MapReduce in Erlang

Tom Van Cutsem
Context

• Masters’ 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 MapReduce style
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
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 (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 $v_i$ with the same $k_i$ 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 $v_i$ with the same $k_i$ 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 $v_i$ with the same $k_i$ are reduced together (remember the invisible “grouping” step)

\[
\begin{align*}
\text{map}: (k, v) & \rightarrow [ (k2,v2), (k2',v2'), \ldots ] \\
\text{reduce}: (k2, [v2,v2',\ldots]) & \rightarrow [ v2'', \ldots ]
\end{align*}
\]

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

- \((K_1, V_1) = (\text{document URL}, \text{document contents})\)
- \((K_2, V_2) = (\text{word}, \text{frequency})\)

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

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

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

**Reduce**

\((\text{“be”}, [1, 1])\) \(\rightarrow\) \([2]\)

\((\text{“not”}, [1])\) \(\rightarrow\) \([1]\)

\((\text{“or”}, [1])\) \(\rightarrow\) \([1]\)

\((\text{“to”}, [1, 1])\) \(\rightarrow\) \([2]\)

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

• (K1,V1) = (document URL, document contents)

• (K2,V2) = (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 \(<\text{word}, \text{documentID}\>\) pairs

  • Reduce: accepts all pairs for a given word, sorts documentIDs and returns \(<\text{word}, \text{List(documentID)}\>\)

• Implementation in Erlang follows later
Conceptual execution model

map: \((K1, V1) \rightarrow \text{List}[(K2, V2)]\)
reduce: \((K2, \text{List}[V2]) \rightarrow \text{List}[V2]\)

mapreduce: \(((K1, V1), \text{map, reduce}) \rightarrow \text{Map}[K2, \text{List}[V2]]\)
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)

• ...

woensdag 27 april 2011
Erlang in a nutshell
Erlang fact sheet

- 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
Sequential programming: factorial

-module(math1).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

> math1:factorial(6).
  720

> math1:factorial(25).
  15511210043330985984000000
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

```erlang
-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]\}

```%
Input = [{K1, V1}]
Map(K1, V1, Emit) -> Emit a stream of \{K2, V2\} tuples
Reduce(K2, List[V2], Emit) -> Emit a stream of \{K2, V2\} tuples
Returns a Map[K2, List[V2]]
mapreduce(Input, Map, Reduce) ->
```
A naive parallel implementation

% Input = [{K1, V1}]
% Map(K1, V1, Emit) -> Emit a stream of {K2,V2} tuples
% Reduce(K2, List[V2], Emit) -> Emit a stream of {K2,V2} tuples
% Returns a Map[K2,List[V2]]
mapreduce(Input, Map, Reduce) ->
    S = self(),
    Pid = spawn(fun() -> master(S, Map, Reduce, Input) end),
    receive
      {Pid, Result} -> Result
    end.
A naive parallel implementation
A naive parallel implementation

master(Parent, Map, Reduce, Input) ->
    process_flag(trap_exit, true),
    MasterPid = self(),

    % Create the mapper processes, one for each element in Input
    spawn_workers(MasterPid, Map, Input),

    M = length(Input),
    % Wait for M Map processes to terminate
    Intermediate = collect_replies(M, dict:new()),

    % Create the reducer processes, one for each intermediate Key
    spawn_workers(MasterPid, Reduce, dict:to_list(Intermediate)),

    R = dict:size(Intermediate),
    % Wait for R Reduce processes to terminate
    Output = collect_replies(R, dict:new()),
    Parent ! {self(), Output}.
A naive parallel implementation

\[
\text{spawn\_workers(MasterPid, Fun, Pairs)} \rightarrow \\
\text{lists:foreach(fun(\{K, V\}) \rightarrow} \\
\quad \text{\textit{spawn\_link(fun()} \rightarrow \text{worker(MasterPid, Fun, \{K, V\}) end) end, Pairs).}
\]

% Worker must send \{K2, V2\} messsages to master and then terminate
\text{worker(MasterPid, Fun, \{K, V\}) \rightarrow} \\
\quad \text{Fun(K, V, fun(K2, V2) \rightarrow MasterPid ! \{K2, V2\} end).}
A naive parallel implementation

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

% Worker must send \{K2, V2\} messages to master and then terminate
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 and merge {Key, Value} messages from N processes.
% When N processes have terminated return a dictionary
% of {Key, [Value]} pairs
collect_replies(0, 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
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")  M  [{rover,1}, ...

Map(2,"cats")  M  [{zorro,2}, ...

Map(3,"cars")  M  [{rover,3}, ...

R  Reduce(zorro,[2])

R  Reduce(rover,[1,3])
```

{rover,"dogs","cars"},
{zorro,"cats"},
...

Text indexing using the parallel implementation

\[
\text{index}(\text{DirName}) \rightarrow \\
\quad \text{NumberedFiles} = \text{list_numbered_files}(\text{DirName}), \\
\quad \text{mapreduce}(\text{NumberedFiles}, \text{fun find_words/3}, \\
\quad \quad \text{fun remove_duplicates/3}).
\]

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

\% the Reduce function
\[
\text{remove_duplicates}(\text{Word}, \text{Indices}, \text{Emit}) \rightarrow \\
\quad \text{UniqueIndices} = \text{sets:to_list}(\text{sets:from_list}(\text{Indices})), \\
\quad \text{lists:foreach}(\text{fun (Index) \rightarrow Emit(Word, Index) end, UniqueIndices}).
\]
Text indexing using the parallel implementation

```prolog
> 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"]}
```
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

  • Didactic implementation

  • Simple idea, arbitrarily complex implementations