Big Data Types
Internally Parallel in an Actor Language

Josef Karakoca

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Big Data Types: Internally Parallel in an Actor Language

Josef Karakoca

Around year 2005 the hardware industry hit a power wall. It was no longer possible to drastically increasing computer performance through decreasing the transistors’ size or increasing the clock-speed of the CPU. To ensure future development multi-core processors became the way to go. The Programming Languages Group at Uppsala University is developing a programming language called Encore that is developed to be scalable to future machines with a few hundred or even thousand processor cores.

This thesis reports on the design and implementation of Big data types. Big data types are locally distributed data structures that allow internal parallelism in the actor model by using several actors in their implementations. Thus, rather than serializing all interaction these data structures are potentially as parallel as the number of actors used to construct them. The goal of Big data types is to provide a tool that makes it easier for an Encore programmer to create parallel and concurrent programs. As part of our evaluation, we have implemented a Mapreduce framework which showcase how of Big data types could be used in a more complex program.
# Contents

1 Introduction 3
   1.1 Motivation ........................................... 3
   1.2 Goal & Requirements ................................. 4
   1.3 Concept of Big Data Types ......................... 4

2 Background 6
   2.1 Encore ................................................. 6
      2.1.1 Active Objects .................................. 6
      2.1.2 Futures ........................................... 7
      2.1.3 Kappa - Encore’s Type System .................. 7
      2.1.4 Syntax of an Encore Program ................. 8

3 Related Work 9
   3.1 Hot Objects in Encore ............................... 9
   3.2 Julia’s Distributed Arrays ......................... 9
   3.3 Distributed Hash Tables ............................. 10

4 Designing Big Data Types 11
   4.1 Big Arrays ........................................... 12
   4.2 Big Hash Tables ...................................... 12
   4.3 Mapping Design with Encore Features .......... 13
   4.4 Problems with Synchronizing Supervisors .......... 13
   4.5 Parallelism and Concurrency ..................... 14
   4.6 Operations to Support .............................. 14

5 Implementing Big Hash Tables 16
   5.1 Big Hash Tables ...................................... 16
   5.2 Structure of Supervisors ............................ 17
   5.3 Creating More Than One Supervisor .............. 17
   5.4 Resize Workers ...................................... 19
   5.5 Support for Generic Types ....................... 19
   5.6 Process of Hashing Key Values .................... 20
   5.7 Matching Values to Workers ...................... 20
6 Evaluation and Optimization
  6.1 What is Mapreduce? ........................................ 21
  6.2 Mapreduce with Big Hash Tables .......................... 22
    6.2.1 Example 1: Word Count .............................. 23
    6.2.2 Example 2: Parallel Breadth-first Search ......... 24
  6.3 Extending Big Hash Tables ................................ 26
    6.3.1 Provided Functionality .............................. 26
    6.3.2 Extended Functionality ............................. 26
  6.4 Optimizing Big Hash Tables .............................. 27
    6.4.1 Flame Graphs ........................................ 27
    6.4.2 Generated Flame Graphs ............................. 27
    6.4.3 Optimizing Message Sending ......................... 28
    6.4.4 Optimal Number of Workers ......................... 28

7 Future Work .................................................. 31
  7.1 Big Data Types as First Class Citizens .................. 31
  7.2 Bestow and Atomic ....................................... 32
  7.3 Comparisons .............................................. 32

8 Conclusion ...................................................... 33
  8.1 Programming in Encore ................................... 33
  8.2 Concluding Remarks ....................................... 33
Chapter 1

Introduction

1.1 Motivation

Around year 2005 the hardware industry hit a power wall [16]. It was no longer possible to drastically increasing computer performance through decreasing the transistors’ size or increasing the clock-speed of the CPU. To ensure future development multi-core processors became the way to go. Today almost all personal computers have more than one core and developing programs with strong parallelism support is more important than ever [17].

Parallel programming introduces many problems that the programmer must solve. The Programming Languages Group at Uppsala University is developing a programming language called Encore [15] that is developed to be scalable to future machines with a few hundred or even thousand processor cores. Encore breaks the norm that program instructions are run sequentially by default. An certain way of programming is encourage and with language support instructions are able to run concurrently and in parallel automatically instead of sequential order.

Actor based arrays and hash tables in Encore have all their content stored in a single actor, an actors is a processes that live on a single thread, have their own memory and communicate with massage passing. Operation inside actors are therefor thread safe and operations can only be executed sequentially. To continue the norm of the Encore Language to not have sequential instruction as default this thesis examines different kinds of big data types which is not limited by having to store all data inside a single actor. Big data types store collections of values in many different actors and make it possible for certain operations to be executed concurrently and in parallel.
1.2 Goal & Requirements

The goal of this thesis is to implement and test concepts of locally distributed data types with inspiration from similar implementations in other actor based languages. Part of the process is to evaluate and design an implementation of locally distributed data types suited for the Encore environment which will be called Big data types.

Evaluated both usability and performance of the final result. Write a none trivial application that use the at lest one of implementations of the locally distributed data types. Discuss and determine if it is necessary to continue development of the implementation or if other approaches are more suitable for the future development in Encore.

1.3 Concept of Big Data Types

Big data types is structured in a way so that its data is divided in different processes on the same computer. Big data types tries to solve problems when large amounts of data is stored inside one actor. For example if heavy independent operations is to be done to every element in an array there is currently no easy way of making the operations run concurrently or in parallel. Instead of iterating through all elements sequentially in one actor Big data types would use many actors to iterate through different parts of the array which make them able to be scheduled on different cores at the same time. With large arrays this could be very beneficial for performance.

Interacting with Big data types should be familiar with how it’s done with the corresponding actor based data types already implemented in the Encore programming language. The final design should embrace core features of Encore and provide abstractions for the concurrency and parallelism needed. For example the design shown in Figure 1.1 the red circles representing different processes would while using Encore each be there own active object.
Figure 1.1: A graphical view of a Big data type where data is divided between different processes (red). The supervisor (blue) has information about the data type, links to all processes and can handle requests that want to access or run operations on data in the different processes.
Chapter 2

Background

The implementation and design part of this thesis is written for a new programming language called Encore. Some of Encore’s most important features are described in this section. Basic knowledge of Encore is needed to fully grasp the content of this thesis.

2.1 Encore

Encore is developed at the Programming Languages Group [5] at Uppsala University and is funded by the European project Upscale [7]. Encore has been in development since 2014 and was released under an open source licence in April 2017.

Encore is a statically typed high-level object-oriented programming language which also use the actor model. Every Encore program has at least one actor but Encore has support for millions of actors even on a machine that only have a few cores. Encore offers many abstractions for concurrency and parallelism beyond the pure actor model and uses a capability type system that ensures no data races occur.

Encore’s compiler is written in Haskell and translates Encore code to C code which is then compiled to a binary executable.

2.1.1 Active Objects

Encore supports both passive and active objects. Passive objects are initialized inside actors and executions of methods are synchronous just like objects in other popular object oriented languages like Java. Active objects are actors and method calls on them are asynchronous. Each active object has its own message queue and processes each message sequentially. One benefit of active objects is that they can be scheduled on different cores without the programmers
involvement. There is however an overhead when using message sending instead of normal method calls [15]. Active objects embraces the actor model but still allow programs to be written in an object oriented fashion by providing passive objects and sequential execution inside them.

2.1.2 Futures

Futures are an important building block of Encore, as messages may return futures. A future is a placeholder for a value that is returned from an asynchronous call. A Future is returned immediately to the caller but the return value is stored later during the programs execution. Reading a value from a future before the future has been fulfilled will block the reader until the value have been calculated and stored [15].

![Figure 2.1: Graphical view of message passing and futures in Encore [15].](image)

2.1.3 Kappa - Encore’s Type System

Encore relies on a reference capability-based type system called Kappa [9; 10] for concurrency control. Kappa ensures that no data races occur by assigning a mode to all class declarations. Modes determine how the object built from classes can be shared during runtime.

The different modes in Encore are defined by local, subord, read, linear and active. Instances of local classes can be aliased inside a single active objects but not shared between them. Instances of subord classes are like local objects but they are restricted by the context of there creation instead of a single active object. Instances of read classes are not allowed to be mutated and are therefore safe to share. Instances of linear classes can not be aliased and can only be
transferred between active objects if consumed with the consume keyword. If a linear object is consumed the ownership of that object is transferred to the new instance. If the linear object is not consumed before transfer a compiling error will occur. At all times there is only one usable reference to a linear object [13]. Instances of active classes are active objects.

2.1.4 Syntax of an Encore Program

```encore
active class Main
  def main() : unit
    print("Hello world!")
  end
end
```

The above code is a simple program written in Encore which prints "Hello World!" to the standard output. All Encore programs consist of at least one active class and one main method.

Encore syntax is a mix of both functional and imperative styles with inspiration from languages like Haskel, Scala and Erlang. Method calls to active objects use the (!) bang operation and are asynchronous. The future returned can be accessed by using the get keyword. Synchronous calls to passive objects use the (.) operation [15]. Encore uses maybe types instead of null pointers. A type Maybe t can either be of type t or Nothing. Patter matching can be used to convert a maybe t type to t. The def keyword is used to declare methods and the end keyword is used to end code blocks.
Chapter 3

Related Work

The thesis goal to implement a version of locally distributed data types were inspired by other applications that strive to not limit performance by the need to run operations sequentially. How these applications works and how they are related to the work in this thesis is described further in this section.

3.1 Hot Objects in Encore

Hot Objects are actors that allow internal parallelism. The rationale for Hot Objects stem from systems where some actors get a lot more traffic than others. Hot Objects introduces locks to allow actors to handle more then one massage at the same time. The introduction of locks only affects the methods inside the Hot Object thus the locks are encapsulation inside actors [19]. A prototype of Hot Objects was implemented in the Encore programming language but at the time of writing this thesis (spring 2017) the implementation is not up to date with the current version of the language.

To understand how Hot objects solve problems similar to Big data types look at the example from section 1.3 where an array live inside a single actor. The problem in that example was that arrays that only lives inside active objects are limited by sequential operations. Hot Objects are not limited by this because they can handle more then one massage at the same time and therefore access different parts of the array in parallel while still avoiding data-races with locks. Hot Objects could achieve a performance boost for large arrays by introducing parallelism in another way than what Big data types suggests in example from section 1.3.

3.2 Julia’s Distributed Arrays

Julia is a high-level, dynamic programming language that focuses on numerical computing. Julia provides functionality for distributed parallel execution,
numerical accuracy and an extensive mathematical function library [4]. The philosophy of Julia is similar to Encore’s in the way they both want to provide ways for the programmer to easily achieve parallelism. The library ‘Julia Distributed Arrays’ [3] was one of the main inspirations for this thesis project.

To Compare Big data type with constructs in the Julia language, look again at the example problem in section 1.3 where arrays live inside actors. To not be limited by having to run instructions on a single core Julia is changing the structure of actor based arrays. With the Distributed Array library the data in an array could be divided between different processes and through library functions operations could be run in parallel interacting with different parts of the data in different processes at the same time. One big different between Julia’s distributed arrays and Big data types is that Julia provide functionality to divide data between different machine which Big data types do not. The idea of both however is for the library to abstract away parts of the work to achieve parallelism to make it easier for the programmer to write parallel programs.

3.3 Distributed Hash Tables

Distributed hash tables also called DHT is important for modern distributed systems, particular in large-scale distributed environments. In a normal Client/Server model most of the resources is centralized at the server, this could be a bottleneck and potentially even a weak point for the system. DHT uses a peer-to-peer (P2P) model where each node is treated equally and a single nodes maintains a portion of the hash table which help with both load balance and the ability to scale into large networks [11].

DHT also inherits properties from hash tables such as the ability to locate and search an element with high efficiency. It does this even when each node do not have a complete view of the system [11]. This and the ability to easily scale inspired the need to introduce a version of big data types which worked like a hash table instead of an array. The structure of Big data types and DHT is however very different. Big data types is not fully decentralized and have specialized supervisors to handle requests. One of the reason that they are different is because they also have different use cases. Big data types is not designed for large scale distributed systems, they are locally distributed on a single machine and want to serve as an alternative to actor based data types and do not try to replace the Client/Server model.
Chapter 4

Designing Big Data Types

Big data types have its data distributed between many actors so that operations on different parts can be scheduled on different cores. The structure of Big data types is represented by two layers, the upper and the lower layer (see figure 4.1). The lower layer is not visible from the programmers point of view while the upper layer is. This is because it makes Big data types more easy to use (see section 1.3 for concepts of Big data types) trough abstracting away how the data is divided. The programmer is do not need to know the low level structure to be able to write parallel program with Big data types.

Figure 4.1: A graphical view of a Big data types with one Supervisor (blue circle) in the upper layer and 4 Workers (red circles) in the lower layer. The grey boxes inside the Red circles represent values that each Worker stores.
The lower layer consist of a number Workers. A Worker is an active objects that stores and operates on parts of the data the Big data type owns. The optimal number of Workers and the number of values each worker stores can vary a lot depending on which application it is used for.

The upper layer is the layer the programmer interacts with and it consist of one or more Supervisor objects. A Supervisor coordinates requests and contain necessary information about the data type, like the number of workers, currently number of inserted elements and what hashing function is in use.

4.1 Big Arrays

In a Big array every Worker stores both a local index for its data as well as a global one representing which part of the array index that the specific worker owns (in other words which elements that are stored inside the worker). The Supervisor need at all times have a synchronized picture of all the workers to not send data to the wrong place. For example if we have an Big array with 20 slots and 2 Workers the first worker would have slots for elements with a global index of 0 to 9 while second Worker have slots for elements with a global index of 10 to 19 (The local index of both Workers would be is 0-9). If the Supervisor get a request to insert an element on index 5 it should pass that request to the first Worker.

Under usage the Big array may be changed in ways so that the data is no longer evenly distributed between workers. An unevenly distribution is bad for performance. For example if a task is carried out on all elements in the Big array some Workers would finish much faster than others resulting in less probability for operations to be scheduled on different cores which is the point of using Big arrays. This problem can be solved by occasionally recreating and redistribute the structure, this is however very costly and should not be done to often.

4.2 Big Hash Tables

The Big hash tables operate in similar fashion as the Big arrays but indexes is not necessary for the hash tables, the data still needs to be divided evenly for it to be efficient. A good hashing algorithm will help to create an even distribution. To determine which Worker owns what value a modulo operation will be used on the hashed key value and the total number of Workers, the modulo operation will return a worker id that can be used as an index in the list with Workers. For example if we have 4 Workers we calculate modulo 4 on the hashed key value, the result will be either 0, 1, 2 or 3. The result will always be the same for the same key value if we have the same amount of Workers.
4.3 Mapping Design with Encore Features

One of the Goals of the thesis (see section 1.2) is to design Big Data type so it suites the Encore programming language. The design is made while thinking about core Encore features such as active objects and message passing. Remember that each active objects handle each message sequentially but messages are sent asynchronously. At the lower level each piece in the Big data type will be constructed by an active object. In a use case where several different processes want to access the same part of the Big data type in Encore each part (each active object) will have a queue of wanted operations, operations on different parts of the data type could therefor be scheduled concurrently and in parallel. The Encore type system also guarantees data-race freedom [13]. Implementing the structure of Big data types described in the sections above in an object oriented language that uses threads is possible but would be different and possible harder compared to an implementation Encore and the actor model.

4.4 Problems with Synchronizing Supervisors

Creating more than one Supervisors can result in problems but it is necessary to be able to send request to the Big data type from more than one active object. If a Supervisor gets a request to change the amount of Workers it can not be processed like a normal request because all Supervisors needs at all times have a correct view of how the big data type is structured. If one Supervisor changes the structure when another Supervisor is sending requests to the lower level requests could get lost. This is the case with Big hash tables due to the modulo operation described in section 4.2, it uses the number of Workers to find which Worker that stores what value and if the number of Workers is changed a incorrect ID will be calculated.

The chosen solution is that the users can’t requests to resize the Big data type, instead of changing the amount of Workers the individual Workers increase in size if needed. Each Worker will have a constant when it will run a function to double the amount of entries, for example when more than 70 percent of the entries is filled. How many times each Worker is resized depend on the input data. This result in an automatic behavior of resizing which always have entries available for the user. With this solution the amount of Workers never change during execution and the algorithm in section 4.2 will therefor work correctly. This solution was chosen because it i simple and fits the goal of further abstraction of the lower layer. The biggest trade-off for this solution is not being able to change the number of Workers. One way however to come around that problem is to create a new Big data type and transfer all data. The need to change the number of Workers varies a lot depending on the application, but if the data size is change drastically it could be good idea to change the number of workers to maximize performance.
Figure 4.2: Each worker resize itself automatically if needed. The blue circle represent a Supervisor and the red circles workers.

4.5 Parallelism and Concurrency

Encore does not guarantee parallelism. This means that it is not possible to request things to run in parallel, only that things are allowed to run in parallel. If instructions are scheduled on different cores is decided by the run-time. If all instruction from the Big data types are scheduled sequentially most operations will be slower because the structure is more complicated than actor based data types.

Defining the amount of Workers for a specific application can be challenging. A larger amount of Workers result in higher possibility for instruction to run concurrently and in parallel but more Workers also result in higher cost for certain operations. Finding the amounts of Workers which result in best performance for the problem and the hardware (different amounts of cores could have a impact on which parameters are good) can best be found with trail and error. Finding the optimal number of Workers is explored further in section 6.5.6.

4.6 Operations to Support

Big arrays and Big hash tables have a similar design, they have similar problems and strengths and is thus discussed together as the same thing. Most operations that already are supported by Encore’s build-in data types are supported and named similarly. For example the hash map in Encore standard library has a
method for removing items named remove(key : k) and Big hash tables corresponding method is named the same and have the same arguments. This makes the Big data types feel like a add-on to normal data types and interacting with them are very similar.

Methods for inserting and deleting many values at the same time are supported. These iterate through the input lists of values and send asynchronous calls to the lower level to delete each value in the Worker where those values are stored.

To understand the flow of operation through the structure of the Big data type we look at a simple operations for inserting values. The Supervisor gets the request and calculates which worker should own the value, then it passes a message with the element that is going to be inserted to the correct Worker. The Worker then reads the message and calculate in which of its local entries that should store the value and inserts it there.

Methods for filtering values of the Big data type with a programmer provided lambda function are available (see 5.2.2 for an example). Methods to return sizing and distribution information about the Big data type exists to help programmers test efficiency of their programs. An example of this is a function that returns how many empty entries each Worker has. Different initializing and converting functions are available like converting an array to a Big Array or initializing a Big array with random values.
Chapter 5

Implementing Big Hash Tables

The design of Big Data types described in chapter 4 have been explored further by implementing Big hash tables in the Encore programming language. This chapter describe details of the structure and how the Big hash tables core features are implemented. Big hash tables are built completely with constructs that exists in Encore. Thus, Big data types are provided as a library.

5.1 Big Hash Tables

A big hash table is implemented by many small disjoint hash tables, each wrapped in an actor. A passive frond-end keeps track of what hash ranges are handles by what small hash table and delegate the work accordingly. Each small hash table is refereed to as a Worker and is impalement by an active class named Worker.

The front-end (or the upper layer) is implemented by a passive class named Supervisor. Supervisors have references to all Workers and decide which worker should own what data just like the design suggest.

Interacting with a Big hash table is very similar to how one would interact with an actor based hash table. The method put(key,value) is used to insert values into the Big hash table and the get(key) method is used to extract values.

```java
1 var ageTable = new Bighash[String,Int](StringID)
2 ageTable.put("Kent",19)
3 ageTable.put("Eric",21)
4 var ericAge = ageTable.get("Eric")
5 println(ericAge)
```
The code example above shows the creation of a Big hash table and usage of the put and get methods. This block of code prints 21 to the standard output when it is executed. When creating a new Big hash table the programmer need to define type used as well as a hash function that hashes the key value type.

5.2 Structure of Supervisors

Methods calls to a Big hash table are first handled in a Supervisor. After the Supervisor has calculated which Worker owns the value that the method want to access it delegates the work to that Worker. See section 5.5 and 5.6 for calculations details for how workers are chosen.

```plaintext
linear class Supervisor[sharable k,sharable v]
  var numOfWorkers: int
  var workers : [(Worker[k,v])]
  var workerSize : int
  var siphash: Siphash
  var hashFunction : k -> uint
```

The Supervisor class is a Linear class which means it can only exist one reference to objects initialized from the class, section 5.3 explain how it still is possible to have more than one reference to a Big hash table. The sharable keyword makes it possible to pass generic values between active objects. The Supervisor class fields hold information about the structure of the Big hash table. It stores the amount of Workers, how big Workers are and which hashing function that is used.

```plaintext
def removeMany(keys:[k]) : unit
  repeat i <- |keys| do
    var hash = this.generateHash(keys(i))
    var workerID = this.modulo(hash,this.numOfWorkers)
    this.workers(workerID) ! remove(keys(i),hash)
  end
end
```

RemoveMany is a method which takes a list of keys that the programmer wants to remove from the Big hash table. The Supervisor iterate through them and for each key calculate which worker own the specific key. Messages are then sent to Workers to remove that key-value pair from its entries.

5.3 Creating More Than One Supervisor

There are in some cases (like the program in section 6.3) beneficial for the Big hash table to be able to be accessed from more then one place, both in the aspect of performance and usability. If all messages is sent to the original reference of
the data type it could become a bottleneck if lower level finish work faster then one Supervisor can delegate. It also make it possible for different active objects to send massages to the bottom level.

The front-end (or the top level) consist of instances of a linear class which means there can only be one reference to that object and it can be passed to other active object only by consuming it [9]. Before the programmer consume the Big hash table it can copied with the copy method. The copy method creates a new front-end by creating a new passive Supervisor object with the same information as the first one. The new Supervisor is then linked to the same Workers at the bottom level.

```plaintext
1 var ageTable = new Bighash[String,Int](StringID)
2 var ageTableCopy = ageTable.copy()
3 activeObject ! f(consume ageTableCopy)
```

The code example shows the the creations of a Big hash table and how to copy the front-end to be able to pass it to another active object and still be able to use it. f is a method in the active object activeObject.

![Figure 5.1: 3 Supervisors (blue) are linked to the same Workers (red). It’s possible for both Supervisors (if they are in different active objects) and the Workers to be scheduled on different cores.](image-url)
5.4 Resize Workers

```plaintext
def resizeIfNeeded() : unit
    if (this.tableSize - this.filledEntries < this.tableSize/4) then
        this.rehash()
    end
end
```

The method resizeIfNeeded() checks if a worker needs to be resized. It runs before inserting any value into a hash-entry. In the current implementation the size of a Worker is doubled every time it is more than 75% full. Starting with a large amount of entries can result in fewer resizes but it takes up more space.

5.5 Support for Generic Types

To function Big hash tables need a unique identifier for each object used as a key. In the Java every object has unique hash code which can be generated with the hashcode() method, Encore provide no such feature. The identifier needs to work both with classes created by the programmer and with standard Encore data types. If two strings contain the same characters they should generate the same identifier. For Big hash tables this problem is solved by asking the programmer to provide a hashing function that translate the key value to an unsigned integer (uint). The hashing function is needed in the initialization of the Big hash table. Below is one example how unique identifier for strings could look.

```plaintext
EMBED
uint64_t strHash(char* input);
BODY
uint64_t strHash(char* input) {
    uint64_t hash = 5381;
    char *str = input;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}
END
fun stringID(s : String) : uint
    var ans = EMBED (uint) (uint64_t) strHash(#s.getData());
    END
end
```

This example contains embed statements which allows embeded C code in Encore files. The algorithm used for hashing strings is called djb2 and was implemented by Dan Bernstein [6]. The Encore function stringID can be used with
a Big hash table where strings are used as key values.

### 5.6 Process of Hashing Key Values

All key values are first hashed to a uint type by a programmer provided function. That uint value is then hashed with a Siphash. Siphashes are optimized for short input’s and was created to provide protection from hash-flooding. Hash-flooding is an attack on a software where the attacker tries to disrupt the system by designing the input so every insertions end up in the same hash entry [8]. Siphashes help Big hash tables to distribute elements evenly by having a diverse output range. This is very important for Big hash tables to be efficient and have a high possibility for concurrency (because it increases the probability that operations are evenly divided between actors).

### 5.7 Matching Values to Workers

When the final key value is hashed it is used to calculate which Worker should own the key-value pair. A modulo operation is used to determine which worker that is, the hashed value modulo total number of workers. For example if we have a hash value of 36 and 32 Workers in the Big hash table, the Worker id is 4 which is a result of 36 modulo 32. That id 4 is used as an index in the list with Worker pointers inside the Supervisor. This is simple solution that with an efficient hashing algorithm will distribute the different keys evenly between Workers.

Here is the put method inside the Supervisor class, it generates a hash and the calculate which worker own the key-value pair, it then calls the put method in the Worker.

```haskell
1 def put(key:k, value:v) : unit
2   var hash = this.generateHash(key)
3   var workerID = this.modulo(hash, this.numOfWorkers)
4   this.workers(workerID) ! put(key, value, hash)
5 end
```

This chapter described the structure of the implementation and some important parts of it. The source code of the Big hash table can be found here: github.com/parapluu/encore/tree/development/modules/standard/Big/HashMap
Chapter 6

Evaluation and Optimization

Part of work for this bachelor thesis included writing a none trivial program that use Big hash table. This was done to showcase how Big data types was intended to be used to write parallel programs. The implemented program described in this section is a version of a Mapreduce framework that uses Big hash tables.

6.1 What is Mapreduce?

The style of computing called Mapreduce have been implemented for several systems. The term Mapreduce was first introduce by Google with an internal implementation (simply called Mapreduce) [12]. One popular open source implementation created by the Apache Foundation is Hadoop [1]. Mapreduce could be a good choice for solving problems that can be run in parallel, have large amount of input data and have access to big numbers of computational nodes. The programmer prepare input and write two functions that are implemented according the Mapreduce protocol. The framework then run a task which have three main stages, “Map”, “Shuffle” and “Reduce”. In the Map stage every node applies one of the programmer-provided function to its local data. The shuffle step then redistributes the data depending on the output from the Map stage. Finally the Reduce step reads each group of data and outputs a result from each group [14]. In figure 6.1 a simple example is showcased to get an overview of the whole Mapreduce process.

Figure 6.1: Shows a simplified view of a Mapreduce process. In this example a mapper finds the number of characters in a text and output each characters with the number of occurrences. All the characters are sorted together and passed on to the reducers. Each reducer chooses the largest number connected to each characters and outputs that to the final result [14].

21
6.2 Mapreduce with Big Hash Tables

For a programmer to be able to run a Mapreduce algorithm two functions needs to be created, these two functions are called Map and Reduce. The Map function job is to sort the input data into different piles and Reduce function job is to take each pile and output a result (as described in section 6.1). The Map and Reduce functions needs to have specific types for the input and output, which types these are defined when the Mapreduce object is initializing. The Map and Reduce functions do not use any Big hash tables, its the Mapreduce Object that are constructed by Big hash tables. The structure of a Mapreduce object is showcased in figure 6.2. In that figure we also see how each level consist of a Big hash table and that types of Big hash tables match with the Mapreduce objects types. Two Mapreduce algorithms have been implemented to showcase how the Mapreduce implementation with the Big hash tables can be used.

Figure 6.2: A Graphical view of the implemented Mapreduce object, each level consist of a Big hash table (each row of four red circles is Big hash table). For a input to go through this structure once is called to run one Mapreduce job. Some algorithms need many iterations to complete, if this is the case the types k2 and v2 matches the types k1, v1.
6.2.1 Example 1: Word Count

The process to run an Mapreduce algorithm with the thesis implementation is described in this section. The first example runs an algorithm that counts the occurrences of words in a text.

```
fun map(key:int, value:String) : [(String ,int)]
  var words = value.to_lower().split(" ")
  var result = new[(String ,int)](|words|)
  repeat i <- |words|
    result(i) = (words(i),1)
  end
  result
end

fun reduce(key:String, values:[int]) : (String,int)
  var sum = 0
  for value <- values do
    sum += value
  end
  (key,sum)
end
```

The first step is to create the Map and the Reduce functions (see these functions for the Word Count in the code snippets above). The second step is to prepare the input data by inserted it into a Big hash table object so it can be passed to a Mapreduce object. To initialize a Mapreduce object 4 types needs to be defined, these are named k1, v1, k2 and v2. These types could be either any data type that exist in Encore or a user defined data type. The k1,v1,k2 and v2 types need to match with the Map and the Reduce functions input and output types (see the function parameters in the code snippet bellow). Lastly a hashing function for the k2 type is needed in the initialization of the Mapreduce object.

```
map(k1,v2) -> [(k2,v2)]
reduce(k2,[v2]) -> (k2,v2)
```

To run a Mapreduce job the run method on the Mapreduce object is called with the prepared input, the Map function and the Reduce function as arguments (see code snippet below).

```
var mapReduce = new MapReduce[int,String,String,int](
  stringID)
var result = mapReduce.run(consume input, map, reduce)
```

The return value from the run method is a Big hash table with the result stored as k2 and v2 types. In the Word Count example the result is different words mapped to how many times that word appears in the input text. For example one key could be the string "hello" and the value stored at key would be an integer representing how many times hello occurred in the text. To get an
overview of the result all values could be extracted from the Big hash table by
looping through the Big hash keys generated from the keys() method and by
extracting values with the get(key) method. Next section show a more complex
Map reduce algorithm but the process to run tasks is similar to the Word Count
example in this section.

6.2.2 Example 2: Parallel Breadth-first Search

Parallel breadth-first search is a more complex Mapreduce algorithm compared
to the Word Count example in previous section. A data type to represent nodes
in a graph is needed for this algorithm and a class named node is used. Each
node has an unique id, an distance to the first node, a color and a list of adjacent
nodes. The color is either 1, 2 or 3 representing white, gray and black. White
means that the node is unvisited, gray that the node was in adjacent to an node
that was visited in the last iteration and black mean that the node already has
been visited.

Integers and nodes is used as types when the Mapreduce object is initialized.
The input to the Mapreduce object is a Big hash table which represent a graph,
the Big hash table have integers as keys and nodes as values. The Parallel
breadth-first search algorithm calculates how many edges all nodes in the Graph
are from the starting node. The algorithm contains many iterations of Mapre-
duce tasks. A Mapreduce task is the process from input to output through the
Mapreduce structure. In the Word Count example there was only one iteration
of this process but this algorithm may need many iterations.

During each Mapreduce task we look at all nodes, if they are colored gray
(gray mean we marked it in the last iteration) we color that node black and go
through all of that nodes white neighbours and make them gray for the next
iteration. The process is repeated until all nodes are colored black, when this
is the case we have an solution. The number of task is equal the depth of the
graph because in each task we process all current nodes neighbours.
fun map(key:int, value:Node) : [(int, Node)]
  if value.color == 1 then
    value.color = 0
    var adjList = value.adjList
    var result = new[(int,Node)](|adjList|+1)
    repeat i <- |adjList| do
      var id = adjList(i).id
      var dist = new Node(-1)
      dist.distance = value.distance + 1
      dist.color = 1
      result(i) = (id, dist)
    end
    result(|adjList|) = (key, value)
    result
  else
  [(key, value)]
end

fun reduce(key:int, values:[Node]) : (int, Node)
  var dmin = 100
  var node = new Node(-1)
  var color = 2
  for d <- values do
    if d.id >= 0 then
      node = d
    else
      if d.distance < dmin then
        dmin = d.distance
      end
    end
    if d.color < color then color = d.color end
  end
  if node.distance > dmin then node.distance = dmin end
  node.color = color
  (key, node)
end

The two Mapreduce algorithm implemented and described in this is section was just handpicked examples. There are a lot of Mapreduce algorithms defined that easily could be converted to Encore code so they can be run with this implementation. The following sections describe details of the Mapreduce implementation and how Big hash tables functionality are used in their structure.
6.3 Extending Big Hash Tables

The implementation of Mapreduce was the first time Big hash tables have been used on a larger scale. The best outcome would obviously be that everything Mapreduce needed could be implemented without changing the Big hash tables source files, this however was not the case. This section describes what functionality that was provided by the first design of Big hash tables and what was missing.

6.3.1 Provided Functionality

Automatic Sorting & Distribution

An important step in Mapreduce is to sort keys and distribute data between different working nodes. The nodes in this implementation are Workers in the Big hash table and every level that holds data in Mapreduce consists of a Big hash table (see Figure 6.2). Sorting and distribution will happen automatically when sending a request to input a value into a Big hash table. The Big hash table inserts incoming data by placing all values with the same key at the same hash-entry and because keys are divided among the many Workers the results is that data is distributed. An operation to insert is thus the only method from the big hash tables Mapreduce need to be able to sort and distribute the data for any Mapreduce step.

6.3.2 Extended Functionality

Multiple Values per Key

Instead of having only one value mapped to each key the Mapreduce stages needs to be able to dynamically append more values to keys. This was not a supported features in the first design. At the bottom level each hash-entry was changed to be a link-list instead of only consist of one value, no old functionality was changed but methods to extend and extract many values from one specific hash entry was added.

Interface for specific Instructions

Much of the work done in the Mapreduce steps are done at the lower level of the Big hash table (in the Workers). For Example when the Map and Reduce functions sends massages between each step it’s the Workers that do the calculations and sends these messages.

It was clear that at this stage the functionality needed could not be created without changing the Big hash table source files. Preferable an interface where these kind of function could be inserted are wanted but this was not a priority. The Big hash table was instead extended with one mapper and one reducer method that each handle one step in the Mapreduce implementation.
6.4 Optimizing Big Hash Tables

Until this stage most time had been spent on adding functionality to the Big hash tables rather than maximize performance. During both the process of implementing the Big hash tables and using them in the Mapreduce program ideas how to improve the them came up. These hypothesis sparked discussion and it was decided that this thesis should include a section on optimization. To evaluate where optimizing could have most impact the Mapreduce program described in section 6.2.2 was used with a tool to measure performance.

All tests described in this chapter was done on the same machine, a MacBook Pro (Retina, 13-inch, Mid 2014). The machine has a 2.6 GHz dual-core Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory. The compiler used was Apples LLVM version 9.1.0 on macOS 10.12 operative-system.

6.4.1 Flame Graphs

Flame graphs is a tool to measure a programs performance, its a open source and is created by Brendan Gregg [2]. Flame graphs approximates how long specific functions hold the CPU during the program execution. It achieves this by tracking the stack and taking samples. Flame graphs is used by the programmer to get an overview of where in the program time is spent. In a Flame graph diagram, each box on the y-axis represents one stack-frame and the length along the x-axis represent how long they were active during the execution of the program. Boxes are sorted in alphabetical order along the x-axis [2] and should not be seen as a time-line.

6.4.2 Generated Flame Graphs

The Flame graph (see figure 6.3) is generated from a run of the Mapreduce program with a parallel breadth-first search algorithm. The algorithm finds the shortest path to every node from the starting node in an undirected unweighted graph (see section 6.2.2).

Many graphs was first generated and most of them was very similar even with different numbers of nodes, similar in the sizes means that with a larger input size there are no part of the Big hash table that becomes a large bottleneck, boxes represents more time but they all have increase a similar percentage.

Figure 6.3 show a run of parallel breadth-first on a graph with 50 000 nodes. The Flame graph shows that the map and reduce functions take up a small part of the total running time of the program, 5.44% and 5.45%. This will vary depending on the algorithm but a high percentage was not expected because of all the other things that are done in the Mapreduce process, there are for example hundreds of thousands values inserted into the Big hash tables during this run.
6.4.3 Optimizing Message Sending

For every key-value pair the map function returns a message is sent to the next stage in the Mapreduce program. When using the Big hash table one would sometimes want to batch many pairs together and send them in one message instead of many separate massage, methods for this functionality was added and test used in the Mapreduce map step. All messages now contains an array with many pairs instead of a single pair.

This resulted in a lot less massage sending but the trade off is that all pairs instead needed to be batched together. In the current implementation a link list is created for each Worker, the length of the result from each map function is unknown therefor arrays can be tricky to work with. For every pair the key value is hashed (hashing was also needed before this optimization) and the values are inserted in the correct linked list. Lastly all the values are extracted from the link-list and inserted to an array to later be sent to a Worker. For each batch of pairs only one massage is sent to each Worker. This solution did not increase the speed, it was slightly slower. It is possible that the best solution would be something in between the two implemented solutions, an arrays could be used as an buffer and a messages is sent when it’s full, this however are not further explored in this thesis.

6.4.4 Optimal Number of Workers

The optimal total number of Workers are around 32 for this instance and could be another value for an other machine (specs for this machine in section 6.5). This number will change depending on the input and which algorithm that are used. There are however some interesting conclusions that could be drawn from the data in figure 6.3. When only having one Worker the algorithm is almost twice as slow as when having 20-30 Workers, this must mean that the algorithm benefits from the concurrency that more Worker enables. When more than 32 Workers are used the cost to initialize these object every step becomes larger than the gain from the concurrency.
Figure 6.3: This a flame graph generated from a run of the parallel breadth-first search algorithm with 50000 nodes.
Figure 6.4: Running the Parallel Breadth-first Search algorithm (see section 6.3.2) on a graph with 10000 vertices and where each vertex has at least 10 edges connected to it. The best value of 5 independent runs for every specific number of Workers is picked and showed in the figure.
Chapter 7

Future Work

The work on Big data types done for this thesis was limited to match the requirement for a Bachelor thesis. There are work that could build on what has been discussed and implemented here and in this chapter some of these ideas are discussed.

7.1 Big Data Types as First Class Citizens

During the process of writing of the specification for this bachelor thesis talks about making Big data types first class citizens in Encore was discussed. For Big data types this mean that a lot of syntactic features could be added that would make it more convenient for programmers to use them. The need of this was mostly wanted to make it easier to work with sets of values by treating a set as a single value. For example a Big data type that stores a user created class should be able to call methods on that class directly with method calls on the Big hash table, these methods calls should interact with some or all objects in Big hash table. To get such feature currently the programmer would have to create and anonymous function that is inserted as an argument to a specific method.

Example with no support, raiseSalary() is a method in the Employee class.

```javascript
1 // Creates a Big array object with Employee objects.
2 var AllEmployees = new Bigvar[Employee](employees_list)
3
4 // Call the method raiseSalary and on all Employee objects
5 AllEmployees.apply(fun(e:Employee) => e.raiseSalay(0.04))
```

Example with First Class Citizens support in Encore.

```javascript
1 var AllEmployees = new Bigvar[Employee](employees_list)
2 AllEmployees.raiseSalary(0.04)
```
7.2 Bestow and Atomic

There is more than one way to implement the concepts behind Big data types. During the same time period as this thesis was written another bachelor student was working on implementing Bestow and Atomic constructs into the Encore Language. Atomic lets an actor specify a sequence of messages that another actor need handle without interleaving for other messages. Bestow lets actors safely interact with an internal object through the interface of that object [18]. These new language constructs make it possible for a completely new implementation of Big data types. Comparing this future implementation with the one done in this thesis would be very interesting, mostly in a performance aspect.

7.3 Comparisons

All tests done for this thesis were run on one computer. In the future it would be interesting to run the same test on different machines with different specs and compare the results. Specially one with a higher number of cores to see how well Big data type scaled. Also comparing the performance of this Mapreduce implementation to a another implementation while using the same algorithm would be interesting.
Chapter 8

Conclusion

8.1 Programming in Encore

To work with a programming language that was in early stages of development was something new for me. First it was a bit of a hassle, documentation was not good and I had to learn how to find information in other ways. It was mostly done by looking at test cases for the language or by asking the developers of Encore directly. Encore changed its syntax once during my development but the changes needed in my source code were minor. It was fun to see how the language progressed and changed while working with it and I am overall positive towards programming in Encore. It was easier than expected to get into thus it had many functional features that I recognized, it helped a lot that I knew the actor model from previous work with the Erlang programming language.

8.2 Concluding Remarks

This thesis reports on the design and implementation of Big data types. Big data types are locally distributed data structures that allow internal parallelism in the actor model by using several actors in their implementations. Thus, rather than serializing all interaction these data structures are potentially as parallel as the number of actors used to construct them. We have implemented Big data types solely through constructs that exist in Encore. Thus, Big data types can be provided as a library on top of which programmers can construct programs. As part of our evaluation, we have implemented one such program, a Mapreduce framework which uses a locally distributed hash table.
Bibliography


