UPPSALA
UNIVERSITET

IT 14 073
Examensarbete 15 hp
December 2014

Actors and higher order functions

A Comparative Study of Parallel Programming Language Support for Bioinformatics

Staffan Arvidsson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Actors and higher order functions

Staffan Arvidsson

Parallel programming can sometimes be a tedious task when dealing with problems like race conditions and synchronization. Functional programming can greatly reduce the complexity of parallelization by removing side effects and variables, eliminating the need for locks and synchronization. This thesis assesses the applicability of functional programming and the actor model using the field of bioinformatics as a case study, focusing on genome assembly. Functional programming is found to provide parallelization at a high abstraction level in some cases, but in most of the program there is no way to provide parallelization without adding synchronization and non-pure functional code. The actor model facilitate parallelization of a greater part of the program but increases the program complexity due to communication and synchronization between actors. Neither of the approaches gave efficient speedup due to the characteristics of the algorithm that was implemented, which proved to be memory bound. A shared memory parallelization thus showed to be inefficient and that a need for distributed implementations are needed for achieving speedup for genome assemblers.

Handledare: Dave Clarke
Ämnesgranskare: Justin Pearson
Examinator: Olle Gällmo
IT 14 073

Table of contents

1. Introduction

Parallel programming is growing in importance, as the future of computer architecture will be made of massively parallel computers. However, the software developed by parallel frameworks such as MPI and OpenMP demands the programmer to add parallelization explicitly which demands lot of time and effort [1]. Many believe that functional programming is the answer to the question of how to make efficient parallel code in an easier way [2,3]. Pure functional languages have no side effects and this reduces problems like race conditions and synchronization [2]. This thesis will aim to study some of the language constructs in the functional language Haskell[4] and the multi-paradigm language Erlang[5] and how they perform in the field of bioinformatics.

Bioinformatics is a field that is growing and the need for computational resources has grown tremendously over the last few years [6]. This is especially true since beginning of the 21st century when the era of next generation sequencing (NGS) arose and sequencing genetic information can be produced at very low cost. To cope with the increased amount of data that all needs to be analysed, it is essential to both develop better algorithms and to create highly parallel programs that can run efficient on many core systems and big computer clusters. Parallelization of bioinformatic programs are currently implemented mostly in the imperative languages C/C++ but there are no studies that I have found that evaluate the applicability of parallel functional programming. A further discussion of this is found under Section 8 for the interested reader. This thesis will try to assess the possibilities for functional languages to contribute to bioinformatics.

2. Background theory

This section contains a brief introduction to the biology and bioinformatics needed to understand the content of this report. If the reader feels comfortable in these areas already, he or she is free to skip it and go to the next section.

Bioinformatics is the study of biological processes using mathematics, statistics and computer science and the field in its entirety is very broad and diverse [7]. This study is restricted to the field of genome assembly, which is the process of recreating the genetic information of organisms out of small pieces of genetic code obtained by sequencing machines. The genetic information in a cell comprises both DNA and RNA molecules. These molecules are built up by sequences of nucleotides, also called bases. Finding the sequence of these bases can answer many important biological questions, for example finding how different species or individuals are related and causes of diseases, to name just a few. The genome size ranges from a few million base pairs (Mbp) in some bacteria to up to hundreds of giga base pairs in some plant organisms.

Sequencing technology has since the beginning of the 21st century become much cheaper as next generation sequencing (NGS) technology arose. The sequencing machines and methods used today usually yield sequence lengths ranging from 50 to 300 bases for short read sequencers and up to 40 kilobases in some long sequence technologies such as Pacific bioscience [8]. The gap between read lengths given by machines to the complete genome lengths demands for a so called high read depth, meaning that each unique base should be sequenced multiple times (usually about ten to a few hundred times). Thus, the amount of data needed to assemble a complete genome is many times greater than the number of bases in an organism's genome, which in itself is very large. Genome assembly has proven to be very complex and computationally it is NP-hard [9]. To get a more in depth description about the field and algorithms in use, take a look at the article by Pop [6] for a computer science approach and Schatz *et al.* [10] for a biological approach.

3. Problem in hand

There are several different assembly algorithms in use today, including de Bruijn graphs, overlap layout consensus graphs (OLC) and greedy algorithms [10,11]. The greedy approaches have the obvious disadvantage of finding suboptimal solutions but can be used when having only a small number of sequences to assemble. De Bruijn graphs and OLC graphs are the methods used today when dealing with NGS data, where de Bruijn graphs are the method that shows the most promising characteristics when it comes to scaling. OLC graphs searches for matches between all sequences, which of course scales badly when increasing read depth and sequence lengths. De Bruijn graphs on the other hand searches for matches between shorter sequences of a fixed length, which scales much better even when increasing depth and sequencing length. This project will focus on de Bruijn graph assembly, as this approach seems to be the most promising and is the approach taken by many others when trying to optimize assembly [11,12].

3.1 Assembly by de Bruijn graphs

The first step in all types of assembly is to clean up the data and remove reads that have a too low quality score (all machines associate a quality score for each sequenced base). This can also be done to reduce sampling size if the sequencing machine produces too much data, as more data might not always produce better result but it will always increase memory requirements and runtime. Separate programs can do this for you, or it can be part of the assembly program.

De Bruijn graph assembly is usually divided into several steps that each performs a specific task. After the initial cleaning step, the next thing to do is to generate *k*-mers out of the sequences. *K*-mers are simply sequences that are of length *k* and are generated by creating all possible sequences of length *k* from each read. Thus, by having an initial read length of 75 bases and *k* of 55 bases, you will generate 21 *k*-mers. See Figure 1 for an illustration, using *k*=3. *K* is a parameter to the algorithm that will influence the specificity versus sensitivity, a larger *k* give more unique *k*-mers but there is a higher probability of having a read error in the sequence as it grows larger and the memory requirements will increase. A smaller *k* gives a higher coverage of the genome and gives a higher probability that each *k*-mer in the genome is present, but the probability of having identical *k*-mers in several positions in the genome increases, giving a graph with more spurious connections between un-linked parts of the genome. This is illustrated in Figure 1, where having *k*=3 give a match between two different reads, but which is faulty. By having a larger *k* value, this match would not occur and these two sequences would be separated in the graph. The *k*-mers are usually stored in a hash table or similar data structure to facilitate fast lookup and they are stored together with the number of occurrences of that *k*-mer (and potentially some more data).

The actual de Bruijn graph is visualised as having each edge corresponding to *k*-mers and each node/vertex to correspond to the (*k*-1) overlap between two succeeding *k*-mers (Figure 1, part 3). The graph is created by iteratively finding the next *k*-mer (edge) and piece all *k*-mers together into the complete graph. However, the graph will contain both faults based on sequencing errors and missing parts of the complete DNA in the organism, leading to spurious matches to other parts of the genome or *k*-mers not matching at all. It will also contain cycles and other complexities based on repetitive sequences in the DNA and identical sequences occurring on multiple locations in the genome. A step that is often called *graph pruning* will try to resolve errors in the graph based on the number of occurrences of *k*-mers, and other information, but that is out of scope of this project (see Miller *et al*. [13] for a thorough discussion on this topic).

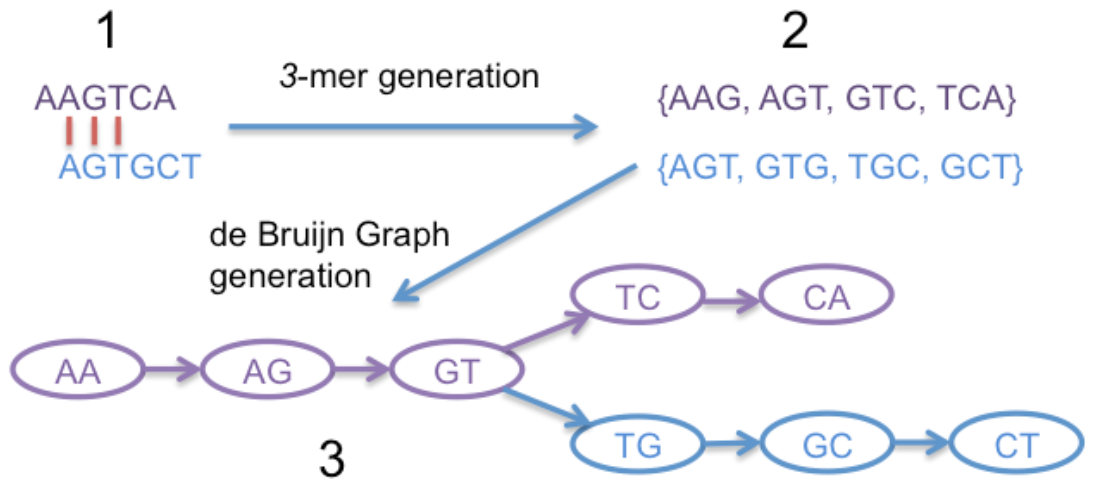

Figure 1 Illustration of the generation of a de Bruijn graph. In step 1 there are two sequences, which have three spurious matches. By using *k*=3, all the possible *3*-mers will first be generated, which can be seen in step 2. Note that these *k*-mers are separated for illustrative purposes; after they are generated there is no difference between two *k*-mers that have the same sequence. The de Bruijn Graph is created in step 3 by finding the (3-1)-base overlap between all *3*-mers, which joins the two input sequences but creating a split end because of their differences.

Once the graph is as created and simplified, the graph can be traversed and yield a set of contiguous segments of DNA (called *contigs*). The remaining step can be part of an assembler or be a separate program by itself and it consists of piecing these contigs together and form the, hopefully, correct sequence of the original genome. This step is called *scaffolding* and uses another type of sequence information called *paired-end* reads. This step will also be outside the scope of this project but more information can be found in the articles and papers referenced earlier.

3.2 Previous work

Assemblers are currently a hot topic in bioinformatics and computer science research. This is due to the large computational resources needed to perform an assembly, which means that the potential gains in runtime and memory requirements are substantial. The memory footprint for assembling large genomes can be as much as 500GB, which was reported by assembly of the white spruce [14], and the execution time even on big clusters can be in the order of days. There are other programs that use different types of algorithms to achieve smaller memory footprints

[15], but do so by sacrificing run time. Most assemblers used today runs on computer clusters or specialized computers with extra memory.

There are several recent papers that report high speedup and minimized memory footprint [11-12, 15-16]. They are all based on C/C++ and Message Passing Interface (MPI) to introduce parallelism. Some older assemblers are only implemented for shared memory architectures, which use OpenMP (Pasqual and Velvet) and Pthreads (SOAPdenovo) [11]. The general algorithm for distributed programs is to hash *k*-mers to different nodes to be able to deterministically store the *k*-mers for easy lookup. Each node has its own hash table or hash map for storing the *k*-mers locally.

One example of a heuristic approach is the one taken by Georganas *et al.* which instead of creating the complete de Bruijn graph to begin with and later simplify it, instead only use well supported *k*-mers which have unique backward and forward extensions [11]. In this way, they create linear chains of sequences instead of tree graphs, which greatly simplifies the algorithm. Georganas *et al.* also show that this algorithm scales very well and claims that it presents efficient scaling on up to 15,360 CPU cores [11]. This heuristic approach might in many cases be a valid approach, as it should almost exclusively produce true contigs and the de Bruijn graph is usually complimented with scaffolding that will piece the contigs together. Another approach that this paper used was the use of Parallel HDF5 reading of the input data, which usually pose a bottleneck for parallelization [11].

4. Algorithmic approaches

As described in the previous section, there are several different approaches to genome assembly as a field and even within de Bruijn graph based assembly. This project started out by implementing two different ways of creating the hash table that corresponds to the de Bruijn graph in assembly programs, described further in the following sections. These two approaches turned out to be very similar algorithmically so implementation was only carried on using the heuristic approach.

4.1 Classical

The classical de Bruijn graph approach is based on chopping up reads into *k*-mers (sequences of length *k*). These *k*-mers are then stored in a hash table or corresponding data structure that allows fast lookup and a count is kept for the number of occurrences of each *k*-mer. This approach can be implemented by one pass over the input data and thus only read the input data once. In a real application, this step is usually preceded by some quality control which filters out bad quality reads and performs statistics on the composition of *k*-mers. In this way, spurious reads can be filtered out and removed from the dataset.

After the *k*-mer hash table is constructed, the program will then create graphs by iteratively finding all possible extensions in both directions. There is however no good explanation in how this is implemented in actual programs in use today, which is one of the reason why this approach was dropped later on in the project to instead focus on the approach described in the following section. Furthermore, the graph that is constructed will be full of complex connections, rising from either spurious sequences given by the sequencing machine and by repetitive parts of the genome. Usually the program would traverse the graph and reduce the complexity by removing loops and other complexities in the graph by using statistics and threshold values. However, the validation of this step would require a lot of testing and outputting results in a graphical manner to make sure that the implementation is done correctly, which would reduce the productivity in the project. This was the other reason for not continuing with this approach, which instead allowed more time to be spent implementing the heuristic version, discussed next.

4.2 Heuristic

This approach was heavily based on the algorithm proposed by Georganas *et al*., which instead of creating a tree-like graph structure only keeps the unique linear parts of the graph [11]. *K*-mer generation is similar to the classical approach, but the algorithm will not only store the *k*-mer sequences but also the bases that are just preceding and succeeding the *k*-mer in the read, which are called forward respectively backward extensions (Figure 2). The algorithm will only keep the *k*-mers that has unique forward and backward extensions and the corresponding de Bruijn graph will only contain linear sequences of *k*-mers.

The de Bruijn graph does not need to be made explicit when using this approach, but can instead reside in the hash table. Contigs can then be created by picking a starting *k*-mer and iteratively traverse the graph in both directions and once there are no more unique extensions, the contig is finished. Traversal of the de Bruijn graph will be done by adding the extensions to the *k*-mer and thus creating an overlap of *k* bases instead of (*k*-1) as in the classical approach (Figure 2). Contig generation is finished when there are no non-traversed *k*-mers left in the hash table. This approach will thus only create contigs that are unique and fully supported by the data. Loops, short branches and other sources of complexity in the classical de Bruijn graph will not be considered until scaffolding begins. However, that will be out of the scope of this project as the main interest is how Haskell and Erlang perform in these applications.

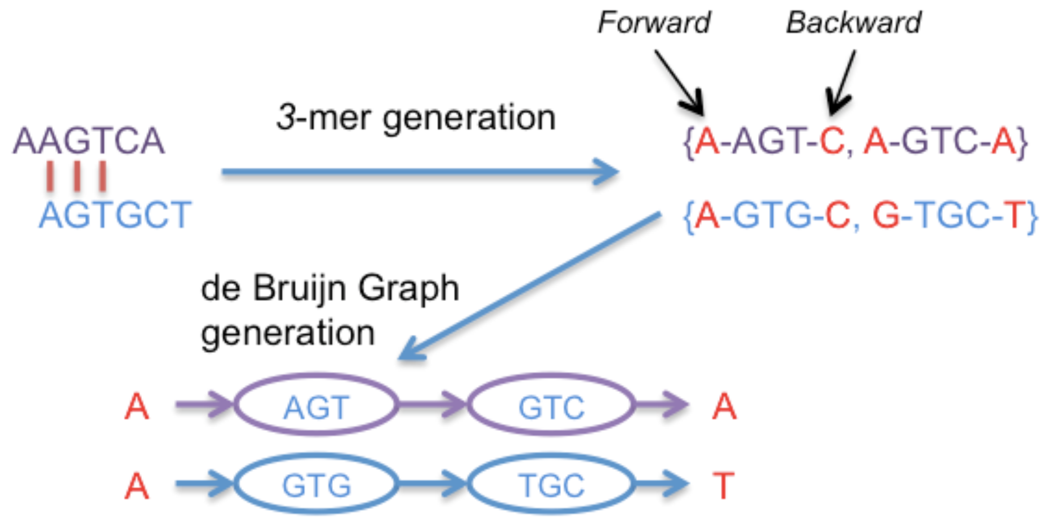

Figure 2 Illustration of the heuristic approach used in construction of de Bruijn graphs. This approach will only yield linear chains of contigs. Forward and backward extensions are marked with red.

4.3 Simplifications made

This thesis work is limited in time and the objective has been to evaluate Erlang/Haskell and their language abstractions in the field of bioinformatics. Thus some simplifications have been made during the project, putting focus on using the data structures that are needed in these types of applications, and not to write programs that are fully functional in terms of assembly of genomes.

The first simplification has been to only use one type of input format of the data. There are several file formats, including different types of binary file types, databases, ordinary text files etc. Furthermore, there are several ways to specify quality scores for input data. But as the quality score only add information for an initial cleaning step, which in itself does not promise any insight into interesting aspects of functional versus imperative programming, these properties have all together been dropped within this project. The program will take a file format called FASTA, which is one of the most known formats, and it is stored as a normal ASCII file.

Another important step in the assembler is that all reads have a *reverse compliment*, which are due to the double stranded DNA and RNA. Each sequence that is read by a machine can come from either of the strands, so by reading one strand we know that the reverse and complemented strand is also present in the DNA of the organism. Real assemblers have to take this into account, some are doubling the input sequences to yield also the reverse compliment of the sequence, and then solve the graph with both sequences present in the edges. This project will neglect this property, as it does not add any interesting aspects.

5. Programming theory

Different implementation strategies were used for Erlang and Haskell to fully evaluate the respective advantages and disadvantages of the two languages. Erlang is a multi-paradigm language that is mainly built for extreme concurrency and lightweight processes and the implementation was thus aimed towards an actor model structure. Haskell on the other hand is a pure functional language, which is highly expressive and use lazy evaluation. This part of the report discusses some important aspects of these programming languages.

5.1 Basic Erlang theory

Erlang[5] is famous for its support for concurrency by using lightweight processes. These processes are much smaller than regular operating system processes and thus provide good means for scalability. The model for concurrency in Erlang is called the *actor model*, in which functions can be made into *actors* and the actors perform computations as response to messages passed to it by other actors. Actors communicate to each other by passing messages asynchronously.

Just as in Haskell, Erlang mostly uses immutable data and the default is to share nothing between processes (actors). But there are some exceptions, the Erlang Built-In Term Storage (ETS) is an example of a data structure that is both mutable and can be shared between actors. This structure also includes a limited support for concurrency, where updates to table entries are guaranteed to be both atomic and isolated. Having a data structure that allows concurrent access from multiple actors removes the need for sending all data to one actor to compute a final result, which often creates a bottleneck in computation.

An important aspect of functional programming, which is found in both Erlang and Haskell, compared to imperative languages, is that they often use recursive data structures. This means for instance that arrays are traded in for lists in the normal case (even though there are some cheats to use array-like structures). Lists are fundamentally different compared to arrays, as they have to be traversed in order to access a value that is not the first in the list and things like getting the length of a list or accessing the last value in the list is *O(n)* operations. However, accessing and adding values to the beginning of a list is made in constant time. The choice of recursive data structures is tightly linked to the absence of iterative constructs (for or while statements) that are completely replaced by using recursion.

5.2 Basic Haskell theory

Haskell[4] is a pure functional language in which code can be divided into either pure or impure functions. Pure code means functions that cannot have side effects and that fulfil referential transparency, that is, the same input to a function must always yield the same output. Impure code on the other hand is all code that does not fulfil these criteria. This means that every time the program for instance reads or write to a file, the code is not pure any more as there are a lot of things that can go wrong and the result of such a function can be different with the same input. Another concept that comes with pure code is that there are no variables, all data is immutable when running pure code. Of course there are ways to go around this in Haskell, but that involves using monads and impure code, which will then remove the advantage of not needing synchronization when parallelizing the code. To show potential advantages and disadvantages of functional code, the implementation has been focused on keeping code pure when using Haskell. Mutable data structures have instead been used in Erlang, which also use immutable data as default but have structures that are mutable.

Haskell is using lazy evaluation, which means that it only evaluates expressions when it actually needs to. Before an expression is evaluated it is stored as a *thunk*, which serves as a way to get the value once it is actually needed by some other part of the program. Lazy evaluation can sometimes be very useful by not having to evaluate everything if it is not used later on, but it can also be a source of decreased performance and increased memory footprint when used wrong, thus the program usually needs profiling and optimization to get it right in the end.

A good aspect of Haskell is that it includes a library for byte strings, which is perfect for applications that rely heavily on reading from files and manipulating strings. The normal String type in Haskell is a list of Chars, and Chars do not have a fixed size because it depends on which type of encoding that is used. As the String is implemented as a list, it also gets all of the laziness that lists have, only evaluating one element at a time and only when they are forced to. By instead using ByteStrings you can guarantee to have a fixed size and take away some of the laziness that is incorporated in the String type. The syntax and functions exported by the ByteString library is very similar to the normal String library so changing between the two types is relatively effortless.

One of the benefits of functional programming is the high level of abstraction, often yielding considerably fewer lines of code compared to identical programs written in an imperative language. One reason for the greater level of abstraction is the presence of higher order functions, which is a function that can take another function as a parameter. Some of the most common higher order functions are map, filter and fold. Map is a function that takes another function and a list as input and applies the function to each index in the list, creating a new list as output. Filter takes a predicate (a function returning either true or false) and a list, applying the function to each index in the list and only keeping the ones that the predicate return true for. Fold takes a binary function (a function that takes two input values) called the accumulator function, an initial value and a list. The fold will first call the accumulator function with the first value from the list and the initial value, the result from the function will then recursively be applied together with the next value from the list. The result from the fold function will be a single value, the value that is returned from the accumulator function after being called with the last value from the list. These higher order functions can in many cases remove the need to explicitly write recursive functions or the need for iterative constructs.

In this project, parallelism in Haskell has been introduced by using the Control.Parallel.Strategies module, which provides a high level approach that exposes parallelism to the compiler. This module gives functionality that can for instance be used together with the higher order function map. To illustrate the simplicity of this approach, take the fictional problem of calculating the Fibonacci number for a list of numbers, which can be written as “`map fib numbers`”. The calculation of Fibonacci numbers is actually dependent of each other, and the calculation of several numbers could be provided much faster by using dynamic programming, but in this example we think of them being independent of each other so they can be calculated in parallel. This can be achieved by simply rewriting the expression into “`` map fib numbers `using` parList rdeepseq ``”, which make each call to fib being sparked into a new thread. The execution of a parallel Haskell program can be visualized by using the tool ThreadScope, an example of parallel execution for the Fibonacci calculation can be seen in Figure 3. The graph shows that the work is done to a great extent on all available cores, yielding a good speedup.

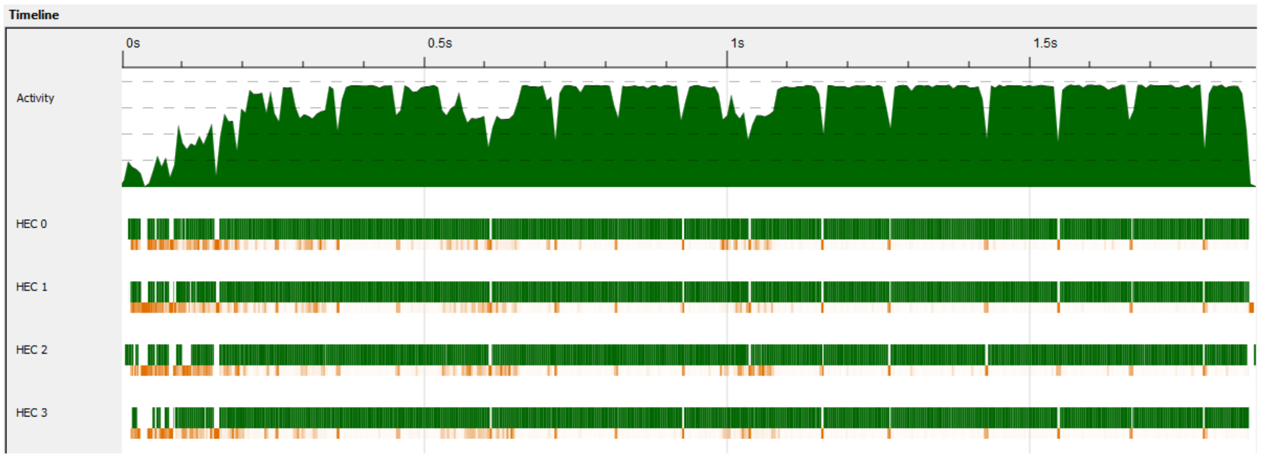

Figure 3 Parallelism shown using the ThreadScope tool [17]. HEC0-3 are the four cores on the CPU and Activity shows the total work done by all cores. The green represent CPU work and the orange/red represent garbage collection activity.

6. Results and implementation

The following sections will describe the results from the thesis work and highlight the advantages and disadvantages of using these languages. Erlang programs have been run at a Linux server belonging to the IT institution of Uppsala University, having an AMD Opteron 16-core CPU and using Erlang R14B04. This system did not support Haskell, so all Haskell programs have been run on a Macbook air, having a Intel i5 dual core CPU, which uses Intels Hyper-threading and thus can give up to four virtual cores. This was not an optimal setup but was the only available one. The data used as input is taken from a real data sample that has been sub-sampled into a size that gave run times that was easier to evaluate (run enough times to get good estimates of the true performance). Details of how this data has been generated can be found in the Appendix under Data generation. All runs using different parameters have been run at a minimum of three times and the shortest runtime is the one used in the comparisons. The shortest runtime is generally used when estimating performance to reduce variance in the results that rise from disturbance from other processes/users, unpredictable cache utilization and other variations in the system.

Parallelization has been focused on the *k*-mer generation step, which has shown to be the most time consuming part of this application [11]. There are multiple potential ways to implement contig generation in parallel but the time constraint of the project forced these ideas to be dropped. Potential strategies would be to create parallelism either at a low level as when extending a *k*-mer in two directions or a higher level by creating multiple contigs in parallel.

6.1 Erlang

To evaluate the actor model in Erlang, two versions were created of each program; one that does everything in serial without any concurrency and one that allows concurrency. The first approach was to go by the standard “share nothing” approach, using a general balanced tree (Gb-tree), see Erlang manual for module “gb_trees” for reference. This approach is further discussed in the following section. Trying to improve the performance further was done by instead using Erlang’s built in term storage, which has some support for concurrent access from multiple processes. Further information is under the coming sections.

6.1.1 General balanced trees – serial version

Data in Erlang are generally immutable and local to processes. The first implementation used one of these data structures for storing the *k*-mer table, namely a map that uses hashing for searching and falls under the category general balanced trees (implemented in the module gb_trees). This data type facilitates fast lookup and efficient space behaviour according to the Erlang reference manual and was a good way to start.

The implementation is divided into tree separate modules:

- A parser module that reads the input FASTA file.
- A contig generation module that traverses a de Bruijn graph and returns the contigs.
- A main program that calls the parser module and creates the de Bruijn graph from the input and then calls the contig generation module.

A performance profile of these diverse tasks can be seen in the left set of stacked bars of Figure 4 for the sequential program. The slowest step by far is the creation of the graph, so that is the step that has the most gain of parallelization.

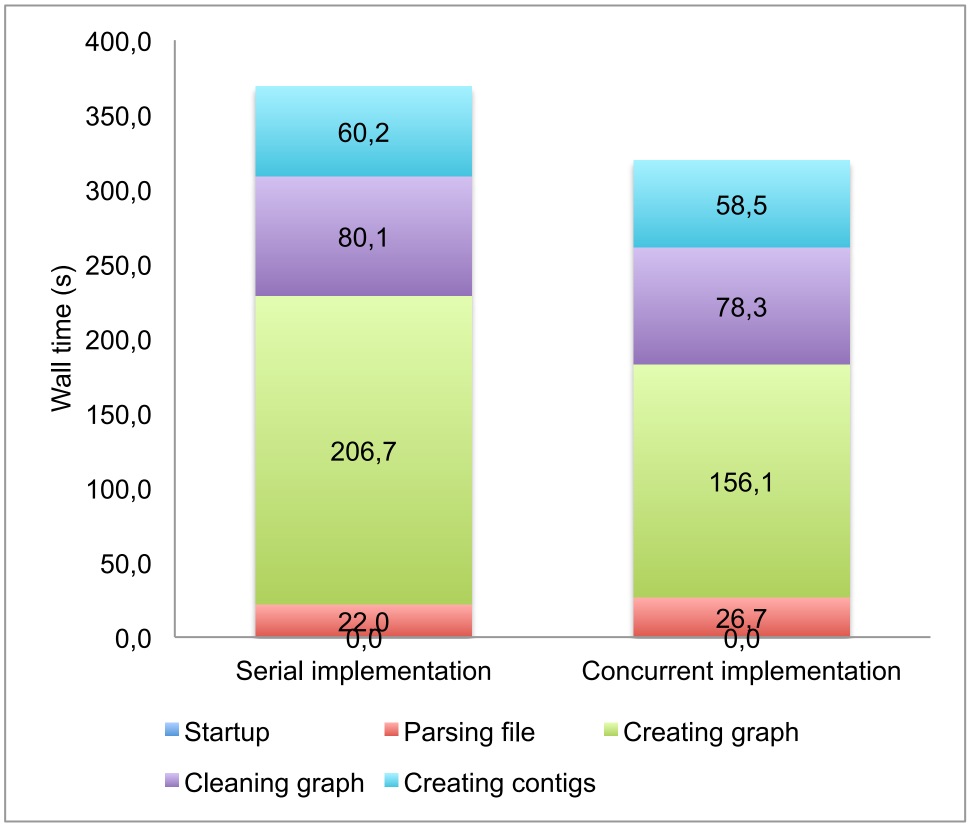

Figure 4 Timing of each part of Erlang code using GB tree as data structure. The left set of stacked bars shows the timing for the sequential implementation and the right set of stacked bars shows the corresponding for the concurrent implementation when using two worker actors. Timings of the concurrent versions for other than two workers should only differ in the green section as the rest is done sequentially. Input is 500 k DNA sequences, 44 bases long and *k*=25.

6.1.2 General balanced trees – concurrent version

The default paradigm of the Erlang language is that processes do not to share any data with each other. When two processes need to communicate, the data is copied from the sending process to the receiving in an asynchronous fashion. There is thus an overhead incurred when processes have to send data between each other. Moving from a serial version into a concurrent version will thus add a time penalty and extra work for the memory that needs to copy what can be quite large amounts of data but the additional parallelism might outweigh this cost. However, the main disadvantage will be that all results must be sent to one process in the end so the final result can be compiled.

When implementing a concurrent program, the different actors have to be identified. In this implementation the following actors were used:

- *Parser* (1): reads the input FASTA file and sends records to *Worker* actors.
- *Workers* (1..N): receives records and create *k*-mers and forward/backward extensions for each *k*-mer. Once done, they send their result to the *Reduce* actor.
- *Reduce* (1): receives data from workers and adds them to the GB-tree. This actor will put together all data into the final result and once finished, return it to the *Coordinator*.

- *Coordinator* (1): Starts up the other actors, coordinates work between them and signals to actors when events occur. The result will be sent here to finally do the cleaning of the graph, which in this implementation is done serially.

A performance profile when using two worker actors can be seen in the right set of stacked bars in Figure 4. Important here is both that the overall runtime has decreased with more than 13 % and that the parsing step takes a little longer than with the serial implementation. The increase in parsing time is because parsing is overlapped with work done by worker actors. This is performed by setting a parameter called chunk size, which stops the *Parser* after reading the set number of records and forces it to sends them to the next *Worker* in the worker queue. The *Worker* will start to work with producing *k*-mer data as soon as it receives data and send its result on to the *Reduce*. The overlap of reading and work induces a small overhead of about 20 % for the read portion of the program, but allows the *Workers* to start earlier. *Workers* are getting data sent to them based on a queue that is implemented in a round robin fashion, using a simple list.

To evaluate the gain in using the actor model compared to the serial implementation, several runs was performed, which gave the speedup graph that can be seen in Figure 5. When compiling the speedup metric, only the parsing and creation of the graph was considered, as there are no changes in the code for cleaning the graph and create the contigs (which is still done serially). The maximum speedup was only a mere 14 % and the speedup was reduced significantly when using more than five worker processes, even though the program was run on a 16-core machine. One aspect of this very poor scaling is due to the need for *Reduce* to compile the final result, which puts a lot of work on only one process that executes sequentially. That process will be a limiting factor for this application and one way to improve on this would be to remove the need for having a single process to compute the final result. This is what led to the implementation using the Erlang built in term storage, which is discussed in the following sections.

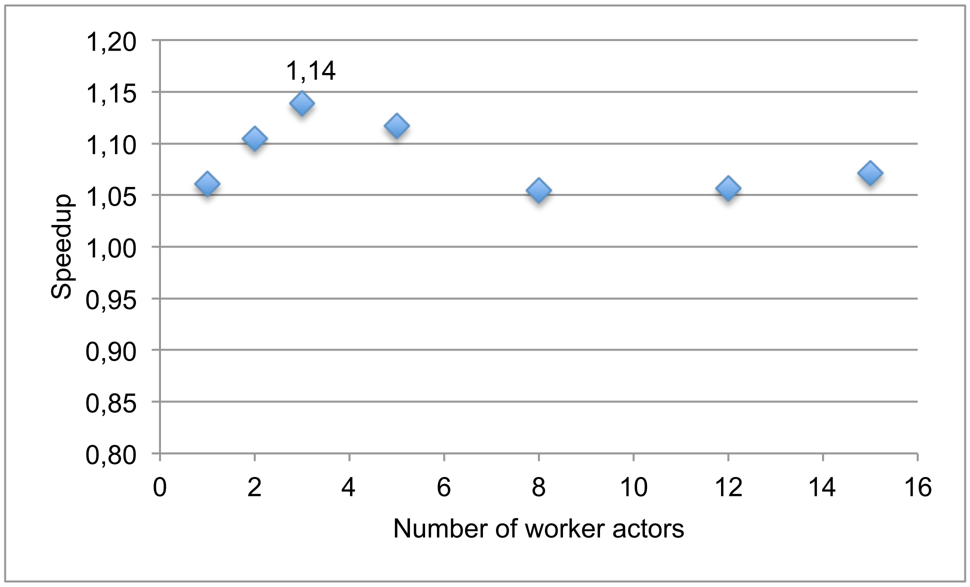

Figure 5 The speedup from using multiple worker actors, using GB tree as data structure for storage. These measurements are based on 500k input sequences of 44 bases long, using *k*=25 and using wall time of parsing of input FASTA file and creation the de Bruijn graph. The cleaning step and contig generation is not considered. Speedup is calculated as $T_{sequential}/T_{\#workers}$. The

speedup registered by using one worker is based on the overlap of parsing the input file and the work of creating the *k*-mer table.

6.1.3 Concurrent data structure (ETS) – serial version

The Erlang built-in Term Storage, from now on abbreviated as ETS, is a storage structure that allows processes to store very large quantities of data in the runtime environment. ETS stores dynamic tables that can be of the types set, ordered_set, bag and duplicate bag. In this case the set type was used, as the *k*-mers do not need to be ordered and the table will only contain one value for each key (the *k*-mer sequence). Access to the ETS structure is given by using the ets module, which gives a set of interface functions. The good thing about ETS is that it has support for concurrent access to the structure, meaning that several processes can read and write to the same data tables.

The implementation of the serial version does not benefit from the support for concurrency, but there was a big performance gain compared to the GB-based programs (Figure 6A). Using ETS instead of a GB-tree gave almost 80 % faster runtime with the same input. Another notable change compared to the GB-based implementation is that the graph-cleaning step, which previously took up around 80 seconds, is now performed in just 0.8 seconds. This is because there is no need to go through the entire ETS table to choose which *k*-mers to keep and which should be removed, now this step can be done by pattern matching to the values in the table. The only overhead this incurs is that a value for validity of a *k*-mer has to be added, but it is enough to keep a one-bit true/false value so the overhead is negligible.

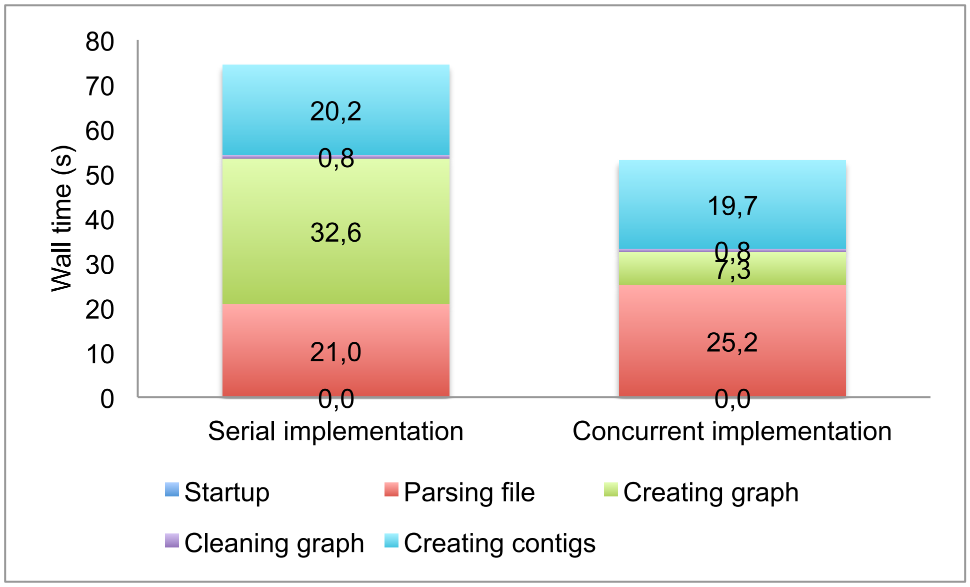

Figure 6 Timing of each part of Erlang code using ETS tables as data structure. Left set of stacked bars shows the timing for the sequential implementation and right set of stacked bars shows the corresponding for the concurrent implementation using two workers. Timings of the concurrent versions for other than two workers should only differ in the green section as the rest is done sequentially. Input is 500 k DNA sequences, 44 bases long and *k*=25.

6.1.4 ETS – concurrent version

In the previous section it was established that the ETS data structure is much faster than the GB-tree. But the reason for choosing it in the first place was that it could be favourable in creating a program that scales better by removing the need for a *Reduce* actor as all *Worker* actors can directly update the shared *k*-mer table. The structure of the concurrent version using ETS is very similar to the one used for GB-trees described earlier with the only exception of the removing of the *Reduce* actor and letting the *Worker* actors directly update the result table. This benefits performance both by removing the need for worker processes to send their results to *Reduce* and there is no need for having the bottleneck of having one process to handle all data in the end. Some of the implementation of this program can be found in the Appendix under Erlang code.

Performance profiling the concurrent version, using two *Workers* shows a similar pattern as in the GB-tree based version (Figure 6B). Parsing takes more time compared to the serial version because of overlapping parsing and work, but the creation of the graph is much faster. The overall speedup is almost 43 % using the time for the complete program. Examining the speedup as a function of the number of *Workers* shows that the best speedup is achieved using only two *Workers* (Figure 7). Having only a *Coordinator*, a *Parser* and two *Workers* give a total of 4 concurrent lightweight processes, which should not be a limit for the 16-core machine used. The probable cause for this poor scaling is that the application is too memory bound to achieve better speedup by adding more CPU cores and processes.

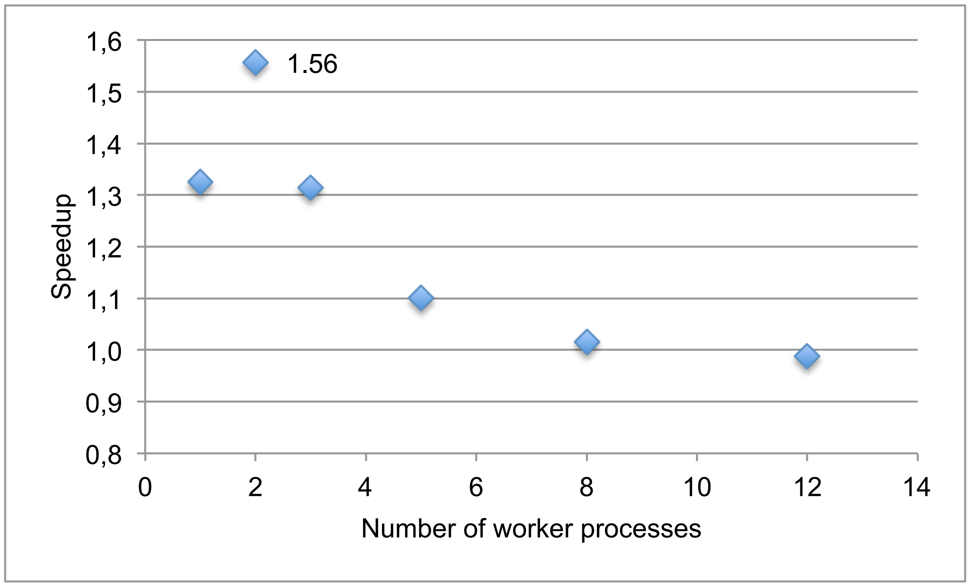

Figure 7 The speedup from using multiple worker actors, using ETS as data structure for storage. These measurements are based on 500k DNA sequences of 44 bases long, using *k*=25 and using wall time of parsing input FASTA file and creation of the de Bruijn graph. The cleaning step and contig generation is not considered. Speedup is calculated as $T_{sequential}/T_{\#workers}$. The speedup registered by using one worker is based on the overlap of parsing the input file and the work of creating the *k*-mer table.

6.2 Haskell

The parallelization strategy used for the Haskell implementation is very different compared to the concurrent approach used in Erlang. The focus during Haskell implementation was to use a high abstraction level and keep code as pure as possible. The program is divided into a set of steps:

- Parse input FASTA file
- Create all possible *k*-mers together with their forward and backward extensions
- Create the final de Bruijn graph
- Create contigs

The parser is very straight forward to implement and will return a list of sequences. Creating all possible *k*-mers can be implemented as defining a function that creates all *k*-mers from a given sequence and simply mapping that function over the list of sequences. Creation of the final de Bruijn graph was done by folding over the list of all *k*-mers and compiling a HashMap for the *k*-mers with unique forward and backward extensions. Parts of the implementation can be found in the Appendix under the section Haskell code.

Profiling a lazy language to see how much time is spent in the different functions is a hard task if it should be done correctly. This is because you have to force the evaluation of one function to be executed completely before the start of the next function, but which can change the dynamics of the program. A rough profiling of the serial program was done to get a lower estimate of how much time is spent on executing code from each function (Figure 8). From the profile, it is possible to tell that the time for creating all *k*-mers is at least 6 seconds, but probably a fair bit more. The end-point of creating the graph is reasonably accurate as a strict version of HashMap is used, which fully evaluates both key and value of its entries. Another important aspect is that the execution time is about half of the execution time used by the initial Erlang program using GB-trees, but it takes about 2,3 times longer time than the ETS implementation (comparing sequential code). The comparison is not completely accurate as the programs are run on completely different systems (a 2.2GHz 16 core server, compared to laptop computer).

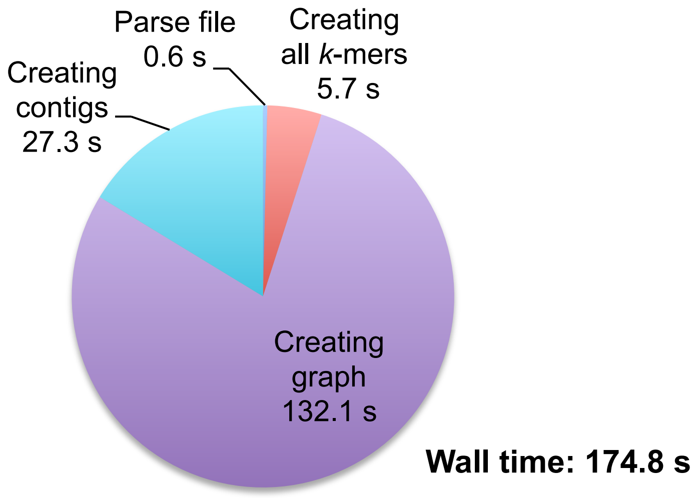

Figure 8 Timing of each part of the execution in the Haskell program when executed serially. There are faults in these proportions resulting form the laziness of evaluation in Haskell. The proportion will give a lower estimate on how much time is spent in each function, but the creation

of contigs and total time are solid values. These measurements are based on a dataset of 500k DNA sequences of 44 bases long, using *k*=25.

6.2.1 Parallel Haskell version

Finding the parallelism of this implementation is rather simple. First of all, the mapping over the function that creates all *k*-mers for a given DNA sequence is completely data parallel and can be computed independently. Parallelization if this part was straightforward by adding a parallelization Strategy from the Control.Parallel.Strategies library. There are several Strategies that can be applied to lists, but the one that gave the best speedup was the parListChunk Strategy that splits the list into chunks of a given size and executes them in parallel. The maximum speedup for this strategy was 2.12 (Figure 9), but the interpretation of this graph is not completely straightforward because of the architecture of the system. The speedup of using 1 virtual core is slightly below 1, which is expected because of the overhead that might be incurred by the compiler when adding threading to the program. When using two virtual cores, the runtime is almost the same as when using only one, which is a bit disappointing. The same pattern is seen when scaling up to using three or four virtual cores, where they both give roughly the same runtime. The ultimate reason for this is not clear, but one hypothesis is that when using two virtual cores, the threads are scheduled to the same processor core and that both threads are stopped when one of them has to wait for memory. In any case, it is impossible to predict whether adding more cores possibly could improve speedup or if the maximum is reached.

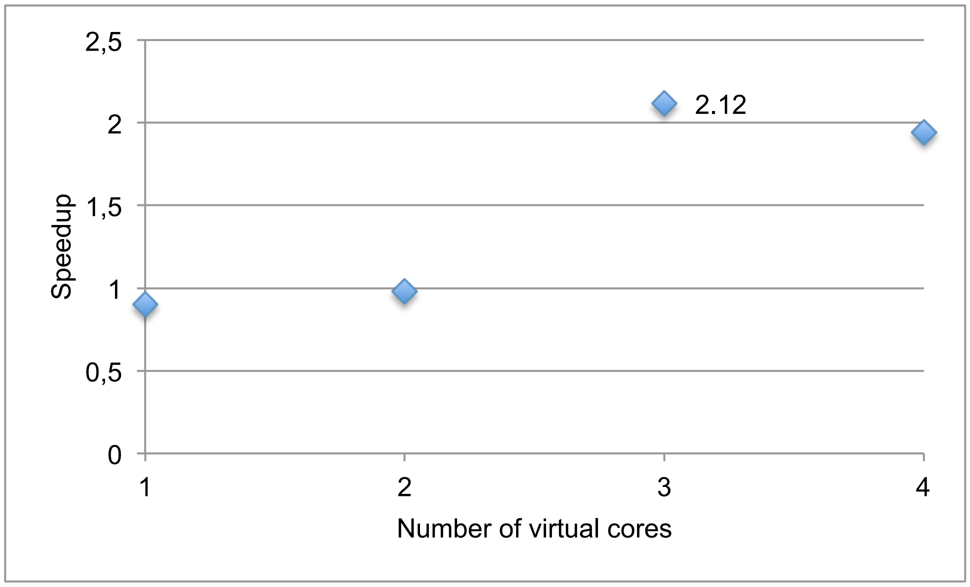

Figure 9 The speedup of using multiple threads. Speedup is calculated as $T_{sequential}/T_{\#threads}$. The computer only has a two core CPU, but uses Intel's Hyper-threading, which allows the system to use four virtual threads. These measurements are based on a dataset of over 3 million DNA reads, 44 bases long, using *k*=25.

Haskell provides a good profiling tool called ThreadScope, which shows the execution of the program and how much work different cores do. Analysing the same 3 million sequences with the same parameter gave the result in Figure 10 in which it is evident that the program is memory bound. The green parts shows the CPU work done and the total activity of the program is seldom up at even using one core at full speed. Zooming in at a smaller time segment reveals a jagged

CPU curve, indicating that the work is done in short bursts and then has to stop and wait for memory. This is rather expected as the program is mostly doing string computations that are memory intensive. The bad scaling of the Erlang program seen previously is easier to describe when the application is proven to be memory bound, adding more processes that can work in parallel do not change the limiting factor, which is the speed and stress put on the memory bus. Together with this result, you can hypothesise that the behaviour for this Haskell program would be similar to the Erlang case, it will scale badly once the limit of the memory bus is reached. The only way to decrease runtime is to use multiple memories, meaning that the application must be distributed on several nodes.

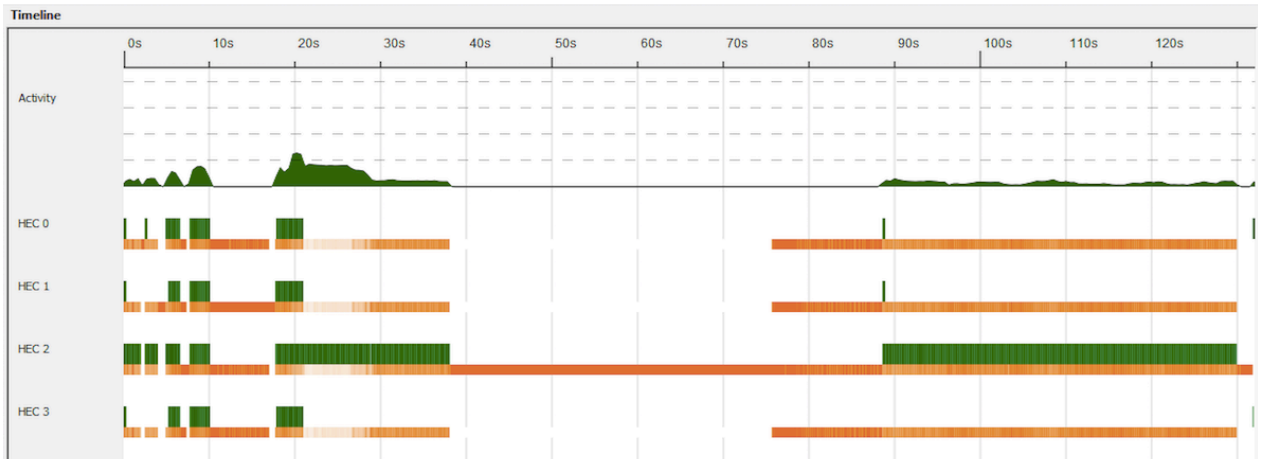

Figure 10 Parallelization of the construction of all *k*-mers, viewed by the ThreadScope tool [17]. Green parts indicate CPU activity and orange/red indicate garbage collection activity. Data used for processing was just over 3 million DNA reads, 44 bases long, using *k*=25.

The parallelization of the first part was performed in a simple and high-level way that gave a good resulting speedup. The remaining steps of creating the graph (that is, fold over all *k*-mers and compute the find the unique ones) and creation of contigs are still left. There seems to be a number of ways that this could be turned into parallel, but none that are as clear and easy to implement as in the case of the parList Strategy. Furthermore they would require the code to be impure to manage it, and would thus leave the niceness of parallel functional programing (not needing locks and difficult synchronizations).

6.3 Code size

A common way to compare language expressiveness is to use lines of code as a metric. The fewer the lines, the more expressive the language is in general. Haskell, which is generally considered to have a high abstraction level ended up with the fewest lines of code in the implementations made in this thesis (Figure 11). This metric was base only the number of lines, not counting lines for commenting, importing libraries or declaration of type synonyms. One of the things that usually increase expressiveness is the use of higher order functions in functional languages. But in this comparison, there are no differences based on this as Erlang supports many of the higher order functions existing in Haskell (like map, fold, reduce). If the comparison had been between Haskell and an imperative language, the difference would potentially be a lot bigger. In the case of “Contig generation”, the serial Erlang and Haskell code yielded roughly the same number of lines of code. This is manly due to the lack of good abstraction of this algorithm and the code is forced into a more imperative programing style. There is also a slight increase in lines of code when dealing with lookups in HashMaps because the return value is in the "Maybe" type, which adds extra code for checking what is in the Maybe and retrieve values form it.

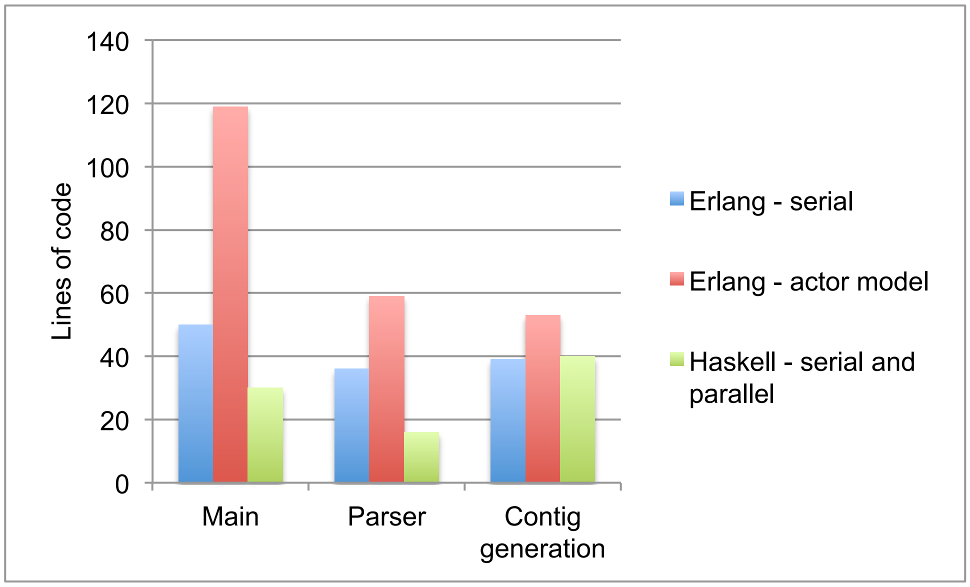

Figure 11 Lines of code used for the different implementations. "Main" stands for a function for running the program and code for generating the *k*-mers. The Erlang programs used for this metric were the ones implemented with the ETS data structures. For Haskell there is no additional overhead when comparing serial and parallel code, the only difference is a couple of libraries that needs to be imported and adding some function calls on the line that is parallelized.

When comparing the serial versus the actor based Erlang programs, there is more than a doubling in the number of lines of code for "Main". This is because the logic of how the actors communicate and the spawning of actors are all done in the "Main" program. The "Parser" only has 65 % more lines in the concurrent version, which are heavily contributed by the added functionality of overlapped parsing and creation of *k*-mers. This extra functionality required some extra checking in the parser function, an additional function for sending data to workers and some interface functions for communication.

6.4 Haskell compared to Erlang

Comparing the two languages used in this thesis is not an easy task. Haskell has proven to reduce code size and have a higher expressiveness compared to Erlang. Parallelization in the obvious tasks, especially when having complete independence between data, is simplified to the mere addition of a few words on a single line and can increase speedup in a good manner. But when dealing with more complicated parallelization strategies, the theory needed to grasp the concepts requires a rather experienced programmer. In some cases the program might be forced into the use of mutable variables to achieve good speedup, meaning that the code will no longer be pure functional. The benefits of parallelization of functional programs as deterministic parallel execution, absence of race conditions and need for synchronization will no longer be valid. Haskell might be very suitable for some algorithms, but in this implementation only a part was possible to parallelize in a pure functional manner.

Erlang on the other hand is an easier language to learn in the beginning. There are no explicit differences between pure and impure code, after all it is not a completely functional language. There is an extra overhead in terms of code size, but it might on the other hand improve readability of the code. Haskell forces functions to only return one type of value, thus having to deal with Maybe-types, whereas Erlang can have multiple return types. Parallelization by using the actor model require a lot more code to be written, deciding on flow of data, how communication should be performed and so on. The complexity of the program thus increases a lot when going from serial code to a good concurrent version.

Assembly programs are both memory bound and contain a lot of string computations (creating substrings, adding characters when traversing graphs etcetera). This means that a programming language that is both fast at string computations and have a small memory footprint would be preferable. Unfortunately both Erlang and Haskell implements strings as lists of chars, which means that each character in the string will take up both the bytes that represent that character and a pointer to the next index in the list. List based strings will thus take up roughly twice as much memory as a strings that are based on arrays. Strings in Erlang are using 32 bits for encoding each character and 32 bits for each pointer in a 32-bit system and twice that in a 64-bit system. Haskell has a similar implementation of normal strings, but also provide another type of string implemented in the Data.ByteString library that instead provides array-based strings of only 8 bits for each character. These byte strings much more suitable for this application as the memory footprint is a lot smaller and computations are a lot faster. Erlang on the other hand provide bit syntax which could be very advantageous for this type of application, the DNA sequences are only containing a four different bases so there is a potential compression into using only 2 bits for each character. Another advantage is that binary data can be sent between processes by only sending references to the binaries (without copying the data), which could be very beneficial in this application. However, binary data was not considered during the thesis because of the time constraint.

6.5 Need for distributed computing

From the results seen in the previous sections, including the bad scaling of the Erlang programs and the output of ThreadScope using Haskell, it is evident that to achieve a good scaling in this application you will have to go into distributed computing. The time for this thesis was not enough to go into an implementation of how this would be carried out. There is a number of papers and assemblers in use that show that distributed assembly is a working approach, both the ones showing a good speedup [11,15], and the ones that use distribution to handle the memory requirements [13,16].

A suitable language for writing assembly programs must thus have support for distributed programming. In Erlang the distributed computing is built into the language the Erlang reference manual [18] has a good tutorial for implementation help. The distribution is handled by running independent Erlang runtime systems on multiple computers, each called a node. Communication between the nodes is handled by the TCP/IP protocol. Haskell on the other hand have some libraries and extensions that allow for distributed computing, but distributed programming is mostly a research topic [19]. There are a few different libraries that could be of interest when implementing an assembler.

7. Conclusion

The conclusion of this thesis is manifold. First of all, genome assembly is a memory bound process that does not scale well with a multicore architecture when using a shared memory system. Only two programming languages has been studied in the this thesis, but looking at implementations made by others in C/C++ the tendency is to go towards distributed programming so the limitation of shared memory parallelization should exist for all types of languages [11,15]. Another reason for going towards distributed computing is that the memory footprint of these programs is very large and using computer clusters is an easier way to run them, instead of having to buy specific machines with extreme amounts of RAM. Both Haskell and Erlang have support for distributed computing, even if distribution in Haskell seems to be more of a research field and not as integrated in the language as in Erlang.

Haskell comes out as generally being a more expressive language compared to Erlang, it produces less lines of code and parallelization can be done in a much more straightforward way. The lazy evaluation strategy in Haskell can also improve how parallelisation is done, which is exemplified in the parallelisation of creating *k*-mers, where the input file can be read in parallel to work done in other threads. Laziness can however be a source of performance decrease, both in space and runtime, which make performance profiling and tuning a critical step in development.

The goal of this thesis was to investigate the applicability of functional programming and the support for bioinformatics applications. Genome assembly was taken as a case study, hoping for it to be representative of problems within the field. However, it might be the case that this application is a bad representative and that other bioinformatics algorithms are far less memory intensive. The results presented here when it comes to speedup might be very misleading, using the same parallelisation strategies on an application that is less memory intensive is certain to scale a lot better. The key finding in this thesis will be the relative ease of how parallelization can be expressed in these languages.

7.1 Future work

As stated previously, distributed programming is a demand for developing a good genome assembler but was something that there was no time to evaluate within the time frame of this thesis. A future work would thus be to do an implementation using distributed computing to see if the potential gain is there or not. Another interesting aspect would be to develop the same type of algorithm in a language that is more commonly used within bioinformatics (like C/C++ for high performance tasks) and compare both performance and the ease of parallelization. The evaluation in this thesis is lacking this comparison, which might be the most important one. Haskell might produce a lot less code and make development a lot faster, but if the performance loss is too big that might be irrelevant.

Another future work would be to try implementation on a set of other bioinformatics applications. The characteristics of other algorithms might benefit to a greater extent when using functional programming, the actor model and not be constrained by the lack of mutable variables.

8. Bioinformatics today

This section provides a short description of the bioinformatics field as a whole. The field is very broad and diverse, with problems ranging from simple scripting to highly optimized code running on big computer clusters. Practically all types of programming languages are used within bioinformatics. Benchmarking has been done on a set of typical applications within bioinformatics, to evaluate how different languages perform. Fourment and Gillings compared the most commonly used languages in bioinformatics (Python, Perl, Java, C, C++ and C#) and found that C/C++ is the fastest and uses the least amount of memory [20]. On the other hand, C/C++ loses in their expressiveness where instead languages as Python and Perl shine [20]. This study did however not assess the ease and efficiency of parallel programming and they did not consider any functional language. Nonetheless, C and C++ are the languages used in most of the popular bioinformatics software available today. Programs like Blast (finds similarity between sequences compared to a database of sequences), Celera (genome assembler), MrBayes (generate phylogenetic trees) are all implemented in C or C++ [21-23].

Another attempt at speeding up bioinformatics computations has been to implement highly parallel programs running on GPUs and accelerators using CUDA [24-27]. These papers have all reported large increase in speedup indicating that this is a viable way to improve performance. GPUs and accelerators are example of many core architectures [28], which means that these are algorithms that function very well for parallel architectures and that the gain of easy parallelization strategies for bioinformatics can potentially be vast.

Functional programming has not been used to any greater extent within bioinformatics when it comes to software that is extensively used. There exists a BioHaskell library[29], very similar to BioPython, BioPerl and other "Bio-" language-extensions, but functional programming is mainly used when creating a first prototype and is later implemented in a faster language. There does not seem to be any extensive study that evaluates the applicability of functional programming within bioinformatics to date.

9. References

[1] Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, *et al*. A view of the parallel computing landscape. Communications of the ACM. 2009; 52(10):56-67.

[2] Hammond K. Why Parallel Functional Programming Matters: Panel Statement. Reliable Software Technologies - Ada-Europe. 2011; 6652:201-205.

[3] Hinsen K, The Promises of Functional Programming. Computing in Science & Engineering. 2009; 11(4):86-90.

[4] Haskell programming language (http://Haskell.org)

[5] Erlang programming language (http://Erlang.org)

[6] Pop M. Genome assembly reborn: recent computational challenges. Brief Bioinform. 2009 July;10(4):354-366.

[7] Isaev A. Introductionto Mathematical Methods in Bioinformatics. Corrected second printing. Berlin: Springer; 2006.

[8] SMRT SEQUENCING ADVANTAGE [Internet] California: Pacific Biosciences of California, Inc. [updated 2014, cited 2014 Nov 13]. Available from: http://www.pacificbiosciences.com/products/smrt-technology/smrt-sequencing-advantage/

[9] Compeau P.E.C Pavzner P.A, Tesler G. How to apply de Bruijn graphs to genome assembly. Nature Biotechnology. 2011; 29(11):987-991.

[10] Schatz M.C, Delcher A.L, Salzberg S.L. Assembly of large genomes using second-generation sequencing. Genome Res. Sep 2010; 20(9):1165–1173.

[11] Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2014, Nov 16-21:437-448.

[12] Liu Y, Schmidt B, Maskell D.L. Parallelized short read assembly of large genomes using de Bruijn graphs. BMC Bioinformatics 2011, 12:354-364.

[13] Miller J.R, Koren S, Sutton G. (2010). Assembly algorithms for next-generation sequencing data. Genomics, 95*,* 315-327.

[14] Birol I, Raymond A, Jackman S.D, Pleasance S, Coope R, Taylor G.A, *et al*. Assembling the 20Gb white spruce (Picea glauca) genome from whole-genome shotgun sequencing data. Bioinformatics. 2013; 29(12):1492-1497.

[15] Chikhi R, Rizk G. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology. 2013; 8(22):1-9.

[16] Simpson J.T, Wong K, Jackman S.D, Schein J.E, Jones S.J.M, Birol I. ABySS: A parallel assembler for short read sequence data. Genome Res. 2009; 19:1117-1123.

[17] ThreadScope performance profiling tool (http://haskell.org/haskellwiki/ThreadScope).

[18] Erlang Reference Manual, User's Guide version 6.2. [Internet] Stockholm: Ericsson AB; c2003-2014 [updated 2014 September 16; cited 2014 December 3]. Available from: http://erlang.org/doc/reference_manual/distributed.html

[19] Haskell.org [Internet] Haskell Wiki: Applications and libraries/Concurrency and parallelism [updated 2002 August 17; cited 2014 December 3]. Available from: https://haskell.org/haskellwiki/Applications_and_libraries/Concurrency_and_parallelism.

[20] Fourment M, Gillings M.R. A comparison of common programming languages used in bioinformatics. BMC Bioinformatics. 2008; 9:82-91.

[21] Camacho C, Coulouris G, Avagyan V, Ma N, Papadopoulos J, Bealer K, Madden T.L. BLAST+: architecture and applications. BMC Bioinformatics. 2009; 10:410-419.

[22] Myers E.W, Sutton G.G, Delcher A.L, Dew I.M, Fasulo D.P, Flanigan M.J, *et al*. A Whole-Genome Assembly of Drosophila. Science. 2000; 287(5461):2196-2204.

[23] Ronquist F, Huelsenbeck J.P. MrBayes 3: Bayesian phylogenetic inference under mixed models. 2003; 19(12):1572-1574.

[24] Manavski S.A, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics. 2008; 9(2):S10.

[25] Bustamam A, Burrage K, Hamilton A.N. Fast Parallel Markov Clustering in Bioinformatics Using Massively Parallel Computing on GPU with CUDA and ELLPACK-R Sparse Format. EEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB). 2012; 9(3):679-691.

[26] Bustamam A, Ardaneswari G, Lestari D. Implementation of CUDA GPU-Based Parallel Computing on Smith-Waterman Algorithm to Sequence Database Searches. Advanced Computer Science and Information Systems (ICACSIS), 2013 International Conference on. Sept 2013. 137-142.

[27] Liu Y, Schmidt B, Maskell L.D. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. Bioinformatics. 2012; 28(14):1830-1837

[28] Rauber T, Rünger G. Parallel Programming for Multicore and Cluster Systems. 2nd ed. Berlin: Springer; 2013.

[29] Haskell library (http://biohaskell.org).

Appendix

I. Data generation

The data used in this project has both been "artificial", by creating DNA sequences in a random fashion and data coming from real sequencing machines. Artificial data has mostly been used during the development phase and the real data has been used when benchmarking the programs to get metrics for this report.

The true data used in this project was downloaded from NCBIs short read archive (http://www.ncbi.nlm.nih.gov/sra/) and can be found by searching for the identification number ERR048903. This data was produced by sequencing paired end data from the bacteria *Escherichia coli*, using an Illumina machine. SRA Toolkit was then used for transforming the paired end data into short reads and obtaining it in FASTA format (data downloaded from the short read archive is usually in SRA format). Only one part of the paired end reads was kept, resulting in a FASTA file with just over 3 million sequences of 44 bases long reads. Sub-sampling was perform by taking the first 500 thousand sequences (1 million rows) from the original FASTA file, to reduce runtime enough to be able to run several times for analysis of the program.

II. Erlang Code

Concurrent Parser_tools

```
%%-------------------------------------------------
%% spawn this to it's own process!
%%-------------------------------------------------
parse_chunk_wise(FilePath, Return_PID, N, Dest_PIDs) ->
        if N=<0 ->
                signal_error(Return_PID, {error, "Cannot pass a chunk_size less than 1!"});
        true ->
                case file:open(FilePath, [read]) of
                        {ok, IoDev} ->
                                parse_chunk_wise(IoDev, N, [], Dest_PIDs,
length(Dest_PIDs)),
                                signal_done(Return_PID),
                                file:close(IoDev);
                        {error, enoent} ->
                                io:format("~n ########### The file specified cannot be
found ###########~n~n"),
                                signal_error(Return_PID, {error, enoent});
                        {error, Reason} ->
                                signal_error(Return_PID, {error, Reason})
                end
        end.

parse_chunk_wise(IoDev, N, Next_name, Dest_PIDs, NextPID) when NextPID =< 0 ->
        parse_chunk_wise(IoDev, N, Next_name, Dest_PIDs, length(Dest_PIDs));
parse_chunk_wise(IoDev, N, Next_name, Dest_PIDs, NextPID) ->
        case scanN_fasta(N, IoDev, [], Next_name, []) of
                {eof, [{[],[]}]} -> ok;
                {eof, Result} ->
```

```
                        return_data(lists:nth(NextPID, Dest_PIDs), Result),
                        ok;
                {_, New_next_name, Result} ->
                        return_data(lists:nth(NextPID, Dest_PIDs), Result),
                        parse_chunk_wise(IoDev, N, New_next_name, Dest_PIDs, NextPID-1);
                {error, Reason} ->
                        signal_error(hd(Dest_PIDs), {error, Reason})
           end.
%% will return {IoDev, [{Name, Sequence}]} IF not encountered eof  %% IF eof reached: returns
{eof, [{Name, Sequence}]}
scanN_fasta(N,IoDev, Result, Next_name,[]) when N =< 0 ->
        {IoDev, Next_name, Result};
scanN_fasta(N, IoDev, Result, Temp_name, Temp_seq) ->
        case file:read_line(IoDev) of
                {ok, Data} -> [First_char|_] = Data,
                        if First_char == $> -> %% Name!
                                %-----------------------------------
                                %Check if there is an old record to add
                                if Temp_name == [] -> % No old record
                                        scanN_fasta(N, IoDev, Result,
string:strip(tl(Data), right,$\n), Temp_seq);
                                true ->
                                        % old record that should be put in the result!
                                        scanN_fasta(N-1,IoDev, [{Temp_name,
Temp_seq}|Result], string:strip(tl(Data), right,$\n), [])
                                end;
                                %-----------------------------------
                        First_char == $\n -> % just continue to next row!
                                scanN_fasta(N, IoDev,Result,Temp_name,Temp_seq);
                        true -> % Data!!
                                scanN_fasta(N, IoDev, Result, Temp_name,
lists:append(Temp_seq, string:strip(Data,right,$\n)))
                        end;
                eof -> % Add last record to result and return it {eof,[{Temp_name,
Temp_seq}| Result]}; {error, Reason} -> {error, Reason}
        end.
```

Concurrent de Bruijn Graph code (Coordinator actor)

```
genDeBruijn(FilePath, K) ->
        % Delete the old table if present
        delete_previous_stuff(),
        % create a new table
        ets:new(?TABLE, [set, public, named_table]),
        % Spawn worker processes
        Worker_PIDs = spawn_and_link_workers(?N_PROC, K),
        distribute_work(FilePath, Worker_PIDs, ?BUCKET_SIZE),

        % Signal 'stop' to all workers
        signal_stop_to_Workers(Worker_PIDs),
        % Wait for all workers to finish
        wait_for_workers_to_finish(Worker_PIDs),clean_tree(),
        {uu_contigs, UUContig_list} = trav_deBruijn:createAllUUContigs(?TABLE,K),
        utils_deBruijn:printContigs(?OUT_FILE, UUContig_list).

% --------------------------------------
wait_for_workers_to_finish([]) -> ok;
wait_for_workers_to_finish(Worker_PIDs) ->
```

```
        receive
                {stop, PID}->
                        wait_for_workers_to_finish(lists:delete(PID, Worker_PIDs));
                {'EXIT', PID ,normal} -> % no problem!
                        wait_for_workers_to_finish(lists:delete(PID, Worker_PIDs));
                Unknown ->
                        io:format("Unknown message to MAIN! ~p~n", [Unknown])
        after 6000000 ->
                io:format("No result in Main....")
        end.
%%-----------------------------------------------
% Spawn worker processes that will do the work
spawn_and_link_workers(Num_proc, K) ->
        if Num_proc =< 0 ->
                {error, "Cannot produce 0 or less workers"};
        true ->
                spawn_and_link_workers(Num_proc, K, [])
        end.

spawn_and_link_workers(0, _K, WorkerList) -> WorkerList;
spawn_and_link_workers(Num_proc, K, WorkerList) ->
        Work_PID = spawn_link(?MODULE, worker_proc, [self(), K]),
        spawn_and_link_workers(Num_proc-1, K, [Work_PID|WorkerList]).

%%-----------------------------------------------
distribute_work(FilePath, Worker_PIDs, Chunk_size) ->
        % spawn parser process!
        Parser_PID = spawn_link(c_parser_tools, parse_chunk_wise, [FilePath, self(),
Chunk_size, Worker_PIDs]),
        % go to receive function, will distribute the work
        wait_for_parser(Parser_PID, Worker_PIDs).

wait_for_parser(Parser_PID, Worker_PIDs) ->
        receive
                {done, Parser_PID} -> ok;
        {error, enoent} ->
                signal_stop_to_Workers(Worker_PIDs),
                exit(normal);
        Unknown ->
                io:format("Wait for parser received unknown msg; ~p", [Unknown])
        end.

%%-----------------------------------------------
signal_stop_to_Workers(Worker_PIDs) ->
        lists:foldl((fun(PID, _) -> stopWorkerProc(PID), 1 end), 1, Worker_PIDs).

%%-----------------------------------------------
% Removes all non-unique k-meres from the table - forms the final k-mer table!
clean_tree() ->
        ets:match_delete(?TABLE, {'_', false,'_','_','_'}).
```

Concurrent de Bruijn Graph code (Worker actor)

```
worker_proc(Main_PID, K) ->
        receive
                stop ->
                        sendEndW(Main_PID),
                        exit(normal);
                {data, Data} ->
```

```
                            add_to_K_table(Data,K),
                            worker_proc(Main_PID, K);
                    _ ->
                            exit("Unknown message received in Worker")
        end.
%%------------------------------------------------
%% Produce k-mers  %% Loop over the list of fasta sequences and do work on each sequence
add_to_K_table([],_K) -> ok;
add_to_K_table([{_, FastaSeq}|Tail_list],K) ->
        addFastaSeqToTable(FastaSeq,K),
        add_to_K_table(Tail_list, K).

%% Will take one Fasta String and process it and update the ETS table
addFastaSeqToTable(FastaSeq, K) ->   generateHashIndexes(tl(FastaSeq), K, hd(FastaSeq)).

%% Recursively goes through an individual dna-seqence and adds to the ETS table
generateHashIndexes(Seq, K, PrevChar) ->
        CheckRun = length(Seq) < K+2,
        if CheckRun ->
                ok;
        true ->
                % still some part of the string left
                {FirstK,[NextChar|_]} = lists:split(K, Seq),
                add_to_table( FirstK, PrevChar, NextChar ),
                generateHashIndexes(tl(Seq),K, hd(FirstK))
        end.

add_to_table( K_mer, PrevChar, NextChar ) ->
        % check if it's already present
        K_exists = ets:lookup(?TABLE, K_mer),
         if []== K_exists ->
                % no such k-mere exists yet!
                ets:insert(?TABLE, {K_mer, true, PrevChar, NextChar, 1});
        true ->
                % have to check what's in that record!
                {_Seq, Unique, Prev, Next, _Mult} = hd(K_exists),

                if Unique == false ; Next /= NextChar ; Prev /= PrevChar ->
                        % this means that it's a non unique UU contig
                        ets:update_element(?TABLE, K_mer, {2,false});
                true ->
                        % update the multiplicity of that k-mere!
                        ets:update_counter(?TABLE, K_mer, {5,1}) % Update position 5
(multiplicity) with 1
                end
        end.
```

Traversing the de Bruijn Graph

```
createAllUUContigs(TableName,K) ->
        {uu_contigs, loopTraverse(TableName,[], K)}.

loopTraverse(Table, ContigList, K) ->
        case ets:first(Table) of
                '$end_of_table' ->
                        % no more entries!
                        ContigList;
                Key ->
                        [{Start_seq,_,Prev,Next,_}] = ets:lookup(Table, Key),
                              ets:delete(Table, Key),

                              {left_res, Left_extend} = travLeft([Prev|Start_seq], Table, K),
                              {right_res, Right_extend} = travRight(Start_seq ++
string:chars(Next,1), Table, 1),
                              loopTraverse(Table, [merge(Left_extend, Right_extend,
K)|ContigList], K)

          end.
%%-----------------------------------------------
%% traverse left
%%-----------------------------------------------
travLeft(Contig, Table, K) ->
          First_K = lists:sublist(Contig, K),
          Match = ets:lookup(Table, First_K),
          if Match == [] ->
                    %Done - return the contig
                    {left_res, Contig};
          true ->
                    [{_K_mer,_,Prev,_,_}] = Match,
                    ets:delete(Table, First_K),
                    travLeft([Prev|Contig], Table, K)
          end.

%%-----------------------------------------------
%% traverse right
%%-----------------------------------------------
travRight(Contig, Table, Iteration) ->
          {_First_part, Last_K} = lists:split(Iteration, Contig),
                    Match = ets:lookup(Table, Last_K),
          if Match == [] ->
                    %Done - return the contig
                    {right_res, Contig};
          true ->
                    [{_K_mer,_,_,Next,_}] = Match,
                    ets:delete(Table, Last_K),
                    travRight(Contig ++ string:chars(Next,1), Table, Iteration+1)
          end.
```

III. Haskell code

Parser part

```
readFasta file_path = do
          input <- getStrings file_path
          let fasta = parse input C.empty C.empty []
          return fasta

parse :: [C.ByteString] -> C.ByteString -> C.ByteString -> [(C.ByteString, C.ByteString)] ->
[(C.ByteString, C.ByteString)]
parse [] seq name result = (name,seq):result
parse (x:xs) seq name res
          | (C.isPrefixOf (C.pack ">") x) && (not $ C.null name) = parse xs C.empty (C.tail x)
((name,seq):res)
          | (C.isPrefixOf (C.pack ">") x)      = parse xs C.empty (C.tail x) res
          | otherwise                          = parse xs (C.append seq x) name res
```

Main Code

```haskell
main = do
        [file_path, k_str] <- getArgs
        seqList <- P.readFasta file_path
        let { k = (read k_str :: Int) ;
                hashIndexes = concat $ (map (createHashIndexes k) seqList `PS.using`
PS.parListChunk chunkSize PS.rdeepseq) ;
                cleaned = genUUMap hashIndexes ;
                contigs = getContigs k cleaned ;  }
        return (contigs)

createHashIndexes :: Int -> FastaRec -> [HashIndex]
createHashIndexes k (_, seq) = createHashList (C.tail seq) k (C.head seq) []
createHashList :: Sequence -> Int -> Char -> [HashIndex] -> [HashIndex]
createHashList seq k prev resList
| fromIntegral( C.length seq ) < k+1        = resList
| otherwise      = createHashList (C.tail seq) k (C.head seq)   (((C.take (fromIntegral
k :: Int64) seq), (prev, (C.head $ C.drop (fromIntegral k :: Int64) seq))):resList)


genUUMap :: [HashIndex] -> DeBruijnGraph
genUUMap hashIndexes = snd $ foldl (reduceToUU) (Set.empty, HM.empty) hashIndexes


-- Checks if the k-mer has been found before (part of the Set) -- if found before - need to
check if the prev and next bases matches -- else insert the k-mere in the "found Set" and
insert into the HashMap
reduceToUU :: (Set Sequence, DeBruijnGraph) -> HashIndex -> (Set Sequence, DeBruijnGraph)
reduceToUU (set, graph) index
| found_before             = check_further set graph index
| otherwise                = (Set.insert (fst index) set, HM.insert (fst index) (snd
index) graph)
where found_before = Set.member (fst index) set

-- Checks if the current k-mer has identical prev and next bases as before -- if identical - do
nothing -- if not found in HashMap - found and removed before - return current set and graph --
else remove the k-mer that now has proven to not be unique
check_further :: Set Sequence -> DeBruijnGraph -> HashIndex -> (Set Sequence, DeBruijnGraph)
check_further set graph index
| lookup == snd index     = (set, graph)
| lookup == ('_','_')     = (set, graph)
| otherwise               = (set, HM.delete (fst index) graph)
        where lookup = HM.lookupDefault ('_','_') (fst index) graph
```