TaskQueue` and the task bag interface `TaskBag`. The loot extraction during work stealing is realized by a call to the `split` method on the task queue. The method returns a task bag containing the extracted tasks.

4.1.3 AllFTGLB

AllFTGLB's fault-tolerance is based on Hazelcast's data structures. Hazelcast provides resilient and distributed data structures. The *IMap* datastructure of Hazelcast is a distributed resilient key-value store. It is used to store the checkpoints of the workers. Since AllFTGLB writes a backup after a worker becomes idle, the checkpoints contain the results of the workers after the computation is done. The final result is then calculated by reduction over all IMap entries. The number of replications of IMap contents is given as a parameter called *backupCount*. The highest possible number of replications is six.

4.2 DistLogFTGLB

As discussed in Chapter 3, DistLogFT propagates the task bag results towards the master bag. For this, a counter, called `remainingResults`, was added to each task bag. The counter keeps track of how many results of children a task bag has to wait for, before the result can be sent to the parent bag. Whenever a result is received from a child, the counter is decremented and if the counter reaches zero, the result is sent to the parent bag. The computation is completed when the master bag's result is available, i.e., when `remainingResults` is zero. As mentioned above, GLB and AllFTGLB use an async-finish structure for termination detection. When all tracked activites are done, the computation is completed. In DistLogFTGLB, such a structure is not necessary, because it is sufficient to observe the master bag's counter on place 0. By realizing the termination detection this way, no additional messages for termination detection are sent, thus reducing the communication volume.

DistLogFTGLB requires the `split` method to always extract the same loot on the same pool state. For the benchmarks, the code from Reference [28] was adapted, which already fulfilled this requirement, see Chapter 5 for further details on the benchmarks.

The implementation of the lifeline steal handshake, as shown in Section 3.2, is as follows:

1. The thief spawns an activity on the victim. The activity adds the request to a thread-safe queue. This message is the lifeline steal request.
2. In the gaps between task processing, the communication phase begins. In this phase workers take out requests from their queues and answer each request by spawning an activity at the thief. This message is the lifeline steal delivery request.
3. The activity ends if there is already another handshake ongoing. If there is no other handshake going on, a flag is set atomically using a compare and set operation to indicate an ongoing handshake.
4. The activity and the handshake ends if the queue is not empty because of a prior lifeline task delivery.
5. Next, an activity is spawned on the victim which adds the thief to another thread-safe queue. This queue is used for keeping track of ongoing lifeline handshake partners who expect task deliveries. This message is the lifeline steal delivery request answer.
6. Since the lifeline handshake is executed asynchronously next to the processing of tasks, the victim might be processing tasks. Again, in the gap between task processing, the victim sends out lifeline deliveries to the lifeline thiefs by spawning activites on them. This message is the lifeline steal answer.
7. The activity inserts the tasks into the thief's pool and ends the ongoing handshake.

The `TaskQueue` interface is extended by the following methods:

- `clearResult()`

  Resets the current worker result to the same value as it was before the computation began. By resetting the worker result after a task bag has been processed, the worker result of GLB can be used as the task bag result.

- `getInitialTasks(int p, int numPlaces)`

  Returns an object of the TaskBag implementation containing the initial tasks of the worker `p`. Parameter `numPlaces` specifies the total number of initial places. The total number is required for calculating the task distribution for static workloads.

- `getEmptyResultAsTaskBag()`

  Returns an empty task bag containing the initial result of the current worker. This method exists for simplicity of the implementation as we can use it instead of `null` values. Merging the tasks and the result of the task bag returned by this method does not result in any changes to the merging worker.

The `TaskBag` interface is changed to an abstract class (see Listing 4.4). Each object is assigned a unique identifier of type `long`. For the 64 bit size of the `long` value, we write the number of the creating worker to the first 32 bit, beginning from the Most Significant Bit (MSB) and write the value of a counter to the remaining bits. Each worker has its own upcounting counter, that increments whenever a `TaskBag` object is created.

Also, each `TaskBag` object contains a variable `remainingResults`, containing the number of results to wait for, before sending the current result to the former victim. The result is saved in a variable `result`.

`TaskBag` objects also contain the history of the task bag. We remember that the history is a path in the steal tree, which is typically not very long. For the history a `LinkedList<HistoryEntry>` was used for simplicity of usage. `HistoryEntry` is a new class in DistLogFTGLB (see Listing 4.3). The application programmer does not have to implement this class. Each entry in the `LinkedList` represents a node in the steal path. A `HistoryEntry` object has the following attributes:

- `int ownerPlace`

  Contains the number of the thief.

- `int splitStep`

  Contains the number of tasks processed before the loot was extracted on the victim.

- `int splitAtStep`

  Contains the number of prior loot extractions that occured after the same number of tasks processed, incremented by one.

- `long bagId` Contains the identifier of the corresponding task bag.

Finally, `TaskBag` objects contain a list of earlier child task bags of the same parent bag, called `sameLevelSplits`.

A class `ReplayUnit` (Listing 4.5) was added to represent replay units. A `Replay-Unit` object contains a task bag `bag` and the corresponding replay tree `replayTree`. The `replayTree` variable is an object of type `Tree`. The `Tree` class was added for DistLogFTGLB and each object represents a tree node of the replay tree. A `Tree` object contains a list of its children in a variable `children` of type `LinkedList` and an attached value of a generic type (see Listing 4.6). For the attached value, the type `HistoryEntry` is used. `Tree`s can be recursively merged with the method `merge(Tree<T> other)` if they have the same root value. The merging of trees is used during recovery (see below).

The tracking of the work stealing is realized by a `HashMap` called `sentLoot`, which maps bag ids to `TaskBag` objects. Whenever a task bag is stolen, the task bag is put into this map. When the result is received from the thief, the task bag is removed from this map. Another `HashMap`, called `currentBags`, contains currently owned task bags, whose results were not sent to the former victim yet. It maps bag ids to task bags. When a task bag is received from a victim, it is added to this map.

When a failure of a place is detected by the APGAS runtime system, the `place-FailureHandler` method is called for every worker. First, all steal requests and anticipated steal answers from the failed worker are deleted. Pending lifeline hand-shakes with the failed worker are canceled by setting `waitingForLifelinePlace` to -1 if `waitingForLifelinePlace` is set to the id of the failed worker. Also, each worker removes the failed worker from their lifeline graph. Next, each worker searches for task bags sent out to the failed worker by linearly searching through the values of the `sentLoot` map. The next part is executed for each found task bag. As mentioned in Chapter 3, these task bags are called *failed task bags*. If the failed task bag is an initial task bag, the tasks are retrieved by a call of `getInitialTasks(failedWorkerId, P)` on the task queue. Otherwise, the tasks are already contained in the task bag from `sentLoot`. The next step is retrieving the histories of all orphan task bags. The global reduction is implemented by using a typical async-finish structure. Within an outer `finish` block, a loop iterates over all places and spawns an asynchronous activity on each of them by using `asyncAt`.

Within this activity, the places check the histories of their task bags contained in `currentBags` for occurences of the failed task bag. The found histories are cropped, such that they only contain the path from the orphan task bag to the failed task bag. The cropped path is then sent back by another `asyncAt` activity. A synchronized list is used to save all collected paths. From the collected paths, `Tree`s are created and afterwards merged to a `replayTree`. The `replayTree` and the failed task bag are then combined into a `ReplayUnit` object and added to a worker global list `replayUnits`. In work stealing, this list is checked first for any replay units.

```java
public class HistoryEntry implements Serializable {
  public int ownerPlace;
  public int splitStep;
  public int splitAtStep;
  public long bagId;

  // constructor omitted

  @Override
  public boolean equals(Object o) {
    if (o instanceof HistoryEntry) {
      HistoryEntry other = (HistoryEntry) o;
      return other.bagId == this.bagId;
    }
    return false;
  }
}
```

Listing 4.3: HistoryEntry class

```java
public abstract class TaskBag implements Serializable {
  public static int lastId = 1;
  public long id = ((long) (here().id) << 32) + lastId++;

  public int remainingResults = 1;
  public DistLogFTGLBResult result;
  public LinkedList<HistoryEntry> history;
  public ArrayList<Integer> sameLevelSplits;

  public abstract int size();
}
```

Listing 4.4: TaskBag class

```java
public class ReplayUnit implements Serializable {
  public Tree<HistoryEntry> replayTree;
  public TaskBag bag;

  // constructor omitted
}
```

Listing 4.5: ReplayUnit class

```java
public class Tree<T> implements Serializable {
  private final T value;
  public LinkedList<Tree<T>> children;

  // constructor omitted

  // merges another Tree with the same root recursively
  public void merge(Tree<T> other) {
    for (Tree<T> child : other.children) {
      final int idx = this.children.indexOf(child);
      if (idx < 0) {
        this.children.add(child);
      } else {
        final Tree<T> thisChild = this.children.get(idx);
        thisChild.merge(child);
      }
    }
  }

  public T getValue() {
    return this.value;
  }

  @Override
  public boolean equals(Object other) {
    final Tree otherTree = (Tree) other;
    return this.value.equals(otherTree.value);
  }
}
```

Listing 4.6: Tree class

5 Experiments

This chapter describes the experiments conducted. First, Section 5.1 introduces the different groups of experiments. Next, Section 5.2 explains the benchmark applications. Thereafter, Section 5.3 shows used software and hardware as well as the program arguments. Then, Section 5.4 presents the results. Next, Section 5.5 discusses the results. Finally, Section 5.6 describes how the correctness of the implementation was tested.

5.1 Overview

Two groups of experiments were conducted. The first group measured the execution time for varying numbers of places and parameters. We call this group *weak scaling experiments*. The second group measured the execution time of runs with places failing at certain points of the execution for AllFTGLB and DistLogFTGLB. This group will be called *recovery overhead experiments*.

In the weak scaling experiments, execution times of GLB, AllFTGLB and DistLogFTGLB were recorded and a *fault-tolerance overhead* is calculated from them. The overhead is calculated by the formula $\frac{t_{system}}{t_{GLB}} - 1$, with t_{system} standing for the measured execution time of AllFTGLB and DistLogFTGLB, respectively. The overhead is presented as a percentage.

The recovery overhead experiments measure the execution times for three different configurations:

1. Failure-free execution with 144 places
2. Failure-free execution with 132 places
3. Execution with 144 places initially and crashes of 24 places at about half of the time measured for configuration 1. Execution continues with 120 places after the crashes.

All executions of these three configurations use a backupCount of 6 for the APGAS runtime system and AllFTGLB. This backupCount is necessary, because the crashes occur almost simultaneously. A smaller value for backupCount was found to lead to an unrecoverable failure of the APGAS runtime system. We denote the measured times of the above three configurations as t_{144}, t_{132} and t_{144-24}. Since for t_{144-24} the computation used 12 places more than t_{132} for the first half of the computation and 12 places less than t_{132} for the second half of the computation, we expected $t_{132} + t_{recovery} \approx t_{144-24}$, with $t_{recovery}$ being the time recovery from all 24 place failures took. $t_{recovery}$ is calculated from the measured times by reformulating the above equation to $t_{recovery} \approx t_{144-24} - t_{132}$. The recovery overhead is then calculated by the formula $\frac{t_{recovery}}{t_{132}} - 1$ and is presented as a percentage. This is only a rough approximation of the recovery overhead, because the second half of the computation has more idle workers than the first half, resulting in less idle workers for the remaining computation after recovery is completed. As we will see, the measured recovery overheads are small, so a high number of place crashes had to be selected for this group of experiments in order to obtain measurable results. t_{144} was measured for reference, but was not used for the recovery overhead calculation.

For the experiments, the Unbalanced Tree Search (**UTS**) and the N-Queens benchmarks from Reference [28] were selected. The source code was adapted for DistLogFTGLB with only minimal changes, such as the implementation of the new methods of the TaskQueue interface. The actual benchmark algorithms were not changed. The benchmarks are described in Section 5.2.

To evaluate the importance of the changed termination detection, we extended the weak scaling experiments by a third system, called *DistLogFinishFTGLB*. We recall from Section 4, that AllFTGLB uses an outer `finish` for termination detection and DistLogFTGLB does not use an outer `finish`. No extra mechanism for termination detection is implemented in DistLogFTGLB, because place 0 notices the end of the computation as a side effect of the result propagation. DistLogFinishFTGLB extends DistLogFTGLB by an outer `finish`. The difference between DistLogFTGLB and DistLogFinishFTGLB is the additional overhead caused by the `finish`.

Additionally, correctness tests are described in Section 5.6.

5.2 Benchmarks

5.2.1 UTS

The UTS benchmark calculates the number of nodes of an unbalanced tree. The tree is generated during the execution, so the number of nodes is unknown at the beginning. A task corresponds to exactly one tree node. The computation begins with the root node as a task on worker 0. For each node, a cryptographic hash function determines the number of children. The maximum tree depth is given as a parameter d and the width of the tree can be influenced by the *branching factor* parameter b. An initial seed for the cryptographic function can be set by parameter r. The result is a single value of type `long` resembling the total number of nodes in the tree. The partial results are combined by the sum operator as the reduction operator.

5.2.2 N-Queens

The N-Queens [12] benchmark calculates the number of possible positionings of N queens on an $N \times N$ chessboard, in a way, that queens can not attack each other. The number of queens is given as parameter q. The computation begins with a task on worker 0 trying to place a queen on an empty chessboard. Each time a queen can be placed on a free field on the chessboard, a new task is created. The new task tries to place another queen on this new chessboard. This is repeated until N queens are placed. A parameter t controls the number of remaining queens, at which a task does not push new tasks to the work queue, but processes the remaining subtree of the computation tree sequentially. The result is a single value of type `long`. The reduction operation is the sum.

5.3 Setup

Experiments were conducted on two different clusters. The clusters differ greatly in size:

- Cluster of University of Kassel (denoted as *Kassel*):

  The partition `FB16` consists of 12 nodes. Each node is equipped with two 6-core Intel Xeon E5-2643 v4 CPUs and 256 GB main memory [35].

- Cluster of Goethe University Frankfurt (denoted as *Goethe*):

  The partition `general1` consists of 472 nodes. Each node is equipped with two 20-core Intel Xeon Skylake Gold 6148 CPUs and a minimum of 192 GB main memory [13].

Both clusters have the Slurm workload manager [32] installed. All runs of the experiments were scheduled using Slurm. The workload manager helps to reserve nodes exclusively for a given time interval. On Goethe, Slurm also counted the used number of core hours, which were limited. For experiments with 440 places, a usage of about 2 hours equals about 1,000 core hours. All experiments were conducted with Java in version 12.0.2 and Hazelcast in version 3.10.6.

For every configuration, five runs were scheduled and the average of these runs is taken as the result of the configuration. Because of the non-deterministic nature of GLB's work stealing, the execution times in the five runs fluctuates. This results in a certain variance in the presented execution times. Increasing the number of runs would solve this issue, but was not possible due to time and resource constraints.

For every used node, the number of started places is equal to the number of cores available on this node. For example, on `FB16`, we started 12 places on each node, because each node is equipped with 12 physical cores.

Table 5.1 shows the parameters used in the weak scaling experiments for each benchmark and cluster. GLB parameter n is set to 511 in all runs, as it was used in the experiments of past papers, e.g., in Reference [11, 28]. With P being the number of initial places, the GLB parameters w, l and z are calculated as in Reference [29]:

$$w = \begin{cases} P-1, & \text{if } P \leq 6 \\ 6, & \text{otherwise} \end{cases}$$

$$l = min\{x \in \mathbb{N}_{>0} | x^x \geq P\}$$

and z is set such that l^z is greater or equal P.

For the backup interval in AllFTGLB, Daly's formula was used to estimate an appropriate interval for a chosen MTBF. For a long MTBF, Daly's formula is as follows:

$$T_{opt} = \sqrt{2\delta M}$$

with M being the MTBF and δ being the duration of checkpoint writing [8]. Assuming $\delta = 0.01s$ and an MTBF of one week, T_{opt} results in about 350 seconds. Since this value is too large for our experiments, we chose a smaller value of 10 seconds. Most runs did not take longer than 15 minutes, so choosing a short interval made measuring the fault-tolerance overhead more easy. Also, DistLogFT does not use this interval, but sends additional information appended to work stealing messages about every 10 seconds. This made 10 seconds a natural choice as a fitting interval for the experiments. Using 10 seconds for T_{opt} and solving the equation for the corresponding MTBF results in an MTBF of about 1.5 hours.

Benchmark	Cluster	Places	Parameters
UTS	Kassel	12,24	$d = 17,\ b = 4,\ r = 19$
UTS	Kassel	36,48,60,72	$d = 18,\ b = 4,\ r = 19$
UTS	Kassel	84,96,108,120,132,144	$d = 19,\ b = 4,\ r = 19$
UTS	Goethe	40	$d = 17,\ b = 4,\ r = 19$
UTS	Goethe	120,200,280,360	$d = 18,\ b = 4,\ r = 19$
UTS	Goethe	440	$d = 19,\ b = 4,\ r = 19$
N-Queens	Kassel	12	$q = 16,\ t = 10$
N-Queens	Kassel	24,36,48,60,72	$q = 17,\ t = 11$
N-Queens	Kassel	84,96,108,120,132,144	$q = 18,\ t = 12$
N-Queens	Goethe	40,120,200	$q = 17,\ t = 11$
N-Queens	Goethe	280,360,440	$q = 18,\ t = 12$

Table 5.1: Parameters used for the weak scaling experiments

5.4 Results

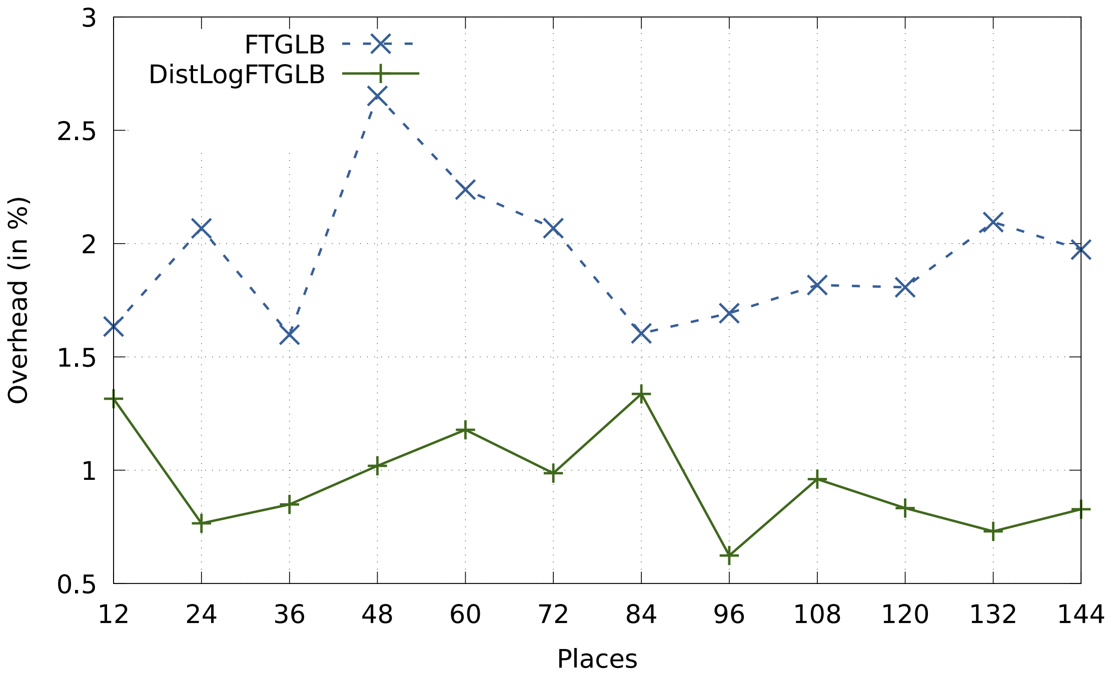


Figure 5.1: UTS on Kassel

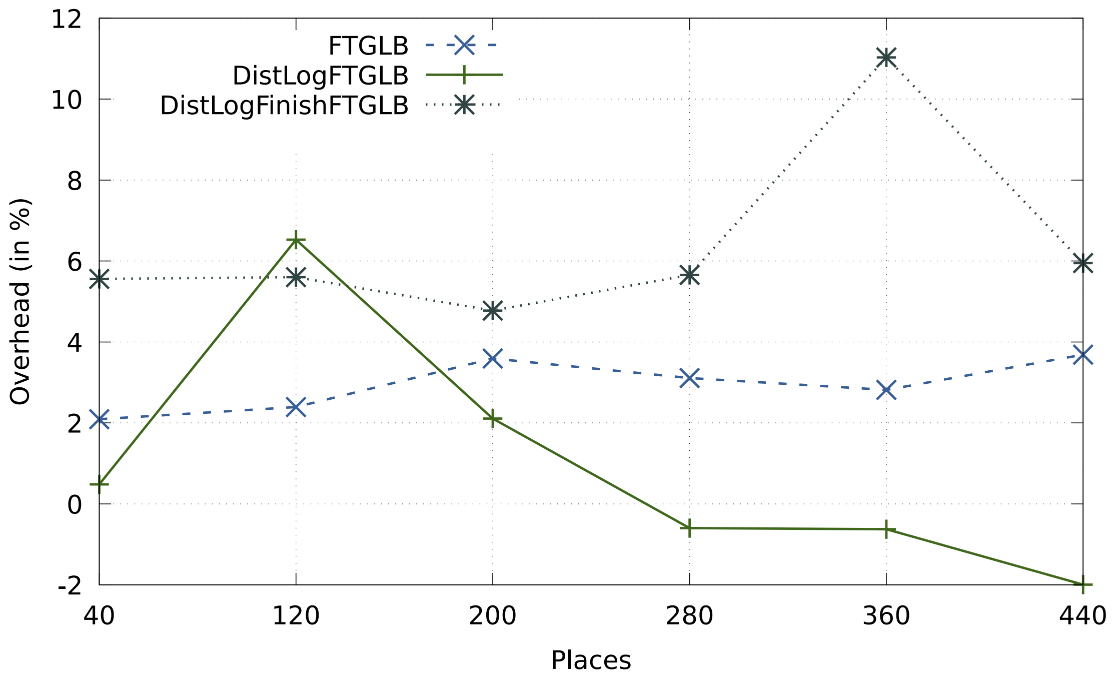


Figure 5.2: UTS on Goethe

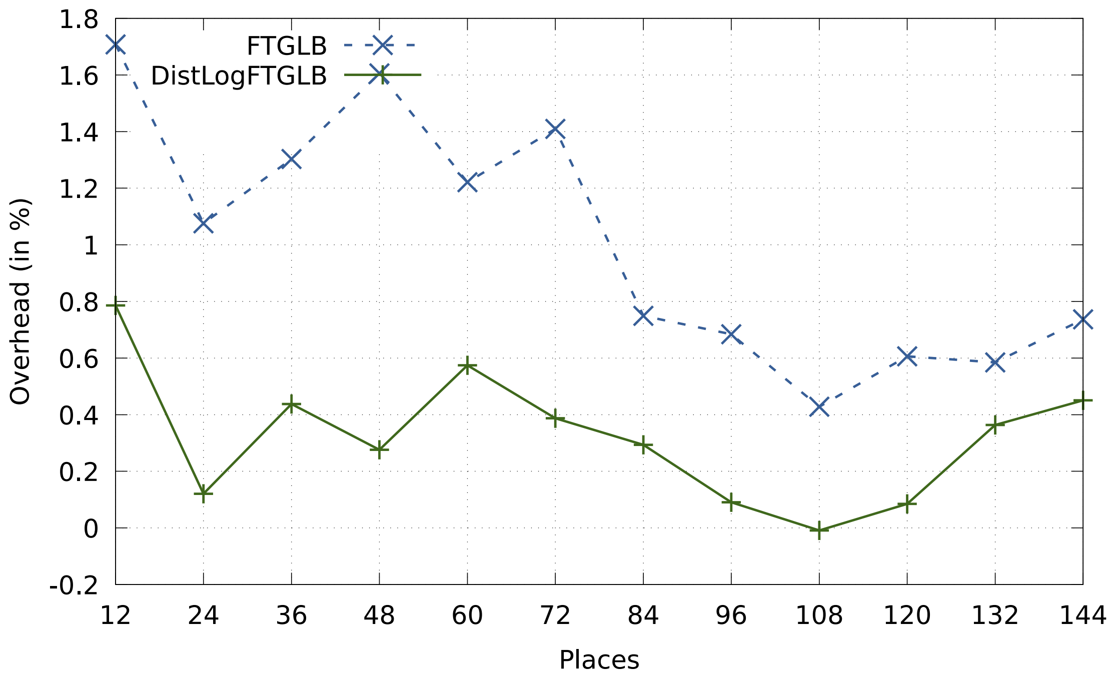


Figure 5.3: N-Queens on Kassel

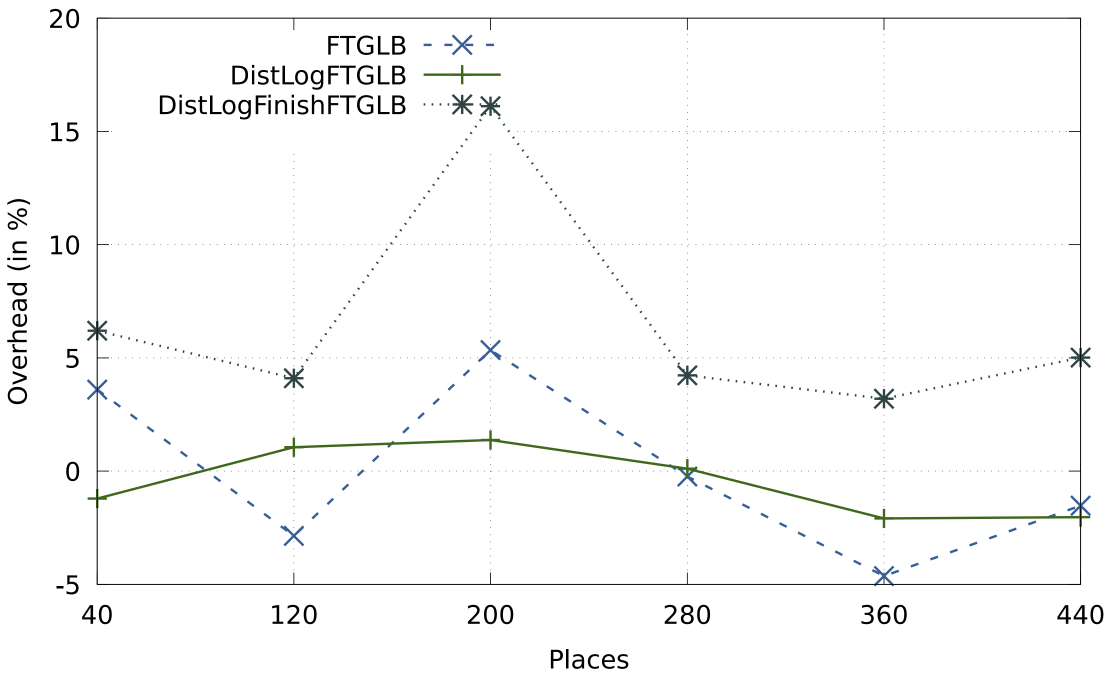


Figure 5.4: N-Queens on Goethe

Places	AllFTGLB	DistLogFTGLB
144	908.68	889.51
132	991.59	967.74
144 - 24	1017.52	1058.10

Table 5.2: Execution times in seconds for UTS with $d = 19$ and backup count = 6 on kassel cluster

Figures 5.1 and 5.2 show the fault-tolerance overheads for UTS on Kassel and Goethe, respectively. In all but one configuration, the overhead of DistLogFTGLB is lower than the overhead of AllFTGLB. For 120 places on Goethe, AllFTGLB has a lower overhead than DistLogFTGLB. On both clusters, UTS showed an almost linear increase of overhead for AllFTGLB.

For 12 places on Kassel, the overhead of UTS on AllFTGLB is at 1.63%, which peaks at 2.65% for 48 places and ends with 1.97% for 144 places. UTS on DistLogFTGLB begins at 1.32% on Kassel with 12 places, peaks at 1.34% for 84 places and ends with 0.83% for 144 places. On Goethe, UTS on AllFTGLB begins with its lowest overhead of 2.09% for 40 places and ends with a peak overhead of 3.68% for 440 places. UTS on DistLogFTGLB shows an overall decrease of overhead proportional to the number of places. It begins with 0.48% for 40 places, peaks at 6.53% for 120 places and ends with an overhead of -1.99% for 440 places.

The N-Queens benchmark's results are depicted in Figures 5.3 and 5.4. The results show an almost linear decrease of overhead for both benchmarks. On Kassel, AllFTGLB begins at the highest overhead of 1.71% for 12 places and ends at 0.74% for 144 places. DistLogFTGLB begins at 0.79% for 12 places and ends at 0.45% for 144 places. The results on Goethe for N-Queens on AllFTGLB show an overhead of 3.60% for 40 places. They show a peak of 5.35% at 200 places and end with an overhead of -1.52% for 440 places. DistLogFTGLB begins at -1.21% for 40 places, peaks at 1.37% for 200 places and ends with an overhead of -2.03% for 440 places.

For the runs of DistLogFinishFTGLB, the overhead was almost always higher than the overhead of the other systems. The overhead of DistLogFinishFTGLB for N-Queens on Goethe in Figure 5.4 has a similar curve than the overhead of AllFTGLB, but is shifted upwards by a about 2% up to 10%.

The recovery overheads are calculated from Table 5.2 as stated in Section 5.1. For AllFTGLB, the recovery overhead is 2.6%. The recovery overhead of DistLogFT-GLB is 3.58 times as high as the recovery overhead of AllFTGLB.

5.5 Discussion

DistLogFTGLB does not send more messages than GLB during failure-free execution, but increases the volume of messages containing loot, which is sent during GLB's work stealing. Most results show a lower overhead for DistLogFTGLB than for AllFTGLB. One reason could be the lower message number of DistLogFTGLB. Another reason can be the different termination detection. Compared to AllFT-GLB and the non-fault-tolerant GLB, DistLogFTGLB also uses less messages for termination detection.

The higher recovery overhead of DistLogFTGLB, compared to AllFTGLB, can be explained by the higher number of messages during recovery and the necessity of reproducing the occured loot extractions from work stealing. Also, more tasks have to be reprocessed in DistLogFTGLB than in AllFTGLB, because of the result propagation. When a worker fails which already received a lot of child results but couldn't send the result to its parent worker, all of the child results are lost. The number of tasks to be reprocessed in such a situation can be very high and can exceed the number of tasks to be reprocessed from a checkpoint in AllFTGLB. The periodic checkpointing in AllFTGLB provides an upper bound for the duration of reprocessing. No such upper bound exists in DistLogFTGLB. In the worst case, it is possible that almost the whole computation has to be repeated.

In Figure 5.2, unexpectedly high fault-tolerance overheads are shown for 120 places and 360 places. The value for DistLogFTGLB and 120 places does not seem plausible since it is higher than the overhead of DistLogFinishFTGLB, which differs to DistLogFTGLB only in sending more messages. We assume both of these unexpectedly high overheads to be errors in measurement caused by external factors. Repeating these experiments was not possible due to time constraints.

DistLogFinishFTGLB showed a significantly higher overhead compared to DistLogFTGLB. Also, even though we observe a decrease of overhead for DistLogFTGLB when increasing the number of places, we can not observe any decrease for DistLogFinishFTGLB. From this, we conclude that the `finish` construct does introduce a significant overhead. With an increasing number of places, the fault-tolerance overhead of DistLogFTGLB decreased below 0%. The reason for the negative overheads can be the different termination detection.

Since some experiments are identical to experiments conducted in Reference [28], we also confirmed their results of the same experiments.

5.6 Correctness

We use the same correctness definition as in Reference [25]: “The algorithm is correct in the sense that the computed result is the same as in non-failure case, or the program aborts with an error message”.

We simulated crashes at certain points during the execution to test the correctness. The crashes were simulated by calls to `System.exit()`. `System.exit()` causes the Java Virtual Machine (JVM) to terminate. During the tests, each worker ran on its own place and each place used its own JVM. A simulated crash on one JVM thus terminates exactly one worker.

Crashes were simulated at certain points in the execution. Single place failure as well as failure of multiple places was simulated. The exact points in the execution are listed in the appendix in Section 7.

We inspected the log files to compare the results and error messages. The tests did not show any wrong results or missing error messages.

6 Related Work

Fault-tolerance for task pools is an active research field. System-level fault-tolerance is the most common way of recovering from permanent node failures. Many libraries which provide it have been published, such as DMTCP [2] and FTI [3].

References [11, 25] propose application-level checkpointing schemes for task pools which save tasks and results in a resilient store. A similar scheme to DistLogFT writes steal relations to a resilient store instead of sending them along with steal answers [28]. Many parallel programming systems support application-level fault-tolerance, e.g., Charm++ [18], Resilient X10 [7] and ULFM for MPI [4].

A different approach to fault-tolerance is algorithm-based fault-tolerance (ABFT). This approach uses properties of the computation to restore lost work without the help of the runtime system, apart from failure notifications. For example, Reference [1] presents an ABFT for matrix computations, which exploits redundancy in matrix computations.

Concurrent checkpointing writes backups next to the computation and can be applied to reduce the fault-tolerance overhead [21]. Reference [33] reports a 10 to 20 times lower fault-tolerance overhead when applying concurrent checkpointing compared to writing the checkpoints inbetween the computation. Our group applied concurrent writing of checkpoints in a scheme of Reference [28].

Reference [36] presents a fault-tolerance scheme for fork-join applications. The scheme broadcasts the location of results that could not get returned to the parent worker. When a worker encounters a task whose location is known, the result is retrieved from the saved location. Later, the scheme was extended by checkpointing in Reference [37]. The later scheme determines a processor, who waits for results of workers and writes them to a checkpoint file on a stable storage. Upon a failure, this processor reads the results of the failed worker and distributes them to the remaining workers.

The parallel runtime system Cilk-NOW [6] employs two fault-tolerance techniques. First, applications can be restarted from periodically written checkpoints. Second, lost work of a failed worker and the work stolen from this worker will be reprocessed by another worker, including finished work. ForkJoinFT improves the second technique by avoiding re-execution of stolen children of failed tasks [19]. A similar fault-tolerance scheme for GLB was presented in Reference [9].

APGAS is under active development. For example, Reference [27] recently extended the APGAS library by locality-flexible tasks similar to tasks in GLB. Additionally, they added cancelable tasks and showed linear speedups for the locality-flexible tasks and overheads for cancelable tasks below 7%.

7 Conclusions

This thesis presented an adaption of a fault-tolerance scheme, which tracks work stealing, and compared it to a scheme, which uses application-level checkpointing. The original scheme from Reference [19] is designed for distributed fork-join programs. We applied the scheme to a reduction-based task pool implementation and provided a description, which specifies the requirements on the task pool variant.

We evaluated the adapted scheme by measuring execution times for two popular benchmark applications and calculated overhead values. The results showed a lower fault-tolerance overhead during failure-free execution for the adaption than for the scheme based on checkpointing. On the backside, the overhead induced by the recovery is higher for the adaption. As a side effect of the experiments, some previous results for AllFTGLB were confirmed with the latest versions of Java and APGAS.

We concluded, that sending less messages for fault-tolerance can reduce the fault-tolerance overhead. Slightly increasing the volume of messages sent during work stealing did not seem to have a significant effect on the execution times. Even though checkpoint/restart is an established technique for achieving fault-tolerance, tracking of work stealing is a viable alternative to it.

The scheme can be extended by checkpointing to reduce the recovery overhead. Workers periodically save their current task bags to a stable storage. Since the saved task bags represent a past state, where stolen tasks are already removed, recovery can skip some reprocessing. Future work should evaluate if such an extension can reduce the recovery overhead and if the checkpointing causes a siginificant increase of the fault-tolerance overhead during failure-free execution. Also, the adaption should be compared to a GLB variant, which uses the same termination detection.

Bibliography

[1] N. Ali, S. Krishnamoorthy, M. Halappanavar, et al. “Multi-Fault Tolerance for Cartesian Data Distributions.” In: *Int. Journal of Parallel Programming* 41.3 (2012), pages 469–493.

[2] J. Ansel, K. Arya, and G. Cooperman. “DMTCP: Transparent checkpointing for cluster computations and the desktop.” In: *Int. Symp. on Parallel & Distributed Processing.* IEEE, 2009, pages 1–12.

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, et al. “FTI: High performance Fault Tolerance Interface for hybrid systems.” In: *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis.* ACM Press, 2011, pages 1–32.

[4] W. Bland, A. Bouteiller, T. Herault, et al. “Post-failure recovery of MPI communication capability.” In: *The Int. Journal of High Performance Computing Applications* 27.3 (2013), pages 244–254.

[5] R. D. Blumofe and C. E. Leiserson. “Scheduling multithreaded computations by work stealing.” In: *Proc. Symp. on Foundations of Computer Science.* 1994, pages 356–368.

[6] R. D. Blumofe and P. A. Lisiecki. “Adaptive and Reliable Parallel Computing on Networks of Workstations.” In: *Proc. USENIX Annual Technical Symp.* 1997.

[7] S. Crafa, D. Cunningham, V. Saraswat, et al. “Semantics of (Resilient) X10.” In: *ECOOP – Object-Oriented Programming.* 2014, pages 670–696.

[8] J. T. Daly. “A higher order estimate of the optimum checkpoint interval for restart dumps.” In: *Future Generation Computer Systems* 22.3 (2006), pages 303–312.

[9] M. Dratwa. “Übertragung eines fehlertoleranten Algorithmus für Fork/Join-Programme auf reduktionsbasierte Taskpools.” Masterthesis. University of Kassel, 2018.

[10] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, et al. “A Survey of Rollback-recovery Protocols in Message-passing Systems.” In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pages 375–408.

[11] C. Fohry, J. Posner, and L. Reitz. “A Selective and Incremental Backup Scheme for Task Pools.” In: *Int. Conf. on High Performance Computing & Simulation (HPCS)*. 2018, pages 621–628.

[12] E. J. Gik. *Schach und Mathematik*. 1st edition. Thun, 1987.

[13] Goethe University Frankfurt. *Goethe-HLR Cluster Usage*. `https://csc.uni-frankfurt.de/wiki/doku.php?id=public:usage:goethe-hlr`. 2019.

[14] Y. Guo, R. Barik, R. Raman, et al. “Work-first and help-first scheduling policies for async-finish task parallelism.” In: *2009 IEEE Int. Symp. on Parallel Distributed Processing*. May 2009, pages 1–12.

[15] Hazelcast, Inc. *The Leading Open Source In-Memory Data Grid*. `http://hazelcast.org`. 2018.

[16] T. Herault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015.

[17] IBM Corp. *Core implementation of X10 programming language including compiler, runtime, class libraries, sample programs and test suite*. `https://github.com/x10-lang/x10`. 2018.

[18] L. V. Kale and S. Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++.” In: *SIGPLAN*. Volume 28. ACM, 1993, pages 91–108.

[19] G. Kestor, S. Krishnamoorthy, and W. Ma. “Localized Fault Recovery for Nested Fork-Join Programs.” In: *IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*. 2017, pages 397–408.

[20] D. Lea. “A Java Fork/Join Framework.” In: *Proc. ACM Conf. on Java Grande*. ACM, 2000, pages 36–43.

[21] K. Li, J. F. Naughton, and J. S. Plank. “Real-time, Concurrent Checkpoint for Parallel Programs.” In: *Proceedings of the Second ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*. PPOPP ’90. Seattle, Washington, USA: ACM, 1990, pages 79–88.

[22] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. “Steal Tree: Low-overhead Tracing of Work Stealing Schedulers.” In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, 2013, pages 507–518.

[23] S. Olivier, J. Huan, J. Liu, et al. “UTS: An Unbalanced Tree Search Benchmark.” In: *Languages and Compilers for Parallel Computing.* Springer LNCS 4382, 2006, pages 235–250.

[24] J. Posner. *Extended APGAS library repository.* `https://github.com/posnerj/PLM-APGAS`. 2018.

[25] J. Posner and C. Fohry. “A Java Task Pool Framework providing Fault-Tolerant Global Load Balancing.” In: *Int. Journal of Networking and Computing (IJNC)* **8.1** (2018), pages 2–31.

[26] J. Posner and C. Fohry. “Cooperation vs. Coordination for Lifeline-based Global Load Balancing in APGAS.” In: *Proc. ACM SIGPLAN Workshop on X10.* ACM, 2016, pages 13–17.

[27] J. Posner and C. Fohry. “Hybrid Work Stealing of Locality-Flexible and Cancelable Task for the APGAS Library.” In: *The Journal of Supercomputing* 74.4 (2018), pages 1435–1448.

[28] J. Posner, L. Reitz, and C. Fohry. “A Comparison of Application-Level Fault Tolerance Schemes for Task Pools.” In: *Future Generation Computer Systems* (Submitted).

[29] L. Reitz. “An Asynchronous Backup Scheme Tracking Work-Stealing for Reduction-Based Task Pools.” Bachelorthesis. University of Kassel, 2018.

[30] V. A. Saraswat, P. Kambadur, S. Kodali, et al. “Lifeline-based Global Load Balancing.” In: *Proc. ACM Symp. on Principles and Practice of Parallel Programming.* ACM, 2011, pages 201–212.

[31] V. Saraswat, G. Almasi, G. Bikshandi, et al. “The asynchronous partitioned global address space model.” In: *The First Workshop on Advances in Message Passing* (2010), pages 1–8.

[32] SchedMD. *Slurm Workload Manager.* `https://slurm.schedmd.com`. 2019.

[33] F. Shahzad, M. Wittmann, M. Kreutzer, et al. “A survey of checkpoint/restart techniques on distributed memory systems.” In: *Parallel Processing Letters* **23** (2013), pages 1340011–1340030.

[34] O. Tardieu. “The APGAS library: resilient parallel and distributed programming in Java 8.” In: *Proc. ACM SIGPLAN Workshop on X10* (2015).

[35] University of Kassel. *Scientific data processing.* `https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html`. 2019.

[36] G. Wrzesińska, R. V. Nieuwpoort, J. Maassen, et al. "Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid." In: *Proc. Int. Parallel and Distributed Processing Symp.* 2005.

[37] G. Wrzesińska, A. Oprescu, T. Kielmann, et al. "Persistent Fault-Tolerance for Divide-and-Conquer Applications on the Grid." In: *Proc. Euro-Par.* Springer LNCS 4641, 2007, pages 425–436.

[38] W. Zhang, O. Tardieu, D. Grove, et al. "GLB: Lifeline-based Global Load Balancing Library in X10." In: *Proc. First Workshop on Parallel Programming for Analytics Applications.* ACM, 2014, pages 31–40.

Appendix

Source code on CD

The source code of the algorithm's implementation and the source code used in the experiments is included on the attached CD.

Correctness test cases

1. Crash of place 2 after processing at least one task, but before answering steal requests. This case simulates the failure of a task bag with no children in the steal tree.
2. Crash of place 2 after sending tasks to another place. This case simulates the failure of a task bag with at least one children in the steal tree.
3. Crash of place 2 after sending tasks to another place. In addition, the place receiving the tasks crashes too. This simulates simultaneous failure of two directly connected task bags in the steal tree.
4. Crash of place 2 after processing some tasks. Places who create replay units due to this crash, crash before the units are assembled.
5. Crash of place 2 after processing some tasks. The place who stole the replay unit crashes before the processing of the replay unit begins.
6. Crash of place 2 and place 3 after about half of the expected execution time. This case does not cover any specific failure situation, but helped to test that no situation was left unconsidered.
7. Crash of all places except place 0.

Execution times

Places	GLBCoopOneSteal	FTGLB	DistLogFTGLB
12	647.67	658.25	656.18
24	327.02	333.78	329.52
36	870.12	884.03	877.51
48	654.21	671.56	660.88
60	523.03	534.74	529.19
72	436.84	445.87	441.15
84	1492.28	1516.21	1512.23
96	1307.88	1330.02	1316.04
108	1161.99	1183.11	1173.16
120	1048.40	1067.35	1057.13
132	954.94	974.95	961.92
144	874.37	891.62	881.61

Table 1: UTS on Kassel: Execution time in seconds

Places	GLBCoopOneSteal	FTGLB	DistLogFTGLB	DistLogFinishFTGLB
40	280.36	286.23	281.71	295.95
120	374.76	383.73	399.22	395.74
200	230.49	238.77	235.35	241.49
280	169.45	174.72	168.44	179.03
360	133.55	137.31	132.72	148.29
440	435.47	451.51	426.78	461.37

Table 2: UTS on Goethe: Execution time in seconds

Places	GLBCoopOneSteal	FTGLB	DistLogFTGLB
12	96.92	98.58	97.68
24	397.97	402.26	398.45
36	265.60	269.06	266.76
48	199.84	203.05	200.39
60	160.37	162.33	161.29
72	133.94	135.82	134.45
84	981.87	989.22	984.75
96	861.79	867.68	862.57
108	767.11	770.39	767.04
120	689.50	693.68	690.09
132	627.40	631.07	629.69
144	575.24	579.47	577.83

Table 3: N-Queens on Kassel: Execution time in seconds

Places	GLBCoopOneSteal	FTGLB	DistLogFTGLB	DistLogFinishFTGLB
40	325.43	337.15	321.48	345.60
120	118.06	114.68	119.30	122.90
200	70.83	74.62	71.80	82.24
280	436.88	435.84	437.36	455.36
360	357.94	341.36	350.46	369.38
440	292.89	288.43	286.94	307.56

Table 4: N-Queens on Goethe: Execution time in seconds