Delft University of Technology

Faculty of Electrical Engineering, Mathematics, and Computer Science
Computer Engineering

A Resiliency-First Approach to Distributed DAG Computations

Thesis by:

Theodorus Cornelis Leliveld

Advisor:

prof. dr. H.P. Hofstee

Committee:

Chair:
prof. dr. H.P. Hofstee

Members:
prof. dr. D.H.J. Epema
dr. ir. Z. Al-Ars

A Resiliency-First Approach to Distributed DAG Computations

Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

submitted on

July 3, 2017

by

Theodorus Cornelis Leliveld
born in Nieuwkoop, the Netherlands

CE-MS-2017-06
Computer Engineering
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4,
2628 CD, Delft
The Netherlands

This thesis is dedicated to my parents, Nico and Gemma, my sister Laure, and my girlfriend Marieke, as without your support I would not be where I am now. Thank you for your endless patience and unconditional love.

Abstract

A framework is introduced for computations with transformations on immutable data. Inspiration is taken from Apache Spark, however the model of computation is generalized from an emphasis on narrow and wide dependencies, to an arbitrary set of transformations that form a directed acyclic graph (DAG). A distributed scheduling algorithm is developed with resiliency mechanisms that can account for stopping failure. Furthermore some properties of the system are derived. Finally future work is discussed showing there is fertile ground for further research and development to extend this work.

Preface

Before you lies the thesis "A Resiliency-First Approach to Distributed DAG Computations". I have written this thesis as partial fulfillment for the requirements of my degree in Computer Engineering from Delft University of Technology. The main part of this work was carried out at the IBM Austin Research Lab, during my stay there from November 17 2016 to April 26 2017.

The topic of this thesis came forth from a mutual interest for distributed computing between my advisor, Peter Hofstee, and I. We were both interested in technologies like Spark but wondered whether there exists a more unifying foundation. During this research project we attempted to find such a foundation, which includes resiliency from the start. This effort accumulated to my thesis, and a paper that is currently under submission.

First I want to thank Peter for being my advisor. It has been a lot of fun working with you, and I have learned a great deal. Your guidance throughout this project has been of tremendous value. It has been a great pleasure working with you.

Next I want to thank everyone from ARL for the warm welcome to Austin. I especially want to thank Jan Rellermeyer, Ahmed Gheith, Andy Martin, and Vinod Muthusamy for their technical advice. From TU Delft I want to thank Zaid Al-Ars for bringing me into contact with Peter and IBM, and for giving me this opportunity.

Contents

List of Figures

Listings

Chapter 1

Introduction

Due to the nature of distributed systems it is difficult to separate the specification of the computation from the details of the execution. Distributed systems rely on a partitioning of the application to reach a sufficient degree of parallelization. This partitioning induces costs such as serialization overhead, network latency bandwidth, and synchronization overhead. When resiliency is a requirement the distributed system needs information about the state in the computation to recover.

This entanglement between computation and execution partly explains the plethora of different distributed computing frameworks. Examples include MPI, Apache Hadoop/MapReduce, Apache Spark, Apache Storm, etc. These frameworks all target different domains. Examples of these include scientific, big data, iterative big data, and real-time computation.

1.1 Objective

In this thesis we set out to find a more rigid foundation for directed acyclic graph (DAG) computations on distributed systems. The key objective is not to engineer a system that has a specific percentage speed increase over some existing framework, but rather a theoretical foundation on which systems can be built. As such we try and specify requirements for certain parts of the system rather than a specific implementation.

For example, a function is required that arranges all the processes in a DAG that enables work stealing in the system. This can be tuned for a specific network topology. A similar function is used to assign tasks to processes. This could be tuned to a specific computation or problem domain. As such the main contribution of this thesis is not an engineered system, but a set of thoughts of how such a system can be organized. In writing this thesis a prototype is developed alongside to ensure the ideas still make sense in a practical implementation.

In developing these ideas, existing frameworks are evaluated. Arguably for DAG computations the most popular framework is Apache Spark.

1.2 Apache Spark

Spark introduces the notion of Resilient Distributed Datasets [2] (RDD). RDDs are immutable data collections that consist of partitions the user can operate on. In general two classes of operations exist. Operations that have narrow dependencies, for which each partition in the result RDD has a dependency on a single partition in the input RDD (that no other partition in the result RDD depends on, one-to-one), and wide dependencies, for which each partition in the output RDD depends on all the partitions in the input RDD.

In Spark terminology, the collection of transformations on immutable data structures is called the operator graph. This operator graph forms a DAG. A powerful property of immutable data structures is that the transformations are referentially transparent, meaning the transformations always produce the same results. Therefore RDDs can be recomputed consistently. This is the fundamental property Spark uses for its lineage resiliency mechanism. If some part of the computation is lost due to failure, the system can recompute the missing parts.

The DAG of transformations is scheduled by a centralized job scheduler in Spark, by cutting it up into transformations on partitions. The job scheduler hands out these pieces of work to executor nodes. These executor nodes compute the results of these transformations on the partitions. In case of failure on an executor, the job scheduler restarts the work on another executor. The job scheduler is redundant to avoid a single point of failure.

This all results in a scalable, resilient and easy to program system, but there are some limitations. The narrow/wide dependency model does not fit all the possible problems that can be expressed with immutable data structures. The centralized scheduler makes the system act in a centralized way, both in terms of coordination and resiliency.

1.3 Contributions

In this thesis we attempt to find a more rich foundation that improves on the aforementioned limitations of Spark. To achieve this, a number of contributions are made in this thesis:

In Chapter 2 a richer model of computation is developed that generalizes the narrow/wide dependency model to arbitrary DAGs. Furthermore a system for generating unique identifiers for tasks in the DAG is identified that is heavily used in the scheduling algorithm. Finally access mechanisms to the DAG on a distributed system are developed.

A distributed algorithm for executing these DAGs on a distributed system is introduced in Chapter 3. This algorithm's scheduling consists of two parts: A static assignment, and a work stealing algorithm. A token based system to manage resources and avoid deadlock is introduced as well. Finally some properties of the system are derived.

An extension to the distributed execution algorithm is shown in Chapter 4. This extension provides resiliency mechanisms that make the system tolerant to stopping failures. Similarly a number of properties were derived for this extension.

In Chapter 5 the prototype implementation is shown to be resilient to faults by means of a fault injection test. In Chapter 6 some practical limitations of this work are discussed, as well as a discussion of the referenced works. Chapter 7 draws some conclusions and in Chapter 8 some future work is presented to show there is fertile ground for further research.

1.4 Implementation

Alongside the theory in these chapters a prototype will be developed in Haskell. The reason for Haskell is twofold. Pure functional languages, such as Haskell, have plenty of machinery to implement embedded domain specific languages. This allows us to write a language for specifying computation, while providing the programmer with more guarantees than conventional languages through the type system. Among this machinery are (generalized) algebraïc data types, monadic abstractions, existential types, phantom types, etc. The other advantage is that a pure language forces a strict methodology to state, as all values are immutable. This makes it easier to reason about programs, which is especially useful in a distributed systems setting.

The distributed system code is built using Cloud Haskell [3], which provides an Erlang/OTP [4] actor model platform. The actor model is a model of concurrent computation, where actors form the primitive of execution rather than threads or processes. Actors can handle messages where during the handling of each message, actors can spawn more actors dynamically, send messages to other actors, and keep local state which can influence the handling of the next message the actor receives.

Writing the prototype of this application in a library that is based on the actor model proved to be of tremendous value throughout the process. It makes it significantly easier to reason about the different processes (or actors) in the system because their behaviour is isolated. Actors do not have any shared state, which allows us to solely reason on the messages an actor may or may not receive. This made it much easier to design for resiliency.

Chapter 2

Model of Computation

Transformations on immutable data structures form a computation that can be expressed as a DAG. From this point forward these computations are referred to as DAG computations. While Spark exposes RDDs as the basic data type to operate on, the notion of tasks is introduced in this thesis. Tasks resemble the smallest piece of work that can be partitioned. Tasks can be combined and transformed arbitrarily as long as they represent an immutable result. This model is an extension of the narrow/wide dependency paradigm that expresses a wider range of computations.

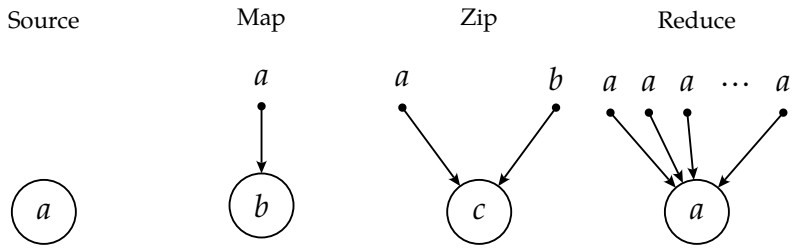
An example where Sparks narrow/wide dependency model is not expressive enough are divide and conquer algorithms. Consider Karatsuba's algorithm for multiplying arbitrary size numbers. In Spark we cannot express the dependencies for this algorithm effectively. The algorithm splits up the multiplication of two big numbers, in three multiplications of numbers half the size in digits. This can repeated recursively. The results of these smaller multiplication can then be combined to provide a result to the original multiplication. The DAG of this computation has neither one-to-one dependency, nor all-to-one dependency, so it does not fit in Spark's narrow/wide dependency model. In the next chapter an extension to this model is introduced, that can in fact express this problem effectively. An example specification of Karatsuba's algorithm in the model introduced in this thesis can be found in Section 2.4.

2.1 Directed Acyclic Graph Computations

Tasks are represented as vertices in the DAG. Tasks have a result type, which is the type of the result of the subcomputation. Two scenarios can be identified: tasks that have no dependencies on other tasks, and tasks that rely on the results of other tasks. Each task must uphold referential transparency. This means a task with dependencies can freely be replaced by its result value, without altering the final result of the computation.

In the prototype, four different kinds of tasks are defined. While these tasks provide a rich language for expressing problems, there is no particular reason one could not add new kinds of tasks to this set as long as they adhere to referential transparency. The different kind of tasks introduced in the prototype are: source for inputting data, map for one-to-one mappings, zip for the com-

bination of two different tasks, and reduce for an n-to-one operation. Example graphs are included for these kind of tasks in Figure 2.1



Source : $a \to \text{Task}\, a$
Map : $(a \to b) \to \text{Task}\, a \to \text{Task}\, b$
Zip : $(a \to b \to c) \to \text{Task}\, a \to \text{Task}\, b \to \text{Task}\, c$
Reduce : $(a \to a \to a) \to a \to [\text{Task}\, a] \to \text{Task}\, a$

Figure 2.1: *Different kind of tasks types with their vertex analogue. The letters in the vertices denote the type of the result of the task. The type signatures for the task constructors are also denoted. Everything between arrows forms an argument to the task constructor except for the last type, which is the result type of the constructor.*

The rationale for these four kinds is as follows. There needs to be a way to produce data into the system. The source task allows this. Transforming individual tasks can be done via the map task. Combining two tasks (of potentially different types) can be done by making use of the zip task. Reduce specifies a n-to-one reduction over a single task result type.

The task kinds, source, map, and zip, allow us to express all DAG computations. For example ternary functions can be expressed by tupling up task results. Consider a function that takes arguments of the types a, b and c and produces a result of type d, i.e. $a \to b \to c \to d$. Tupling up two tasks, one with result type a, and another with result type b results in a task with result type (a, b). Uncurrying the ternary function results in a function from $(a, b) \to c \to d$. This can be trivially expressed by using zip using the tupled up result and a task with result type c. This can be repeated for functions that take four or more arguments by recursively applying this scheme. This scheme might imply performance costs, as more tasks are created that might require the communication of the results depending on the scheduler and execution implementation. Therefore it might be advisable to implement a zip$N$ up to some suitable N in a practical setting. In the prototype reduce was included since this kind of operations is common. However, it does not add more expressiveness to the system.

Implementing these constructors in Haskell can be trivially achieved with algebraïc data types. The constructors can be found in Listing 2.1. The generalized algebraïc data types (GADT) extension was used here for notational purposes, while it strictly is not necessary to express these kinds of types. These

types do however require an existential type extension. The implementation of a (local) evaluation function for these data constructors can be seen in Listing 2.2.

Note that building an abstraction layer like RDDs with just narrow and wide dependencies on top of tasks can simply be achieved by equating RDDs to a list of tasks one can operate on. In such a scheme tasks are equivalent to partitions.

```
data Task a where
  Source :: a -> Task a
  Map    :: (a -> b) -> Task a -> Task b
  Zip    :: (a -> b -> c) -> Task a -> Task b -> Task c
  Reduce :: (a -> a -> a) -> a -> [Task a] -> Task a
```

Listing 2.1: *GADT declaration for DAG computations.*

```
evalTask :: Task a -> a
evalTask (Source a) = a
evalTask (Map f a) = f (evalTask a)
evalTask (Zip f a b) = f (evalTask a) (evalTask b)
evalTask (Reduce f z as) = foldr f z (map evalTask as)
```

Listing 2.2: *Local evaluator function for the data constructors in Listing 2.1.*

Given this definition, a number of problems arise when considering an implementation of a system that can execute this on a distributed system. These problems are:

- How are processes able to access tasks from the DAG.
- How can processes communicate tasks and results.
- How do we compare tasks for equality.

2.2 Distributed Access and Serialization

We tackle the first two problems first. If one could allow each process to access the entire DAG this would solve the problem. Serialization is the process of converting data structures in a suitable binary format such that they can be sent over a network and be decoded without loss of information. If the entire DAG would be serialized, it could be communicated to each process at the start of the communication. This allows each process to access the DAG.

This gives rise to yet another difficulty: it implies serializing a task as the DAG is built up of tasks. Tasks contain functions, and serializing functions is a widely debated topic in the Haskell community for which there is not a clear-cut solution.

Instead we opt to avoid this serialization problem entirely by making use of the recently added static pointer extension. This allows us to create a "static" reference to an object, which provides a reference to a closed expression (in this case the DAG) that is valid across universes. This means the reference is stable and portable in the sense that it remains valid throughout different processes on possibly different machines. Most of all this reference is trivial to

serialize, and thus distribute between processes. The requirement for the use of this extension is that the expression referred to is closed, i.e. there can be no free variables occurring in said expression. This means the DAG computation must be specified completely and cannot contain any free variables at compile time.[1]

This solves the distribution of tasks, but to be able to effectively parallelize the computation over multiple machines, processes need to be able to communicate the results of tasks. This means the result of tasks must be serializable. This can be solved by adding a constraint to all the data constructors that the result type of a task must be serializable. This can easily be achieved by using existential quantification.

2.3 Unique Identifiers

To solve the issue of equality between tasks, the notion of unique identifiers (UID) is introduced. Each task is assigned a UID at construction to be able to identify tasks across processes. The definition of a UID in the prototype is given in Listing 2.3.

```
data UID = UID Integer
  deriving (Eq, Show, Ord, Num, Enum)

data UTask a where
  UTask :: (Serializable a) => UID -> Task a -> UTask a
```

Listing 2.3: *Proposal for unique identifiers and UTask, a Task combined with an id.*

Implementing a scheme that automatically generates these unique identifiers is not as trivial as one may expect. In an object-oriented language, using a singleton design pattern that keeps a counter internally would provide an adequate solution. This does not work for Haskell, which does not allow any form of global state, since it is a pure language without mutable values.

There is a workaround that is similar to the approach taken in the `Data.Unique` library. It internalizes an IO reference that is updated by making use of `inlinePerformIO`. This approach has the downside that it is not consistent across universes, which would break on a distributed system where by definition each process has its own universe.

Another way of looking at this problem, is building an enumeration of the tasks. This is an inherently sequential and stateful process. This is a natural fit for the State monad, the definition of which is in Listing 2.4. In this definition the `s` refers to the state of the computation, in this case the UID space, and the `a` the result of the stateful computation, in our case the task wrapper `UTask a`. A type alias is defined to simplify the definition. A function that inserts a UID space (in this case `[UID]`) inserts the set of UIDs into the defined tasks. The advantage of this approach is that the definition of the computation, and the generation and insertion of the UIDs are completely independent of each other. In other words modifying the UID scheme, or the computation will not affect the other.

[1] This does not mean that there can not be parameters in the computation. They have to be declared as a source task, explicitly declaring them as a "variable" in the DAG.

In the prototype there is function for each kind of task that inserts a UID for the given kind of task. This can be seen in Listing 2.5.

```
newtype State s a = {
  runState :: s -> (a, s)
}

type UIDGen a = State [UID] a

genUIDTasks :: UIDGen (UTask a) -> UTask a
genUIDTasks x = evalState x ([1..] :: [UID])
```

Listing 2.4: *The State monad as introduced in [1] and the type alias for the generation of unique tasks.*

```
popUID :: UIDGen (UID)
popUID = do
  (uid:xs) <- get
  put xs
  return uid

source :: (Serializable a) => a -> UIDGen (UTask a)
source a = do
  uid <- popUID
  return $ UTask SerializableDict uid (Source a)

map :: (Serializable b) => (a -> b) -> UTask a -> UIDGen (UTask b)
map f a = do
  uid <- popUID
  return $ UTask SerializableDict uid (Map f a)

zip :: (Serializable c) => (a -> b -> c) -> UTask a -> UTask b -> UIDGen (UTask c)
zip f a b = do
  uid <- popUID
  return $ UTask SerializableDict uid (Zip f a b)

reduce :: (Serializable a) => (a -> a -> a) -> a -> [UTask a] -> UIDGen (UTask a)
reduce f z as = do
  uid <- popUID
  return $ UTask SerializableDict uid (Reduce f z as)
```

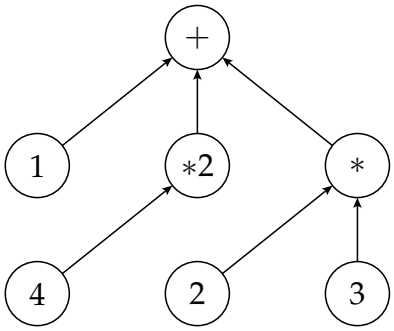
Listing 2.5: *Functions for building up a stateful computation around UTasks and unique identifiers.*

The UID scheme described in this section and access to the full DAG as described in Section 2.2 allows each process to construct a mapping between UIDs and tasks. This mapping should be implemented in such a way that each process in the system starts off with the mapping, or can compute the mapping deterministically without requiring global coordination or communication.

2.4 Example

Figure 2.2 contains an example of how to specify a computation in this model in code, together with the corresponding DAG.

Another example is the multiplication of arbitrary size integers following Karatsuba's algorithm. The code can be seen in Listing 2.6. The two numbers are recursively split up until small enough and than put into source tasks. These tasks are then combined using a zip function. The combine function has three dependencies, so we can express this functions using two zips, one for the actual combination function, the other for tupling up the subresults (`z01`, tupling up `z0` and `z1`).




```
comp = do
  a <- source 4
  b <- source 2
  c <- source 3
  d <- source 1
  e <- map (*2) a
  f <- zip (*) b c
  reduce (+) 0 [d, e, f]

res = eval (genUIDTasks comp) -- results in 15
```

Figure 2.2: *Example of the specification of a computation in code and the corresponding DAG.*

```
karatsuba :: Integer -> Integer -> UIDGen (UTask Integer)
karatsuba x y
  | x < 10 = source (x * y)
  | y < 10 = source (x * y)
  | otherwise = do
    let m = max (numberOfDigits x) (numberOfDigits y)
        m' = m `div` 2
        (highX, lowX) = splitDigits x m'
        (highY, lowY) = splitDigits y m'
        combine (z0', z1') z2' = z2' * 10 ^ (2 * m') +
                                 (z1' - z2' - z0') * 10 ^ m' + z0'

    z0 <- karatsuba lowX lowY
    z1 <- karatsuba (lowX + highX) (lowY + highY)
    z2 <- karatsuba highX highY
    z01 <- zip (,) z0 z1
    zip combine z01 z2
```

Listing 2.6: *Karatsuba's algorithm expressed in tasks as introduced in Chapter 2.*

Chapter 3

Distributed Execution

In Chapter 2 an extensive introduction of the model of computation was given. This chapter focuses on the distributed execution of these DAGs. Two types of processes — task pools and workers — are responsible for making this happen. These two processes represent two levels of concurrency: Task pools on the distributed system level, and workers on the thread level. Workers only communicate with their assigned task pool. Task pools keep a task queue, spawn workers to evaluate said tasks, communicate with their own workers, and communicate with the other task pools in the system. Workers are analogous to executors in Spark. The task pools together take over the responsibilities of the centralized scheduler in a distributed way.

The computation is started by some outside process. This process is called the initiator. The initiator starts a number of task pools, providing them with acces to the DAG corresponding with the computation. Furthermore it sends requests for the tasks, of which the results are desired.

The following definition of a distributed system is used. The system consists of processes $P_1, P_2, \ldots, P_n$. The processes share no mutual clock, nor state and can only communicate by means of messages.

3.1 Task Pool — Worker Interaction

The task pool process starts workers based on the requests it receives. To start the computation, an initial (set of) request(s) must be submitted by the initiator. After the initial request is received, the task pool start arranging the tasks and executing it using their workers.

Each task pool has a number of worker tokens, from now on referred to as tokens, modeling the resources available to execute tasks. When a task pool receives a request for which it has not computed a result, it will either spawn a worker if it has a token available, or queue it in its task queue.

It is not transparent to a task pool if a specific task requires other tasks to be computed first to satisfy its dependencies. Consider the case where there are two task pools, each having spent one token on a worker. The first task pool has a worker that requires a task's result from the second task pool. The second task pool in turn has a worker that requires a task's result from the first task pool. In this scenario both task pools can not give out a token to start

an additional worker to satisfy each others request and thus a deadlock will occur.

This deadlock is solved by the following scheme. Once a worker gets spawned two scenarios can occur. One scenario is the case of a task without dependencies (source). In this case the worker can immediately calculate the result of the task. Once the worker has completed the calculation, it returns its token to the task pool and serves the result of the task to any request it receives.

In the other scenario, the worker has dependencies that need to be resolved. The worker requests the dependencies from its task pool, which in turn routes these requests to their assigned task pools. When the worker asks for dependencies, it implicitly returns its token to the task pool (only once, even in case of multiple dependencies). Once the worker has received all the dependencies, it asks the task pool for a worker token. The task pool either gives the worker a token, if it has one available, or puts it in a separate token request queue. This process is illustrated in Figure 3.1.

Once more tokens become available the task pool starts popping token requests from its token request queue and allowing workers to finish their tasks. If the token request queue is empty it starts popping tasks from its task queue to spawn new workers.

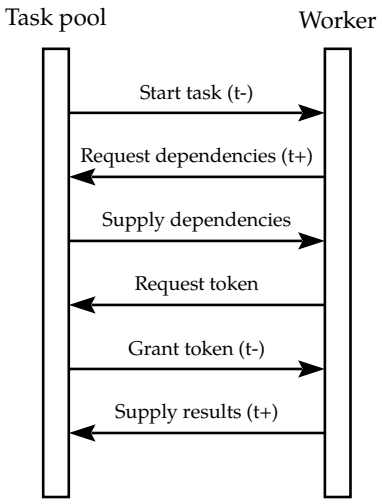


Figure 3.1: *Sequence diagram of worker task pool interaction, where the spending and gaining of tokens by the task pool is illustrated by t-/t+.*

3.2 Worker Process Implementation

In this section the implementation of a worker process will be presented. To this end two basic message types, request and supply messages, are introduced. Requests indicate that the sender is interested in the result of a task. They contain a field for the UID of the requested task, and the process identifier for the process it was sent from. A supply is an answer to a request, containing the result of the task in encoded form (bytestring), the UID of the associated task and the process identifier from which it was sent. The definition of these can be found in Listing 3.1.

Two helper functions for these message types are `sendRequest` and `waitForRequest`. The first function sends a request to some process, by means of its process identifier. The second function blocks until it receives a supply message providing the result for the requested task.

```
data Request = Request ProcessId UID

data Supply = Supply ByteString UID ProcessId
```

Listing 3.1: *Request and Supply message definitions*

As discussed in Section 3.1, a worker's single goal is to evaluate a task. To avoid deadlocks a token system was introduced. To support this system, three additional message types are defined. First a wrapper around a Request is introduced so that task pools can differentiate between requests from their workers and other task pools. Secondly a finish message that tells a task pool that the worker has finished its computation, implicitly giving the worker's token back to the task pool. Last a token request message, that a worker sends to a task pool when it has received all dependencies and needs a token to start the computation before finishing. It contains the process identifier of the worker, and the UID of the task it is computing. The definitions for the message types can be seen in Listing 3.2 and the code for a worker in Listing 3.3.

```
type WorkerId = ProcessId
type TaskpoolId = ProcessId

data Finish = Finish UID

data WorkerRequest = WorkerRequest Request

data TokenRequest = TokenRequest UID WorkerId
```

Listing 3.2: *Finish and WorkerRequest message definitions.*

```
workerProcess :: TaskPoolId -> UTask a -> Process ()
workerProcess parent (UTask uid t) = do
  myPid <- getSelfPid
  case t of
    (Source a) -> do
      signalFinish parent (Finish uid)
      supply dic a uid
    (Map f a) -> do
      sendRequest parent uid a
```

```
      a' <- waitForRequests a
      requestToken parent (TokenRequest uid myPid)
      let !res = f a'
      signalFinish parent (Finish uid)
      supply dic res uid
    (Zip f a b) -> do
      sendRequest parent uid a
      sendRequest parent uid b
      (a', b') <- waitForRequests a b
      requestToken parent (TokenRequest uid myPid)
      let !res = f a' b'
      signalFinish parent (Finish uid)
      supply dic res uid
    (Reduce f z a) -> do
      traverse_ (sendRequest parent uid) a
      a' <- waitForRequests a
      requestToken parent (TokenRequest uid myPid)
      let !res = foldr f z a'
      signalFinish parent (Finish uid)
      supply dic res uid
```

Listing 3.3: *Worker process implementation.*

3.3 Single Task Pool Implementation

We will now expand on the implementation of the behaviour between task pools and workers. For this section only a single task pool is considered, which for now is responsible for executing all the requested work. In Section 3.5 we will discuss the implementation of a multiple task pool system.

Fundamentally task pools are reactive processes, that after each message update their local state (their task queue, task assignment, etc.). Throughout this thesis new functionality will be added to the task pools (and thus the system). This is achieved by introducing new message types and by adding state to the task pools. For now four message types are considered: requests, worker requests, token requests and finish messages. The state of the task pool consists of a mapping between UIDs and their state, as well as the number of tokens that the task pool has.

A task pool starts with a predefined amount of tokens. The task pool constructs a mapping between the UID of the tasks in the DAG, and the state of the task's computation. This task computation state is either that no worker has been assigned yet, or there is a worker assigned including its worker identifier and token status. This mapping is initialized for all UIDs, with no worker assigned. This can be seen in Listing 3.4.

```
data TaskState = Map UID TaskCompState

data TokenStatus =
    Token
  | NoToken

data TaskCompState =
    NoWorker
```

```
  | Worker WorkerId TokenStatus
```

Listing 3.4: *A task pools mapping of the UIDs and the state of their workers.*

On receipt of a request, the task pool checks the current computation state of the task. Two cases can occur:

- There is no worker assigned to this task yet. The task is queued if there is no occurrence of the this task in the task queue. A function is called that checks if there is a token available such that it can be immediately dequeued.
- There is a worker busy computing this task or the worker has finished the computation. In this case the request is forwarded to the worker so that a supply message can be generated from there.

A description of this process can be seen in Listing 3.5.

On receipt of a worker requests, the task pool checks whether the worker still has a token. If that is the case it increases its token count by one, and sets the token status of that worker to no token. In either case the request is then handled the like a normal request. If a token was gained, the task pool checks whether it has any waiting workers in the token request queue, or tasks waiting in the task queue.

On receipt of a finish, a token is also gained and similarly the task pool checks if it can be spend appropriately. On receipt of a token request, the task pool checks whether it can give the worker a token if it has one available, otherwise queuing it in its token request queue. The code for worker requests, token requests, and finish messages can be seen in Listing 3.6.

```
processRequest :: TPState -> Request -> Process ()
processRequest state@TPState{..} req@(Request _ uid) = do
  case lookup uid taskState of
    Just NoWorker ->
      resolveToken state{taskQueue=newTaskQueue} >>= processNextMessage
    Just (Worker wid _) ->
      forwardRequest state wid req >>= processNextMessage

resolveToken :: TPState -> Process TPState
resolveToken state@TPState{..}
  | execToken == 0 = return state
  | execToken < 0 = error "Leaking␣tokens"
  | otherwise = do
    myPid <- getSelfPid
    case (tokenRequestQueue, taskQueue) of
      ((x:xs), _) -> sendToken state{tokenRequestQueue=xs} x
      (_, (x:xs)) -> startWorker state{taskQueue=xs} x
      _           -> return state

startWorker :: TPState -> Request -> Process TPState
startWorker state@TPState{..} req@(Request _ uid) = do
  let Just task = lookup uid taskTree
  myPid <- getSelfPid
  workerPid <- spawnLocal (workerProcess myPid task)
  let newTaskState = adjust (const (Worker workerPid Token)) uid taskState
```

```
      newToken = execToken - 1
  s' <- forwardRequest state workerPid req
  return s'{taskState=newTaskState, execToken=newToken}

sendToken :: TPState -> TokenRequest -> Process TPState
sendToken state@TPState{..} t@(TokenRequest uid pid) = do
  let newTokens = execToken - 1
      newTaskState = adjust (\ (Worker wid _) -> Worker wid Token) uid taskState
      newState = state{execToken=newTokens, taskState=newTaskState}
   in send pid TokenGrant >> return newState
```

Listing 3.5: *Function that handles the receipt of a new request for a task pool, including helper functions for handing out tokens, starting a new worker, and granting a token to a worker.*

```
processWorkerRequest :: TPState -> WorkerRequest -> Process ()
processWorkerRequest state@TPState{..} (WorkerRequest req uid) = do
  let newTokens = case lookup uid taskState of
                    Just (Worker _ NoToken) -> execToken
                    Just (Worker _ Token)   -> execToken + 1
                    _                       -> error "Unexpected state"
      Just (Assignment assPid _) = lookup uid taskAssignment
      newTaskState = adjust (\ (Worker wid _)
                             -> Worker wid NoToken) uid taskState
      newState = state{execToken=newTokens, taskState=newTaskState}
   s <- forwardRequest newState assPid req
   s' <- resolveToken
   processNextMessage s'

processFinish :: TPState -> Finish -> Process ()
processFinish state@TPState{..} (Finish uid) = do
  let newTaskState = adjust (\ (Worker wid _)
                              -> Worker wid NoToken) uid taskState
      newTokens = execToken + 1
      newState = state{execToken=newTokens, taskState=newTaskState}
   in resolveToken newState >>= processNextMessage
```

Listing 3.6: *Functions for handling worker request and worker finish messages.*

3.4 Interaction between Task Pools

Each task pool starts with access to the DAG. Each task pool has a function that assigns UIDs to some task pool. This mapping between UIDs and task pools indiciate which task pool is responsible for the execution of the corresponding task. This function results in the same assignment on each of the task pools. This is the static scheduling, or partitioning of the DAG. To refine potential imbalances in this assignment task stealing is introduced.

The notion of an arrangement of the task pools in a new directed acyclic graph is introduced. This directed acyclic graph is completely different from the computation DAG and is referred to as the task pool graph. Tasks are only allowed to be pushed over the edges (regardless of direction) of the task pool

graph. Each task pool in this graph communicates with its neighbors (again regardless of direction of the edges in the task pool DAG) about the amount of tasks in their task queues. If there is a difference greater then one between in the size of task queues between two task pools, a task gets pushed.

This system reduces the complexity of the number of messages significantly compared to naïvely allowing each task pool to steal from every task pool. In prior work [5] it is proven that when workers are not consuming or producing more tasks this system reaches a balanced distribution of tasks.

When a task gets stolen from a task pool, the two participating task pools update their task assignment to reflect the new situation. This does not have to be known by other task pools as the stolen from task pool will simply forward any request for the stolen task to the task pool it was pushed to. This can potentially produce a high amount of hops for requests if multiple steals happen after each other. In a practical implementation this can be alleviated by only allowing a task to be stolen a certain amount of times.

The static assignment and task stealing allow the tasks to be spread over the system. The static assignment should provide a rough partitioning of the computation (and in fact may be optimized for different kinds of computations), while the work stealing allows a dynamic refinement on top of this.

3.5 Multiple Task Pool Implementation

To determine which task pool handles which tasks (and thus requests) an assignment must be made as described in the previous section. In the prototype each task pool receives a list of the other task pools from the initiator and has a function that recursively divides the DAG over the different task pools. Each task pool has access to the DAG by means of the final tasks, i.e. the tasks no other task depends on. Recursively assigning these tasks and their dependencies guarantees us an assignment for all the tasks in the system.

In the prototype the following assignment function was implemented. The requested tasks are assigned cyclically to the list of task pools. For each of the requested tasks their dependencies the following scheme is applied recursively.

- In case of no dependencies nothing has to be done and the recursion can be bottomed out.
- If there is a single dependency, it is assigned to the same task pool as the parent is, to provide data locality, and this scheme is recursively applied to this task.
- If there are multiple dependencies, each additional dependency is assigned to the next task pool from the task pool list (cyclically) and the scheme is applied recursively to each dependency.

In this scheme some UIDs might be assigned to multiple task pools because more than one task relies on them. We resolve this by favouring the left side of the union operator that merges the assignment. The code can be seen in Listing 3.7.

Because the assignment function is pure and referentially transparent, and each task pool has the same input arguments, we are guaranteed that each task pool ends up with exactly the same assignment.

```
type TaskAssignment = Map UID Assignment

data Assignment = Assignment
  { assTpid :: TaskPoolId
  , assType :: AssignmentType
  } deriving (Eq, Show)

data AssignmentType =
    Static
  | Pushed
  deriving (Eq, Show)

rotate :: Int -> [a] -> [a]
rotate _ [] = []
rotate n xs = (let p = length xs in take p . drop n . cycle) xs

assignTasks :: [UTask a] -> [TaskPoolId] -> TaskAssignment
assignTasks t pids = (foldr union empty
                     . zipWith assignSubTree t
                     . map (`rotate` pids)) [1..]
  where
    assignSubTree :: UTask a -> [ProcessId] -> TaskAssignment
    assignSubTree (UTask _ uid t) pids =
      let assignHead = Assignment (head pids) Static
       in case t of
            (Source _)      -> singleton uid assignHead
            (Map _ a)       -> insert uid assignHead (assignSubTree a pids)
            (Zip _ a b)     -> insert uid assignHead (assignSubTree a pids
                `union` assignSubTree b (rotate 1 pids))
            (Reduce _ _ as) -> insert uid assignHead (assignTasks as pids)
```

Listing 3.7: *Assignment function that builds mapping between tasks and task pools.*

Each task pool can now determine which tasks, and thus UIDs are assigned to it. For these UIDs a mapping is made to the task's computation state. This is the same task computation state as described in Section 3.3, but now this mapping is only initialized for tasks that are assigned to this task pool. This means a third options can now occur when a task pool receive a request: the task is not assigned to this task pool. In that case we forward the request to the task pool it was assigned to. The updated version of request message handling function can be seen in Listing 3.8.

```
processRequest :: TPState -> Request -> Process ()
processRequest state@TPState{..} req@(Request _ uid) = do
  case lookup uid taskState of
    Just NoWorker ->
      resolveToken state{taskQueue=newTaskQueue} >>= processNextMessage
    Just (Worker wid _) ->
      forwardRequest state wid req >>= processNextMessage
```

```
    _ ->
      let Just tpid = lookup uid taskAssignment
       in forwardRequest state tpid req >>= processNextMessage
```

Listing 3.8: *Function that handles the receipt of a new request for a task pool, updated to reflect the case where the task is not assigned to this task pool.*

This assignment allows us to execute the DAG in parallel with multiple task pools. The downside of this static assignment is that there is no notion of dynamic load balancing. Therefore the work stealing from Section 3.4 is implemented. In the prototype the initiator is responsible for generating the task pool graph. The initiator sends each task pool a list of its neighbours in this graph. The construction of the task pool graph could also be done by the task pools themselves, as long as its the same task pool graph for each task pool.

The task pools keep a list of the sizes of the task queues for all their neighbours in the task pool graph. Whenever a task pool's task queue size changes, the neighbours are updated by sending them a message. Whenever a neighbour is asked to execute a task, but does not have a token for it, or a change occurs in its neighbours task queue size, it checks its task pushing conditions to see if it can push tasks to a neighbour. An example task pushing condition is that the task pool's task queues have an imbalance of more than than 2 tasks.

Pushing tasks is done by sending the task pool a push message. That message contains a request of the pushed task. Both task pools involved update their task assignment by replacing the assignment from the pushed from task pool to the pushed to task pool. The other task pools in the system need not be aware of this, as the pushed from task pool will simply forward any requests for the pushed task to the pushed to task pool. As a final step the pushed from task pool forwards all the remaining request it currently has in its task queue to the pushed to task pool.

```
data Push = Push Request

data TaskQLength = TaskQLength TaskPoolId Int

processPush :: TPState -> Push -> Process ()
processPush state@TPState{..} (Push r@(Request _ uid)) = do
  myPid <- getSelfPid
  let ass' = insert uid (Assignment myPid Static) taskAssignment
      newTaskState = insert uid NoWorker taskState
  processRequest state{taskAssignment=ass', taskState=newTaskState} r


processTaskQLength :: TPState -> TaskQLength -> Process ()
processTaskQLength state@TPState{..} t@(TaskQLength pid l) = do
  let nbQ' = adjust (const l) pid neighbours
      newState = state{neighbours = nbQ'}
   in case pushTaskCond nbQ' taskQueue of
        Just pid -> pushTask newState pid >>= processNextMessage
        Nothing -> processNextMessage newState

pushTaskCond :: NeighbourQueue -> TaskQueue -> Maybe TaskPoolId
pushTaskCond neighbours taskQueue = do
```

```
  let ownLength :: Int
      ownLength = length taskQueue
  min' <- safeMinimumBy (compare `on` snd) (M.toList neighbours)
  (\ (pid, len) -> if ownLength - len > 2 then Just pid
                                          else Nothing) min'

pushTask :: TPState -> TaskPoolId -> Process TPState
pushTask state@TPState{..} pid = do
  myPid <- getSelfPid
  case taskQueue of
    (t@(Request _ uid):ts) -> do
        let taskAss' = M.insert uid (Assignment pid Pushed) taskAssignment
            newTaskState = M.delete uid taskState
            newState = state{ taskAssignment=taskAss'
                            , taskQueue=ts
                            , taskState=newTaskState
                            }
        send pid (Push (Request myPid uid))
        broadcast (M.keys neighbours) (TaskQLength myPid (length ts))
        return newState
    [] -> return state
```

Listing 3.9: *Work stealing data types, functions, conditions, and messages. Additionally the message that updates the neighbouring task queue's length and the function that handles this message is also included.*

3.6 Properties of the System

In this section a couple of (informal) proofs are displayed that reason about certain properties of both the workers, the task pools and the system as a whole. For these proofs the following assumptions are made:

1. The processes are fault-free.
2. Each task in the DAG terminates in a finite amount of time once its dependencies have been resolved.
3. No messages are lost, and each message arrives in a finite amount of time.
4. Each computation starts with a set of requests (initial requests) that if satisfied terminate the computation and result in a successful computation.
5. The DAG consists of a finite amount of tasks.

Single Task Pool

In this section we will consider a single task pool with multiple workers.

Theorem 1. *Starvation of tokens can not occur in a task pool if it starts with one or more tokens.*

Informal proof. At the start of the computation the task pool receives one or more requests. Since the task pool has at least one token, it can start an initial worker. For deadlock to occur all tokens must be spent on worker processes that have deadlocked.

Each worker process is assigned a task, that may or may not have dependencies. The task's computation must terminate (assumption 2), so in case a task has no dependency the worker process can not deadlock and we are guaranteed that it will return its token by sending its task pool a finish message.

In the case a worker's task has dependencies, two types of blocking calls are made. The waiting for dependencies, and the waiting for a worker token. When dependencies are requested, the worker may still have a token, but immediately gives it back to the task pool once it has sent this message. Thus when waiting for dependencies the worker can not be in possession of a token. The requesting of a token means that the worker inherently does not have a token. Thus it is not possible for a worker to both have a token and deadlock.

Since this holds for all tasks, and thus workers, the task pool can not experience starvation of tokens.

□

Theorem 2. *A task pool will answer requests with supplies as long as it has tokens available.*

Informal proof. This theorem is equivalent with finishing the computation successfully, as providing answers (or supply messages) to the initial set of requests means a successful termination (assumption 4).

For the initial set of requests, at least a single worker can be spawned. This worker has to execute a specific task that either has dependencies, or does not have any. It will thus either terminate, immediately providing a supply, or spawn more requests and send these to the task pool.

The task pool in turn can spawn more workers (as it has tokens available) for any additional requests it receives and thus this process recursively continues until the (multiple) root(s) of the DAG are found which exist of purely tasks without dependencies. These roots must be found since the DAG is finite (assumption 5). From this point forward, workers will start generating supply messages for these roots of the DAG, which in turn can generate more supply messages in other workers until the initial requests are satisfied.

This has to happen for each worker, as the only thing a worker can block for is the waiting for dependencies. These have to be satisfied otherwise the task pool has not bottomed out the DAG which is in conflict with the previous part of the proof, or waiting for a token, which also cannot occur because we assume task pools have tokens available. □

Theorem 1 concerns itself with availability of tokens, while Theorem 2 ensures that the task pool bottoms out the dependencies such that we can start generating responses. Combining Theorems 1 and 2 we see that in the case of a single task pool, a computation will terminate successfully. The question then becomes, does this hold for multiple task pools as well?

Multiple Task Pools

We now assume a system with n task pools.

Theorem 3. *Each request will arrive to its assigned task pool in a finite amount of time.*

Informal proof. First lets consider just the static assignment. Since the function that assigns tasks to task pools is pure, and the arguments to that function are identical on each task pool, we are guaranteed that each task pool has the same task assignment. Since messages arrive in finite time (assumption 3), and we have to send a single message, we are guaranteed that requests arrive in finite time at their assigned task pool in the static scheduling.

When task stealing is considered it becomes more complicated. The key problem lies in the possibility of a task getting stolen/pushed around infinitely in the task pool graph. A key contribution of Hofstee, et al. is the proof that if the task pools are arranged in a directed acyclic graph[1] a bound function can be defined that shows, that if no more requests and supply messages are being created progress is made towards a balanced distribution of tasks. In other words, a balanced distribution of tasks will be achieved in a finite amount of time if the task queue's size only change due to tasks being pushed.

Considering that our system constantly creates requests and generates supplies, the system constantly alters the size of task queues. This however does not form a problem. Consider the following, suppose a specific task keeps getting stolen indefinitely. At some point this task will halt the system, because it is the only task which blocks progression. This must hold as it must be a dependency of some other task (otherwise it would not be required for the successful termination of the computation).

When this happens, no more requests are being spawned, nor supply messages generated. Thus a balanced task distribution must be achieved in a finite amount of time, due to the bound function from Hofstee, et al. The task that is being bounced around will therefore converge to some specific task pool. This task will reach its assigned task pool in at most $n - 1$ messages. This is the upper bound if it has been stolen by each task pool in the system, the task pool graph is a linked list, and it was stolen from the root of the linked list, to the end (or vice-versa). Since this is a finite number, the requests will arrive in a finite amount of time at the assigned task pool. □

Theorem 4. *Given a system with more than one task pools, each task pool with a more than one token, the computation will terminate successfully.*

Informal proof. The key difference between a single and multiple task pools, is that now task pools rely on other task pools to generate supply messages for their requests. Given Theorem 3 we know that request still arrive in a finite amount of time so we can apply the same reasoning from Theorem 2. Combining this with Theorem 1 concludes that we can successfully execute the computation over multiple task pools under the assumptions provided. □

[1] the task pool graph, different from the computation DAG.

Chapter 4

Resiliency

Up to this point, the actual sending and routing of intermediate results of tasks has been left unspecified. This is due to the routing scheme being heavily tied in with the resiliency mechanisms. A number of assumptions are made:

1. A task pool and its worker processes all fail together, or do not fail at all.[1]
2. Task pools can detect failures of other task pools.
3. Only stopping failures are considered.
4. The initial requests do not get lost.

Inspiration is taken from Stewart, et al. [6]. In our system ensuring requests stay alive such that they will eventually generate a reply is the requirement for resilient execution. To expand on this consider the following. Fundamentally every node can only deadlock because it is waiting on responses of the requests for tasks. From now on this reply is referred to as a supply message. Each task pool has a set of assigned tasks, of which some tasks might be stolen. That information is not globally visible. If a task pool simply checks if failures occur in the destination of its outgoing requests, it might miss that a request has been lost, due to it being stolen to a task pool that failed. Therefore if a task pool forwards a request it has to take ownership of the request, or in other words it becomes responsible for the generation of a supply message to that request.

4.1 Overview

For each task pool an outgoing request set is introduced. Each request contains the address of the (last) sender of the request, the UID of the task and the destination of the request. Once the request reaches the worker that has successfully computed the result, a supply message is generated. When a task pool receives a supply, it finds the corresponding UID in the outgoing request queue, and forwards the supply to the process(es) that requested it. In case of a failure each task pool checks whether they had send outgoing requests to that task pool and if so resubmits the lost requests elsewhere.

[1] This can be dropped but requires extra coordination from the task pools themselves in the form of restarting workers.

Now the final point that has to be solved is where to resubmit requests. Each task pool starts with a set of assigned tasks. In case of a failure these tasks have to be reassigned. In the best scenario, this would not require global communication/coordination and leave all the unaffected task assignments in place, only reassigning the tasks assigned to the failed task pool. A function that reproduces the same reassignment on all task pools solves the problem for a single failure, without global coordination.

An important observation is that task pools might observe failure in a different order than another task pool, i.e. there are no guarantees on the ordering of the failure message arriving at each task pool. This means that task pool P_1 may observe failure f_1 and then f_2 while P_2 may see them in reverse, f_2 and then f_1. The reassignment function must be commutative under composition, to ensure that each task pool still ends up with the same task assignment.

To clarify the previous point. Suppose there are n task pools $p_0, \ldots, p_{n-1}$ and there are $r_0, \ldots, r_{n-1}$ reassignment functions where r_i corresponds to the reassignment when p_i fails. Each reassignment functions takes the task assignment and produces a new task assignment. The properties this function must satisfy are:

1. $r_i \circ r_j = r_j \circ r_i$ for $i, j \in [0, n-1]$, and
2. there may be no occurence of p_i after r_i has been applied to the task assignment.

An example function that satisfies these constraints is the following. Order all the task pools in the assignment according to their process identifier ordering. In case of task pool failure p_i, r_i assigns all the tasks to the task pool that has the next biggest address, making sure it removes any assignment to p_i. In case the biggest address task pool fails, reassign to the task pool with the smallest address. In the case of n task pools this gives us n reassignment function that assigns it to the next biggest address. This function is commutative under composition and removes p_i when r_i executes.

Additional care has to be taken in making sure that the tasks that are being reassigned are the same for each task pool. Differences occur for task pools that have pushed tasks to a task pool that has failed at a later point. Whether a task was assigned or pushed can be tracked in the task assignment in the assignment type field (static or pushed). If a task pool fails that has tasks pushed to it, the pushed from task pool reassigns the task to itself. This happens before the task pool reassigns the tasks that were originally assigned to the failed task pool. This ensures that the assignment stays consistent between all the task pools. The importance of this can be seen in more detail in Section 4.3.

Another consequence of failure is that the task pool graph may become disconnected. This does not form a problem as task stealing is not required for the successful termination of the computation, which is why in the prototype we allow partitions in the task pool graph. This might be suboptimal for performance. A more elegant solution could be a function that rearranges task pools in the task pool graph in such a way that it becomes connected again. This function should be similarly commutative under composition.

4.2 Resilient Task Pool Implementation

The outgoing request set is defined in Listing 4.1. Whenever a request is forwarded the process identifier[2] of the target and the UID of the task are stored. All outgoing requests are grouped by UID. Any additional requests for the same UID are simply appended to that list as they should also be forwarded to the same process identifier.

On reception of a supply, the outgoing request queue is checked if there is a corresponding process identifier and UID combination. There are valid cases where there can be no match when failure is present in the system. Consider the following: Suppose a task pool and its workers have crashed, and we received a message of this fact. We have resubmitted all the work somewhere else. Before the task pool and its workers crashed, it managed to send out a supply message. Our task pool received the failure first, before the supply. Because the request was resubmitted, the system will generate two supply messages for this request.

```
type OutgoingRequest = Map (ProcessId, UID) [Request]

processSupply :: TPState -> Supply -> Process ()
processSupply state@TPState{..} (Supply s uid fromPid) = do
  myPid <- getSelfPid
  let index = (fromPid, uid)
      origR = lookup index outgoingRequest
      oR' = delete index outgoingRequest

  case origR of
    Just xs ->
      traverse_ (\ (Request toPid _ ) -> send toPid (Supply s uid myPid)) xs
    Nothing ->
      return ()
  processNextMessage state{outgoingRequest=oR'}


forwardRequest :: TPState -> ProcessId -> Request -> Process TPState
forwardRequest state@TPState{..} toPid r@(Request fromPid uid) = do
  myPid <- getSelfPid
  let forwardReq = Request myPid uid
      index = (toPid, uid)
      oR' = M.insertWith (++) index [r] outgoingRequest
  case M.lookup index outgoingRequest of
    Just xs ->
      return ()
    Nothing ->
      send toPid forwardReq
  return state{outgoingRequest=oR'}
```

Listing 4.1: *Definition of outgoing request set, a function that handles suply messages, as well as a helper function for forwarding requests.*

With the outgoing request queue in place, all that is left is the reception of failure messages and the resubmission of requests. Cloud Haskell provides

[2] As requests can be forwarded to both task pools and workers.

us with a way to monitor a process, sending us a message if the process fails. Each task pool monitors all the other task pools. On receipt of a failure, all the tasks are reassigned similarly to the function described in Section 4.1. After the reassignment the lost requests are resent to their reassigned task pools. These functions can be seen in Listing 4.2.

```
processFailure :: TPState -> FailureMessage -> Process ()
processFailure state@TPState{..} (FailureMessage _ tpid _) = do
  myPid <- getSelfPid
  let taskAss' = reschedule tpid myPid state
      nbQ' = delete tpid neighbours
      resubmitR = filterWithKey (\ (pid', _) _ -> pid' == pid)
                                                  outgoingRequest
      oR' = difference outgoingRequest resubmitR
      nodes' = Nodes (unNodes nodes \\ [pid])
      newState = state{ outgoingRequest=oR'
                      , taskAssignment=taskAss'
                      , neighbours=nbQ'
                      , nodes=nodes'
                      }
  s <- resubmitRequests newState ((concat . elems) resubmitR)
  processNextMessage s

resubmitRequests :: TPState -> [Request] -> Process TPState
resubmitRequests state@TPState{..} or = do
  myPid <- getSelfPid
  let reassignFunc :: Request -> TPState -> Process TPState
      reassignFunc r@(Request _ uid) ns@TPState{..} =
        let Just (Assignment pid _) = lookup uid taskAssignment
         in if pid == myPid
               then send pid r >> return ns
               else forwardRequest ns pid r

   in foldM (flip reassignFunc) state or


reschedule :: TaskPoolId -> TaskPoolId -> TPState -> TPState
reschedule pid myPid state@TPState{..} =
  let rPid = head . tail . dropWhile (/= pid) . cycle (unNodes nodes)
      reassignUIDs = filter ((== pid) . assTpid) taskAssignment
   in foldrWithKey (reassignTask rPid myPid) state reassignUIDs

reassignTask :: TaskPoolId -> TaskPoolId -> UID
             -> Assignment -> TPState -> TPState
reassignTask reassignPid myPid uid
             (Assignment tpid assType) state@TPState{..} =
  let toReassign = Assignment reassignPid Static
      toMe = Assignment myPid Static
      newAss = case assType of
                 Static -> toReassign
                 Pushed -> toMe
      newTaskAss = insert uid newAss taskAssignment
      newTaskState = case assType of
                       Static -> case (reassignPid == myPid) of
```

```
                                    True  -> insert uid NoWorker
                                                          taskState
                                    False -> taskState
                            Pushed -> insert uid NoWorker taskState

    in state{ taskAssignment=newTaskAss
            , taskState=newTaskState
            }
```

Listing 4.2: *Function that handles failure messages, including a reassignment and request resubmission functions.*

4.3 Properties of the System

The same assumptions are made as in Section 3.6, except now stopping failures are taken into consideration. Stopping failures can only occur on a "unit of failure", which means that if a process fails all its associated workers and task pool also fail.

Theorem 5. *In case of failure all tasks are reassigned in such a way that all requests take a finite amount of time to arrive to their assigned task pool.*

Informal proof. The key point to show is that after each failure the same tasks get reassigned on all task pools in such a way that all requests arrive in a finite amount of time. Considering the static assignment first. The reassignment functions order of observation of failures does not matter, and as such suffices in this aspect. This can be concluded because each task pool ends up with the same assignment, thus it will still only take a single message to send a request to its assigned task pool, which takes a finite amount of time.

Now taking task stealing into consideration. Due to task stealing, task pools might have different task assignments and will have different results from the reassignment function. We observe that a task pool can lose an assignment from being pushed from, or gain an assignment from being pushed to.

A task pool failing after losing an assignment does not form a problem. To illustrate this consider the following. All task pools, beside the task pool that the task was pushed to, will reassign the task to some other task pool. From now on all the work will be redirected there, unless the request is spawned on the pushed to task pool. This task will now have an assignment on two task pools, and can results can be supplied from both. This means tasks can be executed more than once, due to the failure.

On the other hand, failing after gaining a task is a problem. The pushed from task pool, would naively reassign this task, while other task pools would not. This would create a scenario, where the pushed from task pool tries to forward the request to some other task pool, while that other task pool would try to forward it back, creating a loop in the task assignment, effectively forwarding the request endlessly.

From this we can see that pushed to work needs to be reassigned separately first. The easiest way to do this, is revert the push, and reassign the task to the pushed from task pool. Note that his has to be done solely for the work pushed

to the failed task pool. This gives us the guarantee that we end up with a valid task assignment, that ensures tasks arrive at their endpoint in a finite amount of time. □

Theorem 6. *In the presence of stopping failure, the system will provide answers for each request in the system as long as one or more task pools stay failure-free.*

Informal proof. The key point is showing that if a request is lost, we have to resubmit it somewhere to be able to generate a supply. Theorem 5 shows that we will indeed provide an assignment that guarantees that request arrive at their assigned task pool in a finite amount of time. Furthermore the outgoing request queue enables us to resubmit all the lost requests, guaranteeing that we provide responses for the lost requests.

Therefore the reasoning in Theorem 4 holds as long as the resiliency mechanisms have been implemented appropriately. This of course can only be true as long as single task pool remains, in which case all the work is assigned to this task pool. □

Chapter 5

Results

The resiliency as explained in Chapter 4 was tested using a system similar to Chaos Monkey [7]. Chaos Monkey is a service developed by Netflix to actively test the resiliency mechanisms of their cloud by purposefully killing peers. Netflix' services are designed to be failure tolerant. Rather than waiting for failure to occur spontaneously, Netflix chooses to simulate failure to be able to proactively ensure their cloud is indeed resilient. This allows defects in their failure handling to become apparent in a more controlled (during business hours) setting.

In the spirit of Chaos Monkey we engineered a similar failure injection model to verify that our system meets our resiliency requirements. A system was built where randomly a number of processes are killed (in our case a task pool and all its workers). A set of twenty task pools are started. On random uniform intervals between 0 and 2 seconds up to three task pools and their associated workers are killed (using POSIX kill signal). As long as the initial request stays alive the computation terminates successfully. This requirement can be dropped by making the initiator (the outside machine) also resubmit lost requests similarly to the task pools. Alternatively the initiator can send its initial request to all the task pools guaranteeing that the computation will terminate, as long as a single task pool stays alive.

Chapter 6

Discussion

6.1 Discussion of Referenced Work

A number of works were referenced throughout this thesis that served as inspiration. The comparison between this work and Apache Spark has already been clarified throughout this thesis.

The work stealing algorithm from Hofstee, et al. that we started from, does not specify in what way tasks get generated to be able to fill the task queues. This collides with the lazy execution model as it is not clear what the contents of the task queue will be. Introducing the worker tokens, modelling the resources of a task pool, and allowing tasks to be queued gives rise to the notion of task queues in our system.

The resiliency mechanisms were inspired by Stewart, et al. which uses a fork-join model as the basis for their computation. Their workers are spawned by forks, which in turn can recursively spawn more workers. To ensure fault tolerance, every workers supervises all the workers they spawned, restarting them in the case of failure. This creates a hierarchical tree of workers ensuring the resiliency of their spawned workers. Resiliency is guaranteed as long as the root node of this tree survives. Our model does not have this hierarchy, and there are only two layers in the system — task pools and workers. Moving the responsibility of restarting to the responsibility of generating replies to outgoing requests allows us to modify this idea to our framework. The reassignment function is key in ensuring requests have a consistent destination after failures occur.

6.2 Discussion of Results

The work presented in this thesis is compared to Spark, so we should ask ourselves to what extent the ideas proposed are suitable for a real world implementation.

I believe that the fundamental thoughts: the richer model of computation, distributed scheduler, and distributed resiliency mechanisms, form a cleaner foundation than those of Spark. There are however a number of (practical) concerns. This work is built on the actor model. This model does not take into account a number of real world problems such as a finite amount of resources. One of the most important topics in Spark performance is the handling and

freeing up of memory. This is something that is completely open in our system. There is no notion of garbage collection, nor the subsequent freeing of resources. This is a problem that has to be solved before any practical value can be achieved. In Chapter 8 some suggestions are given as to how this problem could be solved.

Another discrepancy between theory and practice in this system is that the resiliency aspects of the system only consider a single type of fault, namely stopping failures. There are other fault categories that can appear in real systems, such as transient, intermittent, Byzantine.

Another origin of failure is that of network partitions. These also come in variety of different categories, for instance permanent and transient. These failures have a similar aspect as transient and intermittent process failures, namely resources being removed from the system as well as them becoming available again.

The notion of being able to gain and lose resources while continuing with the computation can be viewed as a generalization of resiliency. Designing for such a notion is complicated for the system presented in this thesis.

Consider the following. If our system should support the becoming available of resources, this resource should be a task pool to form another unit of failure. This new task pool can only be introduced, or communicated to task pools via a message. On receipt of such a message the task pools should assign some work to this new task pool to utilize the increased capacity. To effectively make use of these gained resources, we would like to assign it uncompleted tasks. Whether or not a task is completed is information that is private to a task pool. Implementing this would effectively require global coordination in the system, which is something we have carefully avoided to stay true to a more distributed approach. Even if we assign no work to this new task pool and purely rely on work stealing, this would mean our reassignment function would have to be commutative with failure of resources *and* gaining of resources. The example reassignment function provided in this thesis and the prototype is not.

There is also the problem that for many real world workloads, we would like to consider scheduling on heterogeneous hardware. FPGAs, GPUs and ASICs are becoming more and more common in the cloud. Even though this is a problem that is not solved for Spark either, their centralized scheduler does make this problem vastly simpler to solve.

The system introduced in this thesis provides fertile ground in the form of a more coherent foundation, but as illustrated it would require a whole lot of engineering work to be in a state where it would be useful in a practical setting. A lot of open problems are left to be solved, and the question remains whether the better fundamentals outweigh the amount of engineering work that would be required.

Chapter 7

Conclusion

In this thesis a framework was presented that allows for the distributed execution of DAG computations in a resilient fashion on a distributed system. An extension to the narrow and wide dependency model from Apache Spark was introduced. This extension allows for a wider range of applications to fit in the model of computation, while not sacrificing the immutability and recovery aspects that are so foundational to the Spark framework. Access mechanisms for the DAG on a distributed system were introduced as well as a scheme for unique identifiers that is heavily tied in with the distributed execution.

The distributed execution algorithm was presented. This algorithm does not require any centralized coordination. The scheduling was implemented in two parts: a static assignment, and task stealing. This two part system provides a rough partitioning of the DAG together with dynamic refinement to achieve a balanced distribution of tasks. A number of performance heuristics are presented in the form of the implementation of the static assignment function and the work stealing topology. These tuning parameters are independent of computation and allow for a good separation of concerns.

An extension to the scheduling algorithm allows task pools to respond to stopping failures and achieve resiliency against these faults. This is achieved with only local information and actions, which greatly simplifies the implementation and reasoning, besides having favourable performance characteristics. Some properties of the system were derived.

In our work the focus was on creating a flexible framework, and exploring the design space around the presented work. An efficient implementation would require more engineering, but given the characteristics presented in this work has the potential to be competitive.

Chapter 8

Future Work

In this chapter a number of possible improvements are presented.

8.1 Network Topology Optimization

This framework leaves a number of options open in the implementation. Among these are the way the framework arranges the processors in a DAG for the work stealing, and the way the tasks are initially assigned to processes.

Both these functions can be adjusted for specific network topologies. This allows for optimizations based on the network topology without altering the computation specification.

8.2 Garbage Collection

If a task pool runs out of memory it needs to select some computed data to release. This can be done safely because if another task pool requests it again, it can just recompute the results. A number of policies can be used to determine which results to evict, for example a least recently used policy where the least recently used task result is evicted.

8.3 Streaming

Streaming is a very common paradigm in distributed computing. In Spark streaming is implemented by cutting up chunks of data and processing these chunks throughout the system. This can be implemented as a DAG computation by modeling all the chunks as separate input vertices. This creates an infinite DAG. As long as some form of lazy construction of the DAG and UIDs is supported by the framework this can be used.

In the case of continuously updating streaming application, all the results will depend on the entire input stream. To avoid enormous costs on recomputation in case of failure, a checkpointing mechanism can be introduced, which given some suitable condition converts some task with dependencies to a new source by persisting it to some redundant readily available medium (like HDFS). This allows all the previous dependencies to be deleted and garbage collected.

This checkpointing mechanism does not have to be limited to streaming applications but can also be useful in long running computations.

Bibliography

[1] Mark Jones. "Functional programming with overloading and higher-order polymorphism". In: *Advanced Functional Programming* (1995), pp. 97–136.

[2] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[3] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. "Towards Haskell in the cloud". In: *ACM SIGPLAN Notices*. Vol. 46. 12. ACM. 2011, pp. 118–129.

[4] Joe Armstrong. "Erlang". In: *Communications of the ACM* 53.9 (2010), pp. 68–75.

[5] H Hofstee, Johan Lukkien, and Jan van de Snepscheut. "A distributed implementation of a task pool". In: *Research Directions in High-Level Parallel Programming Languages* (1992), pp. 338–348.

[6] Robert Stewart, Patrick Maier, and Phil Trinder. "Transparent fault tolerance for scalable functional computation". In: *Journal of Functional Programming* 26 (2016), e5.

[7] *Chaos Monkey Released Into The Wild*. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`.