References

Programming Erlang: Software for a Concurrent World, Armstrong, Chapter 20


How to use Multicore CPU

Use lots of process
Avoid side effect
Avoid sequential bottlenecks
Write "small message, big computations" code
MapReduce

Introduced by Google
Performs parallel computations over large data sets on clusters
Used to update Google's indexes
Word Count
Count how many times each word appears in set of documents

Document 1

The cat in the hat

Document 2

The cat comes back

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>3</td>
</tr>
<tr>
<td>cat</td>
<td>2</td>
</tr>
<tr>
<td>in</td>
<td>1</td>
</tr>
<tr>
<td>hat</td>
<td>1</td>
</tr>
<tr>
<td>back</td>
<td>1</td>
</tr>
</tbody>
</table>
Using MapReduce

The cat in the hat

Map

{the, 1}
{cat, 1}
{in, 1}
{the, 1}
{hat, 1}

Reduce

{the, 3}
{cat, 2}
{in, 1}
{comes, 1}
{hat, 1}
{back, 1}

The cat comes back

Map

{the, 1}
{cat, 1}
{comes, 1}
{back, 1}

Reduce

{the, 3}
{cat, 2}
{in, 1}
{comes, 1}
{hat, 1}
{back, 1}
-module (word_count).
-export ([count/1]).

count (StringList) ->
    ets:new(counts, [set, private, named_table]),
    lists:foreach(fun count_words_in_string/1, StringList),
    Result = ets:tab2list(counts),
    ets:delete(counts),
    Result.

count_words_in_string (String) ->
    Words = string:tokens(String, " "),
    lists:foreach(fun add_word/1,Words).

add_word([]) -> void;
add_word (Word) ->
    case ets:member(counts, Word) of
        true -> ets:update_counter(counts, Word, 1);
        false -> ets:insert(counts, {Word,1})
    end.
Using Word Count

31> word_count:count(["the cat in the hat", "the cat came back"]).
[{"the",3},
{"in",1},
{"hat",1},
{"back",1},
{"came",1},
{"cat",2}]
lists:foreach

foreach(Fun, List) -> void()

4> Numbers = [1, 2, 3].
[1,2,3]

5> Output = fun(X) ->io:format("~p ", [X*2]) end.
#Fun<erl_eval.6.13229925>

6> lists:foreach(Output,Numbers).
2 4 6 ok

foreach relies on side effects. It passes one at a time each element of List as an argument to Fun.
lists:foldl & foldr

foldl(Fun, Acc0, List) -> Acc1

11> Numbers = [1, 2, 3].
[1,2,3]
12> Adder = fun(X,PreviousSum) -> X + PreviousSum end.
#Fun<erl_eval.12.113037538>

13> lists:foldl(Adder,0, Numbers).
6

14> Doubler = fun(X,List) -> [X*2 | List] end.
#Fun<erl_eval.12.113037538>

15> lists:foldl(Doubler,[], Numbers).
[6,4,2]
16> lists:foldr(Doubler,[], Numbers).
[2,4,6]
MapReduce from Text

Illustrates the idea of mapreduce

Does not have features, performance and robustness of Google's version

Example of word count in erlang source code for book
Preliminaries

Creating list of words from a file

%% evaluate F(Word) for each word in the file File
foreachWordInFile(File, F) ->
    case file:read_file(File) of
        {ok, Bin} -> foreachWordInString(binary_to_list(Bin), F);
        _         -> void
    end.

foreachWordInString(Str, F) ->
    case get_word(Str) of
        no ->
            void;
        {Word, Str1} ->
            F(Word),
            foreachWordInString(Str1, F)
    end.
get_word

get_word([H|T]) ->
    case isWordChar(H) of
        true -> collect_word(T, [H]);
        false -> get_word(T)
    end;
get_word([]) ->
    no.

collect_word([H|T]=All, L) ->
    case isWordChar(H) of
        true -> collect_word(T, [H|L]);
        false -> {reverse(L), All}
    end;
collect_word([], L) ->
    {reverse(L), []}.

isWordChar(X) when $A=< X, X<=$Z -> true;
isWordChar(X) when $0=< X, X<=$9 -> true;
isWordChar(X) when $a=< X, X<=$z -> true;
isWordChar(_) -> false.
Calling mapreduce

test() ->
    wc_dir(".").

wc_dir(Dir) ->
    Map = fun generate_words/2,
    Reduce = fun count_words/3,
    Files = lib_find:files(Dir, "*.erl", false),
    L1 = phofs:mapreduce(Map, Reduce, [], Files),
    reverse(sort(L1)).

generate_words(Pid, File) ->
    F = fun(Word) -> Pid ! {Word, 1} end,
    lib_misc:foreachWordInFile(File, F).

count_words(Key, Vals, A) ->
    [{length(Vals), Key}|A].
Defining mapreduce

%% Map(Pid, X) -> sends {Key,Val} messages to Pid
%% Reduce(Key, [Val], Accln) -> AccOut

mapreduce(Map, Reduce, Acc0, L) ->
  S = self(),
  Pid = spawn(fun() -> reduce(S, Map, Reduce, Acc0, L) end),
  receive
    {Pid, Result} ->
      Result
  end.
reduce(Parent, Map, Reduce, Acc0, L) ->
    process_flag(trap_exit, true),
    ReducePid = self(),
    %% Create the Map processes
    %% One for each element X in L
    lists:foreach(fun(X) ->
        spawn_link(fun() -> do_job(ReducePid, Map, X) end)
    end, L),
    N = length(L),
    %% make a dictionary to store the Keys
    Dict0 = dict:new(),
    %% Wait for N Map processes to terminate
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(Reduce, Acc0, Dict1),
    Parent ! {self(), Acc}.
Collect Replies

collect_replies(0, Dict) ->
    Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} ->
            case dict:is_key(Key, Dict) of
                true ->
                    Dict1 = dict:append(Key, Val, Dict),
                    collect_replies(N, Dict1);
                false ->
                    Dict1 = dict:store(Key,[Val], Dict),
                    collect_replies(N, Dict1)
            end;
        {'EXIT', _, Why} ->
            collect_replies(N-1, Dict)
    end.