Experiments with MapReduce in Erlang

Tom Van Cutsem
tvcutsem@vub.ac.be
Context

• Teaching Master-level course on “Multicore Programming”

• Focus on concurrent, parallel and... *functional* programming

• Didactic implementation of Google’s MapReduce algorithm in Erlang

  • Goal: teach both Erlang and typical MapReduce API
What is MapReduce?

• A **programming model** to formulate large **data processing jobs** in terms of “map” and “reduce” computations

• **Parallel** implementation on a large **cluster** of commodity hardware

• **Characteristics:**
  
  • Jobs take input records, process them, and produce output records
  
  • Massive amount of I/O: reading, writing and transferring large files
  
  • Computations typically not so CPU-intensive

Dean and Ghemawat (Google)
*MapReduce: Simplified Data Processing on Large Clusters*
OSDI 2004
MapReduce: why?

- Example: index the WWW
- 20+ billion web pages x 20KB = 400+ TB
- One computer: read 30-35MB/sec from disk
  - ~ four months to read the web
  - ~ 1000 hard drives to store the web
- Good news: on 1000 machines, need < 3 hours
- Bad news: programming work, and repeated for every problem

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
MapReduce: fundamental idea

- Separate **application-specific computations** from the messy details of parallelisation, fault-tolerance, data distribution and load balancing.

- These application-specific computations are expressed as **functions** that map or reduce data.

- The use of a functional model allows for **easy parallelisation** and allows the use of **re-execution** as the primary mechanism for **fault tolerance**.
MapReduce: key phases

- Read lots of data (key-value records)

- **Map**: extract useful data from each record, generate intermediate keys/values

- Group intermediate key/value pairs by key

- **Reduce**: aggregate, summarize, filter or transform intermediate values with the same key

- Write output key/value pairs

Same general structure for all problems, **Map** and **Reduce** are problem-specific

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
MapReduce: inspiration

- In functional programming (e.g. in Clojure, similar in other FP languages)

\[
\text{map} \ (\text{fn} \ [x] \ (* \ x \ x)) \ [1 \ 2 \ 3]) \Rightarrow [1 \ 4 \ 9]
\]

\[
\text{reduce} \ + \ 0 \ [1 \ 4 \ 9]) \Rightarrow 14
\]

- The Map and Reduce functions of MapReduce are inspired by but not the same as the map and fold/reduce operations from functional programming
Map and Reduce functions

- Map takes an input key/value pair and produces a list of intermediate key/value pairs

- Input keys/values are not necessarily from the same domain as output keys/values

\[
\text{map: } (K_1, V_1) \rightarrow \text{List}[(K_2, V_2)] \\
\text{reduce: } (K_2, \text{List}[V_2]) \rightarrow \text{List}[V_2]
\]

\[
\text{mapreduce: } (\text{List}[(K_1, V_1)], \text{map}, \text{reduce}) \rightarrow \text{Map}[K_2, \text{List}[V_2]]
\]
Map and Reduce functions

- All v2 with the same k2 are reduced together (remember the invisible “grouping” step)

\[
\text{map: } (k, v) \rightarrow [(k2, v2), (k2', v2'), \ldots ] \\
\text{reduce: } (k2, [v2, v2', \ldots ]) \rightarrow [v2'', \ldots ]
\]

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
Map and Reduce functions

- All v2 with the same k2 are reduced together (remember the invisible “grouping” step)

map: \((k, v) \rightarrow [ (k2, v2), (k2', v2'), \ldots ]\)
reduce: \((k2, [v2, v2', \ldots]) \rightarrow [ v2'', \ldots ]\)

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
Map and Reduce functions

- All v2 with the same k2 are reduced together (remember the invisible “grouping” step)

\[
\text{map: } (k, v) \rightarrow [ (k2,v2), (k2',v2'), \ldots ]
\]
\[
\text{reduce: } (k2, [v2,v2',\ldots]) \rightarrow [ v2'', \ldots ]
\]
Example: word frequencies in web pages

- (K1,V1) = (document URL, document contents)
- (K2,V2) = (word, frequency)

Map

```
"document1", "to be or not to be"
```

- ("to", 1)
- ("be", 1)
- ("or", 1)
- ...

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
Example: word frequencies in web pages

- \((K1,V1) = \text{(document URL, document contents)}\)

- \((K2,V2) = \text{(word, frequency)}\)

Reduce

\[\begin{array}{c}
\text{Reduce} \\
\text{("be", \([1, 1]\))} \\
\text{("not", \([1]\))} \\
\text{("or", \([1]\))} \\
\text{("to", \([1, 1]\))} \\
\end{array}\]

\[\begin{array}{c}
\text{[2]} \\
\text{[1]} \\
\text{[1]} \\
\text{[2]} \\
\end{array}\]

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
Example: word frequencies in web pages

- $(K1,V1) = (\text{document URL, document contents})$

- $(K2,V2) = (\text{word, frequency})$

Output

- (“be”, 2)
- (“not”, 1)
- (“or”, 1)
- (“to”, 2)

(Source: Michael Kleber, “The MapReduce Paradigm”, Google Inc.)
More Examples

• Count URL access frequency:

  • Map: process logs of web page requests and output <URL,1>

  • Reduce: add together all values for the same URL and output <URL,total>

• Distributed Grep:

  • Map: emit a line if it matches the pattern

  • Reduce: identity function
More Examples

- Inverted Index for a collection of (text) documents:
  - Map: emits a sequence of <word, documentID> pairs
  - Reduce: accepts all pairs for a given word, sorts documentIDs and returns <word, List(documentID)>

- Implementation in Erlang follows later
Conceptual execution model

The diagram illustrates a MapReduce execution model, where:

- **Input** table:
  - K1 | V1
  - k1 | v1
  - k2 | v2
  - k3 | v3

- **Mapper** processes the data:
  - [(a,1)]
  - [(a,2), (b,2)]
  - [(b,3), (b,4)]

- **Intermediate** data:
  - K2 | List[V2]
  - a   | [1,2]
  - b   | [2,3,4]

- **Reducer** collects and sorts according to K2:
  - [1,3]
  - [2,5,9]

- **Output** table:
  - K2 | List[V2]
  - a   | [1,3]
  - b   | [2,5,9]

The process involves:
- Assigning work to mappers
- Collecting and sorting according to K2
- Assigning work to reducers
- Collecting reduced values
The devil is in the details!

- How to partition the data, how to balance the load among workers?
- How to efficiently route all that data between master and workers?
- Overlapping the map and the reduce phase (pipelining)
- Dealing with crashed workers (master pings workers, re-assigns tasks)
- Infrastructure (need a distributed file system, e.g. GFS)
- ...
Erlang in a nutshell
Invented at Ericsson Research Labs, Sweden

Declarative (functional) core language, inspired by Prolog

Support for concurrency:
  - processes with isolated state, asynchronous message passing

Support for distribution:
  - Processes can be distributed over a network
Example: an echo process

- Echo process echoes any message sent to it

-module(echo).
-export([start/0, loop/0]).

start() ->
    spawn(echo, loop, []).

loop() ->
    receive
        {From, Message} ->
            From ! Message,
            loop()
    end.

Id = echo:start(),
Id ! { self(), hello },
receive
    Msg ->
        io:format("echoed ~w~n", [Msg])
    end.
Processes can encapsulate state

- Example: a counter process

- Note the use of tail recursion

```
-module(counter).
-export([start/0, loop/1]).

start() ->
  spawn(counter, loop, [0]).

loop(Val) ->
  receive
    increment ->
      loop(Val + 1);
    {From, value} ->
      From ! {self(), Val},
      loop(Val)
  end.
```
MapReduce in Erlang
A naive parallel implementation

- Map and Reduce functions will be applied in parallel:
  - Mapper worker process spawned for each \( \{K1,V1\} \) in Input
  - Reducer worker process spawned for each intermediate \( \{K2,[V2]\} \)

\[
\text{Input} = \left[ \left\{ K1, V1 \right\} \right]
\]

\[
\text{Map}(K1, V1, \text{Emit}) \rightarrow \text{Emit a stream of } \{K2,V2\} \text{ tuples}
\]

\[
\text{Reduce}(K2, \text{List}[V2], \text{Emit}) \rightarrow \text{Emit a stream of } \{K2,V2\} \text{ tuples}
\]

\[
\text{Returns a Map}[K2,\text{List}[V2]]
\]

\[
\text{mapreduce}(\text{Input}, \text{Map}, \text{Reduce}) \rightarrow
\]
A naive parallel implementation

mapreduce(Input, Map, Reduce) ->
    Client = self(),
    Pid = spawn(fun() -> master(Client, Map, Reduce, Input) end),
    receive
        {Pid, Result} -> Result
    end.
A naive parallel implementation

[Diagram of a naive parallel implementation with nodes labeled 'Master', 'M', 'R', and edges labeled 'spawn_workers' and 'collect_replies']

- Master
  - M
    - {K1, V1}
    - {K1', V1'}
    - {K1'', V1''}

- R
  - {K2, V2}, {K2', V2'}
  - {K2, [V2, V2'']}
  - {K2', [V2']}
  - {K2, V2''}, {K2, V2'''}, ...

- Collect_replies
  - {K2, [V2, V2'']}
A naive parallel implementation

```erlang
class master(Parent, Map, Reduce, Input) ->
    process_flag(trap_exit, true),
    MasterPid = self(),
    spawn_workers(MasterPid, Map, Input),
    M = length(Input),
    Intermediate = collect_replies(M, dict:new()),
    spawn_workers(MasterPid, Reduce, dict:to_list(Intermediate)),
    R = dict:size(Intermediate),
    Output = collect_replies(R, dict:new()),
    Parent ! {self(), Output}.
```
A naive parallel implementation

```erlang
spawn_workers(MasterPid, Fun, Pairs) ->
    lists:foreach(fun({K,V}) ->
        spawn_link(fun() -> worker(MasterPid, Fun, {K,V}) end)
    end, Pairs).

% Worker sends {K2, V2} messages to master and then terminates
worker(MasterPid, Fun, {K,V}) ->
    Fun(K, V, fun(K2,V2) -> MasterPid ! {K2, V2} end).
```
A naive parallel implementation

```erlang
spawn_workers(MasterPid, Fun, Pairs) ->
    lists:foreach(fun({K,V}) -> spawn_link(fun() -> worker(MasterPid, Fun, {K,V}) end) end, Pairs).
```

% Worker sends {K2, V2} messages to master and then terminates
worker(MasterPid, Fun, {K,V}) ->
    Fun(K, V, fun(K2,V2) -> MasterPid ! {K2, V2} end).
```

Fun calls Emit(K2,V2) for each pair it wants to produce.
A naive parallel implementation

collect_replies(\(\emptyset\), Dict) -> Dict;
collect_replies(N, Dict) ->
    receive
      \{Key, Val\} ->
        Dict1 = dict:append(Key, Val, Dict),
        collect_replies(N, Dict1);
      \{'EXIT', _Who, _Why\} ->
        collect_replies(N-1, Dict)
    end.
Example: text indexing

- Example input:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>/test/dogs</td>
<td>[rover, jack, buster, winston].</td>
</tr>
<tr>
<td>/test/cats</td>
<td>[zorro, daisy, jaguar].</td>
</tr>
<tr>
<td>/test/cars</td>
<td>[rover, jaguar, ford].</td>
</tr>
</tbody>
</table>

- Input: a list of {Idx, FileName}

<table>
<thead>
<tr>
<th>Idx</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/test/dogs</td>
</tr>
<tr>
<td>2</td>
<td>/test/cats</td>
</tr>
<tr>
<td>3</td>
<td>/test/cars</td>
</tr>
</tbody>
</table>
Example: text indexing

- Goal: to build an inverted index:

<table>
<thead>
<tr>
<th>Word</th>
<th>File Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>rover</td>
<td>“dogs”, “cars”</td>
</tr>
<tr>
<td>jack</td>
<td>“dogs”</td>
</tr>
<tr>
<td>buster</td>
<td>“dogs”</td>
</tr>
<tr>
<td>winston</td>
<td>“dogs”</td>
</tr>
<tr>
<td>zorro</td>
<td>“cats”</td>
</tr>
<tr>
<td>daisy</td>
<td>“cats”</td>
</tr>
<tr>
<td>jaguar</td>
<td>“cats”, “cars”</td>
</tr>
<tr>
<td>ford</td>
<td>“cars”</td>
</tr>
</tbody>
</table>

- Querying the index by word is now straightforward

(Source: Joe Armstrong, Programming Erlang, 2007)
Example: text indexing

- Building the inverted index using mapreduce:
  - Map(Idx, File): emit {Word, Idx} tuple for each Word in File
  - Reduce(Word, Files) -> filter out duplicate Files

```
Map(1, "dogs")
  Map(2, "cats")
  Map(3, "cars")

M \[\{\text{rover,1}\}, \ldots\]  \[\{\text{zorro,2}\}, \ldots\]  \[\{\text{rover,3}\}, \ldots\]

R \{\{\text{rover,\[\ldots\]}, \{\text{zorro,\[\ldots\]}\}\}, \ldots\]\n
Reduce(\text{zorro}, [2])
Reduce(\text{rover}, [1,3])
```
Text indexing using the parallel implementation

index(DirName) ->
    NumberedFiles = list_numbered_files(DirName),
    mapreduce(NumberedFiles, fun find_words/3,
               fun remove_duplicates/3).

% the Map function
find_words(Index, FileName, Emit) ->
    {ok, [Words]} = file:consult(FileName),
    lists:foreach(fun (Word) -> Emit(Word, Index) end,
                 Words).

% the Reduce function
remove_duplicates(Word, Indices, Emit) ->
    UniqueIndices = sets:to_list(sets:from_list(Indices)),
    lists:foreach(fun (Index) -> Emit(Word, Index) end,
                  UniqueIndices).
Text indexing using the parallel implementation

> dict:to_list(index(test)).
[{
    "rover": ["test/dogs", "test/cars"],
    "buster": ["test/dogs"],
    "jaguar": ["test/cats", "test/cars"],
    "ford": ["test/cars"],
    "daisy": ["test/cats"],
    "jack": ["test/dogs"],
    "winston": ["test/dogs"],
    "zorro": ["test/cats"]}
]
Only the start...

• Add support for fault-tolerance (restart crashed workers using Erlang’s process linking)

• Introduce coarse-grained tasks (each worker process maps/reduces more than 1 key). Process spawning in Erlang is cheap, but still not entirely free.

• Distributed implementation (master and workers on separate nodes)
  
  • Load balancing

• ...
MapReduce for real?

- In Erlang:
  - CouchDB
  - Riak

- Not in Erlang:
  - Hadoop
Summary

- MapReduce: programming model that separates application-specific map and reduce computations from parallel processing concerns.
  - Functional model: easy to parallelise, fault tolerance via re-execution
- Erlang: functional core language, concurrent processes + async message passing
- MapReduce in Erlang
  - Simple idea, arbitrarily complex implementations
  - Erlang is highly suitable to express such distributed algorithms