MBrace: Cloud Computing with Monads

Jan Dzik Nick Palladinos Konstantinos Rontogiannis Eirik Tsarpalis Nikolaos Vathis
Nessos Information Technologies, SA
{jdzik,npal,krontogiannis,eirik,nvathis}@nessos.gr

Abstract

As cloud computing and big data gain prominence in today's economic landscape, the challenge of effectively articulating complex algorithms in distributed environments becomes ever more important. In this paper we describe MBrace; a novel programming model/framework for performing large scale computation in the cloud. Based on the `.NET` software stack, it utilizes the power of the F# programming language. MBrace introduces a declarative style for specifying and composing parallelism patterns, in what is known as *cloud workflows* or a *cloud monad*. MBrace is also a distributed execution runtime that handles orchestration of cloud workflows in the data centre.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed Programming

Keywords cloud monad, distributed programming, big data

1. Introduction

We live in the era of big data and cloud computing. Massive volumes of unstructured data are constantly collected and stored in huge data centres around the world. Data scientists and computer engineers face the formidable task of managing and analysing huge volumes of data in a massively distributed setting, where thousands of machines spend hours or even days executing highly sophisticated algorithms. Programming large-scale distributed systems is a notoriously difficult task that requires expert programmers orchestrating concurrent processes in a setting where hardware/software failures are incessantly commonplace. The key to success in such scenaria is choosing a distribution framework that provides the correct programming abstractions and automates handling of scalability and fault tolerance.

Several programming models have been proposed and many interesting implementations are currently under development: the most widespread paradigm is undoubtedly *MapReduce*, introduced by Google [1] and currently enjoying success in open source implementations such as *Hadoop*; frameworks such as *Akka* [2] offer distribution through the actor abstraction; finally, new programming models like *CloudHaskell* [3] and *HdpH* [4] have recently been proposed by the Haskell community (see section 6).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLOS'13, November 03 - 06, 2013, Farmington, PA, USA.
Copyright © 2013 ACM 978-1-4503-2460-1/13/11. . . $15.00.
http://dx.doi.org/10.1145/2525528.2525531

One of the main criticisms of the MapReduce model and Hadoop in particular, is its reliance on batch processing and linear pipelining of jobs for certain types of execution patterns: streaming [5], iterative [6] and incremental [7] algorithms all require programming models strictly more expressive than MapReduce.

MBrace introduces a novel programming model and execution framework for the cloud that offers a compositional and declarative approach to describing distributed computations. This is achieved with the help of F#'s *computation expressions* that enable fluent, language-integrated *cloud workflows*, in a model also known as a *monad*. Concurrency patterns and overall execution semantics are specified with the help of primitive combinators that operate on such workflows.

```
cloud {
    let job1 = cloud { return 1 }
    let job2 = cloud { return 2 }
    let! [| result1 ; result2 |] =
        Cloud.Parallel [| job1 ; job2 |]
    return result1 + result2
}
```

The above F# snippet defines two small cloud workflows, `job1` and `job2` which are then passed to the `Cloud.Parallel` combinator. This declares that the jobs are to be executed in a distributed fork/join pattern. The workflow further specifies that once both jobs have completed, the parent workflow will resume its computation and return the final result.

Cloud workflows denote deferred computation; their execution can only be performed within the context of a distributed environment, such as the runtime that the MBrace framework provides. The *MBrace runtime* is a scalable cluster infrastructure that enables distributed abstract machine execution for cloud workflows. The runtime interprets monadic structure of workflows using a scheduler/worker hierarchy, transparently allocating computational resources to pending cloud jobs.

For instance, in the cloud workflow declared above, the forked jobs will be scheduled in two separate worker machines transparently designated by the runtime. Once both have completed, computation will resume in another allocated worker, potentially not the same as the one where the forking was originally initiated. What this means is that cloud workflows are executed in a *non-blocking* manner, in which computation state is liberally suspended, transferred and resumed from one machine to another.

The programming model and runtime form the basic components of a wider cloud computing solution provided by MBrace, which also features rich client tooling, cloud workflow libraries, an interactive shell (REPL) and IDE integration [8].

In this paper we present features of the MBrace programming model that involve two fundamental concepts: *distributed computation* and *distributed data*. In section 2 we present the abstractions and combinators that actuate distributed computation completed with examples and an informal description of semantics. We continue with distributed data (section 3), explaining the use and

management of distributed data entities. In section 4 we present an overview of the MBrace runtime. Section 5 offers performance and scalability benchmarks on the MBrace framework compared with Hadoop. In section 6, we give a brief overview of related work and finally, in section 7, we discuss conclusions and future work.

2. Cloud Workflows

Cloud workflows form the essential pillar of MBrace; the programming model they introduce provides the ability to declare abstract and modal expressions in a fluent, integrated manner, to be subsequently executed in the cloud.

Cloud workflows are made possible thanks to a design pattern known as the *monad*. In short, monads are a language feature that allow the definition of language-integrated DSLs in a way where semantic peculiarities are abstracted away from the syntactic interface. Monads have known success in languages such as `Haskell`, `Scala` and `Clojure`.

In F#, monads are manifested in a feature called Computation Expressions [9], which offer a range of overloadable language patterns such as list comprehensions, queries, exception handling and even constructs traditionally associated with imperative programming, such as `for` and `while` loops. What makes computation expressions interesting is the fact that instances can be declared in library code, overloading language constructs without the need to tweak the F# compiler [10].

2.1 A Prelude: F# Async Workflows

A characteristic example of computation expressions in F# are *Asynchronous Workflows* used for asynchronous programming. In `.NET` and other frameworks this has traditionally been associated with callback programming [11], known to result in complex and hard-to-read code. Asynchronous workflows avoid the need of explicit callbacks, giving the illusion of sequential programming [12].

```
let download (url : Uri) = async {
    let http = new System.Net.WebClient()
    let! html = http.AsyncDownloadString(url)
    return html.Split('\n')
}
```

The above workflow asynchronously downloads the content of a web page and resumes to split it into lines once the download has been completed. Async operations are composed with the special `let!` keyword, which can be thought of as syntactic sugar for the callback being passed to the right-hand-side operation. The `return` keyword denotes that the workflow should conclude with the right hand side value. As a sidenote, the two keywords are F# syntactic sugar for the *monadic bind* and *monadic unit* operations, respectively.

F# async workflows can be used to actuate thread parallelism. The combinator

```
Async.Parallel : seq<Async<'T>> -> Async<'T []>
```

where `Async<'T>` is the type signature of an `async` workflow computing a value of type `'T`, combines a given enumeration of workflows into a single asynchronous workflow that executes the given inputs in parallel, following a fork-join pattern:

```
let workflow = async {
    let! results =
        Async.Parallel
            [
                download "http://www.m-brace.net";
                download "http://www.nessos.gr"
            ]

    return Seq.concat results |> Seq.length
}
```

This snippet will download the two pages asynchronously and will resume computation as soon as both operations have completed.

Async expressions have deferred execution semantics: they need to be evaluated by a scheduler that transparently allocates pending jobs to the underlying `.NET` thread pool. A typical executing `async` expression will make jumps between multiple threads as it progresses.

The programming model introduced in F# asynchronous workflows has been successful enough that it has been adopted by other languages such as C# 5.0, Python and Haskell.

2.2 The Cloud Workflow Programming Model

The programming model of MBrace follows very much in the style of F# asynchronous workflows. MBrace introduces *cloud workflows* as a means of specifying distributed computation. A cloud workflow has type `Cloud<'T>`, which represents a deferred cloud computation that returns a result of type `'T` once executed. Building on our previous declaration of the `download` async workflow, we could define

```
let lineCount() =
    cloud {
        let jobs : Cloud<string []> [] =
            Array.map (Cloud.OfAsync << download)
                [| "http://www.m-brace.net" ;
                    "http://www.nessos.gr" |]

        let! results = Cloud.Parallel jobs
        return Array.concat results |> Array.length
    }
```

This is a direct adaptation of the previous `async` snippet. The main differences between this and `async` are the `cloud{}` keyword that delimits workflows and the `Cloud.Parallel` combinator that actuates parallelism. Its type signature is

```
Cloud.Parallel : Cloud<'T> [] -> Cloud<'T []>
```

This takes an array of cloud computations and returns a workflow that executes them in parallel, returning an array of all the results. The parallel jobs are allocated to worker machines in the data centre, just as `async` jobs are scheduled to threads in the thread pool. The parent computation will resume as soon as all of the child computations have completed. Like `async`, `cloud` workflows have deferred execution semantics and have to be sent to an MBrace runtime for evaluation.

Cloud workflows can be used to create user defined higher-order functions. For example, we can define a distributed variant of the classic filter combinator:

```
let filter (f : 'T -> bool) (xs : 'T []) =
    cloud {
        // a nested subexpression that performs
        // the checking on a single element
        let pick (x : 'T) =
            cloud {
                if f x then return Some x
                else return None
            }

        let! results =
            Cloud.Parallel <| Array.map pick xs
        return Array.choose id results
    }
```

Cloud workflows support a multiplicity of overloaded language constructs, including operations such as `for` and `while` loops.

```
cloud {
    for i in [| 1 .. 100 |] do
        let! _ = cloud { return i }
```

```
        in return ()
    }
```

Even though these are sequential operations, the fact that they compose with arbitrary cloud workflows make them powerful tools in scenaria where distributed, iterative algorithms are required. The cloud workflow model also permits higher-order, recursive declarations and is expressive enough to enable complex computation patterns such as the Ackermann function.

An important feature offered by cloud workflows is *exception handling*. Exceptions can be raised and caught in cloud workflows just like any other piece of F# or .NET code. However, the distributed nature of cloud workflows makes this version of exception handling particularly interesting: in MBrace, the symbolic execution stack winds across multiple machines. Thus as exceptions are raised, the computation stack is un-winded across multiple machines as well.

This is a good demonstration of what we feel is one of the biggest strengths of MBrace. Error handling in particular and computation state in general have a global and hierarchical scope rather than one that is fragmented and localised. This is achieved thanks to symbolic, distributed interpretation of what is known as the *free continuation monad* [13, 14], or what we like to call the "monadic skeleton" of a cloud workflow.

Example: Defining MapReduce

MapReduce is a programming model that streamlines large scale distributed computation on big data sets. Introduced by Google in 2003, it has known immense success in open source implementations, such as Hadoop. MapReduce is a higher-order distributed algorithm that takes two functions, `map` and `reduce` as its inputs. The `map` function performs some computation on initial input, while the `reduce` function takes two outputs from `map` and combines them into a single result. When passed to MapReduce, a distributed program is defined that performs the combined mappings and reductions on a given list of initial inputs.

Unlike other big data frameworks, where MapReduce comes as *the* distribution primitive, MBrace makes it possible to define MapReduce-like workflows at the library level. A simplistic variant could be declared as follows:

```
let rec mapReduce (map: 'T -> Cloud<'R>)
                  (reduce: 'R -> 'R -> Cloud<'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
    }
```

This splits the list of inputs in half and passes them recursively into the workflow using the binary parallel operator,

```
<||> : Cloud<'T> -> Cloud<'U> -> Cloud<'T * 'U>
```

Once inputs are reduced to trivial components, the `map` function is applied and forked jobs are gradually joined using the `reduce` function.

Nondeterministic Computation

In addition to `Cloud.Parallel`, MBrace offers the distribution primitive

```
Cloud.Choice : Cloud<'T option> [] -> Cloud<'T option>
```

that *combines* a collection of nondeterministic computations into one. A computation that returns optionals is nondeterministic in the sense that it may either succeed by returning `Some` value of type `'T` or fail yielding `None`. What the `Choice` combinator does is execute input workflows in parallel, returning a result as soon as the first child computation completes successfully (with `Some` result), actively cancelling all other pending children. The combinator returns `None` if *every* child completed without success.

The `Choice` combinator is particularly suited for distributing decision problems, such as SAT solvers or large number factorisation. As an example, we give the implementation of a distributed existential combinator:

```
let exists (f : 'T -> Cloud<bool>) (inputs : 'T []) =
    cloud {
        let pick (x : 'T) =
            cloud {
                let! result = f x
                return
                    if result then Some x
                    else None
            }

        let! result =
            Cloud.Choice <| Array.map pick inputs
        return result.IsSome
    } : Cloud<bool>
```

Local Parallelism

There are cases where constraining the execution of a cloud workflow in the context of a single worker node might be extremely useful. This can be performed using the

```
Cloud.ToLocal : Cloud<'T> -> Cloud<'T>
```

primitive, or its `local` abbreviation. The combinator transforms any given cloud workflow into an equivalent expression that executes in a strictly local context, forcing concurrency semantics largely similar to those of `async` but with the additional capabilities that can be enabled by running on a cloud runtime.

The `local` primitive is particularly handy when it comes to effectively managing computation granularity. As an example, we can implement a simple Fibonacci function, with a fallback mechanism to local parallelism.

```
let rec fib n depth =
    cloud {
        if depth = 0 then
            return! Cloud.ToLocal <| fib n depth
        else
            match n with
            | 1 | 2 -> return 1
            | n ->
                let! (left, right) =
                    fib (n - 1) (depth - 1)
                        <||>
                    fib (n - 2) (depth - 1)
                return left + right
    }
```

3. Distributed Data

Cloud workflows offer a programming model for distributed computation. But what happens when it comes to big data? While the

distributable execution environments of MBrace do offer a limited form of data distribution, their scope is inherently local and almost certainly do not scale to the demands of modern big data applications. MBrace offers a plethora of mechanisms for managing data in a more global and massive scale. These provide an essential decoupling between distributed computation and distributed data.

Cloud Refs

The MBrace programming model offers access to persistable and distributed data entities known as *cloud refs*. Cloud refs very much resemble references found in the `ML` family of languages but are "monadic" in nature. In other words, their introduction entails a scheduling decision by the runtime. The following workflow stores the downloaded content of a web page and returns a cloud ref to it:

```
let getRef () : Cloud<CloudRef<string []>> =
    cloud {
        let! html = download "http://www.m-brace.net"
        let! ref = CloudRef.New html
        return ref
    }
```

When run, this will return a unique identifier that can be subsequently dereferenced either in the context of the client or in a future cloud computation:

```
// compute a cloud ref
let r = runtime.Run <@ getRef () @>
// dereference locally
let data : string [] = r.Value
```

Values of cloud refs are kept in a global *storage provider* that plugs into the MBrace runtime. MBrace transparently manages storage, while it also aggressively caches local copies to select worker nodes. Scheduling decisions are taken with respect to caching affinity, resulting in minimized data transfer.

Cloud refs are *immutable* by design, they can either be initialized or dereferenced. Immutability eliminates synchronization issues, resulting in efficient caching and enhanced access speeds.

An interesting aspect of cloud refs is the ability to define large, distributed data structures. For example, one could define a distributed binary tree like so:

```
type DistribTree<'T> =
    | Leaf
    | Branch of 'T * CloudRef<DistribTree<'T>> *
                     CloudRef<DistribTree<'T>>

// initialize a simple distributed tree
let rec populate (depth : int) =
    cloud {
        if depth = 0 then
            return! CloudRef.New Leaf
        else
            let! l,r = populate (depth - 1) <||>
                          populate (depth - 1)

            return! CloudRef.New <| Branch(depth, l, r)
    } : Cloud<CloudRef<DistribTree<int>>>
```

MBrace also offers a mutable variant of the `CloudRef`, called appropriately `MutableCloudRef`. These permit *conditional* updates, which makes them a powerful primitive capable of defining synchronisation mechanisms such as distributed locks and semaphores. This comes with a performance trade-off, since mutable cloud refs are not cached.

Miscellaneous Data Primitives

The MBrace framework comes with a few additional distributed data primitives:

- *Cloud sequences* allow the storage of immutable *collections* of values that can be dereferenced in a *lazy*, *on-demand* manner.
- *Cloud files* enable the storage of immutable binary blobs.

Distributed Resource Management

The constructs mentioned previously manifest themselves by allocating space in the storage back end of the runtime. They thus occupy resources associated with the global distribution context and are not garbage collectable by individual workers in the cluster. Such "globally scoped" items give rise to the need for distributed deallocation facilities.

The MBrace programming model offers a mechanism for performing such deallocations as well as a syntactic facility for scoped resource management. All of the aforementioned data constructs implement the `CloudDisposable` interface. This can be thought of as a distributed version of the `IDisposable` interface available in `.NET`. Similarities do not stop there; just as `.NET` languages offer the `using` keyword [15] that allows for scoped introduction of disposable resources, MBrace workflows come with the `use` and `use!` keywords that apply to `CloudDisposable` entities. For instance,

```
cloud {
    use! cref = CloudRef.New [| 1 .. 100000000 |]
    try
        if cref.Value.Length > 1000 then
            return failwith "error"
    with e ->
        do! Cloud.Logf "error in cloudRef %O" cref
        return raise e
}
```

Initializing a cloud ref with the `use` keyword provides the assurance that it will be deallocated from the global store as soon as the workflow has exited its scope. In conformance to the standard `using` semantics of `.NET` languages, this will occur regardless of whether the computation has completed normally or exited with an exception.

4. The MBrace Runtime

The MBrace framework includes a distributed runtime capable of executing cloud workflows: the MBrace runtime is an elastic, fault tolerant and multitasking cluster infrastructure and execution engine that includes facilities for managing, monitoring and debugging *cloud processes*, much in the sense of a distributed operating system.

The MBrace execution model follows a scheduler/worker hierarchy: when a cloud workflow is uploaded to the runtime for execution, a scheduler instance is initialized that interprets the monadic structure of the workflow, disseminating continuations to worker nodes as required. Scheduling is load balanced, taking into account CPU and network statistics for each available worker. It is also fault-tolerant, rescheduling any job in the event of a worker being lost.

The MBrace runtime is capable of executing multiple cloud processes concurrently. Cloud processes can be monitored, debugged and cancelled on demand. The runtime enforces a policy of *isolation* in which every cloud process is run at dedicated CLR instances in each worker node: this ensures better resource management and improves overall stability.

It should be noted that the MBrace runtime requires an independent storage service to function. This is used for recording distributed data primitives as well as internal optimizations. MBrace comes out of the box with implementations for FileSystem, SQL and Azure storage providers, while providing pluggable, user-defined implementations is also possible. We plan to include support for other storage services in the future, such as Hadoop's

HDFS or Amazon's S3. A more ambitious direction is implementing an in-house storage service that utilizes storage space from the MBrace cluster itself.

The MBrace runtime is currently deployable in Windows-only machines and has been successfully tested in private data centres as well as Windows Azure. Effort is being made to port MBrace to Mono [16] and Linux.

5. Performance

In this section we discuss performance and scalability of the MBrace framework and how it compares to the fairly established Apache Hadoop framework. Both frameworks were tested while running on clusters powered by Windows Azure. The Hadoop clusters consisted of 4, 8, 16 and 32 data nodes with one head node. The data nodes were A3-Large instances (4 virtual cores, 7GB RAM) and the head node was an A4 Extra-Large instance (8 virtual cores, 14GB RAM). The MBrace clusters consisted of 4, 8, 16 and 32 A3-Large instances used as workers and one A3-Large instance used as a scheduler. We present two benchmarks produced using a couple of fairly common distributed algorithms, *grep* and *k-means*[1].

As a first test, we implemented the distributed grep algorithm over MapReduce in both MBrace and Hadoop. The distributed grep algorithm reads a collection of files and returns the number of lines that match a given pattern. This is a fairly IO-based algorithm that is not very computation-intensive. The input set has a size of 32, 64, 128 and 256 GB depending on the cluster size. The results show that MBrace slightly outperforms Hadoop (see figure 1).

We also implemented the k-means algorithm: k-means is an *iterative* algorithm used for partitioning a set of vectors into a fixed number of clusters, so that a centroid distance minimization condition is satisfied. k-means is an important tool in fields like signal processing and machine learning. However, it is also the prototypical example of an algorithm not directly expressible using the MapReduce model. The problem can be remedied by scheduling sequences of distinct MapReduce jobs, but that could result in performance loss. This is not the case with MBrace, which is not bound by the constraints of the MapReduce model.

We compared the performance of the k-means algorithm found in the Apache Mahout library [17] with our own MBrace implementation. The input was one million, randomly generated, 100-dimensional points, the number of clusters was set to $k = 6$ and the number of iterations was 10. The cluster configuration remains the same as in the grep test. The results show that MBrace significantly surpasses Mahout, with performance improvement reaching one order of magnitude (see figure 2).

6. Related Work

Many interesting and expressive frameworks come from the Haskell community and are based on the idea that strong types and monads can offer a composable model for programming with *effects*. CloudHaskell[3] and HdpH[4] are two new approaches based on monads for composing distributed computations but with slightly different approaches. CloudHaskell is based on a *Process monad* which provides a message passing communication model, inspired by Erlang, and a novel technique for serialising closures. The Process monad is used as a shallow DSL embedding for the channel oriented communication layer and is intended as a low-level layer for building larger abstractions. HdpH is influenced by the `Par` monad[18] and the work on closure serialisation found in CloudHaskell. It provides a high-level semi-explicit parallelism via the use of a distributed generalisation of the `Par` monad. One interesting design decision is the use of global references for communication purposes and data sharing.

[1] All benchmark related source code can be freely referenced at `https://github.com/nessos/MBrace-Benchmarks-PLOS2013`.

Figure 1. Distributed grep performance on Windows Azure

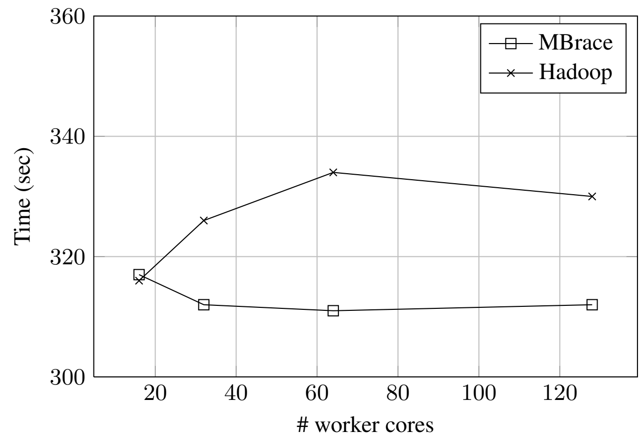
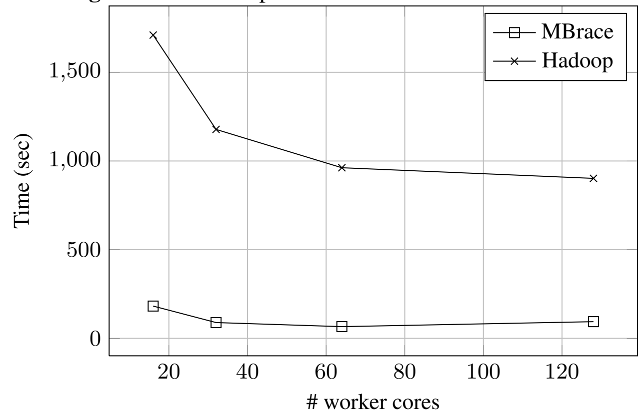


Figure 2. k-means performance on Windows Azure



7. Conclusions

MBrace is a distributed programming model and framework based on the `.NET` software stack. The programming model of MBrace is founded on F#'s computation expressions. Parallelism patterns are introduced using primitive combinators that act on cloud workflows. Cloud workflows are executed by a scheduler running on the MBrace runtime, which transparently allocates cluster resources and ensures fault tolerance.

The MBrace framework includes an execution runtime as well as client tooling for on-the-fly deployment and debugging of distributed code. The framework is currently in its alpha testing stage, however early benchmarks show that it matches, in some cases greatly surpassing Hadoop in performance.

MBrace is a work in progress with many challenges ahead. Among other things, we are currently engaged with providing an embedded DSL for data parallelism in F# and C#, in the form of a `LINQ` [19] provider. We also plan to offer support for the Mono framework [16], which would open up MBrace to the Linux world.

References

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on*

Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, 2004.

[2] Akka Framework. URL `http://www.akka.io/`.

[3] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the cloud. *SIGPLAN Not.*, 46(12):118–129, September 2011. ISSN 0362-1340.

[4] Patrick Maier and Phil Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *Proceedings of the 23rd international conference on Implementation and Application of Functional Languages*, IFL'11, pages 35–50, 2012. ISBN 978-3-642-34406-0.

[5] Longbin Lai, Jingyu Zhou, Long Zheng, Huakang Li, Yanchao Lu, Feilong Tang, and Minyi Guo. ShmStreaming: A Shared Memory Approach for Improving Hadoop Streaming Performance. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 137–144, 2013. .

[6] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010. ISSN 2150-8097.

[7] Cairong Yan, Xin Yang, Ze Yu, Min Li, and Xiaolin Li. IncMR: Incremental Data Processing Based on MapReduce. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 534–541, 2012.

[8] MBrace website. URL `http://www.m-brace.net/`.

[9] Microsoft MSDN, Computation Expressions (F#), . URL `http://msdn.microsoft.com/en-us/library/dd233182.aspx`.

[10] Tomas Petricek and Don Syme. Syntax Matters: Writing abstract computations in F#. *Pre-proceedings of TFP (Trends in Functional Programming), St. Andrews, Scotland*, 2012.

[11] Microsoft MSDN, Asynchronous Programming Model, . URL `http://msdn.microsoft.com/en-us/library/ms228963.aspx`.

[12] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# Asynchronous Programming Model. In *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*. 2011. ISBN 978-3-642-18377-5.

[13] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18:423–436, 7 2008. ISSN 1469-7653.

[14] R.O. Bjarnarson. Stackless Scala With Free Monads. Scala Days 2012. URL `http://blog.higher-order.com/assets/trampolines.pdf`.

[15] Microsoft MSDN, `using` Statement (C# Reference), . URL `http://msdn.microsoft.com/en-us/library/yh598w02.aspx`.

[16] Mono Project. URL `http://www.mono-project.com/`.

[17] Apache Mahout. URL `http://mahout.apache.org/`.

[18] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, 2011. ISBN 978-1-4503-0860-1.

[19] Microsoft MSDN, LINQ (Language-Integrated Query). URL `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`.