MBrace: Cloud Computing with Monads

Jan Dzik    Nick Palladinos    Kostas Rontogiannis
Eirik Tsarpalis    Nikolaos Vathis

Nessos Information Technologies, SA

7th Workshop on Programming Languages and Operating Systems
November 3, 2013
Motivation

- Distributed Computation is Challenging.
- Key to success: choose the right distribution framework.
- Each framework tied to particular programming abstraction.
- Map-Reduce, Actor model, Dataflow model, etc.
Established distributed frameworks

- Restrict to specific distribution patterns.
- Not expressive enough for certain classes of algorithms.
- Difficult to influence task granularity.
- Time consuming to deploy, manage and debug.
What is MBrace?

1. A new programming model for the cloud.
2. An elastic, fault tolerant, multitasking cluster infrastructure.
In This Talk

- Concentrate on the programming model.
  - Distributed Computation.
  - Distributed Data.

- Benchmarks.
The MBrace Programming Model

- A monad for composing distribution workflows.
- Essentially a continuation monad that admits distribution.
- Based on F# *computation expressions*.
- Inspired by the successful F# asynchronous workflows.
A Basic cloud workflow

```csharp
let download (url : string) = cloud {
    let client = new System.Net.WebClient()
    let content = client.DownloadString(url)
    return content
}

: Cloud<string>
```
Composing cloud workflows

```ml
let downloadSequential () = cloud {
  let! c1 = download "http://m-brace.net/"
  let! c2 = download "http://nessos.gr/"

  let c = c1 + c2

  return c
}
```
let downloadParallel () = cloud {
    let! c1,c2 =
    download "http://m-brace.net/"
    <||>
    download "http://nessos.gr/"

    return c1 + c2
}
Distribution Primitives: an overview

- Binary parallel operator:
  \(<||>\) : \text{Cloud<'T>} \to \text{Cloud<'U>} \to \text{Cloud<'T * 'U>}

- Variadic parallel combinator:
  \text{Cloud.Parallel} : \text{Cloud<'T>} [] \to \text{Cloud<'T [>}

- Non-deterministic parallel combinator:
  \text{Cloud.Choice} : \text{Cloud<'T option>} [] \to \text{Cloud<'T option>}

Cloud Monad: additional constructs

- Monadic for loops.
- Monadic while loops.
- Monadic exception handling.
Example: Inverse squares

```csharp
let inverseSquares (inputs : int []) = cloud {
    let jobs : Cloud<float> [] = []
    [ ]
    for i in inputs ->
        cloud { return 1.0 / float (i * i) }
    [ ]

    try
    let! results = Cloud.Parallel jobs
    return Array.sum results

    with :? DivideByZeroException ->
    return -1.0
}
```
How is it all executed?

- Scheduler/worker cluster organization.
- Symbolic execution stack (free monad/trampolines).
- Scheduler interprets “monadic skeleton”.
- Native “leaf expressions” dispatched to workers.
- Symbolic stack winds across multiple machines.
A Map-Reduce implementation

```ocaml
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
  }
```
What about Data Distribution?

- MBrace does NOT include a storage service (for now).
- Relies on third-party storage services.
- *Storage Provider* plugin architecture.
- Future support for HDFS and Amazon S3.
Storage services interfaced through data primitives.

Data primitives act as references to distributed resources.

Initialized or updated through the monad.

Come in immutable or mutable flavors.
Cloud Ref

- Simplest distributed data primitive of MBrace.
- Generic reference to a stored value.
- Conceptually similar to ML ref cells.
- Immutable by design.
- Cached in worker nodes for performance.
Cloud Ref: Example

```ocaml
let createRef (inputs : int []) = cloud {
    let! ref = CloudRef.New inputs
    return ref : CloudRef<int []>
}

let deRef (ref : CloudRef<int []>) = cloud {
    let content = ref.Value
    return content : int []
}
```
type DistribTree<'T> =
  | Leaf of 'T
  | Branch of CloudRef<DistribTree<'T>> *
      CloudRef<DistribTree<'T>>

let rec map (f : 'T -> 'S) (tree : DistribTree<'T>) =
  cloud {
    match tree with
    | Leaf t -> return! CloudRef.New (Leaf (f t))
    | Branch(l,r) ->
      let! l', r' = map f l.Value <|> map f r.Value
      return! CloudRef.New (Branch(l',r'))
  }
Cloud File

- References files in the distributed store.
- Untyped, immutable, binary blobs.
Cloud File : Example

```csharp
let getSize (file : CloudFile) = cloud {
    let! bytes = CloudFile.ReadAllBytes file
    return bytes.Length / 1024
}

cloud {
    let! files = CloudDir.GetFiles "/path/to/files"
    let jobs = Array.map getSize files
    let! sizes = Cloud.Parallel jobs
    return Array.sum sizes
}
```
We tested MBrace against Hadoop.

Both frameworks were run on Windows Azure.

Clusters consisted of 4, 8, 16 and 32 quad-core nodes.

Two algorithms were tested, grep and k-means.

Source code available on github.
Distributed Grep (Windows Azure)

- Count occurrences of given pattern from input files.
- Straightforward Map-Reduce algorithm.
- Input data was 32, 64, 128 and 256 GB of text.
Distributed Grep (Windows Azure)

The MBrace Framework: Performance

MBrace: Cloud Computing with Monads

Eirik Tsarpalis (Nessos IT)
**k-means Clustering (Windows Azure)**

- Centroid computation out of a set of vectors.
- Iterative algorithm.
- Not naturally definable with Map-Reduce workflows.
- Hadoop implementation from Apache Mahout library.
- Input was $10^6$, randomly generated, 100-dimensional points.
**k-means Clustering (Windows Azure)**

The graph shows the performance of MBrace and Hadoop for different numbers of worker cores. The x-axis represents the number of worker cores, while the y-axis represents the time in seconds. The graph indicates that MBrace scales better than Hadoop as the number of worker cores increases.
Conclusions

- A big data platform for the .NET framework.
- Language-integrated cloud workflows.
- User-specifiable parallelism patterns and task granularity.
- Distributed exception handling.
- Pluggable storage services.
- Data API integrated with programming model.
Future Work

- Improved C# support.
  - A rich library of combinators and parallelism patterns.
  - A LINQ provider for data parallelism.
- Support for the Mono framework and Linux.
Thank You!

Questions?

http://m-brace.net