UNI KASSEL
VERSITÄT

Universität Kassel

Bachelor Thesis

An Asynchronous Backup Scheme Tracking Work-Stealing for Reduction-Based Task Pools

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

Lukas Reitz
33211934

Kassel, September 7, 2018

Examiners:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

Contents

List of Abbreviations **iii**

Statutory Declaration **iv**

1 Introduction **1**

2 Background **4**
- 2.1 APGAS Library . . . 4
- 2.2 Hazelcast Library . . . 5
- 2.3 APGAS_GLB . . . 6
- 2.4 FTGLB . . . 8
- 2.5 IncFTGLB . . . 9

3 Algorithm **10**
- 3.1 Overview . . . 10
- 3.2 Steal Backups . . . 14
- 3.3 Regular Backups . . . 14
- 3.4 Asynchronism . . . 14
- 3.5 Recovery . . . 15
- 3.6 Comparison . . . 16

4 Implementation **18**
- 4.1 Steal Backups . . . 18
- 4.2 Asynchronism . . . 19

5 Experiments **20**
- 5.1 Setup . . . 20
- 5.2 Unbalanced Tree Search . . . 20
- 5.3 Betweenness Centrality . . . 21
- 5.4 Synthetic Benchmarks . . . 21
- 5.5 Configuration . . . 21
- 5.6 Results . . . 23
- 5.7 Discussion . . . 27
- 5.8 Correctness . . . 28

6 Conclusions **29**

Bibliography **30**

Appendix **33**

List of Abbreviations

MTBF	Mean Time Between Failures
GLB	Global Load Balancing framework [24]
FTGLB	Fault tolerant GLB framework [15]
IncFTGLB	Incremental FTGLB framework [4]
LocalFTGLB	New framework from this thesis
PGAS	Partitioned Global Address Space model
APGAS	Asynchronous Partitioned Global Address Space library [22]
APGAS_GLB	"APGAS for Java" variant of GLB [16]
UTS	Unbalanced Tree Search benchmark [13]
BC	Betweenness Centrality benchmark [5]
DynamicSyn	Synthetic benchmark resembling UTS [4]
StaticSyn	Synthetic benchmark resembling BC [4]

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, September 7, 2018
Lukas Reitz

1 Introduction

Fault-tolerance is becoming more important as the size of parallel systems grows. Assuming a Mean Time Between Failures (**MTBF**) of 100 years per processing unit, a system consisting of 100,000 processing units will experience three failures per day on average [3].

A common approach to fault-tolerance is checkpointing [7]. It regularly saves a working state of the computation in memory or on disc. System-level checkpointing saves all process data, and thus imposes no additional burden on programmers to achieve fault-tolerance. In contrast, application-level checkpointing saves selected data only, and thus reduces the backup volume. On the backside, application-level checkpointing causes burden on the application programmer, which can be reduced by implementing it in a reusable library [15].

Checkpointing can be either coordinated or uncoordinated. In coordinated checkpointing, all computation units must agree on a time to write a backup. In uncoordinated checkpointing, the units write their backups individually at different times.

Parallel computations are often divided into subcomputations, called tasks, and implemented with a task pool. A task pool is a pattern of parallel computing, in which the tasks are processed by several computational units, called workers. Each worker has its own local pool, from which it takes out tasks, and into which it inserts newly generated tasks. Two established techniques for load balancing with task pools are work-sharing and work-stealing [1]. In work-sharing, overloaded workers give tasks to others. In work-stealing, workers with empty pools steal tasks from others.

The task pool pattern is well-suited for application-level checkpointing, because tasks are appropriate units for restoring a computation. We will later save in our backups the local pool contents, and a partial result, which reflects the results of previous computation.

An example for a library utilizing the task pool pattern is the Global Load Balancing (**GLB**) framework. GLB realizes a reduction-based task pool, i.e., each task has a result, and the overall result is computed by reduction from task results. GLB is based on the lifeline scheme, which was developed for the parallel programming language X10 [18]. X10 and the lifeline scheme operate in a Partitioned Global

Address Space (**PGAS**) setting. The PGAS model describes a cluster as a number of *places*. A place is a part of system memory together with some computational resources. Usually one place corresponds to one node. Each place can access the memory of all other places, but local accesses are faster.

The Asynchronous Partitioned Global Address Space (**APGAS**) model adds asynchronism to PGAS by incorporating tasks, called *activities*. Activities can be spawned locally or remotely, and in a synchronous or asynchronous way. In APGAS, execution starts at place 0. Later, the computation spreads to other places by spawning remote activities.

GLB was ported in two variants to the “APGAS for Java” parallel programming library [16], which implements the APGAS model. The variant used in this thesis deploys cooperative work-stealing, i.e., a thief sends out steal requests and waits for an answer. The answer can contain tasks, but can also be a reject message indicating that the victim has no tasks to share. It is called **APGAS_GLB** in this thesis. APGAS_GLB provides locality-flexible tasks. They are spawned by one worker on its place, and may later be moved around to achieve load balancing.

Recent papers describe two fault-tolerance schemes for APGAS_GLB [4, 15]. Both deploy uncoordinated application-level checkpointing. Backups are written before the computation starts, in regular time intervals, in the event of work-stealing, during recovery, and when the computation ends. The first scheme [15] is called **FTGLB** in this thesis. It updates checkpoints by replacing them with the new state.

The second scheme [4] is called **IncFTGLB** in this thesis. In contrast to FTGLB, IncFTGLB updates checkpoints incrementally by sending differences since the last checkpoint only. IncFTGLB restricts the selection of data structures for the local pool by requiring that work-stealing and task processing must operate at opposite ends. Further, IncFTGLB only saves so-called stable tasks. A task is called *stable*, if it remained in the pool since the last backup. Results from [4] showed an increase of fault-tolerance overhead for some benchmarks, but also a decrease of overhead for benchmarks with large task descriptors. Both schemes are further described in Chapter 2.

This thesis describes a third, new backup scheme for APGAS_GLB called **LocalFTGLB**. A difference to FTGLB and IncFTGLB is that checkpoints don’t contain the task pool data structures, but subsets of tasks with the accumulated results of finished tasks, and lists of logged work-stealing events. Saving work-stealing events avoids the need to write the remaining tasks at the victim, which are potentially large, to the checkpoint.

Another property of LocalFTGLB is asynchronous backup writing. Most backups written during failure-free execution are non-blocking, i.e., they are written in parallel to the task processing.

Further, LocalFTGLB tries to select the backup time in such a way, that the checkpoint size does not increase. Instead of writing the backup after a fixed number of computation steps, an interval is used which provides a lower and upper bound for the number of computation steps until a backup is written.

While asynchronous writing of backups reduces overhead in benchmarks with task descriptors of any size, logging work-stealing events and variation of the backup time mainly reduce overhead in benchmarks with large task descriptors.

The execution time was measured for APGAS_GLB, FTGLB, IncFTGLB and LocalFTGLB in experiments with the same benchmarks as in [4]. Overhead was calculated as the percentage of execution time difference to APGAS_GLB. In two benchmarks, Unbalanced Tree Search (UTS [13]) and Betweenness Centrality (BC [5]), almost all configurations showed a lower overhead for LocalFTGLB than for FTGLB and IncFTGLB. In two synthetic benchmarks with a varying task descriptor size [4], the overhead of LocalFTGLB was at most about the same as in FTGLB and above that of IncFTGLB.

This thesis is organized as follows: Chapter 2 describes APGAS_GLB and the two previous schemes and libraries in greater detail. Then, Chapter 3 explains the new backup scheme and compares it informally to FTGLB and IncFTGLB. Thereafter, Chapter 4 provides implementation details. Next, Chapter 5 describes experiments and discusses results. The thesis finishes with conclusions in Chapter 6.

2 Background

2.1 APGAS Library

The "APGAS for Java" library provides the Asynchronous Partitioned Global Address Space (**APGAS**) parallel programming model for the Java programming language [19, 22]. It is available online as an open source repository [9]. In this thesis, **APGAS** will denote the library instead of the model. As noted in the introduction, the PGAS model partitions memory and computational resources into places. APGAS adds asynchronism by organizing the computation into asynchronous tasks called **activities**. Activities can be spawned locally or remotely. When spawning a remote activity, final and quasi-final variables used inside the activity are copied and sent to the target place. The programmer has full control over data placement. Remote data can be accessed through global references.

A place in APGAS corresponds to a JVM. All places are interconnected with the Hazelcast library [6] at the network layer. Places are identified by integer ids, counted from 0. For intra-place parallelization, APGAS utilizes Java's Fork/JoinPool [11], which offers dynamic load balancing using work-stealing.

The most important constructs of APGAS are:

- `async()`: Spawns a local activity.
- `asyncAt()`: Spawns a remote activity.
- `at()`: Spawns a remote activity and waits for its completion.
- `finish()`: Spawns a local activity, which waits for all activities spawned by it. Exceptions are collected at the innermost enclosing `finish()`.
- `uncountedAsyncAt()`: Same as `asyncAt()`, but is not waited for by an enclosing `finish()`.
- `GlobalRef`: Global heap reference encapsulating an object contained in the local heap, thus making it globally accessable. Accesses to `GlobalRefs` are conducted in a remote activity on the place where the object is residing.

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

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

Listing 2.1: Distributed Hello World implementation in APGAS

Apart from `finish()` and `at()` for inter-place synchronization, APGAS does not provide any synchronization constructs. Instead, Java's constructs for intra-place synchronization, e.g., `synchronized` with `wait()`/`notify()` and atomic variables from the `Atomic` classes can be used.

Listing 2.1 shows the structure of a typical APGAS program. All APGAS constructs are imported statically to make it easier to use them. The `main`-method contains one `finish()` to prevent premature termination. It waits for the completion of all asynchronous activities spawned with `asyncAt()`.

APGAS provides functionality for fault-tolerant cluster-computing. For instance, it supports distributed, fault-tolerant termination detection with the `finish()` construct, by utilizing Hazelcast's `IMap`, see Section 2.2. User-level fault-tolerance is supported, e.g., through exceptions thrown on place failure and a failure handler method, called `placeFailureHandler`. The `placeFailureHandler` can be registered on each place and is called eventually after a place failure is detected. Fault-tolerance in APGAS is subject to two limitations:

- Failure of the origin place of a `finish` (e.g. place 0) can not be recovered from, because APGAS is not able to migrate the `finish` to another place.
- Loss of an `IMap` entry due to too many simultaneous place failures leads to shutdown of the runtime system with an error message.

2.2 Hazelcast Library

Hazelcast is a Java framework providing resilient, distributed data structures [6]. Hazelcast's data structures can tolerate multiple simultaneous place failures, because data is replicated up to a maximum of six times. The parameter specifying

the number of replications is called **backupCount**. When too many places fail simultaneously, data is lost, in which case a handler method is called. Inside it, the application programmer can implement a way to, e.g., safely terminate the application. The handler method can be registered by calling `addPartitionLostListener()` on the `IMap`.

Internally, an `IMap` is divided into partitions, which are evenly distributed across all places. A hash function assigns each entry to a partition. An even distribution is ensured by redistributing partitions when a place leaves or joins the cluster.

If a user program needs to modify multiple entries and partial updates would lead to an inconsistent state, Hazelcast's transactions can be used. Transactions ensure an atomical update of multiple entries. If one or more updates fail, no entry is updated. All entries get locked during a transaction and get automatically unlocked after the transaction is completed.

To save network traffic, an `EntryProcessor` can be used to update `IMap` entries. `EntryProcessor`s are operations on `IMap` entries, which get executed directly on the places where the entry resides. They retain the order of operations and implicitly lock and unlock the entry. `EntryProcessor`s can be executed synchronously, by calling the `executeOnKey()` method, or asynchronously, by calling the `submitToKey()` method. The `submitToKey()` method returns a `Future` object. Calling the `get()` method on this object blocks the current thread until the operation is executed on all places the entry resides.

2.3 APGAS_GLB

APGAS_GLB is a framework that implements a reduction-based variant of the task pool pattern. It provides dynamic load balancing by work-stealing. The APGAS_GLB framework deploys the following task model (cited from [15]):

- 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.

In APGAS_GLB, a place employs exactly one worker, so multiple places must be used on a single machine with multiple cores to make use of the available resources. Every computation begins with at least one task at one place from which new tasks can be generated. If a worker has no tasks, it sends out steal requests to w random victims. After sending out a random steal request, the worker waits for an answer, which can either be a reject message or a message containing tasks. If these attempts were unsuccessful, steal requests are sent to z so-called **lifeline buddies**. Such lifeline steal requests are answered with a message containing tasks, if the victim has tasks to share. Otherwise, it is recorded on the victim and answered with a message indicating the recording of the request. Recorded requests are answered with a message containing tasks when the victim has tasks to share. Lifelines are calculated before the beginning of the task processing, such that the relations form a z-dimensional hypercube [18]. When stealing failed for all $w+z$ requests, the worker goes into an inactive state, from which it can only be reactivated by receiving tasks from a lifeline buddy. A `finish()` is used on place 0, which terminates when all workers are inactive. After that, the reduction phase starts. Here, the partial results from all workers are collected at place 0 and the final result is calculated by reduction.

The main loop of APGAS_GLB is shown in Listing 2.2 (adapted from [16]). When it is exited, the worker enters the inactive state. Accesses to the local pool are protected by Java's `synchronized` keyword used in conjunction with an object.

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

Listing 2.2: Main loop of APGAS_GLB

APGAS_GLB requires the application programmer to implement the local pool class with the following methods:

- `process(n)`: This method processes `n` tasks and returns a `boolean` value which indicates if the local pool still contains tasks. It returns `true` if the pool is not empty, otherwise `false`.
- `merge(tasks)`: This method merges `tasks` into the local pool.

- `split()`: This method splits the local pool to extract loot for the thief in work-stealing.

The programmer must also implement the data structure for transferring tasks, called `TaskBag`. It is an interface with only a method `size()`, which returns the number of tasks contained.

2.4 FTGLB

This section only outlines FTGLB. Further information can be found in [15]. FTGLB implements uncoordinated application-level checkpointing for APGAS_GLB. It consists of two components: checkpointing and recovery.

Checkpointing regularly writes backups of the whole pool data structure to a dedicated `IMap` called `iMapBackup`. Backups are written after $k \cdot n$ computation steps. They are also written during work-stealing, as well as when a worker enters inactive state, and in recovery. To save tasks which are sent by the victim but not yet received by the thief, an additional `IMap` called `iMapOpenLoot` is used.

When a place fails, its recovery is conducted by an alive place, called **recovery place**. When places are arranged in a ring, wrapping around at the lowest and highest place id, it is the closest alive predecessor of the failed place.

First, the checkpoint of the failed place is recovered by merging the saved tasks into the pool of the recovery place. Also, the recovered tasks are removed from the checkpoint. After recovery, the checkpoint only contains the partial result of the tasks which the failed place processed.

Next, tasks stolen from the failed place that are saved in `iMapOpenLoot` get restored. If the thief is still alive, the tasks are sent again to the thief. Otherwise, they get merged into the local pool and into the checkpoint of the recovery place. After recovery, the checkpoint is marked as *recovered* by setting a `boolean` variable `done` of the saved pool data structure to `true`. This is done to prevent late backups from overwriting the already recovered checkpoint. At every writing of a backup, this variable is checked and the checkpoint is only modified if the variable is set to `false`.

Furthermore, each alive place checks if it sent tasks to the failed place, which were not fully received, by inspecting `iMapOpenLoot`. These tasks get re-merged into the victim's pool.

FTGLB requires the following methods to be implemented for the local pool data structure:

- `getAllTasks()`: Returns all tasks currently contained in the local pool.
- `clearTasks()`: Removes all tasks from the local pool.

2.5 IncFTGLB

The IncFTGLB fault-tolerance scheme uses a different checkpointing component than FTGLB, but the same recovery component. It reduces the backup volume in two ways:

- The backup sent contains only the changes to the last checkpoint. Tasks already contained in the checkpoint are not sent again.
- Only stable tasks are saved. The checkpoint still contains an actual state of the pool that existed at some point in time.

Use of IncFTGLB requires that processing of tasks and work-stealing operate on opposite ends of the task queue. Also, after *each* processed task, control must be returned to the framework.

Stable tasks are determined by monitoring the pool between processing of single tasks. The monitoring consists of taking *snapshots*, which contain the task at the top of the queue, the size of the queue and the current partial result. Each time the pool size becomes lower than the size recorded in the last snapshot, a new snapshot is taken. Snapshots are also taken at the beginning of a backup interval. A working state of the pool can be reconstructed from the last snapshot.

IncFTGLB requires a lot more methods to be implemented by the application programmer than FTGLB, increasing the burden. The main differences in the application programming interface are:

- The `process()` method processes only the uppermost task.
- Various new methods for adding, removing, and reading a number of tasks from the bottom and the top of the task queue are required. These methods are used for generating the snapshots and the backups, and applying the backups to the checkpoints in an incremental way.

Recovery is done as in FTGLB. Also, the initial, final, and recovery backups are the same as in FTGLB.

3 Algorithm

The following sections describe the new algorithm. First, Section 3.1 outlines the general ideas. Next, in Section 3.2 and Section 3.3, backup types are described. The asynchronous writing of the backups is described in Section 3.4. Further, in Section 3.5, the recovery of lost tasks is explained. The chapter finishes with an informal comparison to FTGLB and IncFTGLB in Section 3.6.

3.1 Overview

The main objective for designing LocalFTGLB was to reduce the overhead for checkpointing in comparison to FTGLB and IncFTGLB.

Reasons for the overhead in FTGLB and IncFTGLB are:

- Saving tasks, which are already contained in the checkpoint
- Monitoring of the local pool between the backups
- Waiting for network communication during the writing of the backups

LocalFTGLB is based on a different approach to fault-tolerant task pools, which originates from [10], and was adapted to reduction-based task pools in [2]. Reference [10] presents a fault-tolerant algorithm for nested fork-join programs, where tasks are distributed by work-stealing. When a worker steals a task, the steal relation is saved on the victim. These steal relations form a global steal tree [12]. When a worker fails, the lost node in the steal tree gets restored by recomputing a subset of the tasks of that node. Not all tasks necessarily need recovery, because their children might have been stolen by another worker. To determine the tasks to be recomputed, the steal relations from all workers are collected at the worker of the parent node of the lost node. From these relations, a so-called replay tree is formed, which is a subtree of the steal tree with additional information needed for recovery. The replay tree is eventually sent to another worker by work-stealing. Recovery is conducted at the thief of the replay tree by recomputing the lost tasks of the replay tree and reconnecting the tasks of the children of the lost node. Reconnecting includes saving steal relations to the children and informing the workers of

the children about the changed parent node. Tasks of children, that finished before the worker failed, are recomputed, since their result was already returned to the failed worker and thus went lost.

Reference [2] adapted the scheme from [10] to reduction-based task pools. In reduction-based task pools, tasks don't return their result to their parent task, because the final result is calculated by reduction. LocalFTGLB adapts the technique to **track work-stealing** from [2], which is also based on APGAS_GLB, and combines it with backups. In [2], steal relations are sent along the tasks in work-stealing, whereas in LocalFTGLB, steal relations are saved in a distributed data structure.

In APGAS_GLB, if tasks are stolen from a place, the local pool of the victim is split. The split is executed by a call to the `split()` method. Unlike as in FTGLB, in LocalFTGLB, the remaining tasks of the local pool are not written to the checkpoint. Instead, the number of tasks that have been processed at the time of the split is written to the checkpoint. On recovery, the stolen tasks are removed from the pool by calling `split()` at the corresponding computation steps. This technique leads to smaller steal backups and thus reduces the overhead.

Every place saves the state of the local pool at specific times. The saved state is called checkpoint. As in FTGLB and IncFTGLB, the **frequency of updating the checkpoints** can be set by the application programmer to adjust it to the MTBF of the system used. In contrast to FTGLB and IncFTGLB, the application programmer does not provide a fixed number of computation steps after which a checkpoint is updated, but an interval, in which the checkpoint has to be updated. The framework guarantees that the writing of the backup begins earliest after $n \cdot k_{lower}$, and latest after $n \cdot k_{upper}$ computation steps. This allows the framework to choose a state of the pool which contains less tasks than the last checkpoint to be included in the backup. For this technique, the local pool must be monitored to inspect the pool size. In contrast to the monitoring used in IncFTGLB, where the pool size is inspected after the processing of each task, in LocalFTGLB, the pool size is only inspected regularly every n computation steps and only within the allowed interval as described above. This technique is further described in Section 3.3. Within the interval, each place updates its checkpoint when the pool size is lower or equal to the checkpoint's pool size. This reduces the chance of an increase of the checkpoint's size. If the pool size does not become lower or equal to the checkpoint's size until the last computation step of the interval, a regular backup is written, regardless of the actual pool size.

To hide the waiting for network communication when backups are written, **asynchronous checkpointing** is used. Here, a local copy of the tasks and the partial

result is created, and the computation is continued thereafter. The backup is written in parallel to the ongoing computation. In LocalFTGLB, the guarantees of the checkpointing interval as described above are preserved by waiting for the backups to complete at certain points of the execution. The asynchronism is further described in Section 3.4. In a different context, a comparison of synchronous and asynchronous checkpointing was conducted in [21], where a 10-20 times lower overhead was found by deploying asynchronous checkpointing instead of synchronous checkpointing.

LocalFTGLB makes use of Hazelcast's `IMap`. Instead of saving objects of the task pool data structure of each place as in FTGLB and IncFTGLB, only `TaskBag` objects are saved. A `TaskBag` object contains the following data:

- A set of tasks.
- A unique integer identifying the `TaskBag` object.
- The number of regular backups of this `TaskBag` written, including the current one. This number is called **backupId**.
- A partial result which is formed from the results of finished tasks of this `TaskBag`.
- A boolean value indicating if this `TaskBag` object was already recovered (see below).

The `IMap` containing the `TaskBag` objects is called `bagMap`. It holds one entry per place with the place id as key.

At the beginning of the computation, each place saves a `TaskBag` object to the `bagMap`, containing the initial tasks. This backup writing is called *initial backup*.

Another `IMap`, called `splitMap`, is used for **saving the steal answers** by containing a list of `TaskBagSplit` objects for each `TaskBag` object. The unique integer of the corresponding `TaskBag` object is used as the key. Because every answered steal request splits the pool, `splitMap` contains lists of logged pool splits, that occured at certain computation steps. Each `TaskBagSplit` object describes an answered steal request by containing the following information:

- The number of computation steps since the beginning of the processing of the `TaskBag`'s tasks till the answer of the steal request.
- The backupId of the corresponding regular backup. In recovery, the split is only executed if the regular backup contains the same pool state as the pool state the split occured on. A regular backup which is written after the steal

backup implicitly contains the split by not having the stolen tasks in the checkpoint's pool.

The checkpoint of a place consists of the corresponding entry in `bagMap` and the entry in `splitMap`.

After the computation, all `TaskBag`s, which then only contain a result, are collected at place 0 and the final result is calculated by reduction. A crash of any place other than place 0 does not affect the reduction because the final checkpoints saved in `bagMap` contains the final results of each place. The reduction does not access any places directly with APGAS, but fetches the results from `bagMap`.

As in FTGLB and IncFTGLB, the application programmer needs to implement some methods. The number of methods to be implemented is the same as in FTGLB. Disregarding methods needed by GLB, the following methods for the pool data structure need to be implemented:

- `getAllTasks()`: Returns all tasks currently contained in the local pool.
- `clearResult()`: Removes all finished tasks by resetting the partial result to the same state as it was before the beginning of the computation.

The data structure representing a result requires the following method to be implemented:

- `mergeResult(result)`: Merges another result.

The algorithm expects at all times, that **all tasks of the local pool originated from one place**. In GLB, tasks from multiple lifeline buddies can arrive at one place, even when the pool is not empty. In LocalFTGLB, this is changed. Before splitting the local pool, a lifeline buddy asks the thief if the thief's local pool is not empty by spawning an asynchronous activity at the thief's place. This is called *lifeline delivery request*. If the thief does not need tasks or another lifeline buddy is already giving tasks, the activity ends. If a random steal request is in progress, the activity waits until it is finished. If the random steal was successful, the activity ends. Otherwise, an asynchronous activity gets spawned on the victim's place, initiating the split of the pool and the sending of the tasks in the same way as GLB does. If at this time, the victim's pool has become empty, the request gets recorded and is answered again later, by asking if the thief needs tasks. From the time a lifeline delivery request is accepted to the time the tasks arrive and get merged into the local pool, no random steal requests are sent and other lifeline delivery requests are

declined. All of the activities spawned during a lifeline delivery request are waited for by the `finish()` on place 0.

As in FTGLB, a counter is used for deciding when to write a regular backup. The counter is incremented every n computation steps. When initiating the writing of a backup, the counter is reset. Also, a regular backup is written and the counter is reset when the pool becomes empty.

3.2 Steal Backups

Steal backups are written in the event of work-stealing on the victim side. This is the only type of backup where the checkpoint of another alive place is modified. It consists of two parts, because it modifies two checkpoints at once: The steal backup modifying the victim's checkpoint is called $\text{backup}_{\text{victim}}$ and the steal backup modifying the thief's checkpoint is called $\text{backup}_{\text{thief}}$.

$\text{backup}_{\text{victim}}$ contains a `TaskBagSplit` object with all information as described before. No tasks are read or written when writing this backup. $\text{backup}_{\text{thief}}$ contains a `TaskBag` object with the stolen tasks.

Changes to both checkpoints need to be visible at the same time and partial changes are not allowed, because tasks can then either get lost or get duplicated. Thus, the steal backup uses a transaction to modify both checkpoints at the same time.

3.3 Regular Backups

The regular backup contains a `TaskBag` object with the current tasks of the local pool and the current partial result. Before each writing of a regular backup, the backupId is incremented.

3.4 Asynchronism

The regular and the steal backups are written in parallel to the task processing. Starting the asynchronous writing of a backup is called *starting a backup*. Waiting for a backup to be completely written is called *waiting for a backup*.

Because most of the time of backup writing is spent waiting for network communication, asynchronous backup writing can help to maximize resource usage by

processing tasks while waiting for network communication. The longer the writing of a backup takes, the more time is saved.

The algorithm has one main constraint for the asynchronism:

- The maximum number of tasks to be recomputed in recovery may not exceed $2n \cdot k_{upper}$.

Since the number of tasks processed between the starting of backups is at most $n \cdot k_{upper}$, the algorithm waits for the last backup before starting another one.

During work-stealing, no backups may be written by the thief, because the victim writes the backup for the thief. This implies that the thief waits for all backups before sending out random steal requests or accepts lifeline delivery requests. It also guarantees that all regular backups written by the thief are complete. This prevents the victim from overwriting the stolen tasks in the checkpoint.

3.5 Recovery

LocalFTGLB registers a `placeFailureHandler` on each place. The recovery is conducted within this handler. The handler is executed at each place some time after the failure is detected. Because a `placeFailureHandler` can not be added to the scope of a `finish()`, the `restartDaemon` from [15] is used. The `restartDaemon` is an activity inside the scope of the `finish()`, which terminates when all workers are inactive and all `placeFailureHandler`s for all failed places have been executed.

Recovery of a place's checkpoint is executed by the recovery place, which is determined the same way as in FTGLB. The recovery place's checkpoint is denoted as $checkpoint_{recover}$ and the failed place's checkpoint is denoted as $checkpoint_{failed}$. Recall that a checkpoint corresponds to a saved entry in `bagMap` and the corresponding entry in `splitMap`.

During recovery, the recovery place does not answer steal requests, because the local pool is used for processing the recovered tasks and is thus locked. All tasks to be recovered are always processed in the `placeFailureHandler` at the recovery place. The first step in recovering a failed place is to process all tasks in the recovery place's pool. When the pool is empty, $checkpoint_{recover}$ is updated, because a backup is always written when the local pool becomes empty. The failed place's entry in `bagMap` is locked before reading the tasks from $checkpoint_{failed}$, to prevent any late update during recovery. Then, the failed place's `TaskBag` object from `bagMap` is merged into the recovery place's pool. Next, for the `TaskBagSplit` objects in $checkpoint_{failed}$, the pseudocode in Listing 3.1 is executed.

```
1 bag = bagMap[failedPlace] ;
2 currentStep = 0 ;
3 foreach (split in splitMap[bag.id]) {
4   if (split.backupId == bag.backupId) {
5     pool.process(split.step - currentStep) ;
6     pool.split() and discard the extracted tasks ;
7     currentStep = split.step ;
8   }
9 }
```

Listing 3.1: Pseudocode of the recovery of work-stealing events

Since the logged splits are inserted in the order the steals were answered, no sorting is needed. After all splits have been executed, the pool will still contain tasks. These tasks are then processed and the bag's attribute `recovered` will be set to `true`. Finally, $checkpoint_{failed}$ is updated by writing the `TaskBag` object back to `bagMap` and the failed place's entry is unlocked. Because every backup checks the attribute `recovered` and only updates the checkpoint if the attribute is set to `false`, no late update can occur. A late update would lead to multiple recoveries of the same tasks.

When a victim fails after writing the steal backup, but before the thief received the stolen tasks, the tasks need to be restored. Since stolen tasks get written to the thief's checkpoint in $\text{backup}_{\text{thief}}$, it is sufficient, that the `placeFailureHandler` at each place waiting for tasks, checks if its checkpoint contains tasks which have not yet been received. If such tasks are found, they are merged into the pool and are processed as if they were received normally. To prevent the loot from arriving at a later time, each place keeps track of the unique ids of all received `TaskBag` objects. If a `TaskBag` object with the same id was already received, the message containing the tasks is ignored.

When a thief fails between the split of the victim's pool and writing the steal backup, the tasks get re-merged into the local pool of the victim and a regular backup is written. The regular backup is necessary, because merging the tasks does not guarantee them to be inserted at the same position they were removed from.

In the `placeFailureHandler` on each place, all outstanding steal requests to the failed place are treated as if they were rejected. Also, recorded steal requests from the failed place are discarded and no lifeline steal requests will be sent to it anymore.

3.6 Comparison

This section informally compares LocalFTGLB to FTGLB and IncFTGLB.

A difference to FTGLB is that the steal backup for the victim's checkpoint does not contain the remaining tasks of the pool. This is similiar to the incremental

update of the checkpoint as in IncFTGLB, where only the number of tasks stolen is contained in the backup. The number of tasks is removed from the bottom of the checkpoint's pool. In LocalFTGLB, the computation step and backupId are contained in the backup, so that the tasks can be removed later during recovery.

Like IncFTGLB, LocalFTGLB tries to reduce the number of tasks in the checkpoints. IncFTGLB uses snapshots to determine the state of the pool that contains the smallest number of tasks since the last backup. LocalFTGLB writes backups when the pool size is lower or equal than the pool size of the last checkpoint. The technique used in IncFTGLB saves the lowest number of tasks, at the price of a high monitoring cost. The technique used in LocalFTGLB does not necessarily save less tasks if the pool size stays above the pool size saved in the checkpoint, but requires only low monitoring cost.

As another difference to both other schemes, backups are written in parallel to the task processing. FTGLB and IncFTGLB wait for the backup to complete, before continuing to process tasks.

While FTGLB and IncFTGLB store the task pool data structure in `IMap` entries, LocalFTGLB stores `TaskBag` objects. The task pool data structure can contain variables that are not needed for restoring the tasks. For optimal performance, the programmer needs to mark them `transient`, to exclude them from the backups. Since `TaskBag` objects are only used for transferring tasks, they implicitly contain no such variables.

A major difference to FTGLB and IncFTGLB is that recovered tasks are not locality-flexible. During processing of these tasks, no steal requests are answered. In the worst-case, all tasks were stored at the recovery place and/or the failed place, and all other places did not have any tasks. The recovery place would process all tasks, which would lead to a processing time almost equal to the sequential processing time.

4 Implementation

The following sections describe some implementation details.

4.1 Steal Backups

```
1  abstract class TaskBag implements Serializable {
2      static final AtomicInteger LAST_ID = new AtomicInteger(1);
3
4      long bagId = ((long)(here().id) << 32) + LAST_ID.incrementAndGet();
5      int parentPlace = here().id;
6      LocalFTGLBResult ownerResult;
7      boolean recovered = false;
8      long backupId = 0L;
9
10     abstract public int size();
11 }
```

Listing 4.1: TaskBag class of LocalFTGLB

Recall that the application programmer must implement the `TaskBag` data structure used for transferring tasks. While in APGAS_GLB, `TaskBag` is an interface, in LocalFTGLB, `TaskBag` is an abstract class. This change was necessary, because the algorithm requires some variables inside the data structure. Listing 4.1 shows the implementation of the `TaskBag` class.

As described in Chapter 3, each `TaskBag` object has a unique id. The id is implemented as a 64 bit long value. The 32 highest order bits are set to the place id, and the remaining bits are set to the value of a counter, which is incremented with every `TaskBag` object created. The incorporation of both place id and local counter guarantees global uniqueness. The `backupId` is counted from 0 upwards.

The `TaskBag` class contains a variable `ownerResult` that is used for storing a partial result. The variable is only assigned an object representing the current result of the local pool when writing a backup. `TaskBag` objects sent in work-stealing do not have a result object assigned, because the thief's pool already contains the result.

The class `TaskBagSplit` is shown in Listing 4.2. It contains only two `long` variables. Thus, the size of each logged work-stealing event is small.

```
1 class TaskBagSplit implements Serializable {
2   long step;
3   long backupId;
4 }
```

Listing 4.2: TaskBagSplit class of LocalFTGLB

4.2 Asynchronism

Regular backups are written by using an `EntryProcessor`, which is executed within a thread managed by Hazelcast. The `Future` object returned from the `submitToKey()` method of the `EntryProcessor` is inserted into a `LinkedList`, called `runningBackups`. To wait for all backups running, the `get()` method is called on all elements of `runningBackups`. If the `get()` method returns, the corresponding element is removed from `runningBackups`. Access to `runningBackups` is protected by a `synchronized` block, which uses `runningBackups` as the lock object. Since LocalFTGLB allows only one running backup, `runningBackups` contains at most one `Future` object.

Steal backups are written by using a transaction, because multiple `IMaps` are modified. Because Hazelcast's transactions offer no methods for asynchronous execution, the writing of the steal backup is done within an asynchronous activity. For random steals, the activity is spawned with `uncountedAsyncAt(here(), ...)`, otherwise with `asyncAt(here(), ...)`. The task delivery is also conducted in this activity after the steal backup has been written. Because lifeline task deliveries use `asyncAt()`, it is important that the activity does not use `uncountedAsyncAt()` for lifeline steals. Using `uncountedAsyncAt()` would prevent the `asyncAt()` from being waited for by the `finish()` on place 0.

Because activities do not inherit the locks of surrounding `synchronized` blocks, an `AtomicBoolean`, called `blockBackups`, is introduced. This variable is set to `true`, before the activity is spawned, and is set to `false`, when the steal backup has been written. Backups are only started if `blockBackups` is `false`. The waiting is implemented by calling `wait()` on `runningBackups` in a loop with the condition `blockBackups.get() == true`. When another activity sets it to `false`, `notifyAll()` is called on `runningBackups`, waking up any activities waiting for the value of `blockBackups` to change.

5 Experiments

This chapter describes the experiments conducted. First, in Section 5.1, the hardware and software setup is described. Next, the benchmarks used in the experiments are described in Section 5.2–5.4. Section 5.5 lists the used parameters for the experiments. Thereafter, Section 5.6 shows results and Section 5.7 discusses them. The chapter finishes with a description of conducted experiments, which tested the correctness.

5.1 Setup

Experiments were conducted on the FB16 partition of the High Performance Computing cluster of University of Kassel. The partition consists of 12 homogenous nodes connected using InfiniBand. Each node of the cluster comprises two 6-core Intel Xeon E5-2643 v4 CPUs and 256 GB main memory [23]. This results in a total of 144 cores. Benchmarks were scheduled with the installed job scheduler Slurm [20] which guaranteed exclusive usage of the cluster during the execution. All experiments were conducted with Hazelcast in version 3.10.2 and Java in version 10.0.1.

5.2 Unbalanced Tree Search

Unbalanced Tree Search (UTS) calculates the size of a highly irregular tree generated from node descriptors [13]. It starts with one task corresponding to the root of the tree. Each task corresponds to one node of the tree. Processing a task may generate a number of new tasks, which are the children of the processed task's node. The number is calculated using the deterministic SHA1 function until a certain depth d is reached. Both the branching factor b and the depth d can be set as arguments to the benchmark. Additionally, an initial seed r for the random number generator can be passed.

5.3 Betweenness Centrality

Betweenness Centrality (BC) is a centrality measure for a vertex. It was first described in [5]. The benchmark calculates the betweenness centrality for each vertex in a graph generated by the parameters N, a, b, c, d and s, where N is the number of vertices in the graph, the parameters a, b, c, d determine the graph shape, and s is a seed for the random number generator.

5.4 Synthetic Benchmarks

The synthetic benchmarks were introduced in [4] for examining IncFTGLB when task descriptors are large. They are called **DynamicSyn** and **StaticSyn**.

In DynamicSyn, which resembles UTS, the computation starts with one task at one place, and tasks are generated dynamically until a specified maximum depth d is reached. Each task with a depth smaller than d generates at least 1 and at most c new tasks.

In StaticSyn, which resembles BC, the computation starts with an equal number of tasks at each place and no tasks are generated dynamically. The parameter t controls the total number of tasks.

Each task contains a variable ballast which can be set by parameter b in byte. The ballast is realized as a `byte` array, resulting in synthetically enlarged task descriptors. Additionally, a task granularity g can be set, which specifies the computational cost of each task by controlling the number of iterations of a loop calculating π.

5.5 Configuration

Benchmark	Parameters
UTS	$d = 17$, $b = 4$, $s = 19$, $n = 511$, $k = 2048$, $k_{lower} = 1450$, $k_{upper} = 4096$
BC	$N = 2^{17}$, $s = 2$, $a = 0.55$, $b = c = 0.1$, $d = 0.25$, $g = 511$, $k = k_{lower} = k_{upper} = 32768$
StaticSyn	$t = 100000$, $g = 300$, $n = 12$, $k = k_{lower} = k_{upper} = 8$
DynamicSyn	$c = 27$, $d = 7$, $g = 1$, $n = 128$, $k = 64$, $k_{lower} = 48$, $k_{upper} = 96$

Table 5.1: Benchmark and system configurations

Table 5.1 shows the used parameters for the benchmarks. The parameters are mostly the same as used in [4]. The length of the interval $[k_{lower}, k_{upper}]$ was not optimized for shortest execution time, but chosen by an educated guess for each benchmark. The k_{lower} and k_{upper} values were experimentally adjusted to approximately match the number of backups written in LocalFTGLB to the number of backups written in FTGLB. Each benchmark was executed with a backupCount of the used `IMaps` of 1 and 6.

As in GLB, with P being the number of places at the start, the parameter w is calculated by the formula

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

and the parameter l is calculated by the formula

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

The parameter z is calculated such that l^z is greater or equal to P.

For UTS and BC, the places were distributed to use the least number of nodes, but with a maximum of 12 places per node, which corresponds to the number of cores per node. For example, in experiments with up to 12 places, only one node was used. In experiments with 24 places, 2 nodes were used with 12 places on each node, and so on.

For the synthetic benchmarks, all 12 nodes were used, and the 144 places were equally distributed over all nodes. Instead of conducting experiments with a varying number of places, the ballast of the task descriptors was varied.

Each experiment was executed five times. The average of the execution times of each experiment is reported.

Two versions of APGAS were used. For UTS and BC, a fork was used, which contains bug fixes and additional features. It is available as an online repository [14]. The version of May 31, 2018 was used. For the synthetic benchmarks, the APGAS version from [4] was used. With the version from [14], with growing ballast, the execution time grew about 5 times faster than with the version from [4]. Because this might be a bug, no experiments with large task descriptors were carried out with it. The experiments with small task descriptors were conducted before noticing this behaviour. Due to time constraints, they could not be repeated with the APGAS version from [4]. In a group of experiments conducted with randomly picked con-

figurations, the execution times with small task descriptors did not differ in both versions.

Because [4] already showed that IncFTGLB has a higher overhead than FTGLB for UTS and BC, LocalFTGLB is only compared to FTGLB for these benchmarks. Also, all runtime measurements were conducted without simulated failures.

5.6 Results

The fault-tolerance overhead was calculated by the formula $time_{system}/time_{GLB} - 1$, where $system \in \{FTGLB, IncFTGLB, LocalFTGLB\}$. The result is expressed as a percentage. All results show a higher overhead when backupCount is set to 6, instead of 1.

For UTS, the overhead of both LocalFTGLB and FTGLB increases almost linearly with the number of places. Beginning from 12 places, the overhead difference of LocalFTGLB to FTGLB is almost constant at about 2% with LocalFTGLB having the lower overhead. For BC, LocalFTGLB shows a lower overhead than FTGLB for all, but two configurations. With a backupCount of 1 and 96 places, the overhead is about the same as in FTGLB. With a backupCount of 6 and 12 places, the overhead of LocalFTGLB is 25.18% and the overhead of FTGLB is 14.21%. The overhead increases almost linearly until 96 places, and then stays almost constant. For a backupCount of 1, the overhead remains at about 22% for LocalFTGLB, and at about 24% for FTGLB. For a backupCount of 6, the overhead remains at about 37% for LocalFTGLB, and at about 46% for FTGLB.

For StaticSyn, LocalFTGLB has about the same overhead as FTGLB for all configurations. IncFTGLB shows only a small linear increase of overhead with at most an overhead of 5.14% for a backupCount of 1, and 11.25% for a backupCount of 6. With a backupCount of 6, LocalFTGLB has an overhead of at most 102.58% and FTGLB has an overhead of at most 110.45%.

For DynamicSyn, the overhead of FTGLB increases almost logarithmic. FTGLB shows at most an overhead of 88.58% with a backupCount of 1 and an overhead of at most 185.29% with a backupCount of 6. The overhead of LocalFTGLB oscillates around the overhead of IncFTGLB. Both, IncFTGLB and LocalFTGLB show a slower increase of overhead than FTGLB. In most configurations, IncFTGLB and LocalFTGLB stay below the overhead of FTGLB.

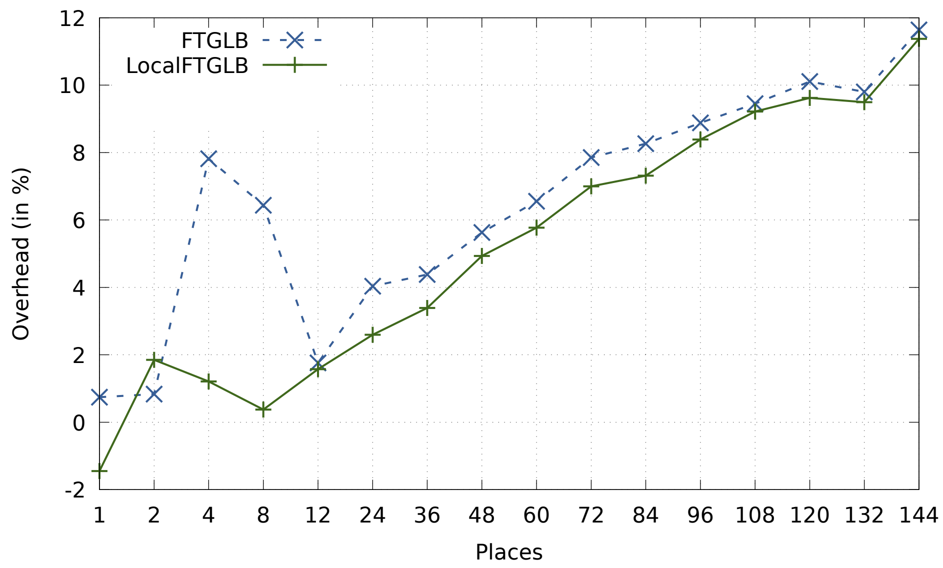

Figure 5.1: UTS with backupCount=1

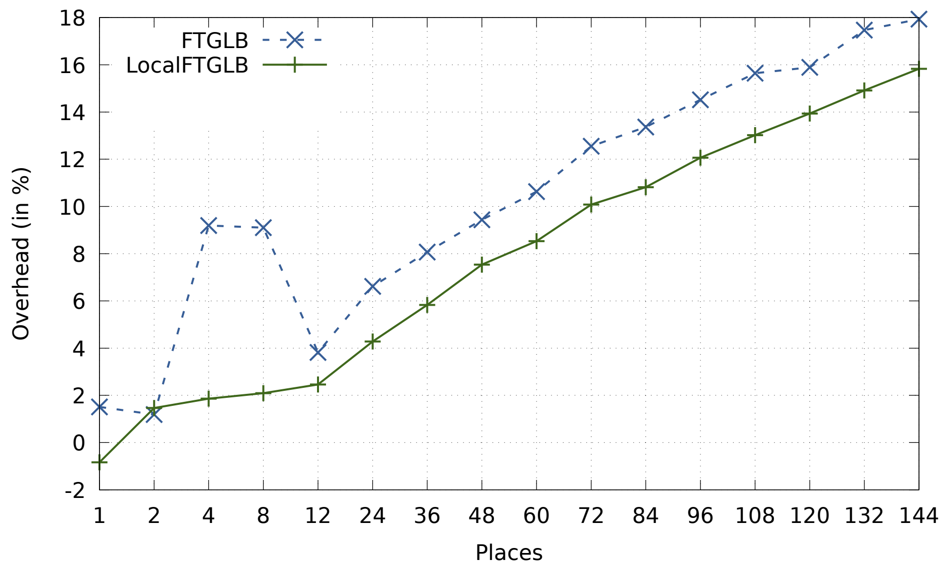

Figure 5.2: UTS with backupCount=6

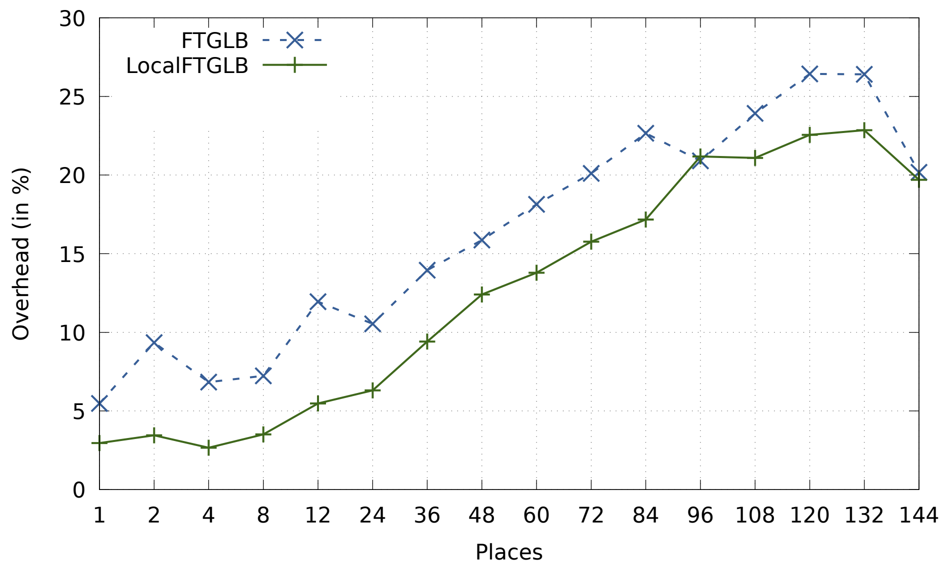

Figure 5.3: BC with backupCount=1

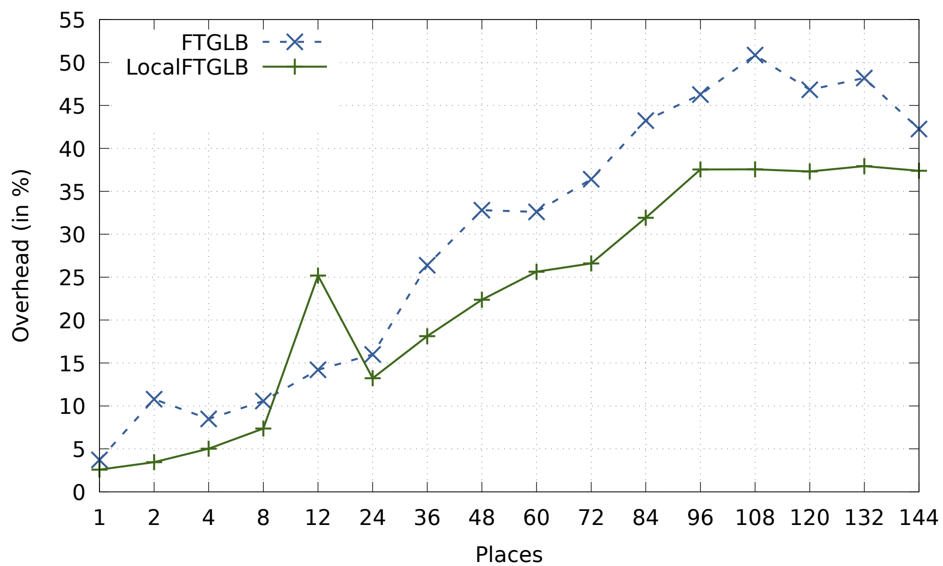

Figure 5.4: BC with backupCount=6

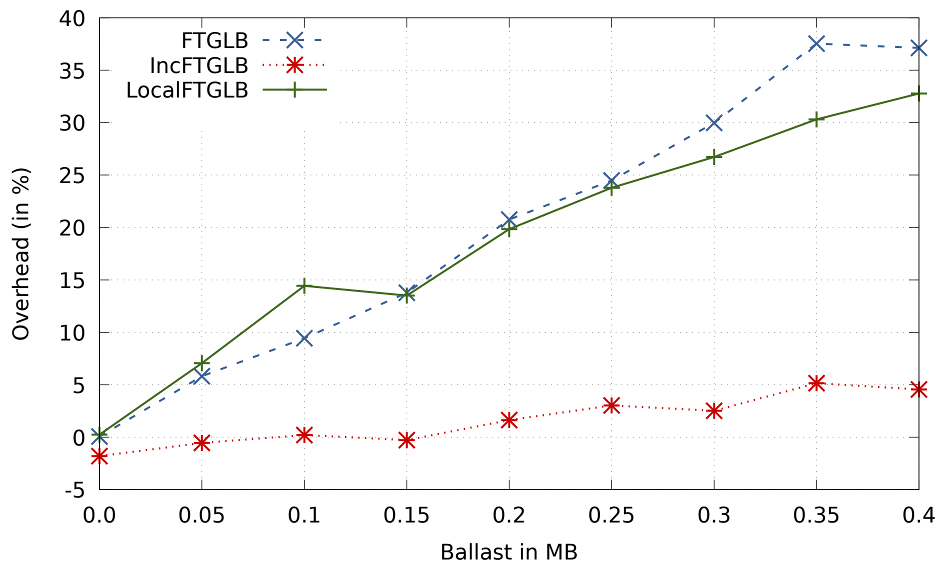

Figure 5.5: StaticSyn with 144 places and backupCount=1

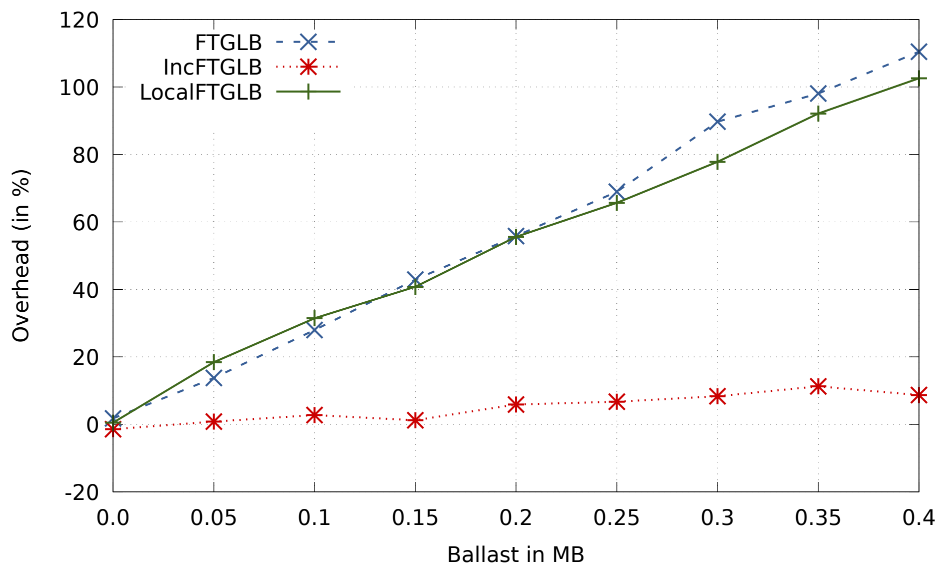

Figure 5.6: StaticSyn with 144 places and backupCount=6

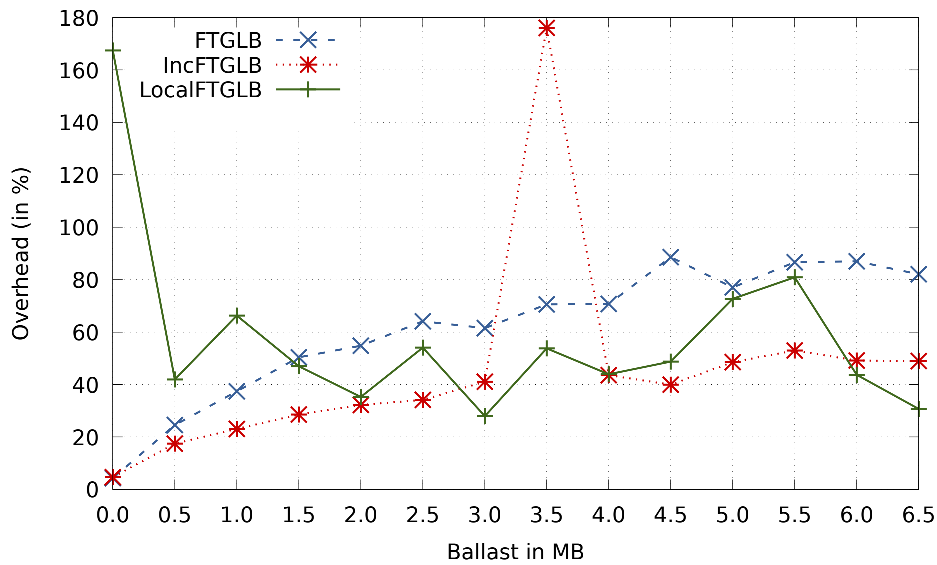

Figure 5.7: DynamicSyn with 144 places and backupCount=1

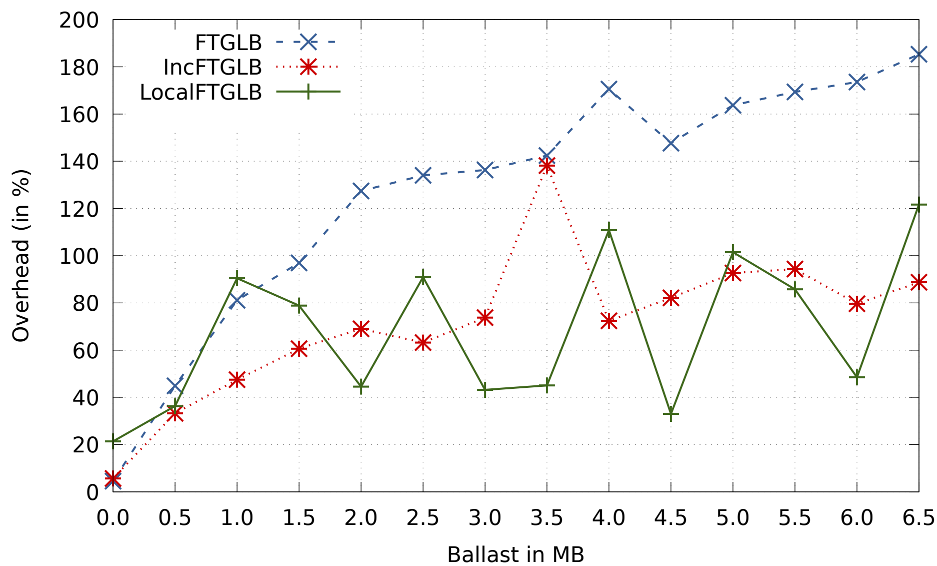

Figure 5.8: DynamicSyn with 144 places and backupCount=6

5.7 Discussion

LocalFTGLB reduced the overhead compared to FTGLB and IncFTGLB in three of four benchmarks. This confirms our observation, that the overheads incurred by saving already saved tasks, monitoring of the pool, and waiting for network communication, which were noted in Section 3.1, could be reduced by the design of LocalFTGLB.

For UTS, the curve of LocalFTGLB is smoother than the curve of FTGLB with 4 and 8 places. One possible explanation is that the asynchronism hides the longer time needed for that configuration to write the backups.

For DynamicSyn and a backupCount of 1, the result of LocalFTGLB with no ballast is more than five times higher than the overhead with the highest tested ballast. The cause of this behaviour is still under investigation.

During the experiments, it was found that the number of steal backups increased with a rising number of places. At the same time, the number of regular backups decreased. This behaviour was to be expected. Because steal backups in LocalFTGLB should have a lower overhead than steal backups in FTGLB, the difference in overhead should become more visible as the number of places increases. The expected effect was not confirmed by the results.

As a side effect of our experiments, the results from [4] for the synthetic benchmarks were confirmed with the latest Hazelcast and Java versions.

5.8 Correctness

The same correctness definition as in [15] is used (cited from [15]): "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".

The correctness of the implementation was tested by simulating crashes of single and multiple places, and inspecting the resulting log files. The simulation of a crash was realized by a call to `System.exit()` at specific points of execution. A list of these points can be found in the appendix.

6 Conclusions

This thesis presented a new scheme for uncoordinated application-level checkpointing for task pools and compared it to previous schemes. The new scheme reduces the overhead for fault-tolerance in failure-free execution. It tracks work-stealing to reduce the backup volume and writes the backups in parallel to the task processing. It also introduces a technique to keep the checkpoint's size low by variation of the backup time.

Results showed a decrease of overhead in most benchmarks. The highest overheads measured were about the same as in FTGLB. In benchmarks with large task descriptors, IncFTGLB still had the lowest overheads of all three implementations. The thesis also showed new benchmark results for APGAS_GLB, FTGLB and IncFTGLB with the latest versions of Java and Hazelcast, which confirmed previous results.

Future work should carry out further performance analysis of the individual components of the algorithm. Also, the recovery component of the algorithm should be improved.

Bibliography

[1] 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.

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

[3] Fault Tolerance Research Hub. *SC'17 tutorial*. `http://fault-tolerance.org/2017/11/11/sc17-tutorial/`. 2017.

[4] 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)*. To appear. 2018.

[5] L. C. Freeman. "A Set of Measures of Centrality Based on Betweenness." In: *Sociometry* **40.1** (1977), pages 35–41.

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

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

[8] 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.

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

[10] 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.

[11] D. Lea. "A Java Fork/Join Framework." In: *Proc. ACM Conference on Java Grande*. ACM, 2000, pages 36–43.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] J. Posner and C. Fohry. "Fault Tolerance for Cooperative Lifeline-Based Global Load Balancing in Java with APGAS and Hazelcast." In: *IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*. 2017, pages 854–863.

[18] 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.

[19] 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.

[20] SchedMD. *Slurm Workload Manager*. `https://slurm.schedmd.com`. 2018.

[21] 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.

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

[23] University of Kassel. *Scientific data processing*. `https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html`. 2018.

[24] 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.

Execution times

Places	GLB	FTGLB	LocalFTGLB
1	7019.79	7072.02	6917.95
2	3723.56	3754.59	3792.47
4	1899.14	2047.62	1922.09
8	1002.20	1066.71	1005.97
12	652.88	664.37	663.16
24	328.32	341.57	336.84
36	220.57	230.24	228.05
48	166.28	175.64	174.48
60	133.96	142.74	141.69
72	112.05	120.85	119.88
84	97.10	105.12	104.20
96	85.67	93.28	92.85
108	76.90	84.16	83.98
120	69.90	76.97	76.63
132	64.17	70.46	70.26
144	58.85	65.70	65.55

Table 1: UTS with backupCount=1: Execution time in seconds

Places	GLB	FTGLB	LocalFTGLB
1	7019.79	7125.68	6961.32
2	3723.56	3767.82	3778.10
4	1899.14	2073.77	1934.50
8	1002.20	1093.43	1023.18
12	652.88	677.80	668.92
24	328.32	350.04	342.38
36	220.57	238.37	233.43
48	166.28	181.97	178.82
60	133.96	148.19	145.39
72	112.05	126.11	123.34
84	97.10	110.08	107.60
96	85.67	98.10	96.00
108	76.90	88.93	86.91
120	69.90	81.01	79.65
132	64.17	75.37	73.73
144	58.85	69.40	68.17

Table 2: UTS with backupCount=6: Execution time in seconds

Places	GLB	FTGLB	LocalFTGLB
1	1475.49	1556.32	1519.20
2	741.50	810.79	767.06
4	488.91	522.34	501.94
8	382.55	410.23	395.97
12	333.24	373.06	351.48
24	168.03	185.74	178.63
36	112.90	128.65	123.53
48	85.60	99.17	96.22
60	69.02	81.54	78.53
72	57.98	69.64	67.13
84	50.25	61.64	58.88
96	44.49	53.79	53.92
108	40.11	49.71	48.57
120	36.67	46.36	44.94
132	33.52	42.37	41.18
144	31.91	38.35	38.20

Table 3: BC with backupCount=1: Execution time in seconds

Places	GLB	FTGLB	LocalFTGLB
1	1475.49	1530.30	1513.53
2	741.50	821.53	767.04
4	488.91	530.39	513.45
8	382.55	423.02	410.72
12	333.24	380.59	417.16
24	168.03	194.91	190.26
36	112.90	142.69	133.36
48	85.60	113.67	104.75
60	69.02	91.52	86.72
72	57.98	79.09	73.40
84	50.25	71.98	66.29
96	44.49	65.08	61.19
108	40.11	60.52	55.18
120	36.67	53.84	50.34
132	33.52	49.67	46.23
144	31.91	45.40	43.85

Table 4: BC with backupCount=6: Execution time in seconds

Ballast in MB	GLB	FTGLB	IncFTGLB	LocalFTGLB
0.0	60.57	60.61	59.47	60.71
0.05	59.83	63.30	59.50	64.05
0.1	59.52	65.14	59.64	68.12
0.15	60.34	68.66	60.17	68.49
0.2	59.50	71.85	60.46	71.30
0.25	59.58	74.18	61.39	73.75
0.3	60.08	78.10	61.59	76.14
0.35	59.26	81.51	62.31	77.22
0.4	60.10	82.42	62.84	79.80

Table 5: StaticSyn with backupCount=1: Execution time in seconds

Ballast in MB	GLB	FTGLB	IncFTGLB	LocalFTGLB
0.0	60.57	61.63	59.70	60.88
0.05	59.83	68.06	60.32	70.85
0.1	59.52	76.17	61.17	78.23
0.15	60.34	86.24	61.06	84.96
0.2	59.50	92.70	62.99	92.56
0.25	59.58	100.68	63.57	98.71
0.3	60.08	114.00	65.08	106.83
0.35	59.26	117.39	65.93	113.87
0.4	60.10	126.49	65.32	121.76

Table 6: StaticSyn with backupCount=6: Execution time in seconds

Ballast in MB	GLB	FTGLB	IncFTGLB	LocalFTGLB
0.0	97.30	101.43	101.81	260.22
0.5	113.68	141.54	133.50	161.35
1.0	129.47	177.88	159.28	215.32
1.5	148.91	223.95	191.39	218.81
2.0	164.43	254.53	217.40	222.38
2.5	200.94	329.77	269.57	309.61
3.0	219.07	353.84	309.02	280.31
3.5	235.48	401.66	324.99	362.15
4.0	252.07	430.31	361.91	362.81
4.5	290.50	547.86	406.50	432.03
5.0	305.88	541.54	454.32	528.22
5.5	325.14	606.92	497.26	588.15
6.0	340.92	637.52	508.48	489.96
6.5	358.13	652.06	533.44	468.02

Table 7: DynamicSyn with backupCount=1: Execution time in seconds

Ballast in MB	GLB	FTGLB	IncFTGLB	LocalFTGLB
0.0	97.30	101.70	102.79	118.09
0.5	113.68	164.72	151.44	154.96
1.0	129.47	234.47	190.93	246.58
1.5	148.91	293.30	239.12	266.35
2.0	164.43	373.97	277.94	237.66
2.5	200.94	470.28	327.97	383.62
3.0	219.07	517.69	380.69	313.62
3.5	235.48	570.86	400.58	341.61
4.0	252.07	682.03	434.68	531.30
4.5	290.50	719.40	529.14	386.46
5.0	305.88	806.68	589.23	616.30
5.5	325.14	875.71	631.86	604.03
6.0	340.92	932.54	612.46	506.33
6.5	358.13	1021.73	676.01	793.86

Table 8: DynamicSyn with backupCount=6: Execution time in seconds

Correctness test cases

1. Crash of place 2 in the `processStack()` method after n computation steps.
2. Crash of place 2 in the `processStack()` method and crash of place 1 at the beginning of the `recoverPlace()` method.
3. Crash of place 2 in the `processStack()` method and crash of place 1 at the end of the `recoverPlace()` method.
4. Crash of place 2 in the `processStack()` method and delayed execution of the `placeFailureHandler()` method by 120 seconds.
5. Crash of place 2 at the end of the `processLoot()` method.
6. Crash of place 2 after the backup written when the pool becomes empty.
7. Crash of all places except place 0.
8. Crash of place 2 between the writing of the steal backup and the sending of the tasks.

9. Crash of place 2 before the steal backup with place 2 as thief is written. The steal backup is delayed by 120 seconds. This tests the prevention of late steal backups to a checkpoint which is already recovered.

10. Crash of place 2 after the regular backup is written. The program waits until the regular backup is written and then simulates the crash.

11. Crash of place 2 after an answered lifeline steal request. The thief waits for 3 seconds until processing the answer to make it highly likely for the `placeFailureHandler()` to be already executed.

12. Crash of place 2 after an answered random steal request. The thief waits for 3 seconds until processing the answer to make it highly likely for the `placeFailureHandler()` to be already executed.