Concurrent Brzozowski DFA construction using Erlang

and how it turned out to be a Google MapReduce algorithm

Hendrik Visagè
SN:91211108

Monday 1st November, 2010

Abstract

This study researched Erlang, a functional language, using a Brzozowski DFA construction algorithm to firstly evaluate Erlang's concurrency features and secondly how much of the Brzozowski DFA construction can be optimally parallelized using Erlang. The algorithms proposed and used, will be shown to be equivalent to the Google MapReduce algorithm. The conclusion of this study was the Erlang's language constructs and concurrency is very expressive and its concurrency is very easy to achieve. However, the actual Brzozowski algorithm did not achieve any speed improvements.

Contents

Listings

Chapter 1

Raison d'etre

1.1 Becoming intrigued

Earlier this year, the author noticed several job advertisements for Erlang programmers, on a South African website. Investigating this further, the author became quite intrigued by the Erlang language, which claims easy concurrency and high availability. What intrigued the author even more, was the fact that Erlang is a functional programming language, and the author have always put of learning a functional language (like Haskell) for various unrelated reasons. These facts alone are not good enough reasons to learn a language, especially when there is no task or project per se to code in the language. For that we need to find a reason to do a task in Erlang. But before we get to that problem statement, let us examine some aspects of the current state of Computer Science.

1.1.1 Today's CPUs

Concurrent algorithms are becoming more important lately as the commodity CPUs shipped on laptops and desktops, are nearly without exception multi-cored or multithreaded. [**?**] already claimed in 1985 that massive parallelism is the future of computing, and it obviously have become a trend with CPUs when Intel's Hyperthreading, Sun's CoolThread CPUs and multi-core AMD Opteron CPUs got introduced.

The idea is to rather have more processing units available than to try and raise the core clock speeds. This apparently helps to keep CPUs cooler, and provide users more processing power. The problem with pushing the GigaHertz, is that we are entering light speed limits, as well as power consumption problems.

However, this necessitates the need for concurrent algorithms and parallel processing to be able to effectively and efficiently use the processing power available in these processing units, as the core clock is not faster, but the

number of processing units increased. On the extreme end today, Oracle[1] in [Ora], claimed their SPARC T3, does 128 threads across 16cores on a 1.65GHz CPU socket.

1.1.2 Determinate Finite Automata (DFA)

DFAs as a matching algorithm have big importance in the matching of patterns. These patterns could be virus signatures, DNA or even network based intrusion detection and prevention. These DFAs are constructed from regular expressions and as these regular expressions become more complex and extended, it is but natural to ask how the DFA construction could be made faster using the available multi-core and multi-threaded CPUs, and that is the reason for the research into the concurrency of this algorithm.

We will have to point out that in this research, we focussed on the construction of the DFA from an expression, and not the actual DFA application to the data to be matched, nor will this study look into the parsing of the regular expressions into expressions useful for our DFA construction.

1.1.3 Erlang

Erlang in a functional language and for programmers used to procedural languages, there is a couple of interesting features (or some might say annoyances) that would make it at least a learning experience to gauge the language. Armstrong boosts about Erlang's built-in concurrency features, and this would be a perfect match to test both the language on multicore CPUs.

1.2 Research focus

The problem this study addresses, is some research into the concurrency possibilities of the sequential Brzozowski algorithm and to implement this in Erlang. This way we will be combining the concurrent features of todays CPUs, the Erlang language that boasts about its concurrency features as well as the quest for a concurrent Brzozowski DFA construction.

The rest of this document will first look at Erlang and its feature set and what makes Erlang (and to some degree functional languages) different from other programming languages especially its support for concurrency. Then we will look at the Brzozowski sequential algorithm and some parallelization proposals. Lastly we will discuss the Erlang implementations for the parallelizations proposals.

[1] previously Sun Microsystems

Chapter 2

Erlang - the language

2.1 Introduction

In this chapter, we will give some brief overviews to Erlang's history and it's language constructs. We will also explain those constructs and ideas that we have used in this project as well as those constructs that is not obviously the same as in other computer languages like the C/C++ languages. For the purposes of explanation in later parts, we will briefly give an Erlang language introduction to some peculiarities. However, for proper detailed explanations we will refer the reader to [Arm07].

2.2 Brief history

Armstrong [**?**] gives a detailed historical overview of Erlang since it's inception in 1986 till his thesis circa 2001. In summary it started in 1981 from the goal *"to suggest new architectures, concepts and structures for future processing systems development"*. Armstrong started to work on adding concurrent processes to Prolog and the language evolved away from Prolog and evolved with its own abstract machines (virtual machines in the JAVA terminology). Erlang have been used in the AXD301 switch which superseded the failed AXD-N switch and it is claimed to have a **NINE** nines[1] uptime in production, while still having around 2million lines of Erlang code.

The main theme found in Erlang is to have small parts that shares nothing while working together, and if any of these small parts experience problems it can not handle, it rather fail fast and let a supervisor process handle the failure and to restart the failed process.

This fail fast is especially nice as there is not suppose to be any shared memory between processes/parts, which means that a failure in one process

[1] 99.9999999% where the usual target for mission critical systems is 5 nines (99.999%) while vendors (from the authors experiences) do not easily target nor claim uptimes higher than 95%

should not impact the other processes by corrupting shared memory. This is quite a different approach from other threading models like the C based Posix threads, where process memory (and thus variables) are shared, and thus have a need for locks and other mutual exclusion methods to prevent the threads from concurrently accessing memory, and thereby corrupting data.

The author would like to point out that this is different from guaranteed and proven correctness used in software development for critical software used in applications like the space shuttles that can not tolerate any glitches, where as the Erlang model tolerates the glitches by restarting the processes.

2.3 Quick Language Introduction

In this section, we will briefly introduce the reader to the Erlang language. This should be sufficient to be able to grasp the code presented in this research, and it will not be a detailed reference. The reader are referred to Armstrong[Arm07] or O'Reilly[**?**] for further in depth explanations and references to the Erlang language.

2.3.1 atoms and Variables

Erlang distinguishes between atoms and variables mostly by the first character being uppercase for variables or a lowercase character for atoms.[2]

Erlang's atoms are similar to C/C++ `enums`, just more generic and not typed like the C/C++ enums which are but typed numbers.

2.4 Functional language features

This section we will briefly glance over some of Erlang's peculiar[3] language features to give the reader a grasp of the expressive power that helped to produce the programs in such short time.

2.4.1 Pattern Matching - function overloading

One of the strengths of Erlang (and the author understood other functional languages too, but have not investigated that) is the way pattern matching is used for code flow paths. Program listing Program 2.4.1 shows this feature with the two functions `area/2`, `area/3` and `area/4`. Remember the atoms start with lowercase letters while the Variables that gets bound to a value, starts with an uppercase letter.

[2] Yes, there are exceptions but that means quoting etc. which have not being used in our code

[3] compared to the C and other procedural type languages

Program 2.4.1 Pattern matching in code flow

```
1  area(square,Side) -> Side*Side;
2  area(cube,Side) -> area(square,Side)*6;
3  area(circle,Radius) -> area(circle,radius,Radius).
4
5  area(circle,radius,Radius) -> Radius*Radius*3.14;
6  area(circle,diameter,Diameter) -> area(circle,radius,Diameter/2);
7  area(triangle,Base,Height) -> Base*Height/2;
8  area(rectangle,Height,Width) -> Height*Width.
9
10 area(box,Height,Width,Depth) -> ((Height*Width) +
11       (Height*Depth) + (Width*Depth))*2.
```

As could be seen in this example, that we used multiple functions (and have them match based on the parameters) rather than having if-then-else or case/switch statements to make code flow decisions. The different distributor states is also handled using these parameter matching.

Guards Another code flow technique is the use of guards (`when` statements) inside functions. These help firstly with pre-conditions (ie. to force only accepting valid values, like positive values for distance), and secondly with another method of controlling the flow of code, but only after the parameters have been matched (Ie. a parameter could match anything, but we want to handle the circle and square different).

These same pattern matching and guards is extended to the message receiving discussed in section subsection 2.5.1 and shown in program Program 2.5.1

Something else to note here, is that the underscore denotes a parameter who's value will be unbounded and ignored. Sometimes a variable with a prepended underscore would be a way to name a variable that would not be used, to prevent compiler warnings.

Notation of functions A convention in the Erlang texts, is to refer to **module:function/arity** for a function for example `lists:map/2` which is read as

module lists

function name map

arity taking 2 parameters

2.4.2 Functions as first class members

By definition a function in a functional language is a first class member, where a function can be passed around like a variable. This do allow for interesting concepts where you have a function definition inside a function call, for example to map a list to its squares, we use something like :

```
1 lists:map(     fun(X) -> X*X end,
2                [1,5,3] )
```

Here we provide a list with elements `[1, 5, 3]`, and `map/2` take each element of that list, apply the provided function (in this case `fun(X) -> X*X`) to that element, and returns a list with the new values `[1, 25, 9]`.

2.4.3 Immutable variables

Variables in Erlang is like algebraic variables that have a fixed value during a run of a function block. For example, once you have bound $X = 1$ and then evaluate $Y = X + 2$ we will have $Y == 3$ and we can not have $X = 2$, later on in that run as $X == 1$ from the first assignment. This prevents side effects from C/C++ constructs like `y=x++`.

The other term that is used instead of assignment, is binding, as a variable gets bound to a value, can can't be unbounded to take on a new value during that run.

Having programmed mostly before in procedural C-type languages, this feature of functional languages have initially had an annoying impact on the thought pattern when trying to grasp the workings of the language, but once grasped the author found it to be natural while programming in Erlang.

2.4.4 Tail recursion

Tail recursion is achieved when the compiler can optimize the code to be a `goto/jump`[4] back to the beginning of the function, perhaps with new parameters. This way there is no returning stack needed that would build up.

One of the important reasons for this feature, is that we can write infinitely recursive servers (functions) without having any memory leaks. This will be shown in some of the techniques used to produce our distributor and receivers in section 3.4.1 and section 3.4.1 without stack space being used.

Program Listing 2.4.2 shows proper tail recursion examples, where the last instruction calls in a flow to `loop/1` is tail-calls.

[4] Yes we all *know* that is a BadThing™ but still CPUs consistent of those instructions and here is a nice GoodThing™ use for them

Program 2.4.2 Right Tail-Recursion

```
1 loop(0) -> true;
2 loop(N) when N > 0 ->
3  io:format("␣iteration:␣~p␣~n",N),
4  loop(N-1).
```

Program Program 2.4.3 show two cases where it is not possible to use tail recursion by the compiler. The first `loop/1` is called before the output, and this means that it needs to return to that spot to do the rest of the work in that function. The `factorial/1` function also needs to return a value, so yet again this is not proper tail recursion and would need to be rewritten for tail recursion.

Program 2.4.3 No Tail-Recursion

```
1 loop(N) when N > 0 ->
2   loop(N-1),
3   io:format("␣iteration:␣~p␣~n",N).
4
5 factorial(0) -> 1;
6 factorial(N) -> N*factorial(N-1).
```

Armstrong ([**?**] and [Arm07]) as well as Cesarini and Thompson [**?**], have in depth discussions and examples related to tail recursion, but for the purposes of this project, the above will suffice.

2.5 Concurrency and distributed programming

Armstrong[**?**] coined the phrase *Concurrency Oriented Programming* to describe how Erlang helps a program to be structured around the concurrency of the application. Armstrong[Arm07] also states that the world is a concurrent place and that in the real world, even though we do things concurrently, we do not share memory as do most threading models in languages like C/C++. As such Erlang is structured so that no process share memory with another process.

What makes this idea of *share nothing* powerful, is that Erlang implements the messaging communication such that both concurrent and distributed processes, communicate in the exact same way. In other words, once you know have the reference PID of the process on the remote node, you can sent a message to it as if it is local, and the response from the remote process can come back to you , without the remote processes knowing whether a local or remote process messaged it.

To create a process[5] in Erlang, we use the `spawn(Fun) -> Pid()`, and to start it on a different (connected) node we use `spawn(Node,Fun)-> Pid()`[6]. As can be seen, both returns a PID to be used for checking and for messages sent to the processes. This makes starting a process locally or distributed just a matter of specifying where, rather than several elaborate methods.[7]

Thus once we have a local concurrent system running, the scaling to a distributed concurrent system would be just adding the node information. Given the ease that we have been able to write a concurrent version we will attempt to do a distributed version too.

2.5.1 Communications

In the real world we use messages to communicate. We also choose to ignore some and to give priority to others. This is the way Erlang processes communicates with each other, by using messages in the same fashion. As we will show later in the code we developed, the processes choose which messages they are interested and even give priority to specific messages.

To communicate with fellow processes, Erlang use asynchronous message passing. This is similar to Ada's rendezvous, but different as the sender do not wait for the receiver to receive, acknowledge nor return a value.

This is so... real world. It is very much like a snail mail letter thrown into a post box... sent and forget.

The receiver will wait only for messages in specific formats, much like the matching of the function parameters in section subsection 2.4.1 and program Program 2.4.1, else it will ignore the message. This "wait till right message" is used later in the AsAvailable distributor (section section 3.4.1) where the distributor will wait for messages from receivers that is available, before it will accept and handle a processing request message. Program Program 2.5.1 shows an example where we start in a waiting state with `loop/1`, and after all the available PIDs have been exhausted (lines 9-11) we go back to that state. In this waiting state we do not care about any `process` messages, as we can not process them without available processors in any case, so we only look and wait for `available` messages. For as long as we have available processors (either more than one in `[Head|Tail]=WholeList` or a single one in `Head` see section subsection 2.5.2), we accept both the `available` and `process` messages on a first come first serve basis.

The author's opinion is that this is one of the best methods of inter process communications, as there are just about no real lock contentions,

[5] An Erlang process is more a light weight thread as it runs inside the VM/Abstract machine

[6] I will exclude the more specialized `spawn_link` and `spawn/3, spawn/4` as they work mostly the same way, just having more tunables

[7] granted the code have to be residing on and available on the different nodes

Program 2.5.1 Receiving messages and timeouts in Erlang

```
1  start_loop() -> loop(waiting).
2  loop(waiting) ->
3    receive
4      {available,PID} -> loop(available,[PID])
5    after 5000 -> throw({timeout_error})
6  end.
7  loop(available,[Head|[]]) ->
8          receive
9                  {available,PID} -> loop(available,[PID|Head]);
10                 {process,{params}} ->
11                                         Head ! {Params},
12                                         Loop(Waiting)
13         after 5000 -> throw ({timeout_eror})
14 end;
15 loop(available,[Head|Tail]=WholeList) ->
16   receive
17     {available,PID} -> loop(available,[PID|WholeList]);
18     {process,{Params}} ->
19           Head ! {Params},
20           loop(available,Tail)
21   after 5000 -> throw({timeout_error})
22 end.
```

and dead lock situations can be easily eliminated (as program Program 2.5.1 shows) the `after` clause that will handle the case when the process have waited too long and none of the right message(s) have arrived.

Process Identifiers and nodes

In Erlang the destination of a message, is the PID (Process Identifier) of the process. Thus each process have it's own message queue, and it is not something special of a process, it is integral inter process communication mechanism for Erlang. Thus all you need to know of a process, is it's PID, and you can sent it messages (whether the process will respond to it, depends entirely on the receiving process)

The reader would be justified to ask what would happen in distributed environments? The answer is simply that a receiving process do not see a different message! Even the sending process still have a PID and sends to that PID the same as before! The reason for this is that the PID data structure already contains the node destination in it.

The only difference is when you spawn a new process, the programmer can tell it where to spawn it, ie. if you want to spawn it on a different node, (instance of the VM) you just use the spawn function that includes the node it needs to spawn the function on. Again refer to [Arm07] for more in depth

explanation, as again we have not ventured in this research to a distributed model, but have to mention that if it were needed, it would not have been a significant change in the code.

Guards in receiving messages

Although none of our code used the guard statements, it have to be noted that it is one of the nice features of Erlang as mentioned in paragraph section 2.4.1. A quick example should suffice for our brief introduction for the reader to compare `loop/2` in program Program 2.5.2 using a guard (the `when` clause) versus the two separate functions (differentiated using the matching of `Tail` not an empty string) in program Program 2.5.1

Program 2.5.2 `loop/2` using guards

```
1  loop(available,[Head|Tail]=WholeList) ->
2    receive
3      {available,PID} -> loop(available,[PID|WholeList]);
4                  {process,{params}} -> when Tail =:= []
5                                          Head ! {Params},
6                                          Loop(Waiting)
7      {process,{Params}} ->
8            Head ! {Params},
9            loop(available,Tail)
10   after 5000 -> throw({timeout_error})
11 end.
```

2.5.2 Parameter List splitting

Program Program 2.5.1 shows another parameter feature that is quite frequently used in Erlang, that being of the splitting of the head (first element) and tail (all BUT the first element) of a list. Also note on the one side the list is split, but the other side we have the while list.

This example could've been rewritten to take the guard

Chapter 3

Brzozowski's DFA construction

3.1 Origins of algorithm

In [Brz64], Brzozowski presented the notion of derivates for regular expressions, and showed how that leads to the construction of a state diagram from recursive derivation of a regular expression. Watson in [Wat95], shows several FA constructions, including Brzozowski's, in generic mathematical presentations. This should help implementors (like programmers and algorithm designers) decide the algorithms to use and be able to implement it in the language they need. Watson was also used and referenced by the implementors of the sequential Erlang implementation, which was used as basis for this research.

3.2 Sequential algorithm

ProgramProgram 3.2.1, shows a Guarded Command Language version of Brzozowski's DFA construction algorithm. This is copied from [SKWH08], with comments inserted to ease the discussions. Based on a brief glance over the code below (and in the source code we used as basis) it appears to be direct implementation of the algorithms presented in [Wat95]. As such the code is considered to be functionally correct and we have not engaged in correctness proving of it.

We will now give a look at this algorithm, and look at how and where this code could be parallelized.

3.2.1 Reduced derivatives

When looking at this algorithm, the only dependency or shared state between iterations and derivatives, is the adding and removal of the derivatives

Program 3.2.1 Brzozowski GCL [SKWH08]

```
func Brz(E, Σ) →
    δ, S, F := ∅, {E}, ∅;
    D, T := ∅, S;
    do (T ≠ ∅) →
        let q be some state such that q ∈ T
        D, T := D ∪ q, T\{q}
        for (i : Σ) → #Inner loop
            d := (d/di) q #Reduced-derivation
            #Already inserted this (d/di)?:
            if d ∉ (D ∪ T) → T := T ∪ {d}
            [] d ∈ (D ∪ T) → skip
            fi
            δ(q, i) := d; #Path insert equivalent to δ(q, i) := (d/di) q
        rof
        #Nullable tests :
        if ε ∈ L(q) → F := F ∪ {q}
        [] ε ∉ L(q) → skip
        fi;
    od;
    return (D, Σ, δ, S, F);
```

to T. This is done in two places, the first is when a derivative is removed from the list/set when any q is taken from T and added/moved to D. The next place is when the newly derived $\frac{d}{di}q$ is checked for existence in $(D \cup T)$ and added to T if not. These two actions should either be atomic or inside critical areas if done through concurrent processes.

3.2.2 Path insertion

The path insertion $\delta(q, i) := d$, again is a critical/serial operation that is effectively just a collection of the $RE, i, \frac{d}{di}RE$ tuples, indexed on the RE, i key. Thus this is not easily parallelized.

3.2.3 Nullable tests

The nullable tests ($\epsilon \in \mathcal{L}(q)$) is an independent once we have the list of reduced-derivative REs (In the code it is the qs). This can be executed in parallel with others.

3.2.4 Sequential implementation

As mentioned in section 3.1, we started with an already implemented sequential Erlang implementation. This made use of Erlang's `lists:mapfoldl/3`, which is similar to `lists:map/2` discussed in subsection 2.4.2, but instead of returning a list, an additional function is applied, that have an accumulator updated as the items are processed. The base implementation used the Σ as the list to process, and a function to do the $\frac{d}{di}RE$, and then adding that derivative into the D list being the accumulator. This a very efficient way of coding it in Erlang and a commendable method in the sequential case!

3.3 Concurrent algorithms

3.3.1 First consideration: ParMap

The first obvious parallelization method comes from doing concurrency over the Σ alphabeth on the inner loop. This is also an easy method as the sequential algorithm makes use of `lists:mapfoldl/3`.

The sequential code use the provided `lists:mapfoldl/3` function. This provided a function to be mapped over the Σ alphabeth list. The function is constructed with the RE/q to be derived and it is given an accumulator parameter. In this implementation, the accumulator is the $\delta()$ storage. The output in the base implementation is then a list of reduced-derivatives which then is uniquely sorted with `lists:usort/1`[1] and already handled derivatives (those in D set) removed and then uniquely merged with the to-do list T.

The first parallelization attempt was to make use of a parallelized-map function as described in Armstrong[Arm07] and then do the fold operation on the received messages. This will spawn a process[2] for each of the Σ_i and then to collect the various reduced-derivatives.

This method is an easy picking, but the granularity is spread over the alphabeth size. In other words with $l = size(\Sigma)$ there will be l processes processing the same RE, and then we will collected all of them (adding to δ, F, T as the messages arrive) and only after all of the l messages have been received, will another set of l processes be spawned. in short it will have bursts of requests, not a queue, which could cause thrashing.

It should further be obvious that a small l will have little concurrency, while a big l might be too much, thus we (as programmer or algorithm designer) have no control over the amount of parallelization, other than the

[1] duplicates removed

[2] Remember erlang processes is not Unix processes, but rather threads inside the virtual machine

size of the Σ alphabeth, which is outside of the programmer's control, but the user's prerogative.

It has to be noted that this was the author's first consideration during the literature study on the Erlang concurrency model as [Arm07] have a easily understood example for `parmap`. Looking back, This might be a faster implementation with lower overhead than the next revisions, but have not been considered for this lack of concurrency control.

3.4 Distribution queues

Addressing the uncontrolled parallelization problem mentioned in subsection 3.3.1, the idea formed to have a central distributor, that will dispatch the processing requests to processing threads. This is not a new idea *per se*, but the author would lie not to mention that the Apple MacOSX 10.6 (Snowleopard)'s Grand Central Dispatcher (GCD) have been a influence for the choice.

In simple terms, a couple of processing threads (or Erlang processes) are started, and then based on a chosen algorithm, the work submitted to the distributor gets sent out to the various processing threads.

3.4.1 First distributor

After considering the issues mentioned in subsection 3.3.1, a distributor is shown in figure Figure 3.1. What needs to be pointed out, is that `mapfoldl/3` or rather any `lists:map/2` was not usable in the same way as done in the serial implementation, and thus `lists:foreach/2` was chosen to iterate over the Σ alphabeth to generate and send the messages to the distributor. Looking back, a `lists:map/2` could have been used, but would have needed yet another paradigm and though pattern shift.

Diagram notation used

The LaTeX symbols used in the text, where not possible to be imported into the UML editor used by the author. For that reason we just give a short mapping between the symbols used in the text and those in the diagrams:

RE the original regular expresion

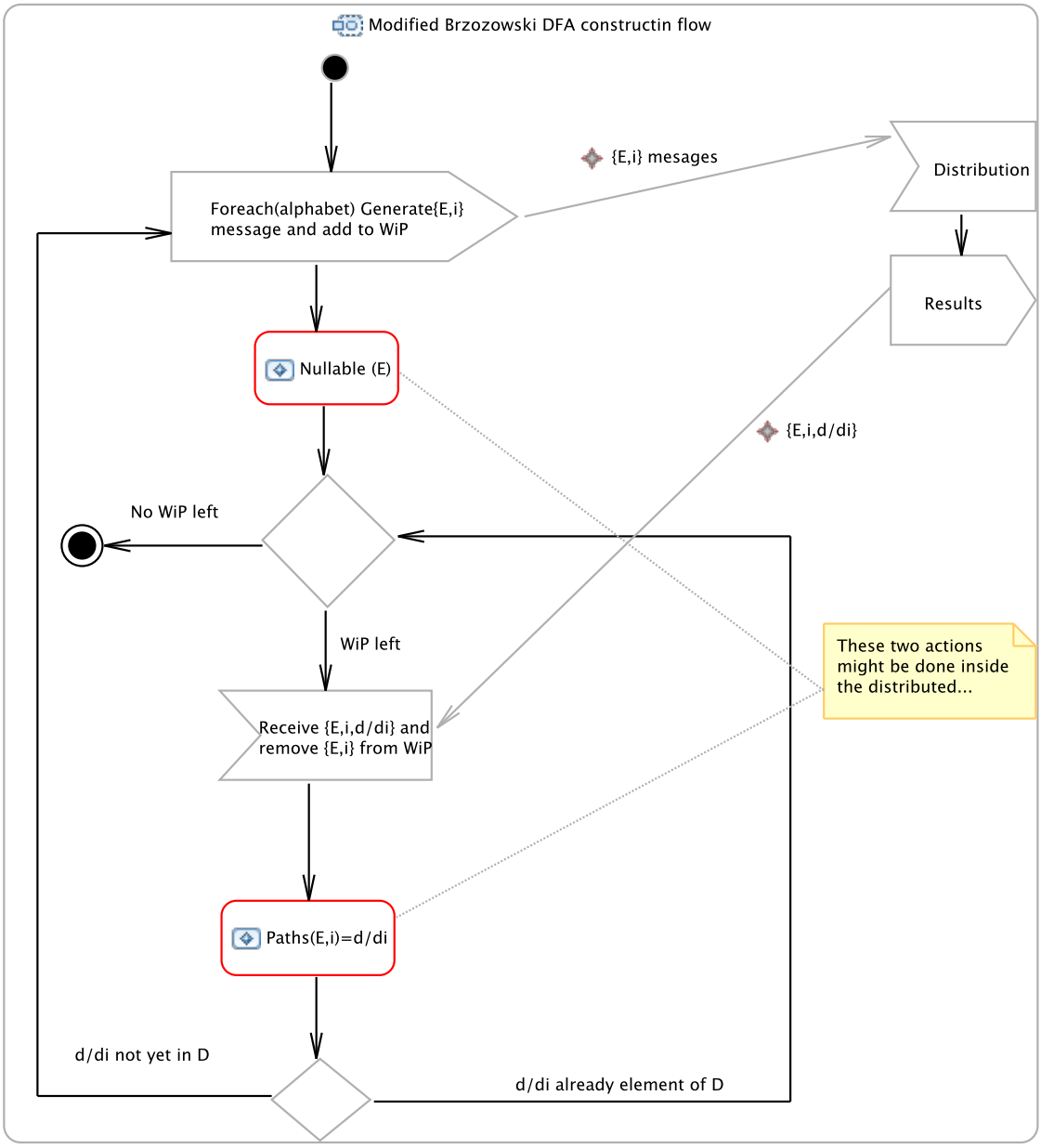
E an expresion, could be the original RE, or part of the RE, in other words a derivative.

$\{E, i\}$ an expresion together with an $i \in \Sigma$

d/di the reduced derivative of E, ie. $\frac{d}{di}E$

WiP Work in Progress - Those messages not yet received.

Figure 3.1 Flow for distribution of $\frac{d}{di}$



Paths(E,i)=d/di the $\delta(E,i) = \frac{d}{di}E$

Nullable(d/di) ie the F list containing $\epsilon \in \mathcal{L}(q)$

Flow description

The sequential algorithm put the original RE on the Todo list T. Then it handles the RE as it would handle the derivatives found. In this algorithm we do something similar, as there is no differentiation between the RE and the $\frac{d}{di}E$s in the first step. This would seen similar to recursiveness of the derivations used in the description of this Brzozowski algorithm in [Brz64]

Core Loop The inner loop for the sequential algorithm is a creation of messages to be send for processing. These messages E, i consists of the E and the letter (i) of the alphabeth (Σ) to derive from.

We also put those messages send in a WiP (Work in Progress) list to keep track of those messages send. We then remove those received, as we do not have any guarantees on the order of messages received given the inherent asynchronous nature of concurrency.

Note: Message parallelization It has to be noted that this algorithm is not concerned with the parallelization of those messages and will not consider it here, as it would be a function and optimization of the distributor. At this point the emphasis will be on the correctness of this algorithm as the distributor will be discussed separately in section 3.4.1.

Nullable($\frac{d}{di}$), Add E to D While writing this and considering the formal aspects to proof the correctness of this algorithm, the Nullable($\frac{d}{di}$) issue needs to be considered in more detail. In the sequential algorithm, this was done at the end of each outer loop. In this algorithm it is also outside and after the "inner loop" but before any of the derivatives are handled. In essence the nullable(E) is handled whenever we try to get more derivatives for an E.

E is also added to D, ie. $D := D \cup E$, to prevent any similar $\frac{d}{di}E$s to be skipped.

WiP test Check for an empty WiP list. If it is empty this process will terminate (perhaps also telling the distributor?). If there is still messages on the WiP list, continue to the receive section.

Receiving $E, i, \frac{d}{di}E$ and $\delta(E,i) := \frac{d}{di}E$ Once a message is received, the corresponding E, i is removed from the WiP list. The Paths is then updated by adding the received $\frac{d}{di}E$ using the $\delta(E,i) = \frac{d}{di}E$ expresion.

Checking $\frac{d}{di}E \in D$ The last step of this algorithm is to check whether the received $\frac{d}{di}E$ have already been considered. This is done by checking the D list. If it has been considered before, the algorithm loop back to the receiving portion, else it loops to the messages generation portion so that this new E could also be processed.

no T todo list, but WiP Note that there is no Todo list (the T) as in the sequential case. This is because the algorithm immediately generates messages for those Todo and put them on the D list. There is however a WiP list that serves the same termination condition as the T todo list in the sequential algorithm.

Thus, the need for the ToDo list of the sequential operation, is replaced with the WiP list in the concurrent algorithms presented here.

Distributors

Based on the stream of messages that the algorithm generates, there is various methods how these messages could be handled, and we coded and researched three options here.

Sequential As a first test to confirm the correct algorithm in at least the sequential case (or the messages all getting processed in the same order as the sequential algorithm), each message received will be processed and the result sent back without any concurrency. This could also be implemented as a single process case of the Round Robin (section 3.4.1) and As-Available(section 3.4.1) distributors, but for simplicity, the decision was made to implement this as a single instance Round Robin case.

Round Robin A Round-Robin processing, is done when you have more than one queue, with a pointer to one of these queues. When a request comes in, it is sent to the queue being pointed to, and then the pointer is advanced to the next queue. This pointer switches to the next queue and assign without regard for queue lengths, and assumes a normal time/size distribution for the requests. It is a simple mechanism, but could cause a single big or time consuming request, to clog a queue and those requests waiting in that queue, in respective of the availability of other queues to process these requests. This similar (but not quite) as picking a pay-point in your supermarket.

In our implementation, the distributor will be given a list of processes that have been spawned and will handle requests. The first one on the list (head of list) will be sent the next available message. This process will then be added to the back of this list and the process repeated for each request sent to the distributor. As would be gathered from the discussion above,

the queues would be at the processing thread side, and not the distributor side.

As-Available The problem mentioned with the Round-Robin distributor, is the fact that a single request could stall requests that could otherwise be processed on available processors, and thus the reason for the AsAvailable distributor. This distributor would have the requests queue at the distributor side, and sent out the requests to those processors that are available.

This is similar to the snake queues at banks, where everybody gets in the same queue, but the person in front gets assigned to the next available teller, thus the queue is at the distributor, not at the teller!

Figure 3.2 State engine for the AsAvailable distributor

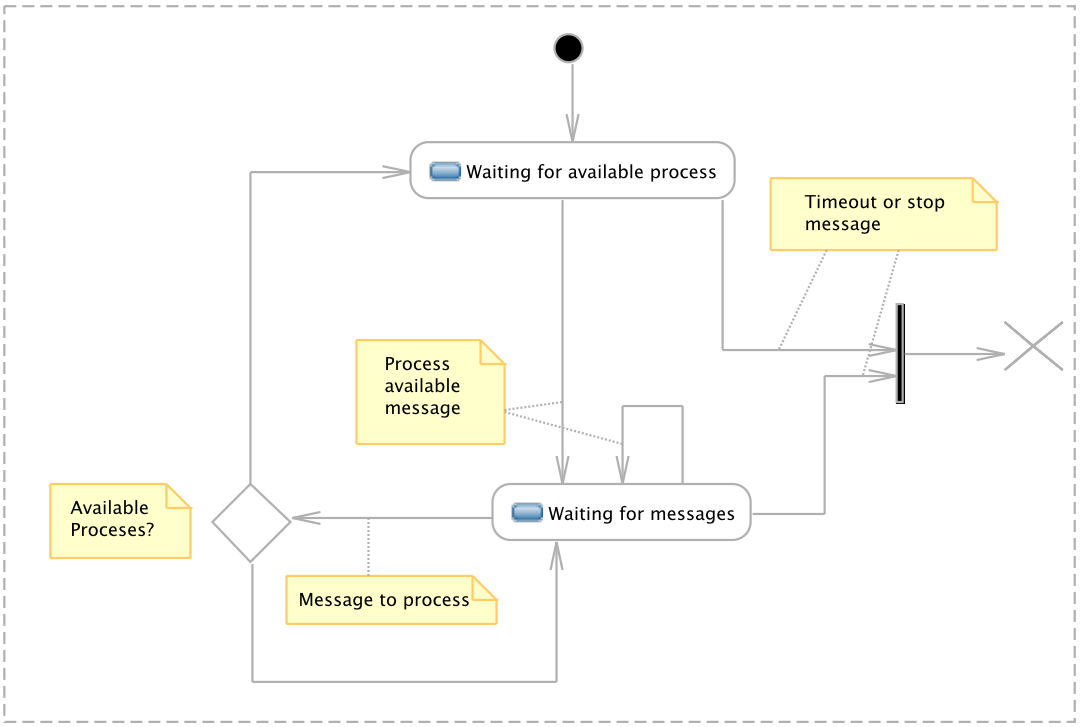


Figure Figure 3.2 shows the state diagram for the As-Available distributor. The distributor will start with no available processes in the WaitingForProcesses state and will wait until a message from a processing thread about its availability. Once it receives an available process message, it will move to the ProcessesAvailable state, where it will wait for both processing and availability messages.

When the distributor receives messages for processing, the distributor will remove the first process (again the head of the list) from the available

list and sent it the message to be processed. The distributor then continue the loop with the tail of the list (minus the process that were sent a message). The distributor stays in this state while it still have processes available, but when it do not have any available processes, it will move to the WaitingForProcesses state.

When a process/thread finished it processing, it will inform the Distributor that it is again available for processing. The distributor will add this to the head of the list of available processes and repeat the ProcessesAvailable state loop.

3.4.2 Nullable also?

At this point of the project's implementation, the author were so astonished by the ease of implementing the concurrency, that the author reconsidered the algorithm and noticed that the `nullable()` part also seemed to be distributable since it does not depend on anything else. After analysis of the nullable implementation from the original code base and having looked at [Wat95], it was concluded that there is no need to keep it inside the inner loop, as it is an independent computation.

The new control and data flow is shown in figure Figure 3.3 and we will explain this code module in more detail, in chapter **??**. Note that the messages had to be augmented to allow for the differentiation of the `reduce(derive(E,i))` and the `nullable(E)` calculation requests. To be honest, it is not strictly needed in this system, as a simple match for {`E`} versus a match for {`E,I`} would have been sufficient. However, I would rather add this functionality, as it would help make the distributor-receiver pairs to be more easily extended and the same distributor-receiver pair be usable by different mappers.

The author made a choice to have the receiver handle both the `reduce(derive())` and `nullable()` computations, as it would keep the distributor simple, but having a seperate receiver(s) for each would not be that difficult to add for Erlang.

In this version, we have moved all the computational parts out of the core loop and delegate it to the receivers. The core loop now only aggregates the results and distribute any results that need to be distributed.

3.5 Map Reduce - the Google connection

After implementation of the second and third versions, a rereading of [Arm07] brought theauthor to Google's MapReduce and a nice figure that explains map reduce. Further researching Google's MapReduce, [**?**] shows how to use MapReduce for counting words in a distributed manner. To do that, the pieces of the document(s) are distributed to mapper processes. The mappers just do the necessary string matching to find a word, and then

send a stream of words with the count of "1" to the reducer. The reducer then take the keys (in this case the words) and aggregate the values(counts).

In Figure 3.4 the third implementation is summarized in a MapReduce fashion. In other words, the Receivers is equivalent to the Mappers as they do the `Nullable()` and `Reduce(Derive())` and sent back a stream of answers (keys being the expresion and sigma or just the expresion) back to the Result receiver. The Result receiver (acting as the Reducer) is doing the aggregation either into the `Finish` list or the Delta dictionary.

Thus even though the initial idea was not based on MapReduce, a MapReduce based algorithm followed from a natural progression while dissecting and refining the algorithm presented.

Figure 3.3 Flow for distribution of $\frac{d}{di}$ and *null*

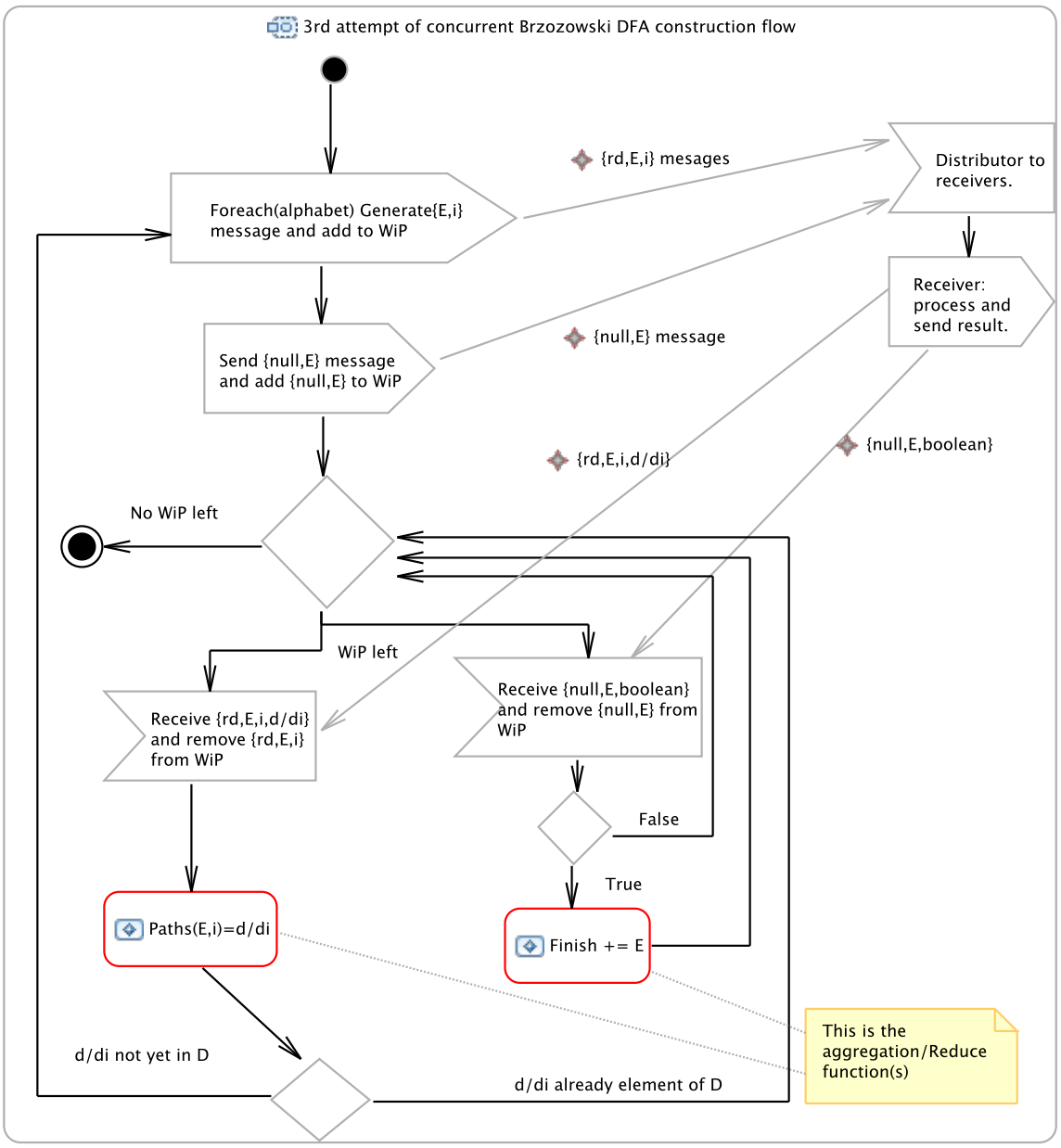
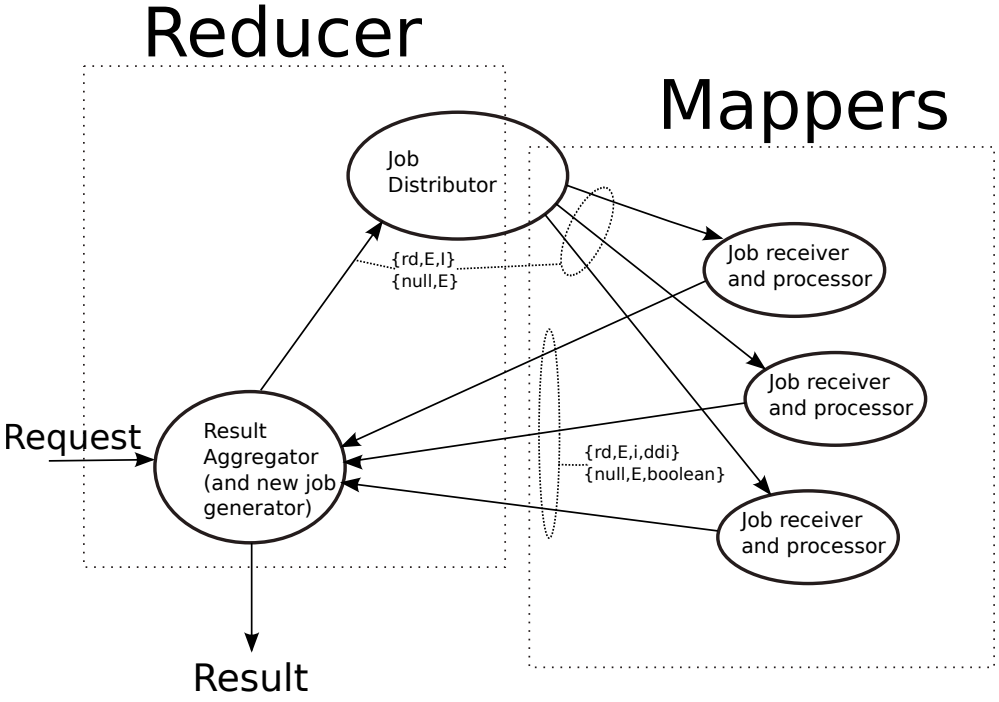


Figure 3.4 MapReduce overview of algorithm



Chapter 4

Code Analysis

4.1 General code remarks

After spending about a weekend coding the second and third iterations, the author have been impressed by the expressiveness of the Erlang language and the ease of the concurrency in this project. It has to be said that once you understand the pattern matching principles of Erlang, the coding do get easier and much more expressive than similar code the author had to write in C/C++.

The author do apologize for the less than profesional looking code formatting, as the ability to output properly formatted code, proofed to be very much the same research effort than the actual code written in Erlang, (which was not the research focus of this project).

4.2 What will be looked at

In the previous chapters we described the history and gave some overviews of Erlang and the algorithms used. In this chapter we will analyze the code to show the expressive nature of the Erlang language and the ease it was to be able to create the concurrent threading of the Brzozowski DFA generation.

The reader would be justified to keep Figure 3.2, Figure 3.1 and Figure 3.3 next to the code to be able to fully understand the code explanations. Even though we will put in an effort follow the flow of the code, the concurrent nature of the code (as the different paths in those figures would attest to) would not make that entirely possible.

The most "important" reason the author would apologize for not following the logical flow of the code to ease the understanding, is the way how the functional language ordering of the code happens. Thus even though the logical ordering/code flow is not that obvious, the authors would argue that the code design and Erlang features would be appreciated by understanding the reasons for the ordering as done in this paper.

4.2.1 Chosen code to explain

Given the bit more complexity of the nullable code of **??**, and it being the last written code, the author decided to use that code as basis for explanation in this chapter.

4.2.2 *Ab Initio*[1]

Referring to Figure 3.4, we will start to discuss the code by starting at the request and entry function. This being the begining of the code flow, as this would set the parameters for the rest of the code portions discussed. This process PID is the results receiver, typically designated with the `Res` variable. This portion also spawns the distributor, and bounds the PID to the `Dist` variable that gets passed on to the core result aggregator function to use as destination for the processing requests.

From there we will look at the Result Aggregator in section 4.4, which is the core of the processing. This function will generate the requests, and pass them on to the `Distributor`, while itself is the `Results` receiver that reduces the mapped data.

After that we will discuss the two different mappers we have implemented. They (having their entry point the `Distributor`) send their processed output to the `Result` receiver.

Lastly we will tied up some loose ends in discussing some of the ancillary code portions used to keep the code clean and simple.

Note: this is not the actual chronological execution of the code, through time and code flow, but rather the flow of the Brzozowski algorithm's processing.

4.2.3 Actual Brzozowski code

As we have used a previously working Brozoswki sequential implementation, and just replaced the core inner loop with our concurrent methods, we will not discuss that part of the code in this document as it was outside this project's scope and function. It will be stated that a brief overview analysis of that code, the author found it to be closely following Watson[Wat95] and that would be a good place to research the specific workings of the Brozoswki DFA generator and the code related to that.

4.2.4 Generated DFA correctness proofs

The author would like to mention that it was not the scope of this work to investigate the correctness of the code per se. The author have, however,

[1] *Ab Initio*: Latin for *from the beginning*. Introduced to this term from the name of a product/company used in massive extract-transform-load (ETL) environments like telcos and financials

checked and confirmed that the DFAs generated and used in this work, have been verified against the original implementation, thus errors from DFA generation, would be traced back to the original sequential implementation used.

4.2.5 Code formatting

Typesetting is never the easiest of things, and typically needs to be taken into consideration as the code is written. This is not the case when the code is first developed, and then we try to type set the code. Though there have been some effort in formatting the code in a readable format, this was not always achieved, especially given the limited width in typesetting, while the actual code is viewed and developed on wide screen displays.

To help with keeping the references as close to the original code as possible, we have used an auto wrapping features of the LaTeX style package called **`listings`**. And to indicate where the source code's original lines are, we have numbered them. Thus those lines without a number in front of them, is still continuations of the numbered line above them.

Comments We have to point out that the % sign is used to indicate the start of a comment line, and those have also been typeset in *italics*.

4.3 Entry Code

Even though it could have been parameterized more generally, the author chose to have one function for each of the different implementations, and also put it in a separate module.

However, the RoundRobin and AsAvailable schedulers have been extracted as parameters. First let us look at the entry call for the AsAvailable scheduler:

```
1  %The entry to hvp2
2  % third Parameter to chose the roundrobin or available scheduler
3  hv_brz(RE, Sigma, available, N) ->
4      TimeOut = 3000,
5      Res = self(),
6      WiP = [],%Only the first (and last) one would be "empty"
7      Finish = [],
8      Dlist = [RE],%Start state.
9      Delta = dict:new(),
10     Dist = spawn(fun () -> hv_dist_avail_start(TimeOut, Res, N)
           end),
11     hv_brzp_null(RE, Sigma, Dist, WiP, Finish, Dlist, Delta);
```

line 4-9 is the parameters we used for timeout values and the reference to ourself (see section 2.5.1) as the results receiver. We initiated the work in progress (`WiP`) and expressions already processed (`Finish`) as empty lists. `Dlist` is set to the list containing the single element of the regular expresion to be considered. These values are initialized the same in all the entry functions.

line 10 spawns the distributor process[2], and using the fourth parameter (bound to the variable `N`) as the number of worker threads to spawn. Also the timeout and our own process information(`Res`) is passed as parameters.

Note: The distributor could also be a single process, not doing any extra threading/distribution (and in retrospect should have been the special serialized test case), as the actual reducer/inner core do not care about the way the processing is spread, just that it sent it to this process, and then receive responses back. For that reason we will first look at the core reducer/result aggregator, and lastly at the distributors and receivers.

line 11 Lastly the call to the actual core of the processing function (`hv_brzp_null` in this case) with the parameters declared and initialized in line 4 to line 9. Notice that the `Distributor` from line 10 is passed as a parameter to this core processing function, and that this line is also verbatim copy of line line 15 of the round robin scheduler!

Doing the RoundRobin scheduler, we do the same as the AsAvailable scheduler, except we spawn a round robin distributor. However, note that line line 12 is the function called with the third parameter matched as the atom `roundrobin` compared to line **??** where the third parameter matched the atom `available`.

```
12 hv_brz(RE, Sigma, roundrobin, N) ->
13 Repeat lines ??-??
14     Dist = spawn(fun () -> hv_dist_rr_start(TimeOut, Res, N) end),
15     hv_brzp_null(RE, Sigma, Dist, WiP, Finish, Dlist, Delta).
```

For special testing, we used a slightly modified distributor starting function, but again we just used a function that used parameter matching to pick the right initialization function.

```
16 %Specialized one with only two named receivers using the RoundRobin
       method
17 hv_brz(RE,Sigma,rr_spec) ->
18   Repeat lines ??-??
```

[2]Remember that Erlang processes are comparable to C/Java threads, but from the program's perspective they are processes

```
19    Dist=spawn(fun() -> hv_dist_rr_spec_start(TimeOut,Res) end),
20    hv_brzp_null(RE,Sigma,Dist,WiP,Finish,Dlist,Delta).
```

4.3.1 Repeated lines

In retrospect, we could have extracted those repeated lines and used an intermediary function to set and pass those on, but it was not essential to the project goals, and the gains would have been minimal and as such not considered, though for production code this needs to be done.

4.4 Result Aggregator a.k.a. Reducer

The function `hvp2:hv_brzp_null7` is the core of the whole algorithm and it implements Figure 3.3. We have explained in Figure 3.4 how this function relates to the reducer in the Google MapReduce.

The reader should notice the use of the function call pattern matching to follow the major paths in the receiver code, and that the receiver calls itself recursively until there is no more work in progress (WiP), ie. tail recursion at play as explained in subsection 2.4.4.

The difference between the empty work in progress (WiP) subsection 4.4.1 and the receive only in subsection 4.4.2, is the empty (`[]`) fourth parameter... and it must be in that order, else the empty WiP state would never be matched.

The derivation of a new expression is done in subsection 4.4.3. Here (again note the order for matching the function parameters) the first parameter is the expression to derive, and did NOT match the atom `receive_only`!

4.4.1 WiP empty (nothing to do anymore

```
21  %The case when the WiP is empty
22  hv_brzp_null(receive_only,Sigma,Dist,[],Finish,Dlist,Delta) ->
23    %io:format("WiP finished"),
24    Dist!{stop},
25    #dfa{ states=lists:sort(Dlist), symbols=Sigma,
          start=lists:last(Dlist) ,transition=Delta, finals=Finish };
```

As we have already mentioned, the fourth parameter is an empty list, and given the fact that we are only receiving (the `receive_only` atom matched in the first parameter) we conclude that we are finished processing as there is no more outstanding derivations, and we can stop the processes, thus we send the `Distributor` a stop message.

Finally we will return the data in a format depicted by the `#dfa` record format. We will refer the author to the specifics of this record format in

[Arm07]. Sufficient to say that it is nothing different from a norma tuple, just a method to name and order the tuple into a record structure named elsewhere in the pre-compiled code, typically a header file.

4.4.2 Receive only

This part of the receiver, is executed based on `receive_only` matching the first parameter, and it only waits for messages from the derivation processes. This should either be a result of a derivation tagged with the atom `rd`, or a result from a nullable test, tagged with `null`. It would remove this request from the WiP list, and recurse back to the same function name, but with the parameters modified based on the result received.

```
26 %Receive only, nothing to derive
27 hv_brzp_null(receive_only, Sigma, Dist, WiP, Finish, Dlist, Delta)
     ->
28   receive
29     {rd,E,I,DDI} ->
30     %io:format("brzp_null_2:"), io:write({rd,E,I,DDI}),
           io:format("~n"),
31     NewDelta=dict:store({E,I},DDI,Delta),
32     case lists:member(DDI,Dlist) of
33       true ->hv_brzp_null(receive_only,Sigma,Dist,
             lists:delete({rd,E,I},WiP),Finish,Dlist,NewDelta);
34       false -> hv_brzp_null(DDI,Sigma,Dist,
             lists:delete({rd,E,I},WiP),Finish,[DDI|Dlist],NewDelta)
35     end;
36     {null,E,true} ->
37       %io:format("brzp_null_2: ~p true~n",[E]),
38       hv_brzp_null(receive_only, Sigma, Dist,
             lists:delete({null,E},WiP), [E|Finish], Dlist, Delta); %
             Add nullable states to F
39     {null,E,false} ->
40       hv_brzp_null(receive_only, Sigma, Dist,
             lists:delete({null,E},WiP), Finish, Dlist, Delta)
41     after 5000 ->
42           io:write(WiP),
43           output_mailbox(1),
44           throw(timeoutRec_only)
45   end;
```

But let us have a look at the options steps a bit more closely:

line 27 This function matched the atom `receive_only` in the first parameter. When there is an expression to derive, it would not match this function header, but rather line 47 in subsection 4.4.3

line 29 we have received a derivation result based on the `rd` atom that matched.

line 31 Modify the Delta dictionary by creating a new dictionary with the Expression and alphabet letter (`E,I`) tuple as a key, and the derived expression `DDI` as the value.

Note: the Delta variable is not changed at all, but a **new** variable with the name NewDelta is created and bounded to the new result.

line 32 Some expression and alphabet inputs might provide the same derivations, we check if we have seen a similar one before, in which case we do not waste time on deriving for the same path multiple times. This is done by checking for the existence of the DDI just received in the list Dlist, of previous expressions that already have been issued derivation requests.

line 33 We have seen this derivation before, so we just remove the E,I from the WiP list, and repeat the process by recursing back.

line 34 here we have not seen this derived expression before, thus we will call ourselves again, but this time with the expression that need further derivation as the first parameter. E,I is also remove from the WiP list.

Line line 96 we have assumed that after 5seconds of inactivity, there is a problem in the distributor or processor, at which point we exit, dumping the state of the mailbox and throwing an exception. This was a very useful method to track a typing mistake where we have sent all the data, but with a mistyped atom!

Debugging statements Even though we all want to present ourselves as invincible and perfect coders, we all make mistakes, and thus we need to debug or fault find our code. Line line 30 is a typical line used to debug the code. This line outputs a user format string with `io:format/2` similar to the Unix `printf(3)`[3]. This is then followed by the Erlang formatting of the data structure passed to `io:write/1`, and finally a linefeed. Line **??** shows an example with some substitution where the `p` is substitute with the value bound to the variable `E` and note that it is a list `[E]` passed on to the function, and the substitution happens based on the list positions.

4.4.3 Expression to derive

This part of the function, have the same code than the receive_only in line 52-62 (except the debugging and comments removed). It is line 47 to line 51 that is the interesting part in this function.

[3] the 3 inside the brackets refers to the section 3 manual pages, which is the "standard" libc functions on Unix

```
46 % When we have an RE/E d/di that needs to be derived/etc.
47 hv_brzp_null(E, Sigma, Dist, WiP, Finish, Dlist, Delta) ->
48   %foreach(Sigma) send message to Dist:
49   lists:foreach(fun(X) -> Dist!{process,[rd,E,X]} end, Sigma),
50   Dist!{process,[null,E]}, %And then also sent a nullable process
         request
51   NewWiP=[{null,E}|add_wip(WiP,rd,E,Sigma)], % foreach(Sigma)
         insert {E,I} into WiP, and add the null to the beginning ;)
```

line 47 If the function parameter matching have not matched the atom `receive_only` in line 27, we consider this to be an expression that needs to be further derived, and the variable `E` gets bound to that expression.

line 49 This line generates the messages for the derivations that need to be processed. The function `fun(X) ->Dist!{process,[rd,E,X]} end`, gets applied `foreach/2` element `X` in the list `Sigma`, and sends out the tuple `{process,[rd,E,X]}` to the `Distributor`, where:

- **`rd`** is the atom to indicate that we need to do a derivation. Compare that to **`??`**, where the `null` would indicate that a nullable check should be done.
- **`E`** The expression that was passed on to this function, and that needs to be further derived.
- **`X`** The element that we need to derive on, ie. $\frac{d}{dX}E$ where $X \in \Sigma$

Thus, with $X \in \Sigma$, and the atom `rd` indicating that the request is to process and return $\frac{d}{dX}E$

A choice during the implementation was made to use a two pair tuple, with the parameters a list, rather than an extended tuple. This choice was just for convenience, but further implementations should consider making it rather an extended tuple.

line 50 sends the `Distributor` a request to process a nullable test of the expression `E`

line 51 generates the new work in progress list. This is done in two steps. First it calls `add_wip/4` (subsection 4.6.1), that extends the current `WiP` with the same list of `{rd,E,X}` that have been sent to the `Distributor`. Then, using the `[Head|Tail]` list expansion method, prepends the `{null,E}` request to the list bound to `NewWip`

The rest of this function (line 52 to line 62), is the same as that in subsection 4.4.2, with the exception of using the newly generated `NewWip` instead `WiP` passed to the function

```
52    receive
53      {rd,E,I,DDI} ->
54        NewDelta=dict:store({E,I},DDI,Delta),
55        case lists:member(DDI,Dlist) of
56          true -> hv_brzp_null(receive_only, Sigma, Dist,
                      lists:delete({rd,E,I},NewWiP), Finish, Dlist, NewDelta);
57          false -> hv_brzp_null(DDI, Sigma, Dist,
                      lists:delete({rd,E,I}, NewWiP), Finish, [DDI|Dlist],
                      NewDelta)
58        end;
59      {null,E,true} -> hv_brzp_null(receive_only, Sigma, Dist,
                         lists:delete({null,E},NewWiP), [E|Finish], Dlist, Delta);
60      {null,E,false} -> hv_brzp_null(receive_only, Sigma, Dist,
                          lists:delete({null,E},NewWiP), Finish, Dlist, Delta)
61      after 5000 -> io:write(WiP),throw(timeOut)
62  end.
```

4.5 Mappers

Initially (and in the code) referred to as the distributor and receiver pairs, these pairs are the Google MapReduce mappers, as explained and shown in **??** and Figure 3.4.

The reader should refer to section 3.4 for algorithmic choices and explanations. What we can expand on in this section, is to mention that the threads are started by the distributor when the distributor is initialized. The distributor also sends the stop messages to the receiver threads.

The difference between the two type of mappers, is the way the distributor keeps track of the receivers and whether the receivers give any feedback to the distributor or not. With that in mind, we will highlight the differences when we discuss the AsAvailable mapper pair, but explain the main interactions and flows with the RoundRobin mappers.

4.5.1 Round Robin mappers

The Round Robin receiver is very simple, in so far as it just process a matched message, sends on the result, and then executes itself recursively... until it receives a {`stop`} message. The RoundRobin distributor is a bit more involved, as it needs to rotate the list of receivers.

RoundRobin Distributor

The distributors consists of two part, the first being the entry portion that spawns the threads, and the actual distributor.

Spawning the receivers After the parameters for the algorithm (like the number of threads to use etc.) have been configured, we create a number of threads based on the parameters passed on to the distributor as can be seen on line 66, that then call the real RoundRobin distributor with a list of receivers on line 67

```
64 %Number of servers variable, should make that a number to pass too,
       but
65 % for the moment this is adequate to test etc.
66 hv_dist_rr_start(TimeOut,Res,N) ->
67   hv_dist_rr(list_start_servers(N,Res),TimeOut).
```

The `list_start_servers/2` function recursively (because Erlang, being a functional language, do not have an iterative for-next loop), generates a list of PIDs, by prepending the list with the result of the `spawn/1` function on **??**, that have started a thread running the `hv_rr_rec` function, described in section 4.5.1.

```
68 %Start N round-robin receivers that will send their results to Res
69 % returning the list of PIDs.
70 list_start_servers(0,_Res) -> [];
71 list_start_servers(N,Res) ->
72   [spawn(fun()->hv_rr_rec("Receiver "++[N+\$0],Res), end)
73    |list_start_servers(N-1,Res)].
```

The simplified special startup case I initial used to test the receiver algorithm's workings is listed in line 75 to line 78. The `list_start_servers/2` with a parameter of `N=2` would have the same effect, just here the reader can clearly see the functional workings of the spawning of the receivers!

```
74 %Two specific and specified servers
75 hv_dist_rr_spec_start(TimeOut,Res)->
76   Rec1=spawn(fun() ->hv_rr_rec("Rec1",Res) end) end,
77   Rec2=spawn(fun() ->hv_rr_rec("Rec2",Res) end) end,
78   hv_dist_rr([Rec1,Rec2],TimeOut).
```

Chain of processes Note how we received the PID of the results receiver from the output of `self/1` in the initialization function, that passed it on to the distributor via the `spawn/1` function's parameter function. That `spawn/1`'s output PID is then saved and used as the next destination for sending the processing requests.

This process is then extended here, where we pass the `Res` PID yet again to the receivers when we initialized them (Note, the Round Robin receivers never know their parent!). In the Round Robin distributor, we **do** save the output of the `spawn/1`, because we will use that to decide who is next inline.

The Real Distributor Now that all the receivers have been started (but not yet discussed which follow in section 4.5.1), the real distributor function on line 79 gets called with a list of receivers and the `TimeOut` value.

```
79 hv_dist_rr([H|T]=Receivers,TimeOut) ->
80   receive
81     {stop} -> lists:foreach(fun(X)->X!{stop} end,Receivers);
82     {process,Param} ->
83       H!{process,Param},
84       hv_dist_rr(lists:append(T, [H]),TimeOut);
85     Other -> io:write(Other),throw(Other)
86     after TimeOut ->
87       io:format("Dist time-out and stopping receivers"),
88       lists:foreach(fun(X)->X!{stop} end, Receivers)
89   end.
```

Looking again at that, function while documenting the code, the author had to pause yet again, and appreciate the beauty and elegance of the functional language aspect of Erlang.

Given the details of code explanations given thus far, the author would only highlight the `[H|T]=Receivers` construct and point the reader to the `list:append(T,[H])` on line 84.

The `[H|T]=Receivers` construct provides a method to firstly extract the head element and the tail list, and secondly provide a single variable bound to the whole list. This way could operate on the whole list when sending the stop messages to the receivers, and on the separate head element when sending the processing request to the next receiver.

On line 84 when append the list `[H]` (containing only the single element `H`) to the back of the list `T` in an efficient way. Thus we move the receiver that have just been sent a message, to the back of the queue!

RoundRobin Receiver

Just looking at the code below, the reader should notice it is nothing more than a switching statements, but encapsulated based on the messages received. But let us take a quick line by line explanation to clear any doubts.

```
90 hv_rr_rec(Name,Res) ->
91   receive
92     {stop} -> false; %Actually *any* terminating statement would
           suffice
93     {process,[rd,E,I]} ->
           Res!{rd,E,I,mds:reduce(mds:deriv(E,I))},hv_rr_rec(Name,Res);
94     {process,[null,E]} ->
           Res!{null,E,mds:null(E)},hv_rr_rec(Name,Res);
95     Other -> io:write(Other),throw(Other)
96     after 3000 -> io:format("Timeout ~p quitting",[Name]),io:nl()
97   end.
```

line 90 The only parameter in there needed, is the `Res` variable, that tells the receiver where to send the results it is computing. The `Name` variable was used in debugging, and is referred to only in the timeout condition on line 96

line 92 This {`stop`} message in line 92, could execute any terminating command, and the receiver would exit. We just choose it to return as `false`, but true would have worked just as well.

line 93 and 94 The crux and reason for this thread's existence, are theses two lines. All they do, is just call the relevant function(s) from the `mds` module, sent the result on to `Result`-receiver, a.k.a. the Reducer, and then loop back to themselves

line 95 although not technically needed, nor useful in the production case, it was a good way to find typing errors in the core-loop. In essence the `Other` variable gets bound to *any* message not matched by line 92 to line 94. Then this get outputted to the console by `io:write/1` in a format that Erlang decided the message was, ie. list or tuple. As encore (to make sure things are noticed) it also throw an exception with `throw/1`

line 96 We made an assumption here that the receiver should never have to wait more than 3seconds[4] before considering a problem with the distributor or other functions. At this point the name given to the receiver is useful in tracking which receiver have been stuck with which request.

Erlang's claims of ease of concurrency At this point in the code analysis, the author would have to emphasize his appreciation of the cleanliness and ease of the concurrency implementation, given the author's experience with TCP/IP on MS-DOS©, SUN-RPC remote procedure calls, Unix pipes and messages. Given Erlang's ease of coding these concurrency features, the author would, at present, concede only to a Smalltalk client server model being easier to implement!

AsAvailable Mappers

As mentioned earlier, the round robin algorithm have the problem that in its simplicity, it is not optimal to balance the load across receivers, thus the reason for the AsAvailable algorithm.

In this section, we will only highlight that which make the AsAvailable different from the RoundRobin mapper.

[4] though this is something that should be parameterized in production code

Spawning the AsAvailable receivers Just to be different, this time (especially as it is only an initialization function) chose a bit different route to spawn the receivers, especially as there was no need to keep track of them like the RoundRobin distributor had to. In that regard I recursed the `hv_dist_avail_start/3` rather than to have a separate function (like the RoundRobin case) as it was easier to do the `N-1` on line 104 than to write a separate function for it.

On reviewing the code, the author noticed line 102 being placed in a position where it will be superfluously be called for each receiver, instead of once and the result cached for each invocation. This being a result of the choice made above.

```
98  %Start the receivers and the distributor
99  hv_dist_avail_start(Timeout,_Res,0) ->
100   hv_dist_available(Timeout,[]);
101 hv_dist_avail_start(Timeout,Res,N) when N>0 ->
102   Dist=self(),
103   spawn(fun() -> hv_rec_available(Timeout,"Receiver␣
          "++erlang:integer_to_list(N),Res,Dist) end),
104   hv_dist_avail_start(Timeout,Res,N-1).
```

The main point to note here, compared to the round robin approach, is on line 103 where the receivers are passed the result aggregator and the distributor's PIDs.

It would be noticed that line 101 have a guard statement in it, as that would prevent fractions and negative numbers to recurse to infinity and would throw an error in that case (no matching function). There are another case this would silently fail at this point (but timeout later) is when the call is made initially with 0 for the number of threads.

AsAvailable distributor When the AsAvailable distributor starts up, it starts with a zero knowledge of any receivers, thus its list of receivers are empty. In this state line 107 matches, and we wait for an `{available,PID}` message. Once we receive such a message, we recurse, just this time with a non-empty list in the second parameter, thus line 114 matches.

```
105 %The Available distributor
106 %First the "empty" case
107 hv_dist_available(Timeout,[]) ->
108   receive
109     {available,PID}-> hv_dist_available(Timeout,[PID])
110   after Timeout ->
111     io:format("timeout␣distributor␣from␣waiting␣state~n")
112   end;
```

In the case that we have a non-empty list of available receivers, the function look very similar to the round robin case, except for line 116 and ??.

```
113 %Receiver list not empty, "normal" case:
114 hv_dist_available(Timeout, [H|Tail]=Receivers)->
115   receive
116     {available,PID}->hv_dist_available(Timeout, [PID|Receivers]);
117     {process,Param}->H!{process,Param},
118               hv_dist_available(Timeout, Tail);
119     {stop}-> lists:foreach(fun(X)->X!{stop} end, Receivers);
120     Other -> throw(Other)
121     after Timeout ->
122       io:format("Timeout distributor fron available state~n")
123   end.
```

line 116 this extra message is what we will receiver from a receiver that is idle and waiting for work. For the sake of simplicity, we prepend this receiver's PID to the front of the queue, thus make use of a LIFO[5] queue strategy.

line 118 This line differs from the RoundRobin case 84 on page 33, in so far as it discards the `Head` (whom we have just sent a process request, thus it is not available any more) and recurse with the `Tail` portion. If the `Tail` is an empty list, we should go back to line 107, else line 114.

AsAvailable receiver The main difference between this and the round robin receiver, is the dual receive blocks, and the reason for those two nearly identical code blocks, is that we have to tell the distributor we are available for processing data, but not only that, but that we need to take a couple of protocol resilience issues into consideration

```
124 hv_rec_available(Timeout,Name,Res,Dist) ->
125   %First we handle all stop/process messages on the queue
126   receive
127     {stop} -> exit(0); %Need to do the exit here else continue to
          next receive ;(
128     {process,[rd,E1,I1]}->
129       Res!{rd,E1,I1,mds:reduce(mds:deriv(E1,I1))},
130       hv_rec_available(Timeout,Name,Res,Dist);
131     {process,[null,E1]} ->
132       Res!{null,E1,mds:null(E1)},
133       hv_rec_available(Timeout,Name,Res,Dist);
134     Other1 -> throw(Other1)%Other1 here, as we already have Other
          below
```

[5] Last In, First Out

```
135     after 0 -> Dist!{available,self()} %Nothing in queue, so we let
            the Distributor know we are available
136   end,
137   %Queue Empty when we get here, so lets wait :)
138   receive
139     {stop} -> true;
140     {process,[rd,E,I]} ->
141       Res!{rd,E,I,mds:reduce(mds:deriv(E,I))},
142       hv_rec_available(Timeout,Name,Res,Dist);
143           {process,[null,E]} ->
144       Res!{null,E,mds:null(E)},
145       hv_rec_available(Timeout,Name,Res,Dist);
146     Other -> throw(Other)
147     after Timeout ->
148       io:format("Timeout ~p quiting ~n",[Name])
149   end.
```

line 127 and 139 The reason we can not exit in the first block with a simple `true`, is that it would then execute the second block, which is not what we want to do, so we forcibly `exit/1` this block

line 134 and 146 remember the discussion in subsection 2.4.3 where a variable inside a function's execution run, may not be bound to more than one value? This is a case where that is enforced by the compiler. It could be argued that it would never happen, but given the possibility that both blocks might get executed, the compiler enforces this rule to prevent the immutable variables to never be changed.

line 135 and 148 these are the single reason why there is a need for two separate blocks, being that line 135 will tell the distributor it is available only when the queue is empty (the `after 0 ->` timeout), and then the receiver will wait (block) for work from the distributor, with a "normal" timeout just like the round robin receiver.

This then closes the major algorithmic code discussions, and only leaves us with the ancillary functions mentioned.

4.6 Ancillary functions

4.6.1 Adding to Work in Progress

The Work in Progress (WiP) is a list of the type of processing that have been sent out to the distributors, and that we still have not received any responses back for. It is a (in our implementation) a simple list, but is perhaps a place to optimize if it grows too big. This is especially as we prepend the new work to the beginning of the list, but the expected responses to be received

next are those at the back of the list that would need to be removed, so in retrospect a FIFO type queue would be a preferred implementation when optimizing.

```
150 %For all the Sigma add {E,i} to the Work In Progress
151 add_wip(WiP, Type, E, [H]) -> [{Type, E, H}| WiP];
152 add_wip(WiP, Type, E, [H| SigmaT]) -> add_wip([{Type, E, H}| WiP],
        Type, E, SigmaT).
```

Again we saw the use of specialization by the use of parameter matching. Here we first check on line line 151 if the fourth parameter is a list with a single element (and we bound the variable `H` to that single element), else we match on line line 152 for a list with more than one element and we bind the head (first element) to `H` while the tail (rest of the list excluding the head) is bound to `SigmaT`.

Mailbox debugging

```
153 output_mailbox(N) ->
154  receive
155    Mess -> io:format("Message␣~p~n:",[N]),
156                 io:write(Mess), io:nl(),
157                 output_mailbox(N+1)
158  after 0 -> exit(123)
159  end.
```

In the early stages of the AsAvailable mappers, we missed the problem related to line 135 and 148 on 37. This little function called as: `output_mailbox(0)` was a lifesaver.

Chapter 5

Code critique

To understand the weaknesses of a system, you will know when not to use it, but also where the improvements would be needed for future work. In that regard we will briefly highlight some issues not mentioned or discussed in this chapter.

5.1 Optimization vs Elegance

The major problem we would acknowledge as a slow and optimizable focus, is the use of lists, with its `O(n)` complexity (and several places it will not be the worst case, but the average case). The idea was not to have the fastest possible solution, but to implement an initial concurrent implementation in Erlang.

Thus, the choice for elegance was always prefered to the choice of speed.

5.2 Coding enhancements

5.2.1 Distributors

The main decision here was that the distributors will not "really" control the receivers (other than to tell those available when a stop message have been received). There are several ways to remedy this especially using the `spawn_link` that would tell the distributor (the PID that spawned the receivers) which have terminated. This way the distributor could make a decision whether to respawn the process or not.

At this stage we have just proved the distribution in a concurrent fashion as the project's goal and would leave these enhancements to implementors of production code that need the resilience and correctness guaranteed.

5.2.2 Work in Progress

We do not check at all whether there are work in progress (WiP) that have not returned to us, ie. a node/process failed while working on an expresion. This also need to be considered and rescheduled in production code, especially when using distributed code when a node fails during processing. Here the `spawn_link` as discussed in subsection 5.2.1 would again be used to inform the work producer (reducer in map-reduce terms) that there were a failure and that it might need to resubmit WiP for recomputation.

Several strategies could be used here, the simplest being that the mapper would only resubmit WiP if notified of a failure and it timed out while waiting for results, meaning that those left in WiP might have been those that have failed. A bit more complex strategy would have the distributor know which job was send to which receiver (mapper) and that it could restart or resubmit that job once it received the failure notice. A control freak case could be that the distributor would also inform the reducer about which mapper received which job, and once the mapper dies, let the reducer know which mapper died so that the reducer can resubmit the job. This last method would also help the reducer to get some performance or processing information from each job.

5.3 Correctness proving

As mentioned in **??**, no formal correctness proofs have been attempted, other than a comparison with the output of the original base sequential algorithm. As another exercise, the constructed DFAs should be constructed in a format that should be compared with the expected DFA construction of known correct implementations. This is left as an exercise outside the scope of this project for time reasons.

Chapter 6

Performance

6.1 Speed comparisons

??

The development and tests were all done on Apple Mac laptops, both having dual core Intel processors, and the results of the tests were discouraging, however it were not surprising. Two things in the tests stood out that would need future investigation: first the size of the tests never took the CPU utilization above 115%, and the second is that the processing time versus the message sizes, is too little to make a difference. But lets look how bad the results were.

In table Table 6.1 we see the regular expression (in the syntax used in the code) that we used to test the performance of the algorithms developed in this project.

We conducted 20 test runs of each algorithm using 2 and 10 threads, and then averaged the results. In table Table 6.2 we tested the "full" ASCI byte range against the regular expression, and in Table 6.1 we only test against the space and the letters a to z and A to Z.

Table 6.1: Expression used for testing

```
{concat,
{union,
"Is dit nog hierso",
{kclosure,"Here"}},
{kclosure,{union,
"Testing",
"My testing"}}}
```

Table 6.2: $\Sigma \in [1 \ldots 255]$

Sequential	145947	
Threads:	2	10
Round Robin	1168392	1111456
RR nullable	1201972	1147706
AsAvailable	1231590	1253817
AA Nullable	1308366	1300956

Table 6.3: Using space, a-z and A-Z

Sequential	28886	
Threads:	2	10
Round Robin	84879	76499
RR nullable	88057	78519
AsAvailable	89504	85985
AA Nullable	98305	94441

6.1.1 Discussion of the results

As were mentioned early in **??** we noticed the CPU utilization never increased above 115%, which was quite discouraging, but given that the Erlang VMs are optimized on Linux and Solaris we were not that surprised, but as time and available systems were not available to test or confirm this hypothesis, we can not make any further remarks on the MacOSX Erlang VM as such.

However, there is another story to be told given the results in tables Table 6.2 and **??** and what [Arm07] also refers to, and is the issue the overheads versus the work done. If the round robin and as-available algorithms are compared, it is obvious that the as-available algorithm have more overhead per message than the round robin (and given the code size differences it is expected). Even just moving the nullable tests to the threads, showed a decrease in performance.

The other interesting results for the two tables, are the overhead of the unused characters in the alphabet in the regular expression, made the performance penalty hit go from a factor of approximate 3 in table Table 6.1 to a factor of over 8 in table Table 6.2. This tells us that the processing done per work-unit is not enough to warrant the overhead of the fine grained concurrency of our algorithms.

Chapter 7

Conclusion

In this project we investigated the concurrency features of Erlang, and applied that to the Brozoswki DFA construction. Erlang's concurrency features are quite expressive (and impressed the author), and the coding for the concurrency were done much quicker than initially anticipated. The authors would acknowledge that the claims of ease of concurrency of the Erlang designers are achievable with minimal effort.

The Brozoswki DFA construction algorithm and the methods chosen to do concurrent processing to derive the DFA, was not able to achieve any speedup on the hardware tested. It will be the authors' opinion that the speedups wil not be easily achieved as the processing needs are much less than the message sizes, and the overhead is more than the actual processing required.

7.1 Future studies/work

As our research focussed on threading the processing over the derivation of each sub-derived expresion for each of the alphabet entries, we concluded that it is too fine grained, and research could be looked at to rather spread the concurrency over each derivation with its alphabet as a processing unit.

Bibliography

[Arm07] Joe Armstrong. *Programming Erlang:Software for a Concurrent World*. Pragmatic Bookshelf, http://www.pragmaticbookshelf.com, 2007.

[Brz64] J.A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[Ora] Oracle.

[SKWH08] T. Strauss, D.G. Kourie, B.W. Watson, and J. Holub. A Concurrent Specification of Brzozowski's DFA Construction Algorithm. *International Journal of Foundations of Computer Science*, 19(1):125–135, 2008.

[Wat95] B.W. Watson. *Taxonomies and toolkits of regular language algorithms*. Citeseer, 1995.

Appendix A

Test listings

A.1 Test listing

The following code were used to generate the tests mentioned

```
lists:sum(
 lists:map(
   fun(X) ->
     element(1,
       timer:tc(hvp1,hv\_brz,
        [{concat,
          {union,
           "Is dit nog hierso",
           {kclosure,"Here"}},
          {kclosure,{union,"Testing","My testing"}}}
         ," "++lists:seq($a,$z)++lists:seq($A,$Z)
         ,available,2]))
     end
    ,lists:seq(1,20)
))/20.
```