BACHELOR'S THESIS | LUND UNIVERSITY

Cactus - The Cal Actor Clojure Language

Marcus Begic, Gabriel Borglund

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2021-06

Acknowledgements

We would like to thank Sven Robertz, who was kind enough to find the time to examine this project, even though it is uncommon to write a bachelor's thesis at LTH.

We would also like to thank our supervisor Jörn not only for teaching us about CAL, Clojure, CSP, language design and more, but also for spending hours in his office teaching us about the systems of global economies, corporations and life in general.

As a last note, thank you to Kirsty Shearer and Alisdair Milliken for their graphical support.

Gabriel Borglund & Marcus Begic, 2021-03-18

Abstract

This thesis explores the possibility and practicality of creating an actor based programming language on top of Clojure. It also gives an insight into the iterations that the language went through to achieve it's current form. We discuss various design choices and limitations of the language, and suggest improvements. Further some measurements were made to see how well an implementation of The Smith-Waterman algorithm scaled on different machines with different number of cores. We find that increased parallelization of the algorithm grants improved performance and that in general the language's performance scales with more parallelization.

Contents

1 Introduction

The first computers where theorized in the beginning of the 19th century with designs such as the Babbage engine [7]. Babbage never built his machine, but he lay the groundwork for later iterations of computing machines. Alan Turing later theorized about computers which could calculate everything that is computable. These ideas led to the core concepts of modern computers. In the early 1950s IBM built the first computer to turn a profit. The IBM 650, which utilized vacuum tubes for its internal workings. However at approximately the same time the TRADIC was introduced. It was the first computer to use transistors instead of vacuum tubes. This transistor idea, was the start of the computer architecture that we still see today. In 1965 one of the founders of Intel, Gordon More observed that the number of transistors on an integrated circuit had roughly doubled every two years. This observation is referred to as Moore's law, but it is to be understood not as a law but as a general projection of where things have been going for the last 70 years. Today however we are beginning to see the end of Moore's law[9]. The size of the transistors in a normal mobile phone are beginning to get close to the limit imposed by physics, where quantum tunneling is set to be a problem. To solve the constant need for speed, another way of gaining performance has been the utilization of parallelization. The idea is not new but it has become more and more utilized in modern systems as the economic and physical limits have started to impose restrictions on the size of transistors. Today it is not uncommon to see an eight core processor in a mobile phone. The problem with parallelization is however the need to not only parallelize the hardware, but also the software.

The current mainstream way of writing parallel processes is utilizing threads. However this mode of parallelization has also proven to be cumbersome. This is something discussed in by Edward Lee in [6], where he reasons about the hardships of using threads as a means of parallelization. He argues that the current way of trying to 'prune' parallel programs written using threads is a fundamentally bad idea, as the programming model still is prone to failure even after deploying rigorous code testing and coding practices.

However the article is not all doom and gloom as he argues for the fact that it is not concurrent programs that are intrinsically bad, it is simply the current model that needs refining.

Luckily threads are not the only model for achieving parallelization.

A different approach is the dataflow model, which is a model where separate computational units send and receive tokens, to and from each other. No state is shared state between these units, thus they can run in parallel. With more units, more parallelization is achieved and thus more efficiency. Kahn[**?**] and Hoare[4] showed that dataflow can provide both computational efficiency via parallelization and formal reasoning about programs.

One such dataflow-language is the CAL Actor Language. Developed in the early 2000's at University of California by Jörn Janneck and Johan Eker, it has since been been applied to various fields and even been adopted by the MPEG group as a standard.[5] These perks made us want to create our own implementation, of a dataflow language, a domain specific language whose performance scales with more parallelization. We reasoned that by extending the Clojure syntax using macros it would be quite efficient to create our language without writing our own compiler. Thus, we created Cactus, a Cal based actor language embedded in Clojure. We then explore whether Cactus' efficiency scales with the number of available processing elements. This exploration took form in the two questions: Is there a correlation between the number of threads in the underlying thread pool and performance? and: How does the width of the stripe in the Smith-Waterman algorithm affect the performance of the algorithm?

In this report the iterative process of designing Cactus is described in chronological order. A number of experiments are designed and conducted to measure some parts of the performance of the language. These tests where conducted to see if the goal of creating a language that would scale with more computing cores, was reached. The results are then shown in the results section. Lastly the features of the language are described, and some examples of how they are used are shown.

1.1 Division of Work

There was no set division of work from the beginning. We worked a lot using pair programming. This to make sure that we both were on the same track. After the first crude version using `core.async` channels, was done, Marcus began work on creating the specialized dataflow channels and the buffers. Gabriel in the meantime, started writing the macros for the cactus language.

Since the dataflow channels were done earlier than the macros (which had to be revised a couple of times), Marcus also started writing the report during this time. For the final weeks of the project both Marcus and Gabriel focused on writing the report and carrying out the testing of the performance of cactus on different computer setups.

2 Background and Related Works

The idea that speed can be achieved via parallelization is not a new idéa, as it dates back to the 50's with the theorizing of supercomputers which were later implemented in the 60's and 70's. The industry has grown massively, as almost all sold microprocessors inside personal computers and phones are multi-core processors. It is also a fact that all servers and data centers in the world today run different kinds of computations in parallel.

The ever-present problem when working with parallel computation are timing issues, which materialize in the form of race conditions and deadlocks. Many attempts have been made to try to solve these problems.

The idea of dataflow, where programs are data-driven. I.e. that the time of execution of operations in a program, are directed by the availability of data instead of a preset order of instructions, is not a new idea either. It is a form of computation that was pioneered by Jack Dennis at MIT[10]

2.1 Cores and SMT

We often refer to cores in the report. However we use the term quite losely as to refer to actual physical cores, but also to refer to cores that have enabled simultaneous multithreading (SMT). That is when the number of cores is refered to as 8, then in the intel processors there would only be 4 physical cores, and 2 SMTs for each core. This seemed to be the better way of handling the number of cores since the performance was more in line with this definition than with the actual number of physical cores.

2.2 Kahns Simple Language for Parallel Programming

In 1974, Kahn published a paper which described the semantics of a simple language for parallel programming [**?**]. The syntax for this language was quite close to ALGOL, but it featured ways to declare channels on which processes could communicate by sending certain types of data. This approach bore some similarity with an actor based language in that the processes were triggered by the availability of data. In that article, Kahn formalize the language by thinking of the program as a set of functions over their respective channel histories. He then proceeded to prove some interesting properties about his program, more concretely he could prove the output of a simple program and he argued that the same way of formalization could be applied to larger programs (he also give examples on how that has been done).

2.3 CAL Actor Language

Dataflow is a group of programming languages and models where the computation occurs in condensed units. Units output data to other units, forming programs[10].

Belonging to this group is the *CAL Actor Language*. In CAL, *actors* form the basic computational units[5]. Actors have ports where they receive data in the form of *tokens*. These can be sent from other actors or networks. They can also place tokens on their output ports. Internally, actors carry none or more *guards*, these are conditions to which the tokens are held before computation can proceed.

Actors may contain one or more sequential computations called *actions* each of which **may** require an input pattern. If the input pattern is met for a certain action, the actor *fires* the action. This means that it consumes tokens from its input ports, performs the computation defined by the action and outputs any tokens it produces on its output ports. If more tokens arrive that don't fulfill any pattern, they are buffered in FIFO queues in the input ports. Additionally, an actor can contain an internal state which can be changed inside of the action body.

A collection of connected actors, is called a *network*, connected networks, actors and networks or basically any combination of the two that takes an input and produces an output, are called *entities*. For example the actor below in Listing: 1 shows an adder actor, that waits for input on the two channels A and B. It then outputs the result on the channel C. However to run this adder it has to run in a network of other actors. The most simple example would be two actors that simply outputs one token, and connect these two to each input port on the adder actor. This cluster of actors is now what we call a network. Sometimes we don't care about weather a producer of tokens is an actor or a network. We then call that producer an entity. Consider a printing actor that receives tokens on its input channel and prints them to the console. It does not matter for the functionality of the printing actor weather it is a network or an actor that is sending it tokens. It simply matters that it is connected to an *entity*.

```
actor Add () A, B ⟹ C :

        action [a], [b] ⟹ [a + b] end
end
```

Listing 1: A simple adder actor in CAL

2.4 Communicating Sequential Processes

In 1978 Tony Hoare published his paper on Communicating Sequential Processes (CSP) [4]. CSP is a formal language that describes communicating asynchronous processes were each process runs on it's own processor. These processes encapsulate some data structures or data and do not share any state with other processes, all communication occurs via message-passing on non-buffered channels were inputs/outputs are explicitly declared. Internally, each process still executes sequentially. Hoare uses Djikstra's guarded commands that check if a condition is fulfilled which dictates if a message can be passed or received, essentially creating a blocking queue. Hoare's model inspired many programming languages such as Go and occam and became the foundation for `core.async`. `core.async` uses CSP based channels between `go` processes and the guarded commands to determine if an process can receive a message, while letting the `go` macro expand to a thread confined process that can sequentially execute Clojure code.

2.5 Lisp

Lisp is a functional programming language notably featuring macros. Calling functions in lisp is done through using paranthesis. For example making an addition the function + is called as follows: (+ 1 2). This expression will of course evaluate to 3. This follows the basic syntax where the first element of the expression (Which is actually just a list), is the name of the function being called, and the rest of the elements are treated as input arguments. This report also heavily depends on macros, however there are great resources online explaining this better than we could ever do in this report. Hence we refer the reader to these. One example is [3].

2.6 Clojure

Being a dialect of LISP, Clojures' source code is a data structure in Clojure. This powerful property allows us to extend the language at compile time using macros. Macros are passed Clojure expressions and can do arbitrary computation at compile-time. The result of which are what any call to the macro runs, at run-time. Macros are defined with the keyword: `defmacro )`

Clojure's `core.async` library contains the `go` function. `go` defines a *process* which is executed asynchronously. Clojure runs on the JVM and one can access all of Java's languages features including threads. `core.async` instantiates a `ThreadPool`, processes are dispatched to the `ThreadPool` and executed. `go` processes can receive and output values through `channels`, the operators `<! (take!)` and `>! (put!)` are used to take and put values on the channels. If there is nothing to take on the channel, `core.async` will park the process. Parking releases the thread occupied by the process and waits until either a value is put on the channel or a value is taken before execution might be continued.

2.7 The Smith-Waterman Algorithm

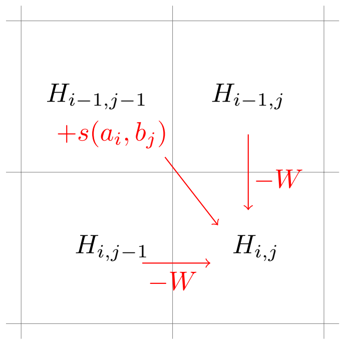
As a driving application and foundation for this project the Striped Smith-Waterman algorithm is used. First written in Clojure using the `core.async` library, and then implemented in Cactus.

Proposed in 1981, the Smith-Waterman algorithm is an important solution to deciding if two sequences of nucleotides are related[8]. Seeing the sequence of nucleotides as strings of characters, the question boils down to finding alignments of local regions in the two strings. These locally aligned regions might be related parts of the DNA.

When comparing two strings A and B, the algorithm can either *match* characters by selecting one identical character from each string, it can *mismatch* by select two different characters and it can *shift*, selecting one character from one string and leaving the other. Each operation contributes with a score or a penalty. The question becomes to find the maximum local score from a sequence of these operations. The match/mismatch operations score for the characters a and b in string A and B, is given by the scoring function $s(a, b)$, usually this value is positive for a match and negative for a mismatch. Shifting for either string will reduce the score by the gap penalty W. The gap penalty may be defined as a function or as a constant. In this project for simplicity sake, the gap penalty is always constant, and the match function simply gives a positive constant score for two identical characters and a negative constant otherwise.

The algorithm determines a scoring matrix were the strings compose the axis of the matrix. Each cell $H_{i,j}$ contains the maximum score for the algorithm until that point, and can be reached from three neighbouring cells $H_{i-1,j-1}$, $H_{i-1,j}$ and $H_{i,j-1}$ by applying the different operations to them. Approaching $H_{i,j}$ from the diagonal or north-west direction translates to matching a character from each string, the value for this operation would then be the value of the previous cell + $s(a, b)$. Approaching from the left or from the cell above is selecting a character from one respective string and shifting on the other. Here the value would be that of the previous cell - the gap penalty W. In the Smith-Waterman the maximum of these values is selected for the cell[8]. Thus the value in the cell $H_{i,j}$, is given by:

$$H_{i,j} = max \begin{cases} H_{i-1,j-i} + s(a_i, b_j) \\ H_{i-1,j} - W \\ H_{i,j-1} - W \\ 0 \end{cases} \tag{1}$$



Crucially, a given cell $H_{i,j}$ will not need to calculate all possible paths to itself, with memoization of the neighbouring cell's maximum score, a time complexity of $O(mn)$ is achieved. After the matrix is composed, the cell with the total max score is found and local sequence alignment string is composed by backtracking from this index in the matrix.

2.8 Striped Smith-Waterman

As any cell in the scoring matrix only depends on it's neighbouring cells, ways to parallalize the algorithm have emerged, one such parallelization is the Striped Smith-Waterman algorithm. Published in 2000, Torbjørn Rognes and Erling Seeberg managed to see a six-fold increase in database searches using Striped Smith-Waterman [2]. Later, Marco Mattavelli et al. published an implementation written in CAL [1].

The matrix is vertically split into *stripes* and each row is computed in parallel. In CAL and Cactus, actors are computing the values in the stripe.

The implementation of the striped Smith-Waterman in Cactus looked something like:

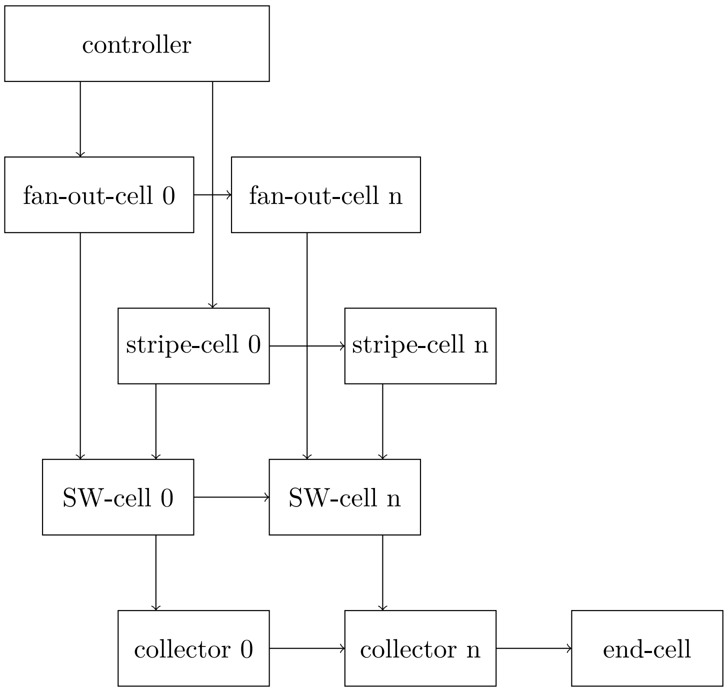


Figure 1: The layout of the actors in the implementation of the Smith-Waterman algorithm. The controller actor sends out sub-strings of length n. One character at a time is then peeled of in the stripe-cells, that then gets sent to the SW-cell. The controller also sends a character that gets sent to fan-out-cell that propagates the character to the rest of the fan-out-cells and to the SW-cells. The peeled characters then get sent to the SW-cells that does the computation, and the value of that computation gets sent to the collector-cells. The collector cells append the score of its corresponding SW-cell to the results vector in such a way that when the vector enters the end-cell, the length is n. The end-cell counts the number of vectors it has recieved, and when it has gotten the correct number of vectors, that is: (/ (length A) n)

Where the controller would write a timestamp to a results file. Then it would continue by sending substrings of B to stripe cell 0. These substrings would be of the length width, which in the figure above is denoted as n. A character from these substrings would then get peeled of by each stripe cell. Given The width 2 and the B string $ABCD$, the controller would send the string AB to stripe-cell 0. Stripe-cell 0 would then send the character A to SW-cell 0. This would happen `(count A)`, times. The stripe-cell 0 would then also propagate the rest of the substring, B to stripe-cell 1. Stripe-cell 1 would then send the character B to SW-cell 1 `(count A)` times. The controller would in the meantime send the next substring to stripe-cell 0, which in this case would be CD, and this substring would also get peeled and propagated through the row of stripe-cells like the first string. The characters in the second substring would of course also get sent to the SW-cells. This part of the network makes sure that SW-cell i, receives the characters at index `((* j width) i)`, where j is 0 to the number of substrings created when dividing B into strings of length width. Given the previous example, and the A string *EFGH*, SW-cell 0 would then receive, *A*, four times (one time for each character in A. That is `(count A)` times), and *C*, four times, while SW-cell 1 would receive *B*, four times, and *D*, four times.

But to calculate a value the SW-cells also needs a character from A. This is where the fan-out-cells comes in. Fan-out cell 0 gets sent each character of A subsequently, and would then propagate that character to the next fan-out-cell. Each fan-out-cell also then sends that character to it's corresponding SW-cell.

This would then mean that each SW-cell has received each character of A exactly once and it's designated characters from B `(count A)` times, and has almost all the pieces of information needed to calculate the scores, except for one, that is the score from its' west neighbour.

Each SW-cell needs three inputs on its' input channels to start the calculation of a score. A character from A, a character from B, and a score from the cell to the west of itself. SW-cell 0 is instantiated with a number of tokens on its west port. Those tokens are `(count A)` number of 0's. This then makes sure that SW-cell 0 can fire its' action `(count A)` times. When SW-cell 0 has calculated its' first value, this is then sent to SW-cell 1. Which in turn could then send its' value to SW-cell 2, and so forth. But each SW-cell also sends its' output value to the collector cell.

The collector cells, are the cells that collect the scores calculated by the SW-cells. These cells receive one value from its' corresponding SW-cell, and it also receives a vector of values from its' western neighbour. Collector 0 is also instantiated with `(* (/ (count B) width) (count A))` number of empty vectors so that it can fire that number of times. Each collector then takes the value it receives from its' SW-cell and appends that to the vector it receives from its' western neighbour, meaning that collector n then outputs a vector of length width to the end cell.

The job of the end cell is then simply to count the number of such vectors it has received, and to print to the same results file that the controller printed to, a timestamp when it has received `(* A-len (/ B-len width) )` such vectors. This would of course mean that all of the values in the matrix have been calculated, and the algorithm has finished running.

2.9 The Theoretical Cap of the Number of Smith-Waterman Cells Run In Parallel

Since every cell in the stripe in the Smith-Waterman algorithm is dependent on the value calculated by its western neighbour, cell n has to wait for cell `(n - 1)` to fire before it can fire. To understand why at most n cells can fire at once, one can imagine firings in discrete steps. In the first step, only the first cell can fire. The second step leads to the firing of the first cell (if there are more rows to be processed that is), but also the firing of the second cell since there now can be a token available on its western port.

The third firing executes the first, second and third cell and so on. This means that at time n, when n steps have executed, all of the actors can fire at once. However, no actor could fire again in that moment, since they are all dependent on the value sent by their western neighbour.

The exception is of course the first cell which will dictate the speed at which firing can be done. However, since all of cells are dependent of its western neighbour, the maximum number of cell firings that can happen at one moment is n, i.e. that all cells in the stripe fire simultaneously.

3 Implementation

In this section we describe the iterative process of implementing the language. We reason about design decisions and chronologically explain the implementation of the channels connecting the actors and the macros that compose the actors.

3.1 Using `core.async`

We use `core.async` as an underlying mechanism for handling concurrency in Cactus. `core.async` is Clojure's standard library for asynchronous programming. We chose it not only because functional suitability but also because it has the biggest support and avoids depending on libraries from third parties. It also had a very useful internal state machine to handle the parking and starting of processes. However, we had to modify aspects of it to suit Cactus.

3.2 DataFlowChannel

Standard `core.async` lacks support for peeking into channels and one cannot see the number of elements that are in the buffer. This was the first reason for modification of the language. Calling `(chan )` in `core.async` would usually instantiate the library type `(deftype ManyToManyChannel )` a `ManyToManyChannel` has capabilities to handle multiple writer writing to multiple readers while handling all the call-backs to the `go` process' state machine. This is wrapped around a Java `LinkedList` that holds the elements placed in the buffer. The initial approach was to add a peek function to the `ManyToManyChannel` type. However, since peeking requires peeking into the LinkedList buffer, the operation becomes $O(n)$, while `take!` and `put!` are $O(1)$ operations. So, a different data structure was required entirely. We removed `ManyToManyChannel` and all the calls to it were changed to calls to a new `(deftype DataFlowChannel )`. This new structure did not require handling multiple readers and writers, since any buffer in CAL can only be read by one actor and written to by one actor. As an internal data structure in the `DataFlowChannel`, we created a circular buffer built on Java's `ArrayList`. A circular buffer gave us $O(1)$ takes and offers to the queue and the underlying `ArrayList` gave $O(1)$ peeking. However, in the current implementation of Cactus the maximum size of the buffer is undetermined, so the circular buffer needed to grow to an arbitrarily large size. Thus, we modified the buffer to grow the `ArrayList` instead of over-writing old elements. When the buffer is full, another `ArrayList` twice as big is instantiated and current elements are copied over. This operation costs $O(n)$ and is not thread safe. So a `mutex` was used, locking the `DataFlowChannel` when resizing occurs.

We removed all call-backs to the `go` process' state machine, this means that internally any call to `put!` or `take!` no longer parked the process. Actors would instead busy-wait and just loop to their guards constantly. In CAL the minimum a guard needs to know from its buffer is the existence of an element at a certain depth. Thus, in the `DataFlowChannel` we implemented the function `(size? n chan)` for internal use, this function takes an integer and a `DataFlowChannel` as parameters and will return true if there exists an element at that depth. This became the underlying logic the guards use.

3.3 Designing The Cactus Macros

3.3.1 Iteration One

The first implementation, before we had the modified `dataflowchannels` and just used the regular `core.async` channels, looked something like:

```
(defn sw-cell [a b an w v aln-v name]
    (go
      (loop [nw 0 n 0 i 0];;Set initial staet
        (let [new-a (<! a) new-b (<! b) new-w (<! w)] ;;Wait for ports
          (let [
                new-nw new-w
                new-n (cell-action nw n new-w new-a new-b)
                ] ;;Assign new local state and execute body
            (>! v new-n);;Set output
            (>! aln-v new-n)
            (if (= i (dec an))
              (do
                (recur 0 0 0)
                )
              (do
                (recur new-nw new-n (inc i))
                )
            )
             ;;Recur
          )
        )
      )
    )
  )
```

Listing 2: The listing shows the first iteration of the Smith-Waterman-cell. This iteration was really just a go process wrapped in a function definition.

that is, a function that took initial values and `core.async` channels as its input parameters. The actor would try to consume tokens from the channels. Since the take `(<! channel)` call would park, the actors would not loop indefinitely but rather would wait for tokens to appear on the channels. This worked for the Smith-Waterman implementation but had the drawback of not allowing multiple actions inside of an actor. This is a feature not needed for the SW-implementation but it is a powerful feature of CAL that would be nice to have in Cactus. This first implementation also featured a very crude version of state, but since the state was implemented as an update of the loop variables in the recur call, changes could not be made in the action body of the actor, but rather, only at the end of an action firing. This has the obvious drawback of not allowing mid firing state changes. That feature could be implemented with nested lets, where each let would update the previous values, and then at the final recur the innermost nested let variables would be the ones sent to the recur call.

In the case of many variable updates this would however turn in to a mess of nested lets, hence that approach was abandoned in later implementations. The final nail in the coffin was the fact that guards where not possible in this implementation.

To see the payload of a token it would have to be consumed. Therefore the concept of guards, where checking the payloads before consuming the tokens is the goal, was not possible.

3.3.2 Iteration Two

The next iteration of the Smith-Waterman-cell was a major update.

```
(defactor sw-cell [a-length] [a-chan b-chan west] ==> [value aligner-value]
  (defstate [nw 0 n 0 i 0])
  (defaction a-chan [a] b-chan [b] west [new-west] ==>
    (let [new-nw new-west
          new-n (cell-action @nw @n new-west a b)
         ]
         (>>! value new-n)
         (>>! aligner-value new-n)
         (if (= @i (dec a-length))
           (do
             (-- nw 0)
             (-- n 0)
             (-- i 0)
             )
           (do
             (-- nw new-nw)
             (-- n new-n)
             (-- i (inc @i))
             )
           )
         )

    )
  )
```

Listing 3: This is the second iteration of the SW-cell. This time around more syntactic sugar has been added. The functionality is however basically the same. The notable exception is the mutable state

At this point a lot of changes to the previous design were implemented, and macros were used to add some nice syntactic sugar. This design does not make a lot of sense outside the context of a defined network, therefore an example network is shown below:

```
(defn -main  [& args]
    (println "started")
    (def A "ABCD")
    (def B "ABCDEFGH")
    (def width 4)

    (println "B␣length␣" (count B))
    (println "A␣length␣" (count A))

    (entities
      (actor controller (controller-actor A B width ))
      (actor stripe (stripe-actor (count A)))

      (actor fanout (fanout-actor ))

      (actor sw0 (sw-cell (count A)))
      (actor sw1 (sw-cell (count A)))
      (actor sw2 (sw-cell (count A)))
      (actor sw3 (sw-cell-printing (count A) (* (/ (count B) width ) (count A)) ) )

      (actor aligner (align-actor A B width))

      (network
        (con (controller :chan-contr-fan-a) (fanout :in-chan) )
        (con (controller :chan-stripe) (stripe :b-chan) )

        (con (stripe :chan-0) (sw0 :b-chan) )
        (con (stripe :chan-1) (sw1 :b-chan) )
        (con (stripe :chan-2) (sw2 :b-chan) )
        (con (stripe :chan-3) (sw3 :b-chan) )

        (con (fanout :chan-0) (sw0 :a-chan) )
        (con (fanout :chan-1) (sw1 :a-chan) )
        (con (fanout :chan-2) (sw2 :a-chan) )
        (con (fanout :chan-3) (sw3 :a-chan) )

        (con (sw0 :value) (sw1 :west) )
        (con (sw1 :value) (sw2 :west) )
        (con (sw2 :value) (sw3 :west) )
        (con (sw3 :value) (sw0 :west)
            {:initial-tokens (vec (repeat (count A) 0))}
            )
        (con (sw0 :aligner-value) (aligner :chan-0))
        (con (sw1 :aligner-value) (aligner :chan-1))
        (con (sw2 :aligner-value) (aligner :chan-2))
        (con (sw3 :aligner-value) (aligner :chan-3))
        )
      )
  (while true )
  )
```

Listing 4: This shows the entire network from controller to aligner

The `defactor` macro was implemented as a macro that expanded to a function definition. The first vector in the definition of a new actor expanded to the function parameter vector in that particular function definition. However a last argument is always added by the network macro. That argument is something called the connections-map.

The connections-map gets compiled inside of the network block in the entities clause, where each con is parsed and added to the connections map. The connections map was just a normal key-value map. When for example: `(con (stripe :chan0) (sw0 :bchan))`, was parsed, the entry `:sw0 :aligner-value chan-0 :aligner :chan-0 chan-0` was added to the final connections map. There were also two special keys inside of the connections-map. The first was the `:number-of-channels` key that kept a tally of the numbers of channels needed to create the entire defined network. There was also an `:arguments` key that held a map of it's own. This map contained the name of the channel that took the arguments and the actual vector of values sent as arguments. To give an example, if the following statement was the first in the network definition:

```
(con (sw3 :value) (sw0 :west)
{:initial-tokens (vec (repeat (count A) 0))}
),
```

the connections-map would get expanded to:

```
{:sw3 {:value channel-0},
:sw0 {:west channel-0},
:number-of-channels 1,
:channel-arguments {:channel-0 {:initial-tokens
(vec (repeat (count A) 0))}
}
}.
```

Since the connections inside the connections-map were just Clojure symbols, the channels had to actually get instantiated and bound to the correct variables. This happened in the entities macro.

The entities macro expanded the network in a series of steps. The first was parsing the last s-expression and creating the connections-map. After that was done, the second step was parsing the actual actor calls.

These are all of the s-expressions that are above the network definition, sort of like constructors if the reader is so inclined.

These got expanded to function calls, that called the expanded `defactor` functions, with the arguments sent to them in the installation expression, and the entire connections-map as was appended to the arguments vector as its last argument.

However these were not called directly but rather were compiled into s-expressions that were then sent to another function that further expanded the s-expressions by wrapping them with let expressions. This is where the channels finally are actually instantiated and bound to the correct channel names. To give an example:

```
(entities
   (actor feeder (has-initial-tokens ))
   (actor printer (print-actor ))

   (network
     (con (feeder :out) (printer :in-0) {:initial-tokens [1 2 3]})
     )
   )
```

Listing 5: A listing showing the entities expression containing a feeding actor and a printing actor. These two are then connected in the network clause.

Would create the connections-map: `{:feeder {:out channel-0}, :printer {:in-0 channel-0}, :number-of-channels 1, :channel-arguments {:channel-0 {:initial-tokens [1 2 3]}}}`, and the entities macro would get expanded to:

```
(let [channel-0 (chan [1 2 3])]
   (has-initial-tokens {:feeder {:out channel-0},
       :printer {:in-0 channel-0},
       :number-of-channels 1,
       :channel-arguments {:channel-0
           {:initial-tokens [1 2 3]}}}
       )
)
```

Listing 6: Showing the the macro-expanded version of an actor that has initial tokens on its output channel: channel-0.

which then would be the actual final s-expression that would get executed at run-time.

This solution also yielded a working implementation of the Smith-Waterman algorithm. It did however have the major drawback of not supporting the instantiating and connecting of actors programatically. The programmer had to write out every single actor and bind it to a variable, only to then connect every actor to the other correct actors channel by channel. There were some trials involving creating symbols programatically, where the programmer would bind actors to variables in for loops, but this never really went anywhere. The programmer would for example have to concatenate a string and a loop variable to create unique variable-names that each actor would then get bound to, but this solution proved very clunky and the programmatic creation of connections did not work at all.

3.3.3 Iteration Three

The previous approach was to limit the programmer as much as possible, letting them just use the special commands implemented i Cactus. This yielded code that was very homogeneous, and structured, but limited the programmer a lot. The final implementation took a more open approach.

This version changed the entire entities macro. It was given a new name: `exec-network`, and instead of having reserved keywords for creating actor calls that got sent the connections-map. The only restriction on the programmer was that the final s-expression in `exec-network` would return a list of connections. To give an example a network could look something like:

```
(exec-network
   (let [has-init (has-init-tokens )
         buffer (buf )
         ]

         (list
           (con (has-init chan-stripe) (buffer in)
               {:initial-tokens [1 2 3]}
               )
           )

     )
   )
```

Listing 7: Showing the updated exec-network expression that replaced the entities expression.

This was made possible by two main ideas. The first: make the only restriction on the network be that the last s-expression inside `exec-network` returned a list of connections, and make the `defentity` (previously `defactor`), return an anonymous function that took, as it's only argument, a connections-map. This made it easy to create actors in a two step process, where first the actor would get called with its initial arguments. Then a second call would happen where the connections-map got sent as the only argument.

This design choice made it possible to make instantiating calls in for loops, and then connect the actors in for loops using the con keyword.

If you for example needed 100 incrementor-cells (an actor that would simply receive an integer i on the in channel and send i + 1 on its out channel) in your network, you would create a for loop in the first let expression in the exec-network expression:
`(let [cells (for [i (range 100)] (incrementor-cell ))])`.

You could then inside of the let expression connect each incrementor-cell to its neighbour in a separate for loop:

```
(let [cells (for [i (range 100)] (incrementor-cell ))]
     (for [i (range (dec (count cells)))]
       (con ((nth cells i nil) out) ((nth cells i nil) in))
       )
   )
```

Listing 8: Listing showing the connecting of actors inside of a for loop.

If you now would send in the token 0 at the in port of the chain 100 would get sent out from the last cell in the chain.

The relative flexibility of this implementation over the last one was obvious. Scaling a network was now only a matter of changing one variable, instead of manually adding more connections one-by-one.

3.3.4 Parking DataFlowChannel

After we got the ability to programatically create actors, we started experimenting with larger numbers of actors and the programs would sometimes deadlock. After some analysis we realized that this occurred when we had more than 8 actors, which had yet been tested in the Striped version of Smith-Waterman. This occurred because the actors were busy waiting. An actor started by checking it's guards by calling the `(size?  n chan)` function on the `DataFlowChannel` and depending on if the pattern matched or not, executed the actions, looped back to the guards and repeated. Since `go` processes were dispatched to a Java `FixedThreadPool`, any processes that was busy waiting would occupy a thread for the duration of it's lifetime. The number of threads `core.async` was originally 8 which was why it worked with the Striped Smith-Waterman which used exactly 8 actors. To see if the size of the `ThreadPool` was the problem, we started by modifying the number of Threads in the ThreadPool and saw that having more actors became possible. However, simply increasing the `PoolSize` lead to two major design flaws.

1. The maximum number of actors that can run in a Cactus program depends on the OS, resources and hardware.

2. The `PoolSize` became dependent on the amount of actors in the program.

Due to 1. the same cactus program could behave differently on different machines or OS'. If one machine could support 12000 threads and instantiates 12000 actors, a less powerful machine might not be able to run the program at all. One of the most beneficial aspects of a dataflow language is that the computation can be seamlessly distributed over hardware, since actors compute asynchronously. This becomes difficult if the mechanism would be hardware dependent or dependent on free memory. Further, in CAL, creating a large number of actors is often required. For example an FIR filter written in CAL is often written with $>$ 2000 actors running simultaneously. Theoretically, the limiting factor for the max `PoolSize` should be the OS, per user Thread limit, but what we found was that some machines with a thread limit of 65000 would still deadlock with a `PoolSize` of above 1024.

Having a `PoolSize` dependent on the amount of actors as in case 2. would introduce further complications. Any program written in Cactus would require a global state, then before instantiation of the actors it would need to calculate how many actors will exist, and then set the `PoolSize` to this. In some programs this might not be known before run time. Just setting the `PoolSize` arbitrarily large would solve this issue but that would mean that a small Cactus program with few actors would start a lot of threads no matter what, while not only being inefficient this would again mean hardware dependency as in 1. Instead we resolved this by further modifying `core.async` and reintroduced a parking operation.

In the original any call to the macro `go` would expand to a state machine function and every call in the `go` block to any parking operators `<!` and `>!` (`(take! chan)` and `(put! chan)`), would park the process until their conditions were met. `take!` would park if there was no value in the buffer to take, and would run the process again when a token was put on it's channel.

`put!` would park if the channel was full, and run again if a token was removed from its channel.

In the state machine, all calls to parking operators were numbered $1..n$ which became the states that the go process could be in. The state machine function itself is a essentially a function inside a mutable Java `AtomicReferenceArray` where each index contained some information. One index would contain the current state the `go` process was in, and one would contain any return value a call to `<!` or `>!` might have. The state machine would call whichever of the parking functions was in it's current state index, these were calls to the `(DataFlowChannel)` were the `take!` and `put!` functions were implemented. If eg. a call in the `go` process was trying to `take!` a value from the channel, the `DataFlowChannel` would check if there was an element in the buffer, if the buffer was empty (and the process would park) the call to `<!` would return `nil`. If there was a value in the buffer the `<!` call would return that value. The state machine would check the return value, if it was not `nil` and there was a token in the channel to remove, it would place it at the return value index in the `Array` and update the current state index. Lastly it would recur and re-run with the new updated state. When it would recur, it passed the updated `Array` and replaced the call to `take!` with the return value at it's return value index and proceed from the new current state from its' current state index (That is, proceeds execution from whatever came after the call to `take!`). Since the recursive call would happen internally, the executing thread would never have been released. Otherwise, if the return value would be `nil` the state machine would simply exit. This would park the process and release the thread allowing it to go back into the `ThreadPool` and execute other processes.

One last requirement of the state machine, was for the process to start again if their conditions are met. We wanted the parked `take!` call in the previous example, to start executing again once a value was `put!` on it's channel. The way this was resolved was by including call-backs to the parking function calls. The call to `take!` would include a call-back function when it was called in the `DataFlowChannel`, if the `DataFlowChannel` would see that the there was no token on the channel to dequeue, it would save the call-back. The call-back function itself was wrapped around the state machine as it was when the`take!` call happened. As the process was parked, a `put!` call from a different process on the `DataFlowChannel` would look if there were any saved call-backs and if there were, it would dispatch the call-back to the thread pool. The dispatched call-back would then be picked up by any free thread in the pool and get executed, this re-runs the state machine in the call-back (the state machine of the `take!` call). This time however, there would be a token on the channel for the `take!` call to dequeue and instead of returning nil, the `take!` would return the value and the process would continue executing.

Once we understood this, we modified the library further to accommodate the `DataFlowChannel` functions that actors required. Guards in Cactus would first check if there are n tokens available on the channel with the predicate `(size? n chan)` call. If the call returned true, the actor would peek at the token i^{th} tokens with `(peek! i)` and bind them to local variables. After binding the variables and executing the action, it would then consume the tokens from the channel by calling `take!` on it. By design, the only time `(peek! i)` would run in any Cactus program is when `(size? n chan)` has returned `true`, and then it would peek at most at the first n elements. If the guards would be true, `take!` would consume only the first n tokens. This would mean that the only function that needed parking was `size?` as the other dequeing functions would always be able to execute when `size?` returned true. We created a call-back for every call to `size` and a `size` function in `CircularArrayBuffer` that returns the number of elements. Calling `(size? n chan)` on a channel then simply calls the `size` function on the `CircularArrayBuffer`, and checks if the return value would be $>= n$. If it was, it would return true. The state machine would update and recur, holding onto the thread. However, if there were fewer than n elements in the `CircularArrayBuffer`, the `DataFlowChannel` would save the call-back function from the `size?` call and return `nil`. The `nil` returning would cause the state machine to not recur and the process would park. Much as in starting the process again would be up to a connected actor calling `put!` on the other end of the channel. `(put! e)` appends element e to the `CircularArrayBuffer`, would checks if there are any saved call-backs in the `DataFlowChannel` and if there are it would dispatch them to the `ThreadPool`. It would then always return `true` to it's own state machine, since we have unbounded channel sizes and `put!` itself would never park an actor. Once a free Thread would receive the dispatched call-back, the actor parked at `size?` would return it's `size?` call and if there would now be enough tokens it would maintain the Thread and proceed to `peek!` etc. If of course it would not park again.

We realized that by just adding a some logic in `put!`, we could remove plenty expensive dispatched call-backs to parked actors on the other end of the channel. Take, for example, one putting actor and one receiving actor that are connected via `chan1`. The receiving actor was parked in it's process at `(size? 10 chan1)` waiting for 10 tokens to arrive on `chan1` while the putting actor would put values on the channel and dispatch call-backs to the receiver every time. For every token 1...9, the putting actor would dispatch the call-back, a Thread would be used from the `ThreadPool` and the parked receiving actor would start executing again, it would then see that `(size? 10 chan1)` has not yet been fulfilled and it would, again park. This would happen until the 10^{th} `put!` call caused the receiving actor to proceed. By only dispatching a call-back on the 10^{th} `put!` we would remove 9 unnecessary runs of the call-back function that each would cost and briefly require a free Thread. We accomplished this by including n in a `(size? n chan)`'s call-back and adding a function in the call-back that would return it. Before `put!` would dispatch the call-back, it would check the call-back's n and if the new size of the `CircularArrayBuffer` was still too small, we would know that the receiver could not proceed anyways and we could just skip the dispatch of the call back entirely.

With this parking `DataFlowChannel` we removed general busy waiting of actors. If a Cactus program could run was no longer hardware or resource dependent and a Cactus programmer could theoretically create as many actors as their memory would allow for, but we discovered one caveat. The only parking function was `size?` and an actor written in Cactus would only call `size?` if it's guards required input tokens in order to fire, however, in both CAL and Cactus, it is entirely possible to write actors that don't require any input tokens to fire. For example the controller in the Striped Smith-Waterman algorithm. This actors would never park and even though it only fires once, it would busy wait and hold a Thread from the `ThreadPool`. With free running source actors like this, whose guards don't depend on any input at all, Cactus faces the same issues as before. 4 controllers running with a `PoolSize` of 4 would hold all Threads and any remaining actors would unfortunately not run. This is something that absolutely needs to be fixed for Cactus to become a viable programming language.

4 Research Component

The main focus of this report was not the research component, but rather the exploration of the possibility of creating an actor centered DSL on top of Clojure. Given this there was still an interesting aspect to measuring the relative performance between different numbers of active threads and actors in an algorithm implemented in the language.

Since the goal of the programming model is to achieve parallelization in a manageable way, the performance has to scale with the number of processing elements available to the program.

In this thesis, an implementation of the striped Smith-Waterman algorithm was used as the basis for evaluating the performance of the language under different circumstances.

4.0.1 Research Questions

The hope for this project was to see some type of performance gain when running more processes in parallel. The thought was that the scaling would likely not be linear, but rather there would probably be an apex to the performance curve where the performance would be maximal given the hardware and application. That is, increasing the number of threads or processes running would not always lead to better performance, but at some point rather the opposite, due to overhead when switching between threads or processes.

It would be desirable that the apex moves to a higher number of threads when having more processing cores.

To evaluate whether or not the language scaled in the desired way, these questions were asked:

- Is there a correlation between the number of threads in the underlying thread pool and performance (controlled for number of cores on the processor running the program)?
- How does the width of the stripe in the Smith-Waterman algorithm affect the performance of the algorithm and is there a difference when the number of threads in the thread pool is changed?

4.0.2 Hypothesis

Since performance gains should be made when more computation is run in parallel, the speed of the execution should increase when the width of the Smith-Waterman stripe increases. However, since there are only a couple of processor cores available, and a set number of threads available in the thread pool, the performance increase should plateau when a certain number of threads are run in parallel. Since running one thread on each core would theoretically lead to no context switching (there will probably be some other programs running on other threads on the processor, but these should have little impact when testing between a wide range of threads.) the hypothesis is that the best performance should be had when running a number of threads that corresponds to the number of cores available on the processor.

If one Smith-Waterman actor cell runs on one thread, we hypothesize that the maximum performance would be achieved when the number of actors correspond to the number of cores in the processor (or in the case of some sort of SMT enabled processor, the number of virtual cores) and the number of threads match that number of cores. This should give the theoretical maximum performance since no overhead is used up for context switching, but the computation is still runs in parallel.

4.1 Methodology

4.1.1 128 x 1024 Smith-Waterman

To determine if there was a correlation between the number of actors and threads in the thread pool and performance, several tests were executed. The Smith-Waterman algorithm was used as a testing application since it was pretty straight-forward to implement and easy to scale. The striped version specifically, was used since it can make use of parallelization in a nice way, were the number of parallelized cells can be easily adjusted.

To test the hypothesis the algorithm was run on two strings A and B. A was of length 128 characters, and B was of length 4096.

The characters in the strings were randomly generated. The algorithm was executed over the strings using a variable amount of computing SW-cells and threads in the thread pool.

The algorithm ran 10 times for each configuration and the mean time for each configuration was then calculated. The configurations were, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048 threads in the thread pool (or until memory ran out, meaning no more threads could be created).

For each number of threads in the pool, each of the following widths were tested, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096. We recorded the time starting from the controller-actor first firing until the end-cell got the final token (See 1 for the layout of the actors, and the appended source code to see the implementation details).

The mean time of the 10 runs for each configuration was then computed for each hardware setup. The hardware used was a: MacBook Air 2014 Intel i5-4260U with 8GB of ram and 2 cores, a MacBook Air 2020 M1 with 8GB of ram and 8 cores, a ThinkPad X1 carbon Intel i7-8565U with 8GB of ram and 8 cores and an Intel NUC i7-10710U with 12 cores and 31.1GB of ram. The MacBooks ran macOS 11, the ThinkPad ran arch-linux with kernel(5.10.0-arch-1), and the NUC ran Kubuntu 20.04.

4.1.2 1024 x 1024 Smith-Waterman

To see if the limiting factor to the performance of the Smith-Waterman algorithm, was the number of SW-cells running in parallel, a new test was executed.

Since the maximum number of actors running in parallel is theoretically capped by the number of actors in the diagonal (See: The Theoretical Cap of the Number of Smith-Waterman Cells Run In Parallel) of the matrix constituted by the A and B string, a new test was executed to see if more performance could be gained by changing the length of the strings.

The test was run with the striped Smith-Waterman on two strings of length $1024x1024$ and a couple of different configurations were tried. The configurations were 8, 16, 32, 64 and 128 threads in the thread pool. With 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 actors in the stripe.

The test was then run 5 times for each configuration, and the mean times of all the runs was calculated. This test was not run on all of the configurations, but rather just on the Intel MacBook Air, and the Intel NUC.

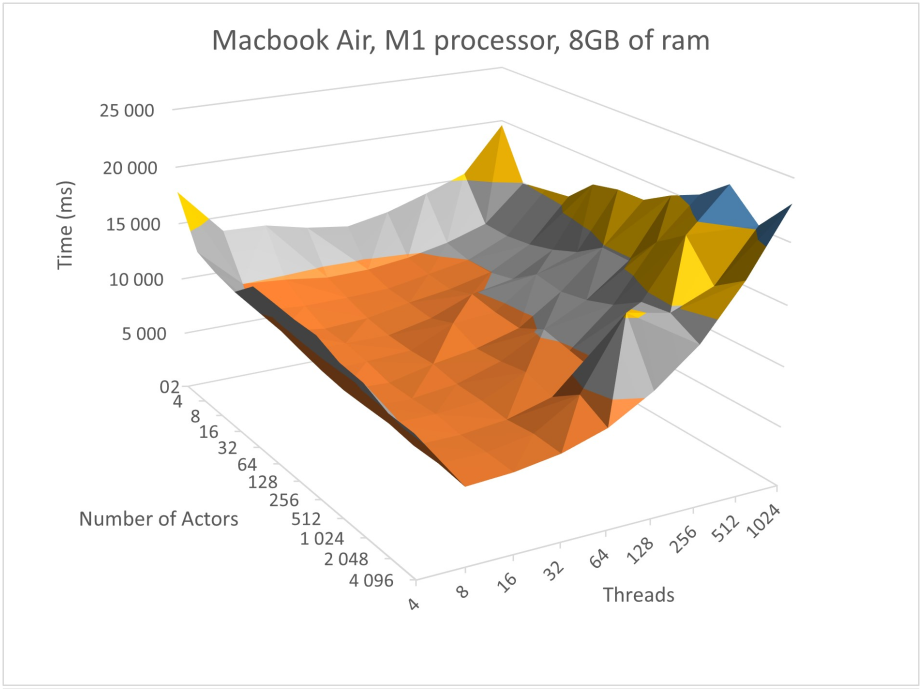
4.1.3 Tighter Interval of Threads

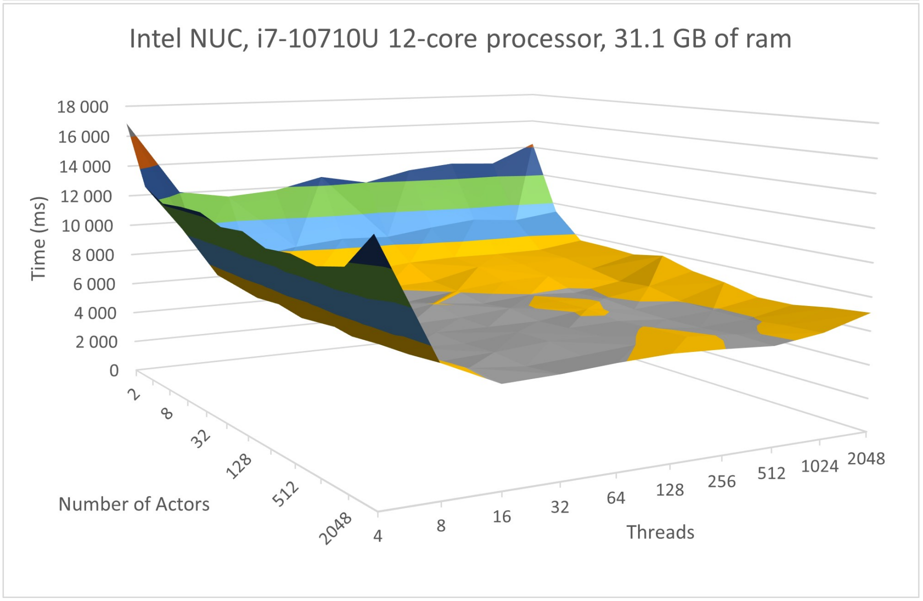
There was also a last test conducted to pinpoint the number of threads best suited for this application. The striped Smith-Waterman algorithm ran 10 times, on two strings A and B of size 128 x 4096. The configurations were slightly different from the previous test using this string size, as this time the thread count varied with incremental steps of one, from 3 to 32 strings using 4 8 16 32 64 128, 512, 1024, 2048 and 4096 actors in the stripe. Each configuration was run 10 times, and the mean times of all the runs for each configuration was calculated.

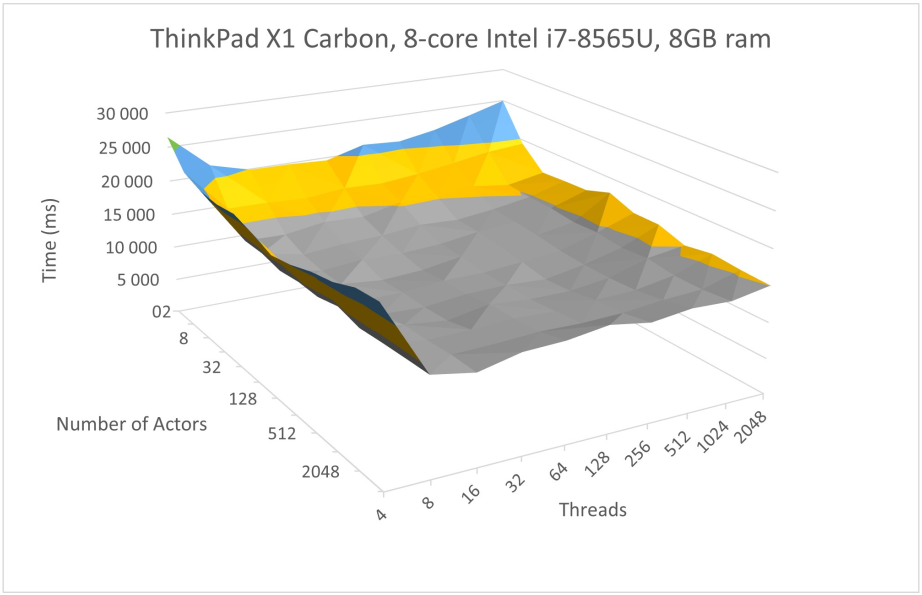
4.2 Results

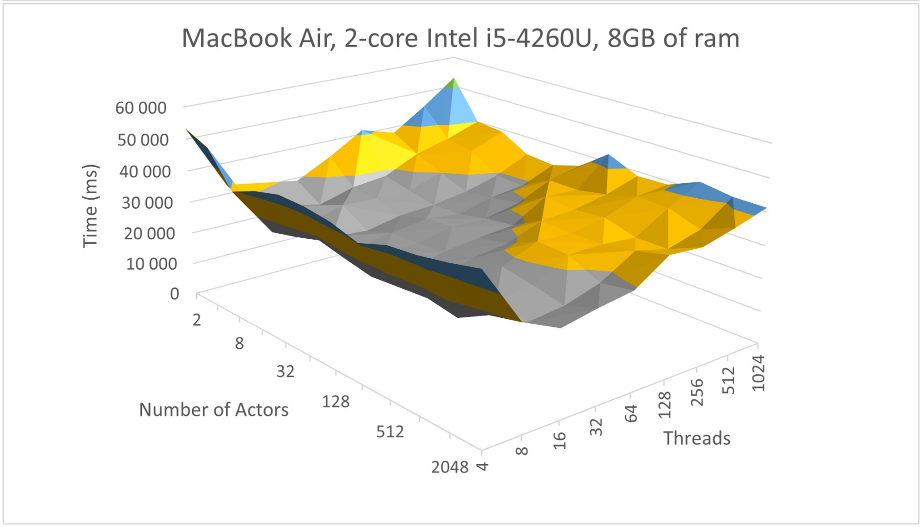
4.2.1 128 x 4096 Smith-Waterman

The results of running the test scripts on the 128 x 4096 strings, are shown in the figures below:

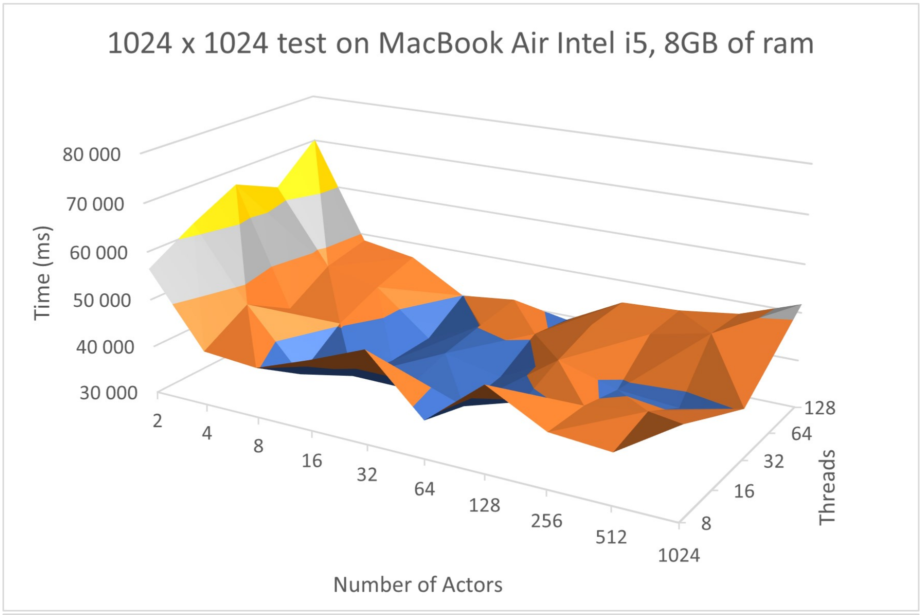


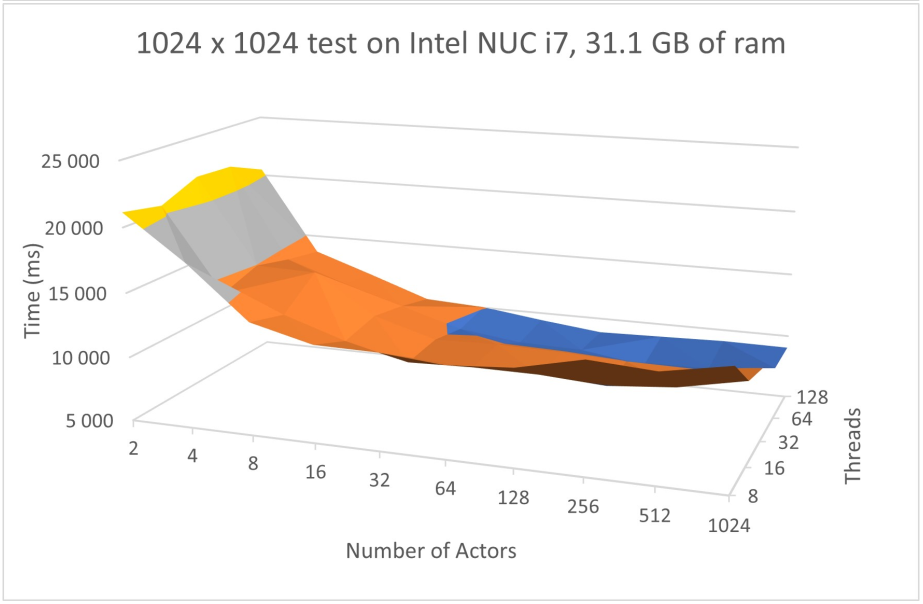



ThinkPad X1 Carbon, 8-core Intel i7-8565U, 8GB ram
Time (ms)
30 000
25 000
20 000
15 000
10 000
5 000
0
2
8
32
128
512
2048
Number of Actors
4
8
16
32
64
128
256
512
1024
2048
Threads


MacBook Air, 2-core Intel i5-4260U, 8GB of ram
Time (ms)
60 000
50 000
40 000
30 000
20 000
10 000
0
2
8
32
128
512
2048
Number of Actors
4
8
16
32
64
128
256
512
1024
Threads

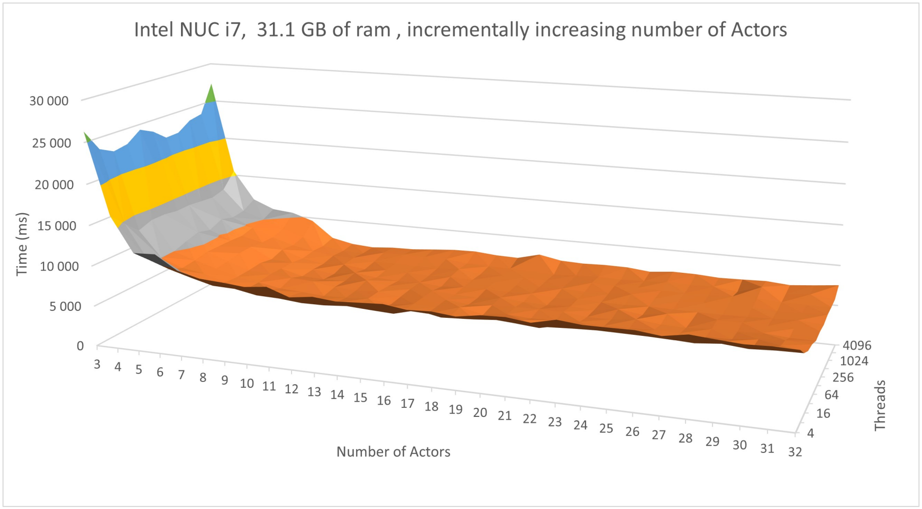
4.2.2 1024 x 1024 Smith Waterman





4.2.3 Running the Algorithm With Different Thread Pool Sizes

The results of running the algorithm with incremental steps of one in the thread pool size is shown in the figure below:



5 Discussion and Future Work

5.1 Thoughts on the Language

The goal of the Cactus language was to create a DSL that allowed a CAL style actor language to be implemented in Clojure, and that language to be able to support a driver application. In this case that driver application was the striped Smith-Waterman algorithm. Another goal was to also see some improvetment when running the algorithm using more processing cores.

The goal set, to allow for the Smith-Waterman algorithm to be run in the language, and scale with more processing entities, was met. The fact that the algorithm was chosen from the start as a target application, was a good driving force for features that needed to be implemented in the language.

It was also a good indication of weather the language was heading in the right direction, for instance during the second iteration of the macros, the language was syntactically quite sweet as a lot of sugar had been added, but the language was also quite limited, as the features where written in a way where the programmer had to use specific keywords, and not just normal Clojure.

The target application then served as a guide when the language became clunky to use, and also clunky to develop.

This led to the third iteration, which was more flexible for the programmer and made the algorithm easier to implement and scale.

5.2 128 x 4096 Smith Waterman

The first test running the Smith-Waterman with a variable amount of threads in the thread pool resulted in some interesting finds. An obvious thing, that was quite expected, was the fact that for nearly all configurations, using a thread pool that only had 4 threads was the slowest of all runs run. This is of course because of the lower amount of parallelization. Another quite obvious find was that there was a "bounce back", where the performance seemed to go down instead of up when adding more threads. This is likely because of the added overhead, and gave proof of the fact that simply adding more threads to the thread pool was not the right way to scale Cactus.

Both of the Mac runs seem to suffer when adding too many threads (that is more than the NUC and the ThinkPad). There was no telling if this was because of the operating system itself or if it was just a coincidence.

The MacBook running the two core Intel i5 was the least well suited for running the SW-algorithm. It was the slowest at finishing the task, which was to be expected with a lower core count. There also seemed to be more penalty for increasing the thread count on the macs compared to the other two computers. As the Intel powered mac performed almost as bad running on 1024 threads as running the program with just 4 threads. Looking at the M1 powered Mac the penalty seemed to be even greater as the test with 1024 threads was even slower than the 4 threads run. However even though the M1 run with 1024 threads was the slowest on the M1, the M1 still beat the Intel i5 Mac with every configuration (as did the 12 core Intel NUC). This was quite in line with the theory, given more parallelization, more speed is expected up until the breaking point where adding more threads would instead make the performance suffer because of overhead. However when more processing cores are available, more threads can be used.

Something else that was noteworthy was the fact that the language seemed very well suited for a large amount of actors. There seemed to be little slowdown in execution speed when using more actors, even though the number of threads stayed the same. This observation lead to the question of where the actual limit resides.

However there seemed to be some spread among the configurations regarding the performance of a large amount of actors. The M1 Mac seemed to get slightly faster or at least stay as fast as the width was increased for all thread counts that were lower than 64. This could be a coincidence as the performance gains were minor. However significant slowdowns were not observed even for the widest stripes.

The ThinkPad and the NUC seemed to be performing slightly worse or equally as good when the width was increased. However these performance losses were also minor, and could possibly be contributed to random chance. Once again significant slowdowns were not observed.

The Intel powered MacBook Air seemed to struggle a bit with a higher actor count than the other machines. The slowdown seems to start when having more than 2^4 actors, and progressively get worse as more actors were added. The same behaviour could be observed in the 1024 x 1024 test. A possibility is that the other machines would suffer the same loss when adding more actors, and that the number at which the performance loss start, is simply higher when having more computing cores.

For nearly all configurations the speed of execution plateaued somewhere around the width of 16 to 128.

When running the striped Smith-Waterman algorithm, the goal was to get the algorithm to run a diagonal along the stripe in the matrix. This was because that way no SW-cell would need to stall to wait for data from the cell to the west of it. To achieve this, a warm up period is needed. Imagine that all cells fire or test their guards synchronously, given a width of the stripe n, the west most cell needs to fire n times for the east most cell to actually start computing. This is of course because the values needs to propagate from the west most cell to the east most cell before the east most cell can start computing its' values. This realization raised the concern that there could be some capping of the performance because of the length of A, and thus the length of that diagonal strip that could fire concurrently.

Running the algorithm with a stripe width of 4096, that is the full length of the B string, could possibly cap the performance, since the maximum number of cells that could fire at once would only be 128 (or something close to that). If that was to be true, that would mean that there was no point in having a wider stripe than the length of A (and of course the length of B). To see if that new hypothesis was true, more tests were run using different string lengths.

It was however, noteworthy, that the performance seemed to stay roughly the same when having too wide of a stripe (up to a different point for different configurations).

This fact is probably not affected by the weight of each computation in each cell, since the actor tests its' guard and only fires if the guard is true, the weight of the computation in a cell that does not fire would not impact the performance of a program. That is of course as previously discussed, if the cells have input channels that need to be checked for tokens.

However this hypothesis was not tested, and would be an interesting topic of future work.

To further test the performance of the language, and to see if the new hypothesis that the length of A was limiting the performance, a new set of tests were performed.

5.3 1024 x 1024 Smith Waterman

Running the same algorithm on a different sized string seemed to have little to no impact on where the apex of performance lied. The NUC had its best result when running the 1024 x 1024 string with 64 threads in the thread pool and with a stripe width of 256. This roughly corresponds to the result of the run on the 128 x 4096 string, where the apex was at 16 threads and a width of 128.

The Mac performed slightly differently, where the apex for the 1024 x 1024 strings was at 16 threads and 8 actors in the stripe, compared to the previous run where the apex was at 32 threads and 16 actors in width.

To be noted is that the Mac run generally performed worse when making the stripe wider than approximately 2^4. This compared to the NUC where the performance generally stayed the same when adding more actors to the stripe.

This find, points to the fact that the size of A and thus the length of the diagonal stripe that should be running once the algorithm has been executing for some time, would not be a bottleneck in performance, and it rather seems like the best performance for the MacBook was to be had at around 2^3 to 2^6 in width, with a thread pool of size 16, as booth of the runs suggested this.

Meanwhile the Intel NUC seemed to fair better with more actors, and the best performance seemed to be had around 2^6 to over 2^{12} actors in width with a thread pool somewhere in the range of 16 to 512 threads.

To further test the hypothesis that the number of threads should roughly correspond to the number of cores available to the program, a last test was performed on the Intel NUC. That test was running the same algorithm on the 128 x 4096 strings, but for each run incrementing the number of threads in the thread pool by one each run.

5.4 128 x 4096 With Variable Thread Count

The test was in line with the previous tests and it showed once again that the scaling of the algorithm in Cactus was not a problem. The best result for running the algorithm on the Intel NUC was to be found at 21 threads and 128 actors in the stripe. This was once again in line with what the previous tests had shown. When Summing the times of execution for each thread, 18 threads seemed to be the best number of threads for fast execution. However the penalty for adding too many threads was again low, while not having enough threads led to major performance losses. Running the program with 18 threads was more than twice as fast as running it with just 4 threads. This further solidified the fact that Cactus scaled quite well, and that having a thread count that is somewhat close or slightly higher than the number of cores is beneficial.

5.5 Future Work

The Cactus language is at this point very much a proof of concept. The goal of creating a programming language well suited for parallel computing has been met. There are however some things that the language has yet to implement. Most notable is the ability to scale a program over multiple machines. The actor model is quite well suited for distributed computation, be that distribution over multiple cores on one machine, or multiple machines entirely. The possibility for Cactus to scale over multiple machines seems promising but was not explored in this thesis.

The current implementation of sub networks inside of Cactus networks is clunky, as the output from the nested network has to be explicitly declared as a network output. It would be nice if outputs from one network to the next could be as seamless, as just connecting two normal actors. There was however not enough time left to figure out a better solution.

As described in 3.3.1, actors that do not depend on any input to fire, or free running sources, will indefinitely busy wait. Since the controller actor does fit this criteria, we realized that the performance of Striped Smith-Waterman had been negatively impacted. Even though it only fires once, controller would busy wait and hold a Thread for the duration of the program. This e.g. prevented us from running Smith-Waterman with less than 3 Threads as well as negatively effecting performance across every test. A future solution for this specific case would be simple: make the controller's guards dependent on a single token on it's channel, instantiate an input channel with exactly one token, consume it and fire. However, a more general solution would require further modifications to the language.

6 Conclusion

All in all, we have created a usable data-flow language written in Clojure. Further we implemented a data-flow version of the Smith Waterman algorithm as an application topic and used it to display our language's scaling with more parallelization. This language is quite 'tinkerable' meaning that any entity researching data-flow languages could fork Cactus and try out design changes and different approaches without having to deal with re-writing a compiler etc.

References

[1] Simone Casale Brunet, Endri Bezati, and Marco Mattavelli. High level synthesis of smith-waterman dataflow implementations. pages 1173–1177, 03 2017.

[2] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23:156—-161,, 2007.

[3] Daniel Higginbotham. Writing macros | clojure for the brave and true, 2017.

[4] C.A.R Hoare. Communicating sequential processes. Technical report, The Queens University, Belfast, Northen Ireland, August 1978.

[5] Johan Eker Jörn W. Janneck. Cal language report specification of the cal actor language. Technical Report UCB/ERL M03/48, University of California at Berkeley, Berkeley, California, December 2003.

[6] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[7] Computer History Museum. The babbage engine. 2021.

[8] T. F. SMITH and M. S. WATERMAN. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195—-197, 1981.

[9] T. N. Theis and H. . P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[10] Wikipedia. Dataflow. 2020.

7 Appendix

7.1 Language Description

7.1.1 Basic Structure of an Actor

There are two types of entities in Cactus, actors and networks. They share some similarities in terms of functionality and syntax. Lets take a look at a basic actor:

```
(defentity relay-once [] [in] ==> [out]
  (defstate [fired false])
  (defaction in [a] ==> (guard @fired)
      (>>! out a)
      (-- fired true)
    )
  )
```

This is a very basic actor that simply relays a token exactly once. If it receives more then one, only the first gets relayed. This actor takes zero arguments at instantiation, however an unlimited number of arguments could be required if they were declared inside the first vector in the `defentity` expression. The entity requires one channel to be connected on the in-port and one channel to be connected on the out-port.

7.1.2 Defining Actor State

The state is defined as a vector of variable names and values inside the `defstate` expression, which has to be placed right after the vector of output channels. To use the variables defined in the state, the variable names have to be prepended with an `@`, like in the defined guard.

7.1.3 Defining Actor Guards

The `guard` s-expression takes exactly one expression as its argument, which has to be a predicate. It may contain the arguments, tokens from channels or state variables. The guard is checked when there are enough tokens on the input channel for one firing. If the guard is true, the action will fire. If the guard is false, it will not.

7.1.4 Defining Actions

An entity can contain multiple actions. Each action has to specify a number of channels, and a variable vector corresponding to each channel. Given the actor above, the actor will wait until there is one or more tokens on the in-channel. If there is at least one token, and the actor has not yet fired, the actor will fire. If the bindings vector would contain two variables, the actor would wait for two available tokens before firing. In the same fashion if there were more channels specified as input-channels, say in-0 and in-1 and both of those had bindings vectors which contained say two variables. Then the actor would wait for two tokens to be available on each channel before firing.

The `defaction` s-expression has to begin with the specification of input channels and bindings vectors and an optional guard. After that the programmer is free to add whatever normal Clojure code they want. There are however some keywords that don't make sense outside of the scope of an action. Those are `(- ... )` and `(»! ... )`, where the first one indicates an update to a state variable and follows the syntax: `(- state-variable new-value)`. Note that when updating the state variable no `@` is required before the variable name. The second one is the output statement, and is used as follows: `(»! output-channel value)`.

7.1.5 Execution of Networks

To execute a program written in cactus, one uses the the `exec-network` keyword. An example of a very simple network would be:

```
(exec-network
  (let [c0 (has-init-tokens )
        c1 (relay "hej")
        p0 (printer "")
        ]

        (list
          (con (c0 out) (c1 in) {:initial-tokens ["hej␣d"]})
          (con (c1 out) (p0 in))
          )

        )
   )
```

Which is a network where one actor that simply has initial tokens is connected to a relay, that relays the tokens to a printing actor. The connection between c0 and c1 contains a token which has the string "hej då" as value. That token is sent to the printer via the relay, and the printer prints out the value.

To use the `exec-network` one creates instances of entities by calling their names. For instance to create the printer actor, we bind the return of (printer "") to the variable p0. Since the printer actor takes one argument, namely the prefix to whatever string it receives as a token on its input port, one such string is supplied. In this case it is the empty string.

When the entities have been instantiated, the programmer is asked to simply return a list of connections via the con keyword. Each con takes two or three arguments. The first is the sending entity and the port on which the sender will place a token. In this case, the first entity is an actor called has-init-tokens and it connects the out port on the has-init-tokens called c0, to the in port on the relay actor called c1. This connection is also given a third argument, which is an arguments-map, containing the key `:initial-tokens`, which will, as the name suggest, be the initial tokens on the channel. There is one more keyword that can be used inside of network entities, and that keyword is `endpoint`, but more on that later.

7.1.6 Defining a Network Entity

There are two entities described above. One is an actor, and the other is a network. To define a network to be used inside of another network one uses the keyword: `defnetwork`. This expression takes, similarly to `defaction`, a number of input channels and corresponding bindings vectors. These will make sure the network waits for the correct number of tokens before executing the network defined inside the expression.

An example of a `defnetwork` could be:

```
(defentity nw1 [] [in] ==> [output]
  (defnetwork in [x] ==>

      (let [feed (feed-one x)
            end (endport output)
            pr (printer "")
           ]

        (list
          (con (feed out) (rel in))
          (con (rel out) (end out))
          )

      )
    )
  )
```

This network could then be used just like a normal actor inside of a `exec-network` expression. There is however two key difference between running a network inside of an entity and simply running it in an `exec-network` expression, and that is the ability to listen for tokens on the input ports, and the ability to put tokens on an output port. Handling inputs works exactly the same as for a normal actor. However to output on a output port one does not use the normal `(»! ... )` form. Instead an `endport` has to be declared. In the example above an `endport` is created for the output port 'output'. The name of the port is given at instatiation and then to send a token on the output channel a token is sent from rel out to end out, which in tern will make the token appear on the output channel. This is also the general case. To output on a output channel, one instantiates an `endport` with the name of the output channel as its only argument, and then tokens are sent to that instance of the `endport` on port 'out'.

7.2 Simple Adder Actor, Cactus and CAL Comparison

```
actor Add () A, B ⟹ C :

        action [a], [b] ⟹ [a + b] end
end
```

```
(defentity Add [] [A, B] ==> [C]
  (defaction A [a] B [b] ==>
    (>>! C (+ a b))
    )
  )
```

7.3 Iteration One

7.3.1 The SW-cell

```
   (def match 8)
(def mismatch -3)
(def penalty -2)

(defn score [a b]
  (if (= a "") 0
    (if (= b "") 0
      (if (= a b) match mismatch)
    )
  )
)

(defn cell-action [nw n w a b]
  (max
   (+ nw (score a b))
   (+ w penalty)
   (+ n penalty)
   0)
 )

 (defn sw-cell [a b w v aln-v name]
    (go
      (loop [nw 0 n 0 i 0];;Set initial state
        (let [new-a (<! a) new-b (<! b) new-w (<! w)] ;;Wait for ports
          (let [
                new-nw new-w
                new-n (cell-action nw n new-w new-a new-b)
                ] ;;Assign new local state and execute body
            (>! v new-n);;Set output
            (>! aln-v new-n)
            ;(println (str i name ": " new—n "\n\n"))
            (recur new-nw new-n (inc i)) ;;Recur
          )
        )
      )
    )
  )
```

7.3.2 The Network

```
(defn -main  [& args]
    (print-actor chan-4-print)

    (sw-cell chan-con-1 chan-con-b1  chan-con-1-zero chan-1-2 chan-aln-1 "0")
    (sw-cell chan-con-2 chan-con-b2  chan-1-2 chan-2-3 chan-aln-2 "1")
    (sw-cell chan-con-3 chan-con-b3  chan-2-3  chan-3-4 chan-aln-3 "2")
    (sw-cell chan-con-4 chan-con-b4  chan-3-4 chan-stop chan-aln-4 "3")

    (aligner chan-aln-1 chan-aln-2 chan-aln-3 chan-aln-4 chan-4-print)


    (fan-out-actor chan-con-b chan-con-b1 chan-con-b2 chan-con-b3 chan-con-b4)

    (>!! chan-str-1 "abb")
    (>!! chan-str-2 "aaa")

    (<!! (controller chan-str-1 chan-str-2 chan-con-1 chan-con-2 chan-con-3 chan-con-4 chan-con-b chan-con-1-zero

 )
```

7.4 Iteration Two

7.4.1 The SW-cell

```
(defactor sw-cell [a-length] [a-chan b-chan west] ==> [value aligner-value]
  (defstate [nw 0 n 0 i 0])
  (defaction a-chan [a] b-chan [b] west [new-west] ==>
    (let [new-nw new-west
          new-n (cell-action @nw @n new-west a b)
         ]
         (>>! value new-n)
         (>>! aligner-value new-n)
         (if (= @i (dec a-length))
           (do
             (-- nw 0)
             (-- n 0)
             (-- i 0)
             )
           (do
             (-- nw new-nw)
             (-- n new-n)
             (-- i (inc @i))
             )
           )
         )

    )
  )
```

7.4.2 The Network

```
(defn -main  [& args]
   (println "started")
   (def A "JLSNDFLSEPFSKDFJESADLFJASIJDFLCMSKSDFPSJEKFSPDJL")
   (def B "HEJA")
   (def width 4)

   (println "B-length:" (count B))
   (println "A-length:" (count A))

   (entities
     (actor controller (controller-actor A B width ))
     (actor stripe (stripe-actor (count A)))

     (actor fanout (fanout-actor ))

     (actor sw0 (sw-cell (count A)))
     (actor sw1 (sw-cell (count A)))
     (actor sw2 (sw-cell (count A)))
     (actor sw3 (sw-cell-printing (count A) (* (/ (count B) width ) (count A)) ) )

     (actor aligner (align-actor A B width))

     (network

       (con (controller :chan-contr-fan-a) (fanout :in-chan) )
       (con (controller :chan-stripe) (stripe :b-chan) )

       (con (stripe :chan-0) (sw0 :b-chan) )
       (con (stripe :chan-1) (sw1 :b-chan) )
       (con (stripe :chan-2) (sw2 :b-chan) )
       (con (stripe :chan-3) (sw3 :b-chan) )

       (con (fanout :chan-0) (sw0 :a-chan) )
       (con (fanout :chan-1) (sw1 :a-chan) )
       (con (fanout :chan-2) (sw2 :a-chan) )
       (con (fanout :chan-3) (sw3 :a-chan) )

       (con (sw0 :value) (sw1 :west) )
       (con (sw1 :value) (sw2 :west) )
       (con (sw2 :value) (sw3 :west) )
       (con (sw3 :value) (sw0 :west) {:initial-tokens (vec (repeat (count A) 0))} )

       (con (sw0 :aligner-value) (aligner :chan-0))
       (con (sw1 :aligner-value) (aligner :chan-1))
       (con (sw2 :aligner-value) (aligner :chan-2))
       (con (sw3 :aligner-value) (aligner :chan-3))

       )
     )

  (while true )

)
```

7.5 Iteration Three

7.5.1 The SW-cell

```
(defentity sw-cell [a-length] [a-chan b-chan west] ==> [value aligner-value]
  (defstate [nw 0 n 0 i 0])
  (defaction a-chan [a] b-chan [b] west [new-west] ==>
    (let [new-nw new-west
          new-n (cell-action @nw @n new-west a b)
         ]
         (>>! value new-n)
         (>>! aligner-value new-n)
         (if (= @i (dec a-length))
           (do
             (-- nw 0)
             (-- n 0)
             (-- i 0)
             )
           (do
             (-- nw new-nw)
             (-- n new-n)
             (-- i (inc @i))
             )
           )
         )

    )
  )
```

7.5.2 The Network

```
(exec-network
   (let [controller (controller-actor A B width)
         sp-cells (for [i (range width)] (stripe-cell (count A)) )
         fo-cells (for [i (range width)] (fanout-cell ) )
         sw-cells (for [i (range width)] (sw-cell (count A)) )
         col-cells (for [i (range width)] (collector-cell ))
         end (finnish-line width (count A) (count B) )
         init (has-init-tokens )

         buffer (buf )
         b1000 (buf )
         ]

         (concat

         (list
           (con (controller chan-stripe) ((nth sp-cells 0) vec) )
           (con (controller chan-contr-fan-a) ((nth fo-cells 0) in) )
           )

         (for [i (range (dec width))]
           (con ((nth sp-cells i) vec-out) ((nth sp-cells (inc i)) vec))
           )

         (list
           (con ((nth sp-cells (dec width)) vec-out) (b1000 in))
           )




         (for [i (range (dec width))]
           (con ((nth fo-cells i) next-fo) ((nth fo-cells (inc i )) in) )
           )

         (list
           (con ((nth fo-cells (dec width)) next-fo) (buffer in) )
           )




         ;connectig the fo cells to the sw cells
         (for [i (range width)]
           (con ((nth fo-cells i) sw-out) ((nth sw-cells i) a-chan) )
           )

         ;Connecting the sp cells to the sw cells
         (for [i (range width)]
           (con ((nth sp-cells i) char) ((nth sw-cells i) b-chan) )
           )



         ;Connectig the sw cells to eachother
```

```
    (list
      (con ((nth sw-cells (dec width)) value) ((nth sw-cells 0) west) {:initial-tokens (vec (repeat (count 
      )

    (for [i (range (dec width))]
      (con ((nth sw-cells i) value) ((nth sw-cells (inc i )) west) )
      )

    (for [i (range width)]
      (con ((nth sw-cells i) aligner-value) ((nth col-cells i) score) )
      )

    (list
      (con (init out) ((nth col-cells 0) vector) {:initial-tokens (vec (repeat (* (/ (count B) width) (coun
      )

    (for [i (range (dec width))]
      (con ((nth col-cells i) out) ((nth col-cells (inc i)) vector))
      )

    (list
      (con ((nth col-cells (dec width)) out) (end in))
      )

  )
)
)
```

7.6 SW-cell in CAL and in Cactus comparison

```
actor SWC (i, j, match, mismatch, gap) A, B, W, NW, N ⟹ V :

        action [a], [b], [w], [nw], [n] ⟹ [v]
        var
                v1 = nw + if a = b then match else mismatch,
                v2 = w + gap,
                v3 = n + gap,
                v = max(max(v1, 0), max(v2, v3))
        end
end
```

```
(defentity sw-cell [a-length] [a-chan b-chan west] ==> [value aligner-value]
  (defstate [nw 0 n 0 i 0])
  (defaction a-chan [a] b-chan [b] west [new-west] ==>
    (let [new-nw new-west
          new-n (cell-action @nw @n new-west a b)
         ]
         (>>! value new-n)
         (>>! aligner-value new-n)
         (if (= @i (dec a-length))
           (do
             (-- nw 0)
             (-- n 0)
             (-- i 0)
             )
           (do
             (-- nw new-nw)
             (-- n new-n)
             (-- i (inc @i))
             )
           )
         )
    )
  )
```

7.7 DataFlowChannel

```
(deftype DataFlowChannel [^ringbuffer buf, ^Lock mutex, ^LinkedList sizes]
  cactus.impl/ReadPort
  (peek!
    [this i handler]
    (.lock mutex)
    (let [val (box (.peep buf i))]
      (.unlock mutex)
      val))

  (size
    [this n handler]
    (.lock mutex)
    (if (<= n (.len buf))
      (do
        (.unlock mutex)
        (box true))
      (do
        (.add sizes handler)
        (.unlock mutex)
        nil)))
  impl/ReadPort

  (take!
    [this handler]
    (do
      (.lock mutex)
      (let [val (box (.plop! buf))]
        (.unlock mutex)
        val)))
  cactus.impl/WritePort

  (put!
    [this e handler]
    (.lock mutex)
    (.offer! buf e)
    (let [iter (.iterator sizes)]
      (loop [sizers []]
        (if (.hasNext iter)
          (let [elem (.next iter)]
            (if (<= (cactus.impl/size-depth elem) (.len buf))
              (do
                (let [func (cactus.impl/fun elem)]
                  (dispatch/run (fn [] (func true)))
                  (.remove iter)
                  (recur (conj sizers func)))))))))
    (.unlock mutex)
    (box true))
```