NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE STUDIES

UNDERGRADUATE THESIS

Map-Hash-Repeat: A Cloud Programming Pattern for Iterative Computation

Anastasios D. Kalogeropoulos

Supervisors: Yannis Smaragdakis, Associate Professor NKUA
Aggelos Biboudis, PhD Student NKUA

ATHENS

JANUARY 2014

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΠΤΥΧΙΑΚΕΣ ΣΠΟΥΔΕΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Map-Hash-Repeat: Ένα Προγραμματιστικό Πρότυπο για Επαναληπτικούς Κατανεμημένους Υπολογισμούς

Αναστάσιος Δ. Καλογερόπουλος

Επιβλέποντες: Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ
Άγγελος Μπιμπούδης , Διδακτορικός φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2014

UNDERGRADUATE THESIS

Map-Hash-Repeat: A Cloud Programming Pattern for Iterative Computation

Anastasios D. Kalogeropoulos

R.N.: 1115200700061

Supervisor: Yannis Smaragdakis, Associate Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Map-Hash-Repeat: Ένα Προγραμματιστικό Πρότυπο για Επαναληπτικούς Κατανεμημένους Υπολογισμούς

Αναστάσιος Δ. Καλογερόπουλος

Α.Μ.: 1115200700061

Επιβλέπων: Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ

ABSTRACT

In this thesis we describe an initial exploration of the Map-Hash-Repeat pattern of cloud computation, in the context of the F# programming language and the MBrace cloud framework. Map-Hash-Repeat can be viewed as a generalization of the well-known MapReduce pattern, where a) the reduce phase is replaced by “hashing”, i.e., re-distribution of results over the node topology; b) both the map and the reduce phases are repeatedly iterated; and c) the final result is produced (possibly via a full reduce phase) upon reaching an iteration fixpoint.

This pattern seems to capture a general mode of computation, e.g., easily simulating iterative algorithms, in addition to traditional MapReduce functionality.

Our prototype is in a functional setting, using the F# programming language, and leverages an existing framework for cloud computation to encode the Map-Hash-Repeat pattern succinctly.

SUBJECT AREA: Cloud computations, iterative algorithms

KEYWORDS: MapReduce programming model, F#, MBrace cloud programming pattern

ΠΕΡΙΛΗΨΗ

Στα πλαίσια αυτής της πτυχιακής εργασίας περιγράφουμε μια αρχική εξερεύνηση του προτύπου Map-Hash-Repeat για επαναληπτικούς κατανεμημένους υπολογισμούς, με χρήση της προγραμματιστικής γλώσσας F# και του Mbrace framework. Το Map-Hash-Repeat αποτελεί μια γενίκευση του γνωστού προγραμματιστικού προτύπου Map-Reduce, όπου: 1) το reduce στάδιο αντικαθίσταται με διαμοιρασμό των αποτελεσμάτων στην τοπολογία των κόμβων, 2) το map και το reduce στάδιο επαναλαμβάνονται και 3) το τελικό αποτέλεσμα προκύπτει (πιθανώς από ένα reduce στάδιο) όταν φτάσουμε σε κάποιο σταθερό σημείο.

Το πρότυπο φαίνεται να υποστηρίζει ένα γενικότερο τύπο υπολογισμών, για παράδειγμα υλοποιεί απλούς επαναληπτικούς αλγορίθμους σε συνδυασμό με την κλασσική λειτουργικότητα του MapReduce.

Το πρωτότυπο αυτό αποτελεί ένα λειτουργικό περιβάλλον, χρησιμοποιώντας την προγραμματιστική γλώσσα F#, και αξιοποιεί ένα ήδη υπάρχον framework για υπολογισμούς στο δίκτυο ώστε να υλοποιείται συνοπτικά το πρότυπο Map-Hash-Repeat πρότυπο.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Υπολογισμοί στο δίκτυο, επαναληπτικοί αλγόριθμοι

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: προγραμματιστικό μοντέλο MapReduce, F#, προγραμματιστικό μοντέλο MBrace

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my supervisor Mr. Yannis Smaragdakis for suggesting the main idea of my undergraduate thesis. I am also thankful for his support throughout the period that I have been working on this thesis.

Secondly, I would also like to thank Aggelos Biboudis for his contribution and his continued support by providing suggestions and comments not only to the programming part but also to the context of this text in order to take its final form.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

PROLOGUE

This undergraduate thesis has been implemented since April of 2013 in the University of Athens at the department of Informatics and Telecommunications.

1. Introduction

Map-Hash-Repeat is a cloud programming pattern for iterative computations. It evolves the MapReduce programming model and it is used to simulate iterative algorithms in an easy way where both map and reduce phases are iterated and the final result is obtained (via a reduce phase) once we reach a fixpoint. To elaborate this, a map phase takes place at first and then the reduce phase is replaced by a re-distribution of the previously computed results over the node topology. The same procedure goes on until reaching an iteration fixpoint.

The MapReduce programming model is used for processing and generating large data sets in a distributed way over several machines. A MapReduce program consists of a *map* phase which processes some input data and generates ("maps") it to intermediate <key, value> pairs according to the user's specifications and a *reduce* phase that takes a collection of intermediate <key, value> pairs and "reduces" them according to the same intermediate key, in order to produce the final result.

Our prototype implementation of the Map-Hash-Repeat programming pattern is based on the F# programming language in combination with the MBrace cloud computing model. F# is an open-source programming language. Primarily, F# is a functional language, but it also supports imperative, object-oriented and functional styles of programming. Some of the main features of the F# language which are used in our Map-Hash-Repeat implementation are: *code quotations, asynchronous workflows, pattern matching, discriminated unions and recursion*. We also used some of the basic features of the MBrace cloud computing framework which are the *cloud workflows* and the *cloud refs.*

MBrace is a cloud programming model/framework which introduces an expressive and integrated way of performing large-scale computations running in the cloud. With the help of the F# programming language, MBrace offers a declarative style for describing distributed computations using the F# computation expressions which specify parallelism patterns known as *cloud workflows* or *monads* and as a result it express many different kind of algorithmic patterns, such as MapReduce and other iterative algorithms.

2. Background

2.1 MapReduce

MapReduce [1] is a programming model for processing large data sets in a distributed manner over several machines (tasks). It is used to deal with problems that can be seperated into smaller independent sub-problems and require operations on large data sets. Its concept is very simple and the main idea is that the inital problem is divided into sub-problems, which are being distributed over the topology into tasks. There is a *boss* who takes the initial input and distributes it to the *workers.* So, each worker tries to solve a part of the inital problem. When completed, the boss node collects all the solutions from each worker and combines them in some way to form the output, which will be the final result of the initial problem.

In more detail, the MapReduce framework consists of two user-specified basic procedures: Map() and Reduce(). During the "Map" step, the Map() function takes as input a series of <key_i,value_i> pairs and they are "mapped" into a collection of intermediate <ikey_j,ivalue_j> pairs that are being distributed to worker nodes. Then, in the "Reduce" step, each worker performs a computation ("reduce") over the data it received and then passes the result to the boss in a form of <okey_i,ovalue_i> pair. Finally, the boss node collects all data from the workers and produces the final output.

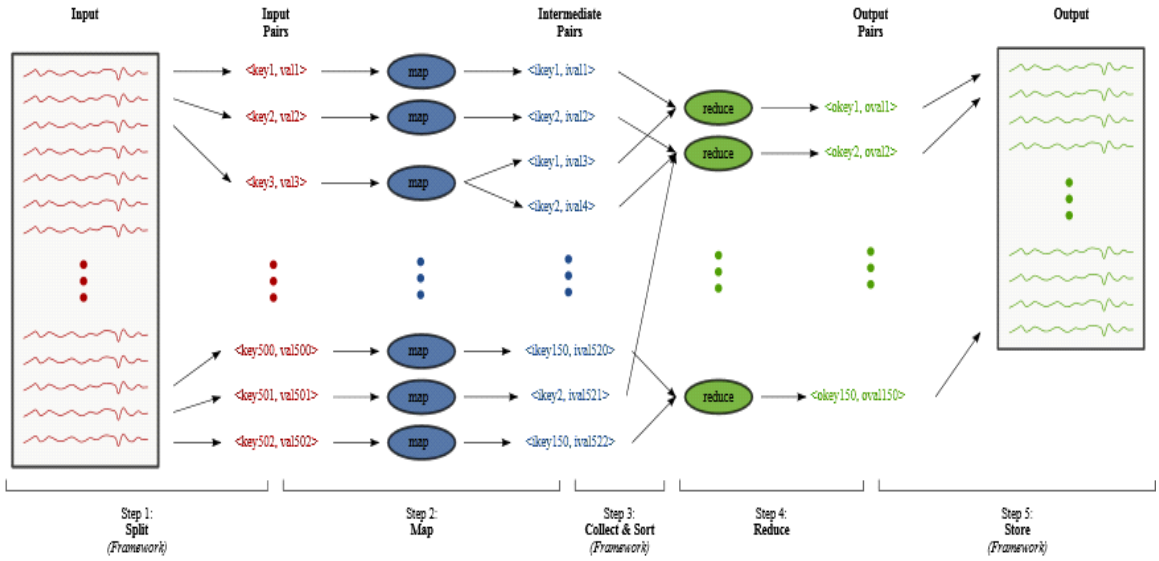

Figure 1: MapReduce process[1]

1 WebMapReduce in Education,

http://webmapreduce.sourceforge.net/education.php

Figure 1 shows the map and the reduce phase. In Step1, the input data is split into key-value pairs and the Map() function takes these pairs as input. The output can be one or more intermediate key-value pairs, as Step 2 shows. In Step3, all intermediate key-value pairs are grouped by key and in the next step each unique key is being given to the reducer including all its values. Then, the reducer performs a computation over its values and returns one or more key-value pairs. In the end, all ouput pairs are collected by the framework. The most important element is that the model is ideal for parallel programming because each key-value pair can be computed independently.

One of the most common examples in MapReduce is the wordcount example. In this example, the goal is to count the number of occurrences of each word in a large set of documents. This problem can be easily implemented using the MapReduce programming model. The *map* function would split the input lines into words and then would emit a <word,1> pair for each word. Then the pairs with the same key are grouped together and are passed to the same worker for the *reduce* step. So, the *reduce* function takes a collection of <key,value> pairs and sums the values of the pairs that have the same key. Finally, the output pairs are in the form of <word,occurences> and are passed to the boss.

2.1.1 MapReduce-based Frameworks

There are many implementations of the MapReduce framework. One of them is the well-known Apache Hadoop [2] project. Hadoop is an open source MapReduce implementation written in Java. It is designed to solve big problems that require computationally extensive operations. These operations can be executed in parallel on a large number of independent machines. Hadoop uses its own distributed file-system called *Hadoop Distributed File System (HDFS)* to provide access to the data from all the servers.

2.1.2 Criticism of MapReduce

Despite the fact that many programmers support the idea of the MapReduce programming model, many people, mainly from the database community, claim that MapReduce represent a major step backwards. To support their view, they argue that MapReduce has a serious performance problem when it comes to access huge amount of data because it uses no indexes. Its techniques are nothing new and they are more than 20 years old. They also assert that MapReduce is missing some main features that are provided by any modern DBMS and it is not compatible with many DBMS tools. However, MapReduce suits well for processing huge amounts of unstructed data and enables data distribution over the network.

2.2 Programming in F#

2.2.1 Quoted Expressions

In F#, many applications need to know not only the structure of a block of source code but also the way it operates. In order to do that, F# provides a specially delimited expression which is called *quoted expression* so that the expression is compiled into an object which represents this expression. With this feature, the programmer can see the abstract syntaxt tree of the code. These quoted expressions can be executed then by alternative means such as an SQL query or as JavaScript in a client-side Web browser. In other words, this mechanism lets the programmer convert a computational representation of the program to an abstract syntax representation of the same language with which you can for instance analyse, execute, print or compile those programs in other ways.

Quoted expressions come in two ways: typed or untyped. To add type information, one can place the quotation markers <@ @> around an expression.

For example, the following code shows a typed quotation:

Table 1: Quoting an expression – typed quotation

```
open Microsoft.Fsharp.Quotations

let expr1 : Expr<int> = <@ 1 + 1 @>

//val expr : Quotations.Expr<int> = Call (None, op_Addition, [Value (1), Value (1)])
```

As you see in the Table 1, the act of quoting an expression gives you back the expression as data. The generic type parameter Expr<int> is the result of the expression <@ 1 + 1 @> which in this example is an integer.

In order to obtain an untyped quotation, one can use the <@@ @@> quotation markers around an expression, as in the following example below:

Table 2: Quoting an expression – untyped quotation

```
open Microsoft.Fsharp.Quotations

let expr2: Expr = <@@ 1 + 1 @@>

//comment

//val expr : Quotations.Expr = Call (None, op_Addition, [Value (1), Value (1)])
```

In Table 2, the result Expr of the expression is not a generic type.

2.2.2 Asynchronous Workflows

Another important feature of the F# programming language are the *asynchronous workflows [5].* With asynchronous workflows F# provides a programming model which enables the programmer to indicate which computations are going to be executed in the background threads in an asynchronous way as the execution continues on the current thread, without blocking it. For instance, background threads can perform different kinds of work such as responding to I/O requests, sleeping or waiting to acquire shared data. Asynchronous workflows avoid the need of explicit callbacks and the user writes the code as if it was executed sequentially.

To create asynchronous workflows, the code is wrapped in the *async* computation expression builder . A computation expression builder is an F# feature which lets the user define the characteristics of his own computation expression by creating a custom builder class and defining its methods. The syntax for defining a computation expression of type *builder-name* is `builder-name { expression }`. This type specifies code that controls the execution of the *expression.*

The result of an *async* computation is an `Async<'T>` object which can be thought of as an asynchronous computation that will compute a value type 'T. In order to begin an asynchronous operation the keywords *let!* (let-bang) and *do!* (do-bang) *a*re being used. Within asynchronous workflow expressions, the language construct `let! var= expr` means “perform the asynchronous operation *expr* and bind the result to *var* when the operation completes. Then continue by executing the rest of the computations”. The difference between *let* and *let! i*s that the first one just stores the result as an asynchronous operation, while the second executes the asynchronous operation and returns the data.

To start an asynchronous workflow, the simplest way is to use the Async.Start method which takes an `Async<unit>` as a parameter in order to begin the execution but in order for the async operation to return a value, the Async.RunSynchronously method is called immediately after the Async.Parallel method.

The Async.Parallel method takes a `seq<Async<'T>>` and starts to execute all the asynchronous computations in parallel, and then the Async.RunSynchronously method is called to initiate the execution of the asynchronous computations.

The following example shows how to use asynchronous workflows to fetch several Web pages in parallel.

Table 3: Fetching Web pages length asynchronously

```
open System.IO
```

```
open System.Net

let webPages = ["google", "http://google.com"; "yahoo", "http://yahoo.com"]
let fetchAsync(name, url: string) =
   async {
      printfn "Request for %s" name
      let req = WebRequest.Create(url)
      let! resp = req.AsyncGetResponse()
      printfn "Response for %s " name
      let stream = resp.GetResponseStream()
      printfn "Reading response for %s " name
      let reader = new StreamReader(stream)
      let html = reader.ReadToEndAsync().Result
      printfn "Read %d characters for %s " html.Length name
      return (name,html.Length)
   }

Async.Parallel [for name, url in webPages -> fetchAsync(name, url)]
|> Async.Ignore
|> Async.RunSynchronously
```

The types of these functions are:

Table 4: Function return types

```
val webPages : (string * string) list

val fetchAsync : name:string * url:string -> Async<string*int>
```

Running the code in F# interactive, the output is:

Table 5: Output fetchAsync method

```
Request for yahoo
Request for google
Response for google
Reading response for google
Read 46800 characters for google
Response for yahoo
Reading response for yahoo
Read 91516 characters for yahoo
```

As one can see in Table 5, there are simultaneous web requests. The most important thing is that the threads which are responsible for these requests are not blocked.

At the example in Table 3, we use the namespaces System.IO and System.Net provided by the .NET framework for reading to data streams and using Internet resources easily. The

webPages function contains a list of tuples and the fetchAsync function takes as input two strings, a name and a url. The output is an Async<string*int> type object. The next three lines perform the execution of the program. The pipeline operator |> is defined as shown in Table 6.

Table 6: Pipeline operator

```
let (|>) x f = f x
```

The above operator it applies the first operand (x) to the function given in the second operand (f).

The Async.Parallel method creates an asynchronous computation and executes all the given fetchAsync computations in a fork/join pattern. This computation is passed to the Async.Ignore metho,using the pipeline operator, to run and ignore its result. Finally, the RunSynchronously method executes in parallel all these sequence computations and awaits the result.

2.2.3 Pattern matching

Pattern matching is a very powerful and flexible feature of the F# programming language that is used to examine data structures and their values against one or more conditions. Instead of using series of *if...then...else* statements, F# offers the pattern matching. In general, patterns are used to compare data with a structure, or to decompose data into its parts or to extract information from structures. They are used in the match expression that has the following form:

Table 7: Match expression used for pattern matching

```
match test-expression with
        | pattern1 [when condition] -> result-expression1
        | pattern2 [when condition] -> result-expression2
        | ...
```

In the form in Table 7, *test-expression* is the expression that will be matched. Most of F#'s types can be used as text-expressions. The *pattern* specifies how to deconstruct the text-expression. The *when condition* is optional. For a successful matching, the pattern must satisfy the test-expression and the when condition must evaluate to true. The *result-expression* is returned in a successful match. The patterns are tested one by one and the procedure stops once a match is found.

A simple example for matching constants is described.

Table 8: Pattern matching example

```
match 5 with
        | 5 -> “Match!”
        | 2 -> “This pattern will never be tested.”
```

In the above example, F# compares the test-expression 5 with the pattern 5 and considers the match as valid. Then, it evaluates the result-expression and returns the string “Match!”. As one can notice, the pattern 2 will never be tested. This means that the order of the patterns matters.

2.2.4 Discriminated unions

Discriminated unions are data types with a finite number of different representations. They can be though of as the union data type in C. The are defined using the keyword *type* followed by a name and then the union cases seperated by the pipe symbol | . Discriminated unions have the following form:

Table 9: Discriminated union type

```
type typeName =
   | Case1 [of datatype1] [* datatype2] ...
   | Case2 [of datatype3] [* datatype4] ...
   | ...
```

For instance, a type is described below, which can me either an integer or a boolean.

Table 10: Example of the discriminated union type intOrBool

```
type intOrBool =
        | Int of int
        | Bool of bool
```

2.2.5 Reference Cells

A ref cell is used to hold a mutable value, meaning that the user can change it anytime. The *ref* operator is used before a value to allocate a new ref cell for the given value. To assign a new value to a reference cell one can use the *assignment operator* := and to read its contents one can use the *! (bang)* operator, as the following example shows.

Table 11: Reference cell example

```
let r = ref 3
printfn “%d” !r
r := 5
printfn “%d” !r
```

In the above example we create a ref cell that contains the value 3 and then we change its value to 5.

To clarify this, reference cells point to a memory address. This means that if one has several ref cells pointing to the same address, changes at that memory address will affect all the ref cells pointing to it.

2.3 Programming in Mbrace

2.3.1 Concepts

The programming model of MBrace [3] is very similar to the form of the F# asynchronous workflows. Just as in asynchronous workflows one uses the syntax `async {expression}` to set up a computation expression that runs asynchronously, MBrace offers *cloud workflows* to introduce distributed computations. Their execution can be performed only within a distributed environment. The framework provides the MBrace *runtime* that provides an environment that abstractly distributes the execution of cloud computations. The environment handles quite good data distribution but when it comes to big data it is not so efficient. MBrace offers a mechanism to manage global data, known as *cloud refs.* In the following section these concepts are described in details.

2.3.2 Cloud Workflows

MBrace provides cloud workflows in order one to define distributed computations. Their style bears a strong resemblance to F# asynchronous workflows. In the same way that asynchronous workflows are scoped by the `async { }` builder and are being executed in

threads, cloud computations use the cloud { } expression builder known as *cloud block* and are being executed in a distributed way. Their execution is postponed until they are being sent to the MBrace runtime for evaluation. The type of a cloud computation is ICloud<'T> and once executed, the output is of type 'T. All cloud blocks must be declared with the [<Cloud>] attribute.

The following example declares a simple computation.

Table 12: Hello World example

```
[<Cloud>]
let HelloWorld() =
   cloud {
      return "Hello world"
   }
```

In Table 12, one can see that this computation returns an ICloud<string> type. This result should be dereferenced in order to obtain the *string* type. To instantiate a local MBrace runtime, the user has to execute the following command:

Table 13: A local MBrace runtime initialization

```
let runtime = Mbrace.InitLocal 4
```

The code in Table 13 initializes a cluster of 4 nodes locally on the current machine.

To execute a cloud computation, one can run the following code. The result is type of string with value “Hello world”.

Table 14: Execution of the HelloWorld cloud computation

```
Runtime.Run <@ HelloWorld() @>
```

We can now define the same example in Table 3 using *cloud blocks* as the following table shows.

Table 15: Fetching Web pages length in parallel

```
[<Cloud>]
let cloudFetchAsync() =
   cloud   {
      let jobs =
         Array.map (Cloud.OfAsync << fetchAsync) webPages
```

```
    let! results = Cloud.Parallel jobs
    return results
}
```

The *Array.map* function applies the given function to each element of the array and returns a new array.

This section explains some basic MBrace combinators that are used in the above example.

- **Cloud.OfAsync**

With this combinator, one can use asynchronous computation expressions inside cloud blocks. Its type signature is:

Table 16: Type signature of Cloud.*OfAsync combinator*

```
Async<'a> → ICloud<'a>
```

The combinator does not change the semantics of the *async* computation. It is useful when one needs to adjust existing asynchronous workflows in cloud workflows. For instance, we can use the *Cloud.OfAsync* combinator to embed the following asynchronous workflow, which causes the execution to stop for *x* milliseconds, in a cloud block.

Table 17: Asynchonous workflow example

```
let sleep x =
   async  {
      do! Async.Sleep x
   }
```

Then, the cloud computation is declared in Table 18.

Table 18: Cloud computation expression with an embeded asynchronous workflow

```
[<Cloud>]
let cloudSleep()=
   cloud {
      do! Cloud.OfAsync (sleep 3000)
   }
```

- **Cloud.Parallel**

The *Cloud.Parallel* combinator is used to execute cloud computations in parallel. The signature of the type is:

Table 19: Type signature of Cloud.Parallel combinator

```
Cloud<'T> [] → Cloud<'T []>
```

As one can see from the above table, the combinator takes as input an array of cloud computations and executes them in parallel. The result is an array containing the result of each computation. It is worth mentioning that each computation is allocated in different worker and is executed in a fork/join pattern. In other words, the boss waits until all the workers terminate and then joins the results to return the final result.

The following example shows the use of the *Cloud.Parallel* combinator.

Table 20: Cloud.Parallel combinator example

```
[<Cloud>]
let parallelExample() =
   cloud {
      let func x y = cloud {return x + y}
      let jobs = [|for i in 1 .. 1000 -> func i (i*i) |]
      let! results = Cloud.Parallel jobs
      return results
   }
```

The *parallelExample* cloud workflow creates 1000 cloud computations that are being executed in parallel using the Cloud.Parallel combinator and finally the results are being collected.

2.3.3 Cloud Refs

Cloud refs are very similar to F#'s *reference cells* but they are *distributed* and *immutable* by design. Once they are being declared, they require a real-time decision by the runtime as far as the locality and the load mechanism is concerned. The MBrace runtime uses some techniques, such as caching local copies to workers, to manage the values of the cloud refs and lessening duplicate copies of repeating data. The type of the cloud refs is ICloudRef<'T>, where 'T is the type of the contained value.

Using the *fetchAsync* function we defined in Table 3, one can create a cloud workflow to get the length of a web page, but this time a reference to the result will be returned that will contain the value instead of the actual value.

Table 21: Using cloud refs to fetch the length of a web page

```
[<Cloud>]
let getRef() =
   cloud   {
      let! jobs = Cloud.OfAsync <| fetchAsync ("google", "http://google.com")
      let! ref = CloudRef.New jobs
      return ref
   }
```

The *CloudRef.New* method creates a new cloud ref that contains the given value. The result of the *getRef* function is of type *ICloudRef<string*int>*, which is a reference to the result. To get the cloud ref, one can run:

Table 22: Receive a cloud ref

```
let cloudRef = runtime.Run <@ getRef() @>
```

In order to get the actual value of the above cloud ref, the programmer has to run the following line:

Table 23: Dereferencing a cloud ref

```
let data = cloudRef.Value
```

In the context of a cloud computation, the user can get the value of a cloud ref using the *CloudRef.Read* method. This method takes a cloud ref ICloudRef<'a> and returns a workflow ICloud<'a> that contains the enclosed value.

However, there is a mutable version of cloud refs; the *MutableCloudRef* type. The difference between the immutable and the mutable version is that a mutable cloud ref can update its contained value and it can be deallocated manually by the user. These imply that the values can not be cached. Mutable cloud refs usually are used to define user-specified data structures or to create synchronization mechanisms like semaphores. The type of the MutableCloudRef is IMutableCloudRef<'T>, where 'T is the type of the contained value.

The following describes some useful functions that are used to manage the mutable cloud refs.

Table 24: Functions that manage mutable cloud refs

Name	Type	Description
MutableCloudRef.New	'a → ICloud<ICloudRef<'a>>	Takes a value and creates a

		new MutableCloudRef.
MutableCloudRef.Read	IMutableCloudRef<'a> → ICloud<'a>	Returns the value of the given MutableCloudRef.
MutableCloudRef.Force	IMutableCloudRef<'a>*'a → ICloud<unit>	It tries to update the value of the given MutableCloudRef with the given value without checking if it was updated by someone else.
MutableCloudRef.Set	IMutableCloudRef<'a>*'a → ICloud<bool>	It tries to update the given MutableCloudRef with the given value and returns true if on success. If its value has been modified since the last time that it was read it returns false.
MutableCloudRef.SpinSet	IMutableCloudRef<'a>*('a → 'a) → ICloud<unit>	It updates a MutableCloudRef using the function given in the second argument.

2.3.4 MapReduce in Mbrace

The MBrace framework provides a library that one can define and execute MapReduce tasks. The recursive mapReduce method is very simple. It takes a user-specified *map* and *reduce* function, an identity value to be used as a termination condition and a list with the input values. The algorithm splits, if needed, the input into halves and passes each half in two *mapReduce* calls and executes them in parallel.

Table 25: mapReduce library in MBrace

```
[<Cloud>]
let rec mapReduce (map : 'T -> ICloud<'R>)
              (reduce : 'R -> 'R -> ICloud<'R>)
              (identity : 'R)
              (input : 'T list)   =
   cloud  {
      match input with
         | [] -> return identity
         | [value] -> return! map value
         | _ ->
            let left,right = List.split input
            let! r1,r2 =
               (mapReduce map reduce identity left)
                  <||>
```

```
            (mapReduce map reduce identity right)
        return! reduce r1 r2
}
```

In the above code, the *List.split* method is an Mbrace function that takes a list and splits it in half, returning the two new lists in a tuple. The *parallel decomposition operator* <||> is an abbreviation of the *Cloud.Parallel* combinator but only for a 2-dimensional array as input.

3. Map-Hash-Repeat

3.1 Overview

Map-Hash-Repeat is a programming pattern for supporting iterative computations in the cloud. Its main idea is based on the MapReduce programming model but there are differences in how the *reduce* phase is defined as well as in the computation of the final output. Map-Hash-Repeat requires an initial node topology in memory to apply the pattern. This limitation is necessary because the algorithm needs an initial input to generate the final solution in the next iterations. Once this has been done, the algorithm executes iteratively two main steps: the first being a map step that is similar to MapReduce's map and a second being the reduce step defined in a different manner. During the map phase, intermediate values are computed in one step. The reduce phase that follows is different from that in MapReduce: instead of *reducing* these values, these are being distributed over the topology. The concept is that during the n^{th} iteration the algorithm uses the values computed during the previous iteration to compare them with the new ones. The algorithm terminates once in all nodes the previous computed value and the new one are the same.

This implies that each node has to keep the value it computed in the previous iteration. Once all nodes terminate, the iteration stops and the output is produced. In the meantime, the node topology can change appropriately according to the newly computed values. The importance of this redistribution is the fact that the new topology is based on the newly computed values, which helps the algorithm to converge faster.

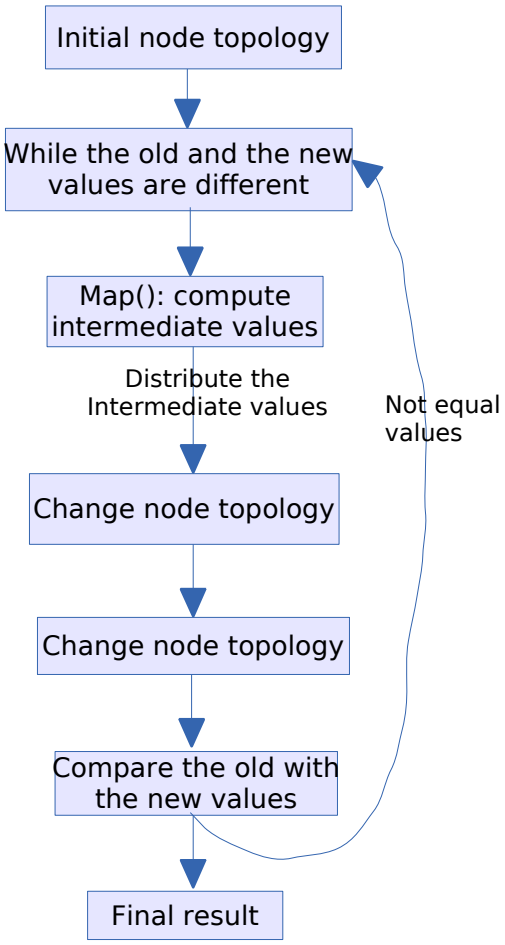

Figure 2: Iterative Map-Hash-Repeat programming model

3.1.1 Description of the algorithm

The following algorithm describes the Map-Hash-Repeat programming pattern.

Table 26: MapHashRepeat pseudocode

```
computeValues := computes the new values that will be used for the next iterations
createNewTopology:= creates a new topology according to the already computed values
keepPrevState := swap the prevValue with the newValue

function recursive mapHashRepeat(input1, input2) :
        newValues := computeValues(input1,input2);
        for each node n in input2:
                n.Values := add(newValues[n]);

        newTopology := createNewTopology(input1,input2);

//returns an array of boolean values, one from each node, on whether
//the previous value equals the new one or not
        states :=
                [|
                        for each node n in input2:
                                prevValue := n.getPrev;
                                newValue := newTopology[n];
                                keepPrevState(prevValue,newValue,n);
                                if newValue == prevValue:
                                        return true;
                                else:
                                        return false;
                |]

        finalState := true;
        for each state s in states:
                finalState := state && true;

        if finalState == true:
                return (input1,input2);
        else:
                mapHashRepeat(input1,input2);
```

3.1.2 Programming Interface

To simulate an iterative algorithm in the Map-Hash-Repeat programming model one should define some user specified functions that are used during the execution of the main algorithm.

The following table describes these functions.

Table 27: Map-Hash-Repeat user specified functions

	Name	Type	Description
1.	createNodes	int->int → ICloud<IMutableCloudRef<Node<int,(float * float),Set<int>>> []>	This function is used to create and distribute the initial data to the MBrace workers. The user needs to pass as arguments the number of the initial data and the number of the clusters. With the help of the *cloud refs*, offered by the MBrace programming model, this function returns a cloud computation of an array that contains references of the data and the center nodes. Dereferencing these computations, one can get the values of the data and the center nodes. Each node is of type Node.
2.	createNeighbors	IMutableCloudRef<Node<int, (float*float),'oldV>>[] -> ICloud<IMutableCloudRef<Node<int,(float *float),'oldV>> [] *IMutableCloudRef<Node<int,(float * float),'oldV>> []>	This function takes as an input the cloud references of two arrays of type Node. The goal is to create an initial topology. The output is a tuple in a cloud computation which contains the cloud references of both inputs.
3.	compute	IMutableCloudRef<Node<int,(float * float),'oldV>> [] → IMutableCloudRef<Node<int,(float *float),'oldV>> [] → ICloud<unit []>	This function is used to calculate the computation that will be used in the future iterations of the algorithm. It takes as parameters the cloud references of t two arrays of type Node.
4.	computeNeighbors	IMutableCloudRef<Node<'Id,(float*float),'oldV>> [] → IMutableCloudRef<Node<'Id,(float * float),'oldV>> [] → ICloud<Map<'Id,Set<int>>>	This function takes as input two arrays of type Node and calculates the new topology in the future iterations. It returns a mapping specifying how the new topology is defined.

5.	isDone	IMutableCloudRef<Node <'Id,'newV,'oldV>> → Set<'Id> → ICloud<bool>	This function is used to check whether we reach a fixpoint or not. It takes as input a cloud reference of type Node and a set of values and checks if the previous computed values are the same with the old ones and returns a boolean value.

3.2 Examples

3.2.1 K-means

K-means is an algorithm which is used to identify the best way to partition a specific dataset into *k* clusters in which the distance between each member of the cluster and the cluster's centroid is the minimum. The user specifies a set of input data and the number *k* of the clusters to be created. Each element in the input data can be a d-dimensional array. The main idea is that k initial centroids are randomly defined and then k clusters are being created. Then, each element in the dataset is assigned to the cluster with the closest centroid. For each cluster, the algorithm recalculates the centroids and all the elements are reassigned to the clusters depending on the new centroid values. The same procedure is repeated until the elements can no longer change clusters.

K-means always converges in polynomial time but it is important to mention that it does not always find the optimal solution. Also, despite the fact that K-means clustering is an NP-hard problem, there are heuristic methods which can be used in order for the algorithm to converge in polynomial time.

3.2.2 Map-Hash-Repeat implementation of K-means

The Map-Hash-Repeat cloud programming pattern can simulate easily iterative algorithms, one of them is K-means, which was described above. In our example, for the sake of simplicity, 2-dimensional points are used for the implementation of the K-means algorithm using the Map-Hash-Repeat pattern. The basic idea is to distribute the elements from the dataset to MBrace workers and then execute the K-means algorithm in order to define the final clusters. In this section the K-means algorithm is described in terms of Map-Hash-Repeat terminology:

The dataset is a generic data type and is defined as a discriminated union Node which can have the value N with the following set of fields:

- a node identifier 'Id
- a type parameter 'value1 which will keep the coordinates of the points
- another type parameter 'value2 which keeps the set of nodes that were in the cluster in

the previous iteration in order to be checked with the new one and determine if it has been changed

- a set of identifiers Set<'Id> which keeps the identifiers of the nodes that are closest to the center node

As mentioned, In order to enable execution specific to K-means, one should define the functions in the Table 27. These functions are described in Table 28.

Table 28: Map-Hash-Repeat user specified functions specially for K-means

	Name	Type	Description
1.	createNodes	int->int → ICloud<IMutableCloudRef<Node<int,(float * float),Set<int>>> []>	The user needs to pass as arguments the number of the initial data and the number of the clusters. This function returns a cloud computation of an array that contains references of the data and the center nodes. Dereferencing these computations, one can get the values of the data and the center nodes. Each node is of type Node. The first field is a unique identifier. In order to distinguish the data nodes from the centers, the identifiers of the latter are negative numbers. The second field is a tuple of two floats that specify the coordinates of the corresponding node. The last two fields are both empty sets.
2.	createNeighbors	IMutableCloudRef<Node<int, (float*float),'oldV>>[] -> ICloud<IMutableCloudRef<Node<int,(float *float),'oldV>> [] *IMutableCloudRef<Node<int,(float * float),'oldV>> []>	This function takes as input the cloud references of the data nodes and the center nodes. Firstly, it seperates the data from the centers of the first input parameter and then it calculates the euclidean distance between each data element and each center. After finding the closest center node for each data node, the function updates the values of the centroids by adding the identifier of the closest data node in the set of the identifiers of the corresponding center. The output is a tuple in a cloud computation which contains the cloud references of both data nodes and the updated centers.
3.	compute	IMutableCloudRef<Node<int,(float * float),'oldV>> [] → IMutableCloudRef<Node<int,(float	This function takes as parameters the cloud references of the data nodes and the centers. Then, for every center, it takes the coordinates of the nodes who are closerst to it and calculates the new centroids which will be the

		*float),'oldV>> [] → ICloud<unit []>	coordinates of the centers of the new clusters. Afterwards, it replaces the previous coordinates of the centers with the new ones.
4.	computeNeighbors	IMutableCloudRef<Node<'Id,(float*float),'oldV>> [] → IMutableCloudRef<Node<'Id,(float * float),'oldV>> [] → ICloud<Map<'Id,Set<int>>>	This function takes as input parameters the cloud references of the data nodes and the centers and calculates the euclidean distance between each data node from all the centers. From all these calculations, the function gets the one with the minimum value and associates the data node to the closest center. Finally, it returns a mapping of each center identifier as a key and a set of identifiers of the closest data nodes to this specific center as a value.
5.	isDone	IMutableCloudRef<Node<'Id,'newV,'oldV>> → Set<'Id> → ICloud<bool>	This function checks whether the cluster has been changed or not. It takes as input a center node and a set of identifiers. These identifiers belong to the data nodes which are closest to the given center. If the current set of the identifiers of the center is the same as the given set, then the function returns true, otherwise false.

Once one has defined the previous functions, the recursive “mapHashRepeat” function is invoked by the Map-Hash-Repeat framework which takes cloud references of the data nodes and the centers alongside with the functions 3,4,5 from Table 28 as input parameters in order to execute the internal algorithm. In every iteration the algorithm calculates the new centroids using the “compute” function and then assigns each node the closest center using the “computeNeighbors” function. In the next step for each center the algorithm checks whether the new set of the identifiers of the closest nodes which it calculated before is the same as the old one or not. If the two sets are the same then a boolean value “true” is stored in an array, otherwise a boolean value “false” is stored. Finally, the array contains as many boolean value as the number of the center nodes and if all these values are true it means that the clusters no longer change and the algorithm terminates returning the data nodes and the center nodes. On the other hand, if even one of the values is false the “mapHashRepeat” function is called with the new computations and the same procedure is followed.

In Appendix I, one can see the full implementation of the K-means algorithm in the Map-Hash-Repeat pattern.

3.3 Implementation

Our implementation of the Map-Hash-Repeat programming model is based on the F# programming language and the Mbrace cloud framework. The main reason why we chose

this programming language is because the Mbrace framework is written in F# and takes advantage of many of the language features. Also, many common programming tasks, like list processing, are much more simpler in F#. The following table shows the implementation of the Map-Hash-Repeat programming model in F#.

Table 29: Map-Hash-Repeat programming model

```
[<Cloud>]
let rec mapHashRepeat (input1 : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])
                      (input2 : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])
                      (computeValues : (IMutableCloudRef<Node<'Id,'newV,'oldV>> [] ->
                                        IMutableCloudRef<Node<'Id,'newV,'oldV>> [] ->
                                        ICloud<Map<'I,'C>>))
                      createNewTopology
                      isDone  = cloud {

   let keepPrevState (node : IMutableCloudRef<Node<'Id,'newV,'oldV>> ) newV = cloud {
      let! cloudNode = MutableCloudRef.Read(node)
      match cloudNode with
         | N(v1,v2,oldSet,currentV)  ->
            let newData = (v1,v2,currentV,newV)
            do! MutableCloudRef.Force(node,N(newData))
   }

   let! newValues = computeValues input1 input2


   let distribute =
      Array.map (fun input -> cloud {
         let! cloudInput = MutableCloudRef.Read(input)
         match cloudInput with
            | N(v1,v2,oldV,newV)  ->
               do! MutableCloudRef.Force(input,N(v1,newValues.[v1],oldV,newV))
      }) input2

   let! _ = distribute |> Cloud.Parallel

   let! newTopology = createNewTopology input1 input2
   let checkAll =
      [|
         for input in input2 -> cloud {
            let! cloudInput = MutableCloudRef.Read(input)
            match cloudInput with
               | N(v,_,_,_) when Map.containsKey (v) newTopology ->
                  let comp = newTopology.[v]
                  let! ok = isDone input comp
                  do! keepPrevState input comp
                  return ok
         }
      |]

   let! check = Cloud.Parallel checkAll
```

```
    let fixpoint = check |> Seq.fold (fun acc item -> acc && item) true
    match fixpoint with
        | true -> return (input1,input2)
        | false ->
            return! mapHashRepeat input1 input2 computeValues createNewTopology isDone
}
```

The programming model requires an initial node topology to apply the above function. The algorithm applies the user-specified computeValues function to the input lists and then distributes in parallel the new values to the second input list each time to the appropriate element. A new node topology is created using the user-specified createNewTopology function according to these new values. Afterwards, each element from the second input is checked in parallel to determine whether its value computed in the n^{th-1} iteration is the same as the new value, or not. An array of boolean values is created that contains one value for each element from the input2. This value is true if the previous value equals the new one, otherwise is false. At the same time, the new value is stored in the element for future iterations. Finally, the array is “reduced” by applying the boolean *and operation* to each element and returns an accumulator of the results. If the value of the accumulator is true, it means that we reached a fixpoint. Otherwise, the next iteration begins by calling the again the *mapHashRepeat* function.

The seq.fold function is used to determine whether we reached a fixpoint or not. It takes a function, an accumulator and a sequence of elements and applies the given function to each element while keeping each time the result in an accumulator. Its type is declared below.

Table 30: Seq.fold method

```
Seq.fold : ('State -> 'T -> 'State) -> 'State -> seq<'T> -> 'State
```

3.4 Related Work

There are many improved implementations based on the well-known MapReduce programming model but not many of them support iterative computations in the cloud. *Twister [4]* is a MapReduce runtime, implementing a programming model that supports iterative MapReduce computations in an efficient way. *Twister* separates the data into two types: static and dynamic. Static data is used in every iteration and remains the same in each computation while the dynamic data are the output of every iteration and in many algorithms are used in the next iteration. Instead of reloading the static data in each iteration, Twister introduces a new “configure” step to load the static data to the map/reduce tasks using long running (cacheable) map/reduce tasks that exist during each computation. There is also an optional reduction phase known as “combine”, which one can use to combine the results from all the reduce tasks in a single value.

4. Conclusion and Future Work

In this thesis we introduced the Map-Hash-Repeat cloud programming pattern for iterative computations, which is relevant to the well-known MapReduce pattern. Our implementation is based on the F# programming language and the MBrace programming model. We have discussed the basic concepts of the MapReduce programming model and the main features of the F# programming language that were used in our prototype in combination with some of the features of the Mbrace framework. We have also presented in details the Map-Hash-Repeat algorithm and its concepts and then we discussed the K-means algorithm and its implementation using the Map-Hash-Repeat programming pattern.

In the future, we plan to create a library that will provide the support to develop more iterative algorithms using the Map-Hash-Repeat programming model.

Abbreviations

HDFS	Hadoop Distributed File System
SQL	Structured Query Language
DBMS	Database Management System

Appendix I

Here we implement the K-means algorithm using the Map-Hash-Repeat programming pattern.

```
type Node<'Id,'newV,'oldV
          when 'Id : comparison and 'oldV : comparison> =
                                    | N of 'Id*'newV*'oldV* Set<'Id>

[<Cloud>]
let rec seqMap (f : 'T -> ICloud<'S>) (inputs : 'T list) : ICloud<'S list> =
    cloud {
        match inputs with
        | [] -> return []
        | x :: xs ->
            let! v = f x
            let! vs = seqMap f xs
            return v :: vs
    }



[<Cloud>]
let createNodes (numData : int) (k : int) = cloud {
    let rnd = System.Random()
    let! initVals = [|
                      for i in 1..numData ->
                          MutableCloudRef.New(N((1,(rnd.Next(0,11) |> float,rnd.Next(0,11) |> float),(Set.empty : int
Set),Set.empty)))
                  |]
                  |> Cloud.Parallel
    let! initCenters =
        [|for i in 1..k ->
            MutableCloudRef.New(N((-i,(rnd.Next(0,11) |> float,rnd.Next(0,11) |> float),(Set.empty : int
Set),Set.empty)))
        |]
        |> Cloud.Parallel
    return Array.append initVals initCenters
}

[<Cloud>]
let createNeighbors (nodes : IMutableCloudRef<Node<'Id,'newV,'oldV>> []) = cloud   {

    //find the centers (nodes with id < 0)
    let! getCenters =
        [|for node in nodes -> cloud {
            let! cloudNode = MutableCloudRef.Read(node)
            match cloudNode with
                | N(id,_,_,_) when id < 0->
                    return Some node
                | N(id,_,_,_) ->
                    return None
        }|]
        |> Cloud.Parallel
```

```
//centers
let centers =
    getCenters
    |> Array.choose (fun x -> match x with
                              |Some(ref) -> Some ref

                              | None -> None)
let! getNodes =
    [|for node in nodes -> cloud {
        let! cloudNode = MutableCloudRef.Read(node)
        match cloudNode with
            | N(id,_,_,_) when id > 0->
                return Some node
            | N(id,_,_,_) ->
                return None
    }|]
    |> Cloud.Parallel


let dataNodes =
    getNodes
    |> Array.choose (fun x -> match x with
                              |Some(ref) -> Some ref
                              | None -> None)
//distance
let dist (x1, y1) (x2, y2) : float =
    let xDistance = x1 - x2
    let yDistance = y1 - y2
    xDistance * xDistance + yDistance * yDistance

//get the coordinates of the centers
let! centerCoords =
    [|for center in centers -> cloud {
            let! cloudNode = MutableCloudRef.Read(center)
            match cloudNode with
                | N(id,coords,_,_) ->
                    return (id,coords)
    } |]
    |> Cloud.Parallel

//calculate distances between each node and each center and return (centerId,minDistance) pairs
let! minPairs =
    [| for node in dataNodes -> cloud {
        let! cloudNode = MutableCloudRef.Read(node)
        match cloudNode with
            | N(_,coord,_,_) ->
                return (Array.map (fun (id,c) -> (id,dist c coord)) centerCoords) |> Array.minBy snd
    }
    |]
    |> Cloud.Parallel

//create clusters
let addN pairMap = cloud {
    match pairMap with
        | (nodeId,clusterId) ->
            for center in centers do
                let! cloudNode = MutableCloudRef.Read(center)
                match cloudNode with
                    | N(id,newv,oldSet,newSet) when id = clusterId ->
```

```
                    do! MutableCloudRef.Force(center,N(id,newv,oldSet,newSet.Add(nodeId)))
                | N(id,newv,oldSet,newSet) ->
                    do! MutableCloudRef.Force(center,N(id,newv,oldSet,newSet))
    }
    let! _ =
        minPairs
        |> Array.mapi (fun i (clusterId,minDist) -> (i+1,clusterId))    //node i will be added to cluster with id x
        |> Array.toList
        |> seqMap addN

    return (dataNodes,centers)
}

//compute the new centers
[<Cloud>]
let compute (nodes : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])
          (centers : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])  = cloud {

    let newVals (center : IMutableCloudRef<Node<'Id,'newV,'oldV>> ) = cloud {
        //get the refs of the nodes who belong to the center
        let getNeighbors (center : IMutableCloudRef<Node<'Id,'newV,'oldV>> ) = cloud {
            let! cloudNode = MutableCloudRef.Read(center)
            match cloudNode with
                | N(_,_,_,setN) ->
                    let neighborIds = Set.toList setN
                    return [for n in neighborIds -> nodes.[n-1]]
        }
        //get the coordinates (2nd element) of the given node
        let getCoords (node : IMutableCloudRef<Node<'Id,'newV,'oldV>> ) = cloud {
                let! cloudNode = MutableCloudRef.Read(node)
                match cloudNode with
                    | N(_,coords,_,_)  ->
                        return coords
        }
        //get neighbors REFS from the given center
        let! neighborRefs = getNeighbors center

        //concatenate given center's coordinates with the neighbors' coordinates and return a list with tuples:
(nodeId,coordinates)
        let! cloudNode = MutableCloudRef.Read(center)
        match cloudNode with
            | N(id,_,_,_) ->
                let! clusterCoords = seqMap getCoords neighborRefs
                return (id, clusterCoords)
    }

    //for each cluster, returns a list with the coordinates which will calculate the new center
    let! allCoords =
        [|for center in centers -> newVals center|]
        |> Cloud.Parallel

    //calculates the x coordinate for each new center
    let xs =
        Array.map (fun (id,coords) -> let xSum = List.fold (fun acc (x,y) -> acc + x) 0.0 coords
                                      xSum/(coords.Length |> float)) allCoords
    //calculates the y coordinate for each new center
    let ys =
        Array.map (fun (id,coords) -> let ySum =  List.fold (fun acc (x,y) -> acc + y) 0.0 coords
                                      ySum/(coords.Length |> float)) allCoords
```

```
    let newCenters =
        Array.mapi2 (fun i x y  -> (-(i+1),(x,y))) xs ys
        |> Map.ofArray

   return newCenters
}

[<Cloud>]
let computeNeighbors (dataNodes : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])
                     (centers : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])    = cloud {

    let dist (x1, y1) (x2, y2) : float =
            let xDistance = x1 - x2
            let yDistance = y1 - y2
            xDistance * xDistance + yDistance * yDistance

    //get the coordinates of the centers
    let! centerCoords =
        [|for center in centers -> cloud {
                let! cloudNode = MutableCloudRef.Read(center)
                match cloudNode with
                    | N(id,coords,_,_) ->
                        return (id,coords)
        } |]
        |> Cloud.Parallel

    //calculate distances between each node and each center and returns (centerId,minDistance) pairs
    //first pair is for node with id = 1, second pair for node with id = 2 etc.
    let! minPairs =
        [| for node in dataNodes -> cloud {
            let! cloudNode = MutableCloudRef.Read(node)
            match cloudNode with
                | N(_,coord,_,_) ->
                    return (Array.map (fun (id,c) -> (id,dist c coord)) centerCoords) |> Array.minBy snd
        }
        |]
        |> Cloud.Parallel

    //returns a list with (cId,nId) which means that
    //node with id nId belongs to cluster with id cId
    return
        minPairs
        |> Array.mapi (fun i (cId,minDist) -> (i+1,cId) )
        |> Seq.groupBy snd |> Seq.map (fun (cId,seqnIds) -> (cId, (Seq.map (fun (nid,cid) -> nid) seqnIds) |>
Set.ofSeq ))
        |> Seq.toArray
        |> Map.ofArray
}

[<Cloud>]
let rec mapHashRepeat (dataNodes : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])
                      (centers : IMutableCloudRef<Node<'Id,'newV,'oldV>> [])
                      (calcCenters : (IMutableCloudRef<Node<'Id,'newV,'oldV>> [] ->
                                     IMutableCloudRef<Node<'Id,'newV,'oldV>> [] ->
                                     ICloud<Map<'I,'C>>))
                      computeNeighbors
                      isDone  = cloud {
```

```
    //change the old set of ids(neighbors) with the new one (comp)
    let changeV (node : IMutableCloudRef<Node<'Id,'newV,'oldV>> ) comp = cloud {
        let! cloudNode = MutableCloudRef.Read(node)
        match cloudNode with
            | N(id,coords,oldSet,currentV)  ->
                let newData = (id,coords,currentV,comp)
                do! MutableCloudRef.Force(node,N(newData))
    }

    let! newCenters = calcCenters dataNodes centers

    let distribute =
        [|for center in centers -> cloud {
            let! cloudNode = MutableCloudRef.Read(center)
            match cloudNode with
                | N(id,coords,oldSet,newSet)  ->
                    do! MutableCloudRef.Force(center,N(id,newCenters.[id],oldSet,newSet))
        }
        |]
        |> Cloud.Parallel

    let! neighborPairs = computeNeighbors dataNodes centers

    let checkAll =
        [|
            for node in centers -> cloud {
                let! cloudNode = MutableCloudRef.Read(node)
                match cloudNode with
                    | N(id,_,_,_) when Map.containsKey (id) neighborPairs ->
                        let comp = neighborPairs.[id]
                        let! ok = isDone node comp
                        do! changeV node comp
                        return ok
            }
        |]

    let! check = Cloud.Parallel checkAll
    let ok = check |> Seq.fold (fun acc item -> acc && item) true
    match ok with
        | true -> return (dataNodes,centers)
        | false ->
            return! mapHashRepeat dataNodes centers calcCenters computeNeighbors isDone //(ref true)

}

[<Cloud>]
let isDone (node : IMutableCloudRef<Node<'Id,'newV,'oldV>> ) comp = cloud    {
    let! cloudNode = MutableCloudRef.Read(node)
    match cloudNode with
        | N(id, coords, setN,currentSet) -> return currentSet = comp
}

let runtime = MBrace.InitLocal 4
//number of data points, number of clusters
let allNodes = runtime.Run <@ createNodes 7 2 @>
let (nodes,centers) = runtime.Run <@ createNeighbors allNodes @>

let (finalNodes,finalCenters) = runtime.Run <@ mapHashRepeat nodes centers compute computeNeighbors
isDone  @>
```

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113].

[2] Apache Hadoop. http://hadoop.apache.org/

[3] Dzik, Jan, et al. "MBrace: cloud computing with monads." Proceedings of the Seventh Workshop on Programming Languages and Operating Systems. ACM, 2013.

[4] Ekanayake, Jaliya, et al. "Twister: a runtime for iterative mapreduce." Proceedings of the 19th ACM #International Symposium on High Performance Distributed Computing. ACM, 2010.

[5] Syme, Don, Tomas Petricek, and Dmitry Lomov. "The F# asynchronous programming model." Practical Aspects of Declarative Languages. Springer Berlin Heidelberg, 2011. 175-189.

[6] http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html (accessed 18/1/2014)

[7] F# Language Reference, http://msdn.microsoft.com/en-us/library/dd233181.aspx (accessed 18/1/2014)