School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Bioinformatics of the Immune System

by

Lasath Fernando

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted:	June 2, 2015	Student ID:	z3373562
Supervisor:	Dr. Bruno Gaëta	Topic ID:	0278

Abstract

This document describes the requirements to theses submitted for the Bachelor of Engineering in Software Engineering degree at the School of Computer Science and Engineering. Requirements described are that of both of context and layout of the theses. The document is written using the LaTeX template provided by the school.

Acknowledgements

This work has been inspired by the labours of numerous academics in the Faculty of Engineering at UNSW who have endeavoured, over the years, to encourage students to present beautiful concepts using beautiful typography.

Further inspiration has come from Donald Knuth who designed TEX, for typesetting technical (and non-technical) material with elegance and clarity; and from Leslie Lamport who contributed LATEX, which makes TEX usable by mortal engineers.

John Zaitseff, an honours student in CSE at the time, created the first version of the UNSW Thesis LATEX class and the author of the current version is indebted to his work.

Abbreviations

HMM Hidden Markov Model

BLAST Basic Local Alignment Search Tool

XML Extensible Mark-up Language

NCBI National Centre for Biotechnology Information

Contents

List of Figures

Chapter 1

Introduction

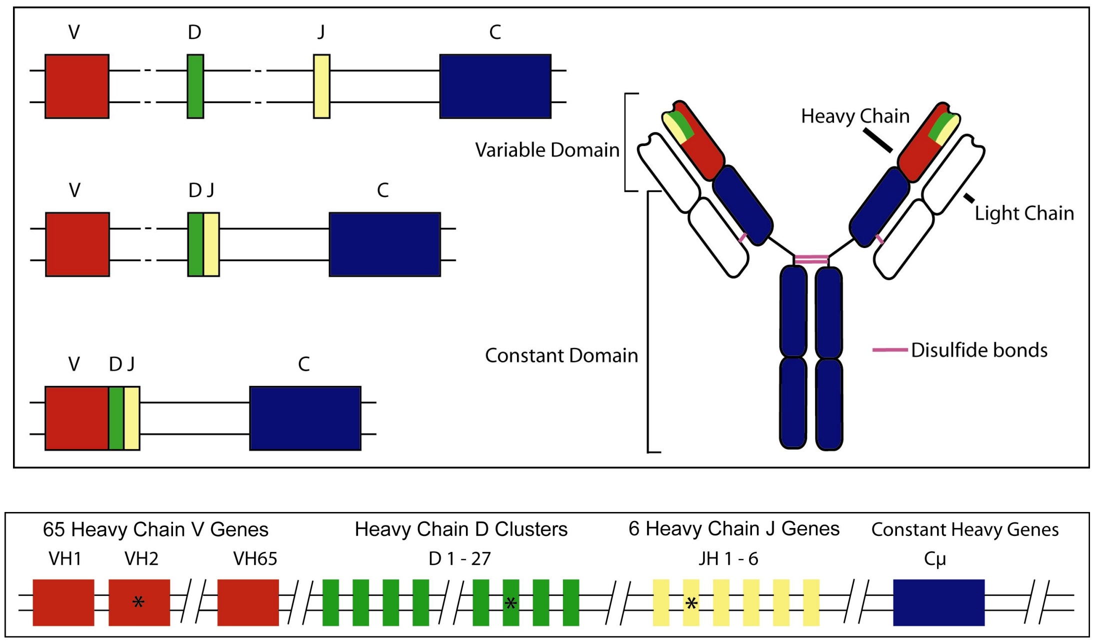
1.1 Immunoglobulin recombination

The human immune system faces many challenges when dealing with foreign pathogens invading the body. One of them is identifying and classifying antigens that are not only vastly diverse, but also constantly mutating and evolving.

It does so by creating immunoglobulins (also called antibodies) whose job is to identify and bind to certain antigens, effectively marking them for destruction by the rest of the immune system. The process, by necessity is highly selective - a given antibody will only bind to a very specific antigen.

However, to deal with the highly diverse and evolving nature of foreign antigens, antibodies must mutate very rapidly. This allows them to be as diverse as the antigens they face, and adapt to bind to newly encountered pathogens.

The process (essentially randomly) selects 3 germline genes from a repertoire in the person's DNA potentially consisting of between 38 and 45 IGHV genes [14, 15], 23 IGHD genes [13] and 6 IGHJ [23] genes. These selected genes are then mutated (adding, removing or changing nucleotides) to generate further diversity. The antibodies that successfully bind to the antigen emit a chemical signal, which then encourages the



Katherine Jackson

Figure 1.1: The IGHV, IGHD and IGHJ gene recombination process

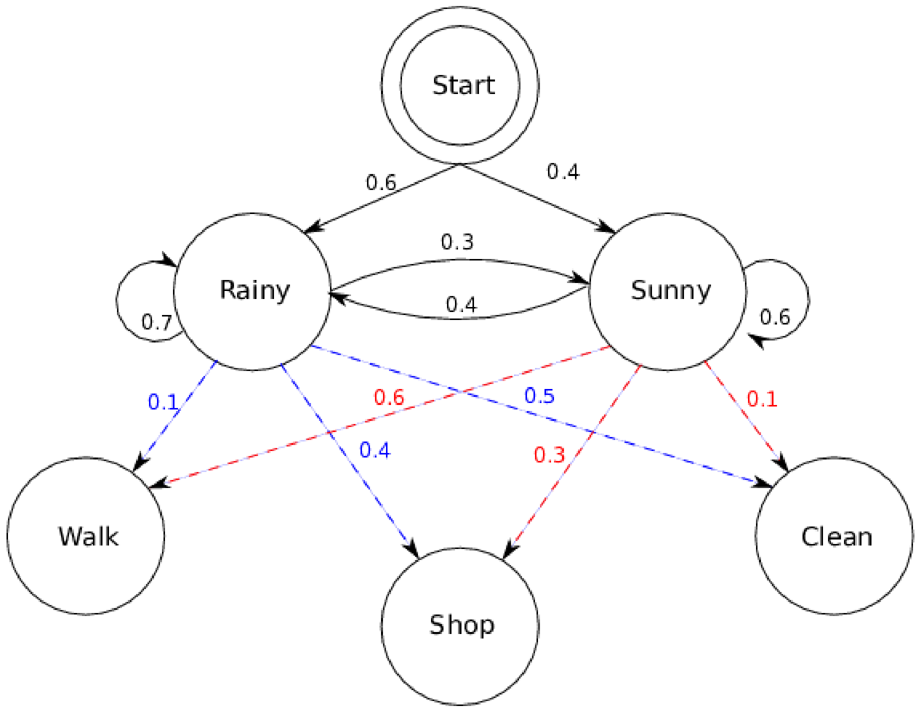


Figure 1.2: An example Hidden Markov Model [33]

production of similar antibodies. This entire process is illustrated in Figure 1.1 and explained in more detail by Gaëta *et al.*[7].

1.2 iHMMuneAlign

As a result of this gene recombination and mutation process, the germline genes often don't provide much information on rearranged genes found in the B-Cells that create the antibodies themselves.

Developed in 2007 by a team of researchers at UNSW, iHMMuneAlign attempts to simulate the process of immunoglobulin gene rearrangement.[7].It represents the recombination process of immunoglobulin heavy chain genes as a hidden Markov model, and then uses it to find the most probable set of germline genes that created the rearranged gene.

First, it selects the most likely IGHV germline gene from its repertoire of possible V genes, using the NCBI's Basic Local Alignment Search Tool (BLAST) [1]. It then constructs a Hidden Markov Model with that IGHV gene, and all the possible IGHD and IGHJ genes, together with other states representing the somatic mutation.

It then uses the Viterbi algorithm to determine the most likely sequence of states (genes and their mutations) that caused the observed emissions (the rearranged sequence given as input).

1.3 Hidden Markov Models

A Hidden Markov Model is effectively a Markov Chain, where the states cannot be observed directly. They are used in a wide variety of machine learning applications, including speech recognition [6].

An example of a HMM is shown in Figure 1.2. In this example, if the sequence of emissions `Walk` was observed, the probability of it being `Sunny` is

$$I\,(\text{Sunny}) \times E(\text{Sunny}, \text{Walk}) = 0.4 \times 0.6 = 0.24$$

and the probability of it being Rainy is

$$I(\text{Rainy}) \times E(\text{Rainy,Walk}) = 0.6 \times 0.1 = 0.06$$

where $I(X)$ is the initial probability of X and $E(X, Y)$ is the emission probability of state Y in state X.

If `Walk, Clean` were observed. The probability of it having been Rainy then Sunny is

$$I(\text{Rainy}) \times E(\text{Rainy,Walk}) \times T(\text{Rainy, Sunny}) \times E(\text{Sunny, Clean}) = 0.6 \times 0.1 \times 0.3 \times 0.1 = 0.0018$$

where $T(X, Y)$ is the probability of transitioning from state X to state Y.

The Viterbi algorithm uses dynamic programming to recursively explore these probabilities, and selects the sequence of states with the highest probability at the end.

Chapter 2

Analysis

Since the high level aim of this project is to improve the performance of iHMMuneAlign, it was necessary to analyse the existing implementation in order to gain insight into its structure, as well as to create realistic aims.

2.1 Run Time

One of the primary attributes of performance, is run time. We begin by measuring the run time of the existing implementation and -more importantly- how it grows with workload.

The current implementation consists of a Java executable and a shell script, to allow running in batches. The Java executable (which does the actual processing) takes a range of input from the script and attempts to process that many in parallel. It then invokes Java program after breaking up the entire range into batches of a given size.

Since we are interested in the rate the total runtime changes with workload as well with increasing parallelism, both were measured. It was run with a total workload ranging from 1 sequences to 100 sequences, with batch size ranging from 1 to 8. The results are shown in Figure 2.1.

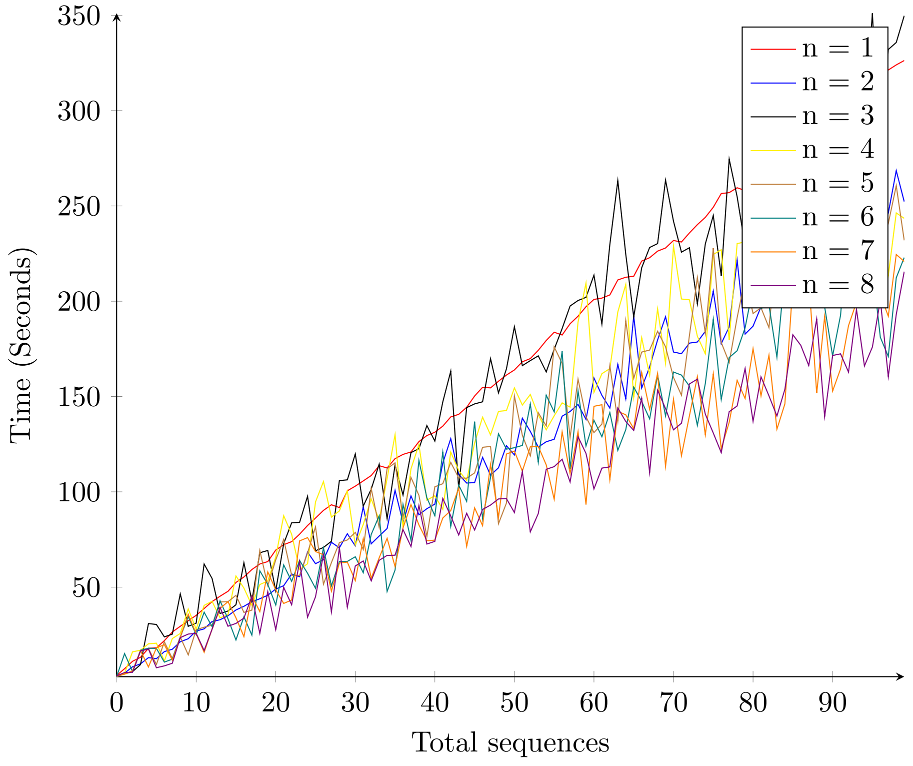


Figure 2.1: Total time taken (in seconds) to process the number of sequences (Wall-clock time). n is the number of sequences per invocation of the program

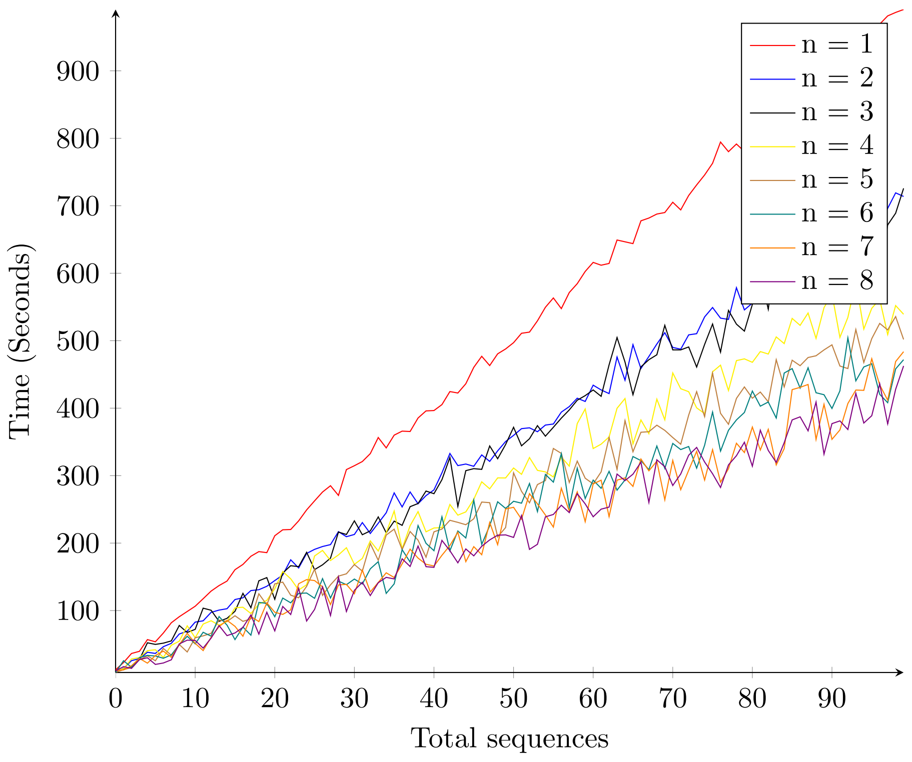


Figure 2.2: Total time spent doing work by each CPU on behalf of the process during execution (CPU Time). n is the number of sequences per invocation of the program

Note that this was performed on a machine with an AMD Phenom II 1090T (6-core) processor, 8GB of RAM and a solid state hard drive.

2.1.1 Interpretation of Results

With a batch size of 1, the expected linear increase was present. However, increasing the batch size shows a more sporadic decrease in runtime. There could be a number of causes for this, including the threads being assigned different CPUs by the scheduler on some runs but not others, increasing cache misses. Nonetheless, it wasn't investigated further, due to time constraints, and the fact that we didn't need such high accuracy for this test. However, there is an overall decrease in total execution time, as the batch size increases.

This suggests that the existing implementation is concurrent, and will utilize all the available processing power on the CPU. However, that should have diminishing returns once the amount of threads exceeds the total number of CPU cores on the machine, which is not the case here - execution time continues to decrease at roughly the same pace as n increases.

Looking at the total CPU time spent on the process, rather than total execution time provided a theory (which was later confirmed in Section 2.2.2, on page 11) to explain this behaviour. For each invocation of the Java program, there is a significant amount of overhead. For example, the Java Virtual Machine (JVM) needs to be initialised, it needs to load the program, analyse and perform any JIT compilation (Just-in-Time compilation, to improve execution time [26]). The program also needs to read in the repertoire data files for each gene and process them. On top of all that, any data that are being cached by the operating system on behalf of the program will be invalidated. Since this only needs to be done once per batch, as the batch sizes increases (and the number of batches decrease), this overhead also decreases.

This is clearly apparent in Figure 2.2, which shows that the total work done by the program to process all the sequences is reduced significantly as the n increases.

Comparing the total executable time (wall clock time) and the total CPU time of each run shows, that there was roughly double the CPU time as execution time. This indicates that the program has 2 threads of execution, but that it doesn't increase threads with workload.

2.2 Profiling

In order to determine the most appropriate approaches for improving performance and overcoming the limitations of the original implementation, further analysis is required. Specifically, we are interested in which parts of the program consume the most CPU time and memory, as well as how they increase with the workload.

The method for collecting this data is called profiling. Since the original implementation of iHMMuneAlign was written in Java, we can use various interfaces in the Java Virtual Machine (JVM) to gather this data. The JVM has an interface called Java Native Interface (JNI)[20], that allows external programs and native libraries (usually written in C/C++) to interact with Java objects. Alongside that, from Java version 1.5 onwards, the JVM also provides another interface, called Java Virtual Machine Tool Interface (JVMTI), which allows external tools to be notified as certain events (such as object allocation, method call) occur during execution.

In Figure 2.4, the data for a run of iHMMuneAlign with 6 sequences, instrumented by jProfiler [25] (a profiler implemented with the above method) is shown.

2.2.1 CPU Call Graph

Figure Figure 2.4 shows the time, and the percentage of total execution time spent in each method invocation of a single execution of iHMMuneAlign.

It begins with the equivalent of the `main()` function for a thread using Java's built in threading library - `Thread.run()`, which in turn calls `startAlignment()`. However, it

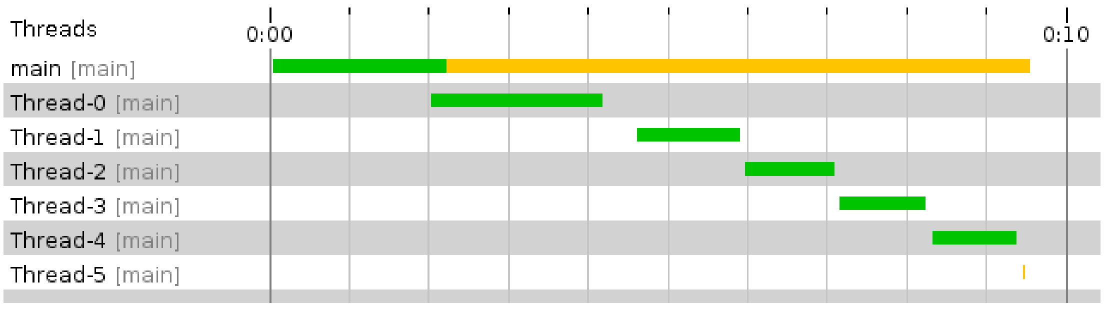
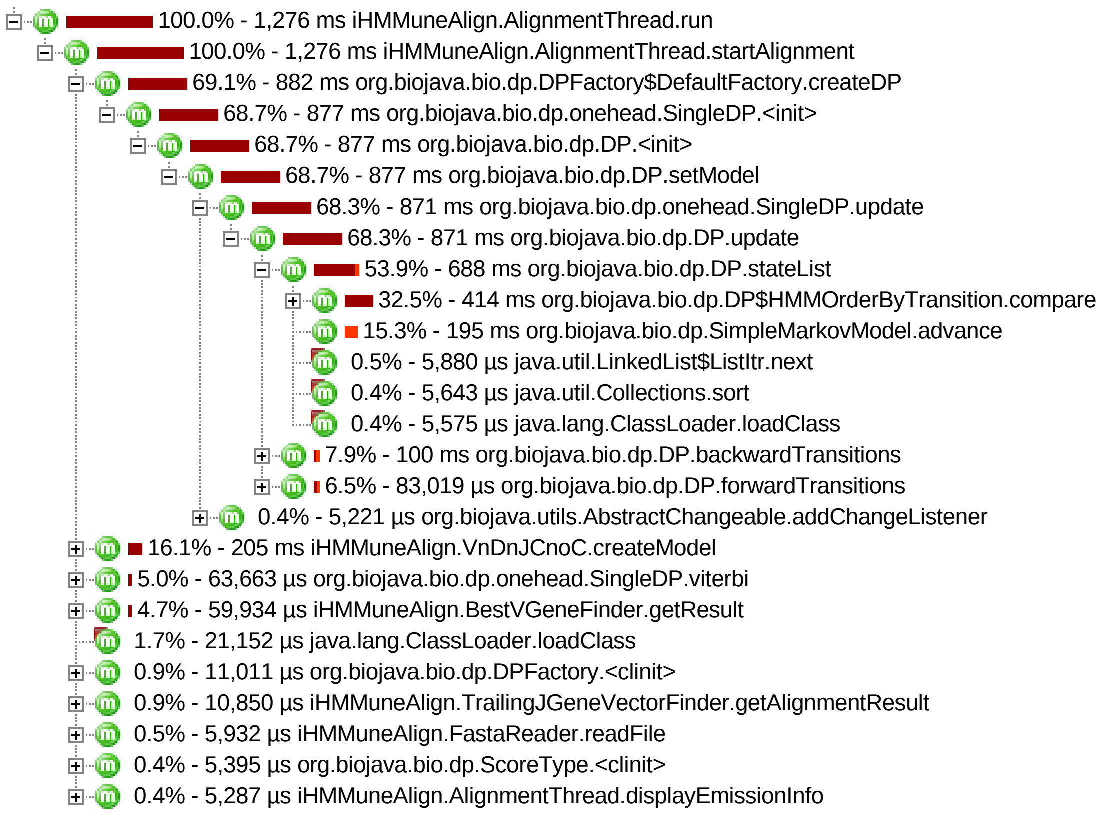


Figure 2.3: Time spent by each thread; running (🟩) and waiting (🟨)



```
100.0% - 1,276 ms iHMMuneAlign.AlignmentThread.run
  100.0% - 1,276 ms iHMMuneAlign.AlignmentThread.startAlignment
    69.1% - 882 ms org.biojava.bio.dp.DPFactory$DefaultFactory.createDP
      68.7% - 877 ms org.biojava.bio.dp.onehead.SingleDP.<init>
        68.7% - 877 ms org.biojava.bio.dp.DP.<init>
          68.7% - 877 ms org.biojava.bio.dp.DP.setModel
            68.3% - 871 ms org.biojava.bio.dp.onehead.SingleDP.update
              68.3% - 871 ms org.biojava.bio.dp.DP.update
                53.9% - 688 ms org.biojava.bio.dp.DP.stateList
                  32.5% - 414 ms org.biojava.bio.dp.DP$HMMOrderByTransition.compare
                  15.3% - 195 ms org.biojava.bio.dp.SimpleMarkovModel.advance
                  0.5% - 5,880 µs java.util.LinkedList$ListItr.next
                  0.4% - 5,643 µs java.util.Collections.sort
                  0.4% - 5,575 µs java.lang.ClassLoader.loadClass
                7.9% - 100 ms org.biojava.bio.dp.DP.backwardTransitions
                6.5% - 83,019 µs org.biojava.bio.dp.DP.forwardTransitions
            0.4% - 5,221 µs org.biojava.utils.AbstractChangeable.addChangeListener
    16.1% - 205 ms iHMMuneAlign.VnDnJCnoC.createModel
    5.0% - 63,663 µs org.biojava.bio.dp.onehead.SingleDP.viterbi
    4.7% - 59,934 µs iHMMuneAlign.BestVGeneFinder.getResult
    1.7% - 21,152 µs java.lang.ClassLoader.loadClass
    0.9% - 11,011 µs org.biojava.bio.dp.DPFactory.<clinit>
    0.9% - 10,850 µs iHMMuneAlign.TrailingJGeneVectorFinder.getAlignmentResult
    0.5% - 5,932 µs iHMMuneAlign.FastaReader.readFile
    0.4% - 5,395 µs org.biojava.bio.dp.ScoreType.<clinit>
    0.4% - 5,287 µs iHMMuneAlign.AlignmentThread.displayEmissionInfo
```

Figure 2.4: The call graph generated by jProfiler, for a single run of iHMMuneAlign

becomes immediately apparent that the majority of execution time from there is spent in the BioJava library.

68% of total execution time is spent constructing a `org.biojava.bio.dp.SingleDP` object, which will allow various algorithms (including Viterbi) to be performed. It takes a `org.biojava.bio.dp.MarkovModel` object, which represents the Markov Model, and converts it to its own internal format. This conversion involves sorting all the states, which is what 53.9% of the whole program's execution time is spent on.

For our purposes, this conversion is unnecessary, as we can generate the states ourselves, and can easily do so in the order necessary to run Viterbi on them. The actual call to the `DP.viterbi()` method, which actually runs the Viterbi algorithm to determine the most likely sequence of states, takes only 5.0% of the total execution time.

2.2.2 Threads

An important aspect of the existing implementation is its threading model - i.e. how it distributes the workload between multiple threads of execution (which can then run on multiple CPUs or CPU cores). Since we already determined in Section 2.1.1 that it does indeed use multiple threads, we need to look more closely at exactly how it does so.

For this, we have our profiler record events in the `java.util.concurrent` package, which include thread creation, destruction and blocking (waiting for another thread to complete). The results of this, shown in Figure 2.3 confirm our earlier hypothesis. The program has two threads active most of the time; one processing the gene, and one (main thread) waiting for it to complete, before starting the next one.

The fact that they use a threading library indicates that the original authors intended to make iHMMuneAlign a concurrent program, processing multiple gene sequences in parallel. While their architecture would have been less performance efficient than using a thread pool (as they have to pay the cost of thread creation and destruction for each sequence), it would still have caused a notable improvement if they allowed the threads to execute in parallel.

Nonetheless, the fact that the main thread doesn't create each thread until its predecessor terminates indicates said efforts of parallelism were abandoned. It is widely regarded in industry that concurrent programming is rather difficult and the original authors most likely deemed completing the project was more prudent than making it concurrent.

No further investigation was done to determine what would need to be done in order to make the current implementation concurrent, as it will most likely be rewritten for this project.

Chapter 3

Proposed Solution

3.1 Re-implementing From Scratch

Based on the above analysis, it is clear that the original codebase has many pitfalls, and limitations that make it difficult to proceed further.

3.1.1 Code Maintainability

Most of the logic for processing a gene sequence is contained in the

```
AlignmentThread.startAlignment()
```

method, which over 2000 lines long. The project doesn't use any revision control systems, and the fact it is littered with commented out code most likely indicates that it didn't do so at any point in the past either.

There are no unit tests that cover most of the program logic, and it would be difficult to write any, since the logic hasn't been split up. This makes it rather difficult to make changes, without introducing any new subtle bugs, or undesired behaviour.

The original authors are no longer available to provide assistance for understanding their codebase, and the style guide they followed (or lack thereof) makes it difficult to read and follow.

It is understandable that the original authors did not prioritise long term code maintainability, as they could have been rushed for time, or they may have deemed it unlikely their could would be used and maintained 7 years later.

3.1.2 Concurrency / Thread Safety

Concurrency must be considered from the very early design decisions onwards, for a program to be efficient and parallel. The fact the original authors abandoned their efforts for parallelism indicates that there may have been concurrency bugs, like race conditions [19] that are very difficult to trace.

Access to shared data must be carefully controlled, and synchronised to avoid data corruption (and race conditions). The potentially high overhead of synchronisation should lead to program structure designs where shared data between threads is minimal. In addition, the thread safety of any libraries used must be carefully considered - if a class is not thread safe, it shouldn't be shared between threads; and if a class is not re-entrant, there shouldn't even be separate instances in different threads.

3.1.3 Libraries and Language

Some of the libraries that the original authors used have since been deprecated. The specific libraries are mentioned in more detail below. Considering that some libraries account for most of the performance inefficiencies of the program (as shown in Section 2.2.1), they will certainly need to be replaced. Since the libraries' APIs heavily influenced the structure of the code, replacing them while maintaining the existing code structure will be a difficult task.

There is also the language itself to consider: Java programs tend to be 2-3 times slower than their equivalent C++ counterparts, and even more so in CPU bound tasks like this [3]. Reimplementing from scratch will allow us to do so in a different language. C++ is a good candidate - it is quite popular for constructing large scale, high performance applications [8]; it contains high concepts from high level languages, while still allowing for low level optimisations, including memory management.

3.1.4 Implementation Stages

The initial version of the new implementation will be loosely based off the already existing implementation. Once the code is rewritten and structured in a manner that lends itself to easy modification, the real optimisation (such as threading, and caching computation) can begin.

3.2 Dependencies of iHMMuneAlign

Apart from the codebase, the current implementation has several dependencies that perform significant amounts of computation. This section outlines how the new implementation will replicate their features.

3.2.1 BioJava

BioJava [11]is a general purpose Bioinformatics library, that contains several useful features for iHMMuneAlign. However, since iHMMuneAlign was written, BioJava has undergone a complete restructuring [22], and most -if not all- the modules the original implementation uses have been deprecated.

Fasta Reader

FASTA was originally the format used by a program that bore its name to store databases of DNA sequences [21]. These days, it is commonly used in Bioinformatics to represent large quantities of gene sequences within a file. BioJava could parse files FASTA format and convert them to its in-memory representation, which were then used by iHMMuneAlign. Since we no longer intend to use BioJava, an alternate method was required to parse the inputs and repertoire files for each genes was required. Since the FASTA format is a very simple format, a custom parser could be built quite easily. In addition, the NCBI C++ Toolkit can also be used to trivially parse FASTA files.

Bioinformatic Data Structures

BioJava contains a large collection of pre-built data structures to represent commonly occurring biological objects and concepts (E.g. RichSequenece, FiniteAlphabet). Here we have several viable solutions.

Bio++ is a set of general purpose Bioinformatics libraries written in C++ [5]. Much like BioJava, it also provides various pre built data structures that represent various biological entities, like gene sequences. However, how well they correspond to the used data structures from BioJava and how useful they are when re-implementing is yet to be seen.

Qt is a C++ framework, originally created by Trolltech and now maintained by Digia [34]. It has various components that assist in the rapid building of user interfaces, as well complex systems and algorithms. While they are less specific to biology, they are designed for speed and efficiency. Once the initial version is built using Bio++, if Bio++'s data structures are deemed a significant performance bottleneck, new ones will be custom built using Qt's support structure.

STL The Standard Template Library (STL) that accompanies C++ also contains a series of very high performance, generic data structures and algorithms [18]. However, it is widely considered to be less user friendly than its alternatives, like Qt. For this reason, Qt's data structures were selected as the primary candidate for a custom implementation, and if they cause a significant performance limitation, they will be replaced by their STL counterparts.

Boost is another, highly comprehensive set of libraries that extend the functionality of the STL [4]. Select features from Boost often end up in subsequent revisions of the STL. Should functionality of the STL be insufficient to replace the Qt implementation once it's been deemed a performance bottleneck, Boost will be used to compensate.

3.2.2 DP Module

We have determined in Section 2.2.1 that the DP module (in the `org.biojava.bio.dp` package) accounts for more than two thirds of the total runtime, with half on a data conversion that is ultimately unnecessary.

Since the iHMMuneAlign was written, BioJava has undergone a major overhaul [22] and the entire module is now deprecated. For replacing the parts of its functionality that could have been useful moving forward (namely, the implementation of Viterbi algorithm), we have two choices:

3.2.3 External Library

Considering Hidden Markov Models are a very popular for a variety of applications [6], there are several C++ implementations available for use. For this, two of them were evaluated

MLPack is a scalable C++ machine learning library [2], by the Georgia Institute of Technology. It has a module to deal with Hidden Markov Models, which includes an implementation of Viterbi. However, the implementation entertains the possibility of every state transitioning to every other state, and as such performed too slow in our tests.

StochHMM is a library and set of tools dealing with Hidden Markov Models, by the University of California [16]. Its evaluation was done by feeding in the model as a text file to its Viterbi solver, which crashed. The cause of the crash was not determined, as we moved on to other solutions.

3.2.4 Custom Implementation

After the failures of the other approaches, the possibility of implementing the algorithm ourselves was evaluated. A proof of concept implementation was created, tested with a roughly equivalent workload, and showed an order of magnitude improvement over the original solver from BioJava.

Later on in the project, specifically when the model generation code is implemented, its performance can be evaluated more accurately. The final decision regarding its use will be made at the time, depending on the performance of the alternatives.

3.3 Concurrency

Since the advent of multi-core and multi-processor machines, concurrency has become essential for high performance applications [27]. Since clock speeds have stagnated, yet the parallelism available in modern hardware continues to rise designing our implementation to be concurrent from the start will ensure that it can take advantage of the speed improvements granted by future generations processors.

The nature of our problem presents us two obvious ways of splitting it up to distribute over multiple threads of execution.

3.3.1 Per Sequence

The fundamental idea is to have each processing a different input gene sequence at the same time. This is the approach the authors of the original implementation attempted, although their implementation created a new thread for each sequence.

Creating a thread requires at least one system call (which is very expensive [31]) and destroying requires another. Additionally, the operating system has to do more book-keeping, as well as preparing a bunch of extra resources for the thread, which then have to be cleaned up - causing a significant performance cost [32].

A much better performing design would be to create a pool of worker threads at the start, and a queue of sequences for them to process. Each thread will then pop the sequence at the top of the queue, process it, write out the result, pop the sequence that is now at the head of the queue and repeat. In this case, the primary limitation for parallelism would come from the queue itself - by virtue of being a shared data structure, it would require synchronisation to maintain consistency [9].

If the synchronisation overhead of the queue is found to be a significant limitation on parallelism, it can be skipped altogether. Since all the inputs sequences are passed to the program in a file, the file itself can be partitioned - then each thread would read, parse and process all the sequences within a partition.

Doing so has the advantage that it will reduce the synchronisation necessary between threads. On the other hand, if the sequences in some partitions require significantly less work than others, some threads would finish much quicker than others and be idling instead of doing useful work. This can be mitigated somewhat, by using a technique called work stealing [10], where threads that have finished all their jobs take pending jobs from other threads.

3.3.2 Per Processing Stage

The other way, is to consider each sequence as a series of tasks, and perform each one in a different thread. It's infeasible to speed up the processing of an individual sequence using this method, as most (if not all) of these tasks would depend on the results of the previous, and initialising a thread could take longer than some of the tasks.

Considering that the goal of the project is to scale well with increasing numbers of sequences, the following approach makes more sense; tasks can be done in batches. I.e. have n threads process the first stage of $n \times k$ sequences (where k is an integer constant), and then have all n threads process the second stage and so on. This has the primary advantage of having much greater data locality (decreasing the working set, page faults, cache misses) and increasing performance significantly [30].

Per contra, this approach will be heavily reliant on multiple work queues to distribute the tasks of different sequences between threads. As explained in the above section, the synchronisation overhead of the queues may become a serious bottleneck on parallelism - even more so, due to the much heavier reliance on it.

Chapter 4

Design & Implementation

4.1 Initialization

Since iHMMuneAlign is rather configurable, it requires a large amount of input data for each run that cannot be embedded in the code directly. This data has to be read during program initialisation and stored in memory such that it can be efficiently used in their required of the algorithm.

4.1.1 IGHD/IGHJ Repertoires

The use-case for the germline IGHD and IGHJ repertoires are farily straight forward. For each nucleotide in each gene, a state needs to be created that represents the probability that gene caused it to be there.

Since this needs to be done for each gene/nucleotide in order, there's no need for random access, or efficient searching. They're stored in a contiguous block of memory in a dynamic (variable size) array. This provides good cache/memory locality [**cache-locality**], while being easy to use.

4.1.2 IGHV Repertoire

Unlike the other two genes, not every IGHV gene needs to be in the model. Instead, a BLAST search is performed, and (for now, at least) only the most likely hit –the closest germline IGHV gene– is used.

However, the BLAST results do not in fact give the sequence in its output; It only presents the relevant matching/aligned section of the sequence. Fortunately, it also gives the (unique) name of the gene, as specified in the repertoire file it was given.

Consequently, an index must be built from the gene repertoire that maps each gene name to its corresponding sequence. Considering this lookup needs to be done for each model, it should be as efficient as possible. This index is currently stored as a hash-table in memory, allowing for efficient, constant time lookup [17].

4.1.3 Mutation Ratios

A large part of what iHMMuneAlign attempts to model is mutation of various germline genes, during the recombination process. While it is known that some nucleotides are more likely to mutate to a certain one than others, determining this is out of scope for iHMMuneAlign [7]. As a result, iHMMuneAlign takes in these probabilities as input.

A simple map from a nucleotide to nucleotide would not be sufficient, as it depends on the two surrounding nucleotides as well. These ratios are given in a format specific to the previous implementation of iHMMuneAlign; an example can be seen in Figure 4.1.

The probabilities are parsed into double-precision floating point numbers; the string on the left is split up, with all the unnecessary characters (arrow and brackets) dropped. Doing so allowed for more efficient hashing when put into a hash-table in memory, effectively forming an index much like the IGHV Repertoire.

```
A(A->C)A  0.200711181
A(A->C)C  0.234567901
A(A->C)G  0.090360982
A(A->C)T  0.249740516
A(A->G)A  0.698044251
A(A->G)C  0.679012346
A(A->G)G  0.798111707
A(A->G)T  0.533465028
A(A->T)A  0.101244567
A(A->T)C  0.086419753
```

Figure 4.1: An extract of a mutation probabilities file

4.2 BLAST

The first step is to determine the most likely germline V gene is in the target sequence. Fortunately, this can be determined quite efficiently by aligning the input sequence with all known V genes [7]. For that we opted to use the excellent alignment tool by NCBI: BLAST [1], like the original implementation.

BLAST is distributed as source code, and can be used in two ways:

4.2.1 Invoking Binary

There are binaries for all the common tasks, that take in sequence repertoires as files in FASTA [21] format. The previous implementation used the `blastn` executable, which takes in a sequence in FASTA format, and writes the results (matching sequences in the repertoires) out to an XML file.

The primary advantage of using blast in this manner would be the ease of development. Since the binaries come pre-packaged, very little extra code would be required to invoke the binary from within iHMMuneAlign, and parse the XML output. However, doing so incurs a (potentially significant) performance penalty at run time due to the cost of:

Process Creation The operating system has to create a new process for the blast executable, perform all its bookkeeping operations, and clean up after it is completed.

Reading Input Files The executable would then have to read the database files from disk again, instead of using the copy that already in memory (residing in iHMMuneAlign's address space).

Serialising Results BLAST, like most high performance programs, uses its own internal intermediate representation for the input and germline genes[1]. Once the alignment search is completed, it will then need to convert that data into a human readable, or standardised format that can be parsed by iHMMuneAlign.

Transferring Results Modern operating systems provide a certain level of isolation, such that a process can not directly access data in a section of memory belonging to another process [28]. As a result, the previous implementation required the BLAST executable to write its results to disk so it could be read.

While IO operations are generally expensive, ones involving disk access tend to be especially so [29]. This is due to the fact that disks are several orders of magnitude slower than main memory (RAM).

As explained in Section 2.1.1 (on page 8), these overheads were partially responsible for large performance inefficiencies in the original implementation.

4.2.2 Linking Shared Library

BLAST+ can also be compiled as a library, that can be statically or dynamically linked the target program using it. This would the alignment search to be performed directly in the process of iHMMuneAlign, effectively eliminating all of the overheads listed in the previous section.

The disadvantage of this approach is that it requires extra work during the development of iHMMuneAlign. This includes the work required for:

Acquiring Libraries The BLAST+ binary executables are quite prevalent in bioinformatics research and as such, they are readily available for most research/development platforms. The libraries however, are much less common, as they are generally used by developers working on production tools relating to BLAST+.

This would require researching the build process used by the BLAST+ libraries. This is a non-trivial task, as the documentation is somewhat lacking, and makes implicit assumptions about the build environment which do not necessarily hold outside of NCBI lab machines.

Programming for Libraries The binaries accept the input data and germline repertoires in FASTA format, and it outputs in XML. Both of these are widely used, standard formats; There are a range of utilities in most languages to deal with them. As a result, very little code would need to be written specifically for it.

Using the data-structures that comprise the internal representations used by the BLAST directly, would be much more complicated. And once again, the documentation regarding were not as comprehensive as they could have been.

Distributing Libraries The binaries are quite widespread, well known, and can be assumed to be already available on machines that iHMMuneAlign would run on. Thus, it would be the users' responsibility to ensure they have it installed on their machine, as it's a required dependency.

That same assumption can not be made for the library, and as a result we must distribute it with the program. Also –and more significantly–, the libraries must be available to the compiler when iHMMuneAlign is being built; This would significantly increase the work required to port iHMMuneAlign to a new platform, as well as making it infeasible to distribute primarily as source code.

Due to the performance considerations described in Section 4.2.1 (on page 23), the we opted to link to the BLAST libraries, and perform the alignment search directly in iHMMuneAlign. However, after spending a significant amount of time attempting to overcome the difficulties detailed in Section 4.2.2 (on page 24), we elected to use the binaries for now.

At the time of writing, the current implementation of iHMMuneAlign assumes the `blastn` binary has already been invoked, and expects to be given its results in XML format. This process can be made into a script easily, and will be revisited at a later date, once more impactful performance optimisations have been exhausted.

4.3 Pipeline

The entire process of creating a HMM to represent immunoglobulin recombination and solving it using Viterbi's algorithm can be split up into several smaller problems. These can be viewed as stages in an assembly line where the result of each one feeds into the next, terminating with the final result. In computing, this is called a pipeline.

4.3.1 Parsing Blast Results

At this stage, iHMMuneAlign does not integrate with the BLAST+ libraries, nor does it invoke the binaries directly; this is assumed to have been performed by the user or a start-up script. Consequently, the first stage of the pipeline is to parse and convert all required information from BLAST+ into an internal format, allowing for efficient model generation.

The BLAST+ binaries output their results in XML format. An extract of such an output file is shown in Figure 4.3. This file was generated by running `blastn` on the AJ512650.1 example sequence that was provided with iHMMuneAlign: "*Homo sapiens partial mRNA for immunoglobulin heavy chain variable region (IGHV gene), clone 43*".

It contains various meta-data about the program, information about input and configurations, and details of a few best matching sequences (hits). We only care about the highest scoring hit; specifically the name of the germline gene, the start of the aligned match.

As mentioned in Section 4.2.2 (on page 25), XML is a widely used format, supported

Figure 4.2: Overview of the Model Solving 'Pipeline'

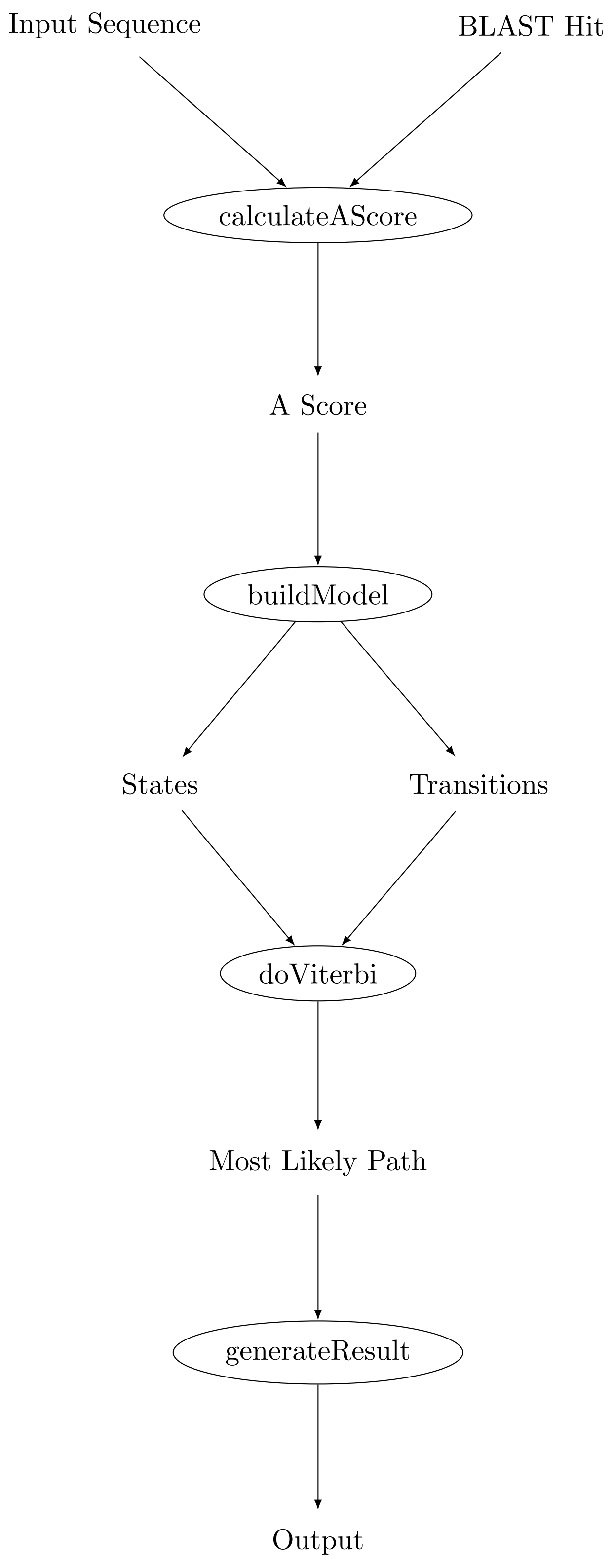


Figure 4.3: An extract from a BLAST+ output file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE BlastOutput PUBLIC "-//NCBI//NCBI␣BlastOutput/EN" "http://www.
   ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.dtd">
<BlastOutput>
        ...
  <BlastOutput_iterations>
    <Iteration>
      <Iteration_iter-num>1</Iteration_iter-num>
      <Iteration_query-ID>lcl|1_0</Iteration_query-ID>
      <Iteration_query-def>AJ512650.1| Homo sapiens partial mRNA for
          immunoglobulin heavy chain variable region (IGHV gene), clone
          43</Iteration_query-def>
      <Iteration_query-len>351</Iteration_query-len>
      <Iteration_hits>
        <Hit>
          <Hit_num>1</Hit_num>
          <Hit_id>lcl|IGHV3-9*01(L1)</Hit_id>
          <Hit_def>No definition line found</Hit_def>
          <Hit_accession>IGHV3-9*01(L1)</Hit_accession>
          <Hit_len>298</Hit_len>
          <Hit_hsps>
            <Hsp>
              <Hsp_num>1</Hsp_num>
              <Hsp_bit-score>452.567</Hsp_bit-score>
              <Hsp_score>460</Hsp_score>
              <Hsp_evalue>1.75847e-129</Hsp_evalue>
              <Hsp_query-from>4</Hsp_query-from>
              <Hsp_query-to>286</Hsp_query-to>
              <Hsp_hit-from>16</Hsp_hit-from>
              <Hsp_hit-to>298</Hsp_hit-to>
              <Hsp_query-frame>1</Hsp_query-frame>
              <Hsp_hit-frame>1</Hsp_hit-frame>
              <Hsp_identity>268</Hsp_identity>
              <Hsp_positive>268</Hsp_positive>
              <Hsp_align-len>283</Hsp_align-len>
              <Hsp_qseq>GAGTCGGGGGGAGGCTGGGTACAGCCTGGCAGGTCCCTGA...</
                  Hsp_qseq>
              <Hsp_hseq>GAGTCTGGGGGAGGCTTGGTACAGCCTGGCAGGTCCCTGA...A</
                  Hsp_hseq>
            </Hsp>
          </Hit_hsps>
        </Hit>
      </Iteration_hits>
  </BlastOutput_iterations>
  ...
</BlastOutput>
```

Note that most lines have been removed, to highlight the important parts of the file. Sections that have been removed are marked with ellipses.

by many libraries and frameworks in C++. Section 3.1.3 (on page 14) proposed the use of C++ frameworks such as Qt[34] and Boost[4]; both of which support reading and writing XML files.

They are both large and extensive frameworks that are hard to distribute, much like BLAST+. Unlike the BLAST+ libraries or the NCBI C++ toolkit however, they widely used and as a result, much more readily available on lab machines that iHMMuneAlign will be used in. Nevertheless, their adoption was delayed until absolutely necessary; in an effort to make iHMMuneAlign easy to distribute.

Presently, a small and lightweight, header-only library called PugiXML[12] is used to parse the `blastn` output file. It very efficient, and sufficiently easy to use; but its continued use will be re-evaluated should iHMMuneAlign becomes dependent on Boost or Qt.

4.3.2 Calculating A-Score

4.3.3 Generating Model

Generating V/D/J States

Generating Transitions

4.3.4 Running Viterbi

4.3.5 Printing Results

Chapter 5

Evaluation

5.1 Goals

To reiterate, the primary goal of this project was to improve the performance of iHMMuneAlign; the HMM based gene alignment tool. The best method of achieving that was determined to be re-implementing from scratch, for reasons explained in Section 3.1 (on page 13).

The secondary goal was to improve the maintainability of iHMMuneAlign; to avoid necessitating such rewrites in future. These two were quite well achieved, as will be demonstrated in the following sections.

However, due to time constraints, not all features of the old implementation could be recreated in the new one. This was a conscious decision – we prioritised performance and maintainability over feature completeness, in order to demonstrate the possibilities.

5.2 Current State

As detailed in Section 4.2 (on page 23), iHMMuneAlign no longer does the BLAST pre-alignment for the IGHV gene directly. For the moment, there is a BASH script which invokes `blastn` and then iHMMuneAlign with its output. For all the benchmarks in this section, invoking the new implementation of iHMMuneAlign will actually refer to launching this script.

Currently, iHMMuneAlign does not support gene sequences with gaps in them. If it encounters one during alignment, it will abort that sequence. This is an important feature, and will be re-added at a later date.

It is important to take note however, that this will not effect the performance comparisons below: the inputs will not result in any gaps during alignment, and the performance penalty of checking if gaps exist is paid by both implementations.

Presently, iHMMuneAlign does not model N and P addition. The previous implementation models N-addition; with evidence it also modelled P-addition at an earlier point in time. Modelling these require having extra states in the model, which will make a significant impact on total workload (and thus, performance). In order to allow for a fair comparison of performance, extra D and J states are added to the model. This is of course, a detriment to correctness and part of the concious effort to prioritise performance at the expense of correctness.

Due to the limitations listed above, the output of iHMMuneAlign no longer matches exactly with the previous implementation. However, the overall improvement of iHMMuneAlign is an ongoing process and all important features will be added back in.

5.3 Performance

The new implementation of iHMMuneAlign was designed from scratch with performance in mind, dictating choice of language, frameworks/libraries, as well as various architectural decisions.

Note that all the benchmarks and analysis detailed in this section were performed on a machine with an Intel® Xeon® CPU E5-1650 v3 @ 3.50GHz, 32GiB RAM and a Samsung® 840 Pro SSD. The CPU contains 6 physical cores, and uses Intel® Hyper-Threading technology to provide 12 logical cores.

5.3.1 Runtime

While there are many different aspects when it comes to the performance of computer programs, in this case we care about run-time specifically.

Method Details

The benchmark to measure and compare the total run-times was fairly straightforward.

Input Data
was chosen meticulously, to ensure processing each sequence would only require features supported by both implementations. To reach a sufficient workload, duplicating sequences were required; while not ideal, it should suffice for our experiments.

Execution Environment
was the machine (hardware detailed above) running Arch Linux (Kernel 4.0.4). Furthermore, the benchmarks were run within runlevel 3 [24]; a lower level environment used during machine initialisation. As such, there were no other non-essential tasks being performed on the system and should minimise interference.

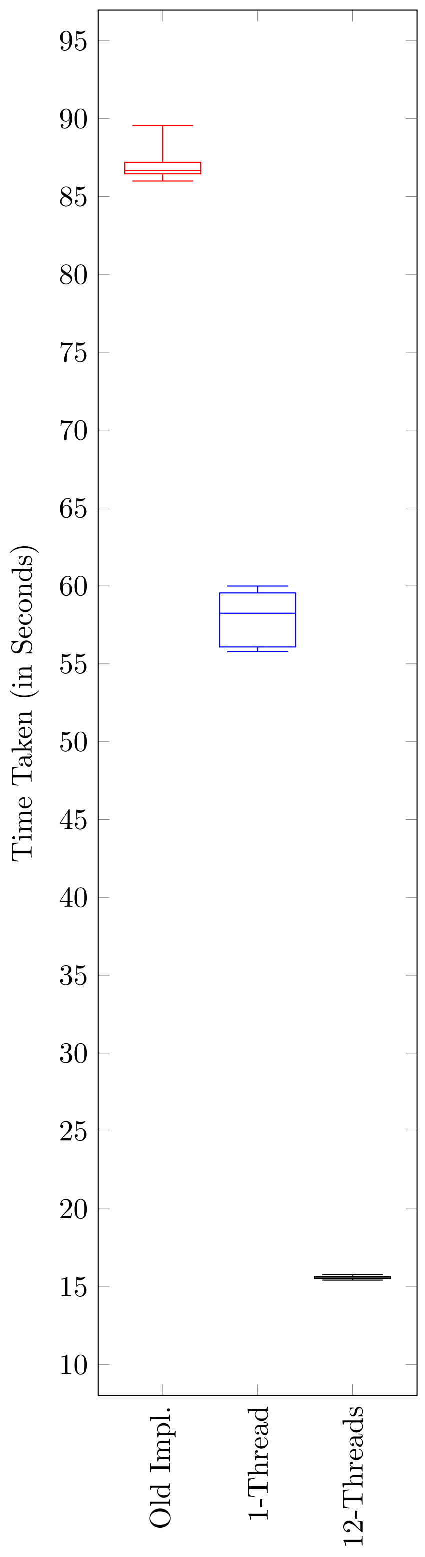


Figure 5.1: Total (Wall-clock) time taken to process 300 sequences

Measurement

of results was performed using the shell's inbuilt `time` utility. This invokes a given program and uses the various operating system features to determine the following:

Real time The total (Wall-clock) time from program invocation to termination.

CPU time Sum of the amount of time spent actually working on behalf of the program by each CPU

System time The amount of time spent by the kernel working on behalf of said program

Sample Size

A sufficiently hight sample size is required to show that the recorded data are statistically significant i.e. not by random chance. However, since a large number of tests are performed in these benchmarks, there is a trade-off between statistical precision and completing in a reasonable time. Ultimately, a value of 20 were most suitable – each program was invoked 20 times per configuration.

Interpretation of Results

The total execution times for both implementations can be seen on Figure 5.1. Since the previous implementation will only use 2 threads, it was run with 2 threads. The new implementation however, will create and use as many threads as there are available CPU cores on the target machine (12 in this case).

Also note that the graph is a box plot for all 3 measurements, rather than just the first two. However, in the case of the third measurement (new implementation without any artificial constraints) this is difficult to see, as the variance is so low relative to the difference in execution time between it and old implementation.

It is readily apparent from the graph that the new implementation is approximately 5 times faster on this machine. While that factor will increase on more powerful machines, it should also be noted that the new implementation is more efficient than the previous.

This can be seen by comparing the CPU times between executions, as presented in **??**. The mean CPU time for the new implementation (restricted to two threads) was 62.24 seconds, while the old one used 230.55 seconds. In simple terms, this means that the old implementation must to roughly 4 times the work to end up with the same results. This can be attributed to various factors including the overhead of Java, as well as the (non performance oriented) structure of the old program.

5.3.2 Parallelism

As described in detail in Section 3.3 (on page 18), parallelism is essential for modern high performance applications. While total computational power in modern machines are increasing faster than ever, single threaded performance has stagnated. Clock speeds are no longer increasing significantly; all that extra power comes in new cores, which require increasing parallelism to leverage.

It is clear from the analysis performed in Section 2.2.2 (on page 11), that the previous implementation has a very low degree of parallelism. In fact, it would cease to gain any benefit from a machine that has more than two cores, which nearly all modern machines do.

Figure 5.2 shows that there Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

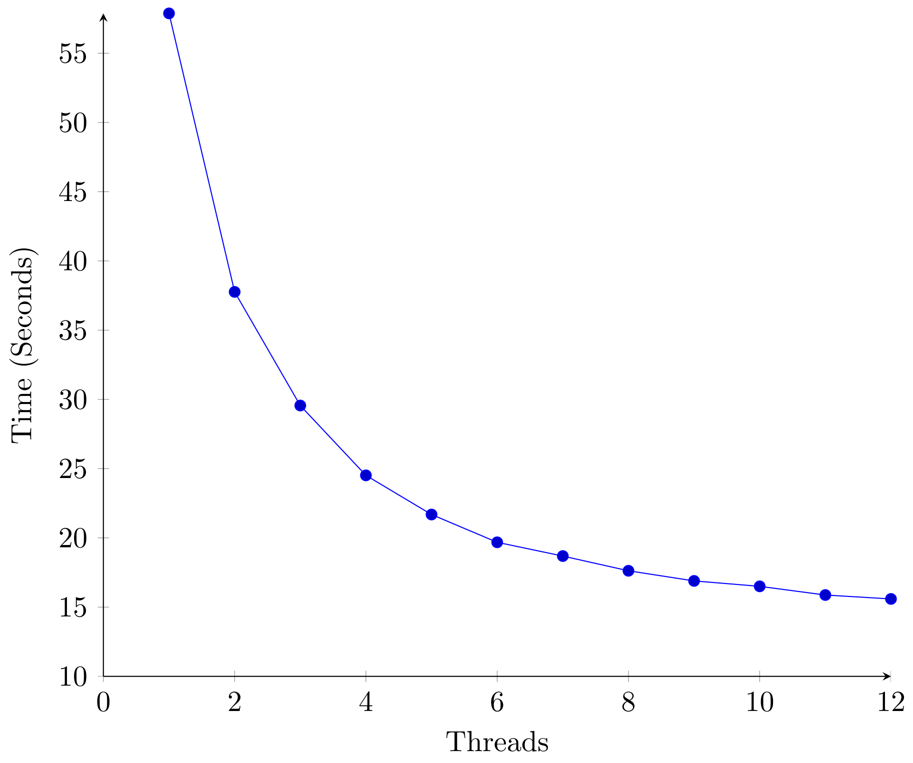


Figure 5.2: Total (Wall-clock) time taken to process 300 sequences, using a variable number of threads

Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.4 Maintenance

The second goal of this project is to make the code for iHMMuneAlign much more maintainable, allowing others to add new features as well as perform further optimisations.

5.4.1 Revision Control

The previous codebase had been maintained since its inception in 2007 until present day by various people. Many features have clearly been added and removed since then, resulting in large chunks of dead (unused) or commented out code.

If this project were under revision control, it could have for code to have been deleted instead of being commented out (they can still be viewed later using revision history). Furthermore, it would have also allowed new features and experiments to have been constructed in branches; only entering the main branch when they were deemed suitable.

The new implementation of iHMMuneAlign has used the Git revision control system since its creation, all the progress that lead to the feature-set and code structure it has today is recorded in its history. As a result, it could avoid several issues, going forward, that plague the previous codebase.

5.4.2 Separation of Logic

One of the largest barriers to understanding the source code of the previous implementation is its lack of clear structure. Several key parts of the logic behind model generation are distributed throughout the code, making components dependent on each other in subtle ways.

This is not only makes the code much harder to follow, but also virtually impossible to modify without introducing subtle bugs; increasing the workload as the developer must watch for this.

The structure of the new implementation is laid out in detail in Section 4.3 (on page 26); with a dedicated section in the code for each stage in the pipeline, plus an initialisation section. This lack of inter-dependence (loose coupling) not only resulted in ease of optimisation (especially with regards to parallelism), but ease of understanding – a developer only needs to keep the details of that stage in mind.

5.4.3 Unit Tests

The previous implementation did not contain any form of automated testing – there were some debugging code, but it still required some sort of manual inspection.

Another consequence of the code being highly interdependent is that it is exceptionally difficult to test an individual component. If unit tests had been written from that start, it would have forced the original developers to structure the code such that components can be tested individually.

The new implementation has a series of unit tests, that were written along with the features they test. This will ensure that as developers add new features, or perform new optimisations, any changes in behaviour introduced will cause tests to fail. Not only will this encourage developers to make new improvements by reducing the risk of breaking features, but it will mean they waste less time tracking down bugs.

While there are several widely accepted techniques for writing maintainable code – which I've followed–, there is no real way of measuring code quality quantitatively; in reality only time will tell.

Chapter 6

Conclusion

A thesis requirements/template document has been created. This serves the dual purposes of giving students specific requirements to their theses — both style and content related — while providing a typical thesis structure in a LaTeX template.

6.1 Future Work

Extract the requirements from the template in order to have very concise requirements.

Bibliography

[1] Stephen F. Altschul et al. "Basic local alignment search tool". In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410. ISSN: 0022-2836. DOI: 10.1016/S0022-2836(05)80360-2. URL: http://www.sciencedirect.com/science/article/pii/S0022283605803602.

[2] Ryan R Curtin et al. "MLPACK: A scalable C++ machine learning library". In: *The Journal of Machine Learning Research* 14.1 (2013), pp. 801–805. URL: http://dl.acm.org/citation.cfm?id=2502606;%20http://www.mlpack.org/mlpack_jmlr.pdf.

[3] Matthias Kalle Dalheimer. "A comparison of qt and java for large-scale, industrial-strength gui development". In: *Klarlvdalens Datakonsult AB, Tech. Rep* (2005).

[4] Beman Dawes, David Abrahams, and Rene Rivera. "Boost C++ libraries". In: *URL http://www. boost. org* 35 (2009), p. 36.

[5] Julien Dutheil et al. "Bio++: a set of C++ libraries for sequence analysis, phylogenetics, molecular evolution and population genetics". In: *BMC bioinformatics* 7.1 (2006), p. 188.

[6] Robert J Elliott, Lakhdar Aggoun, and John B Moore. *Hidden Markov Models.* Springer, 1994.

[7] Bruno A. Gaëta et al. "iHMMune-align: hidden Markov model-based alignment and identification of germline genes in rearranged immunoglobulin gene sequences". In: *Bioinformatics* 23.13 (2007), pp. 1580–1587. DOI: 10.1093/bioinformatics/btm147. eprint: http://bioinformatics.oxfordjournals.org/content/23/13/1580.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/23/13/1580.abstract.

[8] Lois Goldthwaite. "Technical report on C++ performance". In: *ISO/IEC PDTR* 18015 (2006).

[9] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012. ISBN: 978-0-12-370591-4. URL: http://books.google.com.au/books?hl=en&lr=&id=vfvPrSz7R7QC&oi=fnd&pg=PP2&dq=art+of+multiprocessor+programming+Herlihy,+Maurice&ots=e_fBsPVMeP&sig=D6u_nmnRdj6Lu5Ojg29_j8O_PtI.

[10] Maurice Herlihy and Nir Shavit. "Work Distribution". In: *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012. Chap. 16.4. ISBN: 978-0-12-370591-4. URL: http://books.google.com.au/books?hl=en&lr=&id=vfvPrSz7R7QC&oi=fnd&pg=PP2&dq=art+of+multiprocessor+programming+Herlihy,+Maurice&ots=e_fBsPVMeP&sig=D6u_nmnRdj6Lu50jg29_j80_PtI.

[11] R. C. G. Holland et al. "BioJava: an open-source framework for bioinformatics". In: *Bioinformatics* 24.18 (2008), pp. 2096–2097. DOI: 10.1093/bioinformatics/btn397. eprint: http://bioinformatics.oxfordjournals.org/content/24/18/2096.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/24/18/2096.abstract.

[12] Arseny Kapoulkine. *PugiXML.* URL: http://pugixml.org/.

[13] CEH Lee et al. "Reconsidering the human immunoglobulin heavy-chain locus". In: *Immunogenetics* 57.12 (2006), pp. 917–925. URL: http://link.springer.com/article/10.1007/s00251-005-0062-5.

[14] Marie-Paule Lefranc et al. "IMGT, the international ImMunoGeneTics information system®". In: *Nucleic Acids Research* 33.suppl 1 (2005), pp. D593–D597. URL: http://nar.oxfordjournals.org/content/33/suppl_1/D593.short;%20http://nar.oxfordjournals.org/content/33/suppl_1/D593.long.

[15] Honghua Li et al. "Genetic diversity of the human immunoglobulin heavy chain VH region". In: *Immunological reviews* 190.1 (2002), pp. 53–68. URL: http://onlinelibrary.wiley.com/doi/10.1034/j.1600-065X.2002.19005.x/full.

[16] Paul C Lott and Ian Korf. "StochHMM: a flexible hidden Markov model tool and C++ library". In: *Bioinformatics* (2014), btu057. URL: http://bioinformatics.oxfordjournals.org/content/early/2014/01/30/bioinformatics.btu057.short.

[17] W. D. Maurer and T. G. Lewis. "Hash Table Methods". In: *ACM Comput. Surv.* 7.1 (Mar. 1975), pp. 5–19. ISSN: 0360-0300. DOI: 10.1145/356643.356645. URL: http://doi.acm.org/10.1145/356643.356645.

[18] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* 3rd. Addison-Wesley Professional, 2009. ISBN: 0321702123, 9780321702128.

[19] Robert H. B. Netzer and Barton P. Miller. "What Are Race Conditions?: Some Issues and Formalizations". In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623. URL: http://doi.acm.org/10.1145/130616.130623.

[20] Oracle. *Java Native Interface Specification.* 2014. URL: http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[21] WilliamR. Pearson. "Using the FASTA Program to Search Protein and DNA Sequence Databases". English. In: *Computer Analysis of Sequence Data.* Ed. by AnnetteM. Griffin and HughG. Griffin. Vol. 24. Methods in Molecular Biology. Humana Press, 1994, pp. 307–331. ISBN: 978-0-89603-246-0. DOI: 10.1385/0-89603-246-9:307. URL: http://dx.doi.org/10.1385/0-89603-246-9:307.

[22] Andreas Prlić et al. “BioJava: an open-source framework for bioinformatics in 2012”. In: *Bioinformatics* 28.20 (2012), pp. 2693–2695. DOI: `10.1093/bioinformatics/bts494`. eprint: `http://bioinformatics.oxfordjournals.org/content/28/20/2693.full.pdf+html`. URL: `http://bioinformatics.oxfordjournals.org/content/28/20/2693.abstract`.

[23] Jeffrey V Ravetch et al. “Structure of the human immunoglobulin μ locus: characterization of embryonic and rearranged J and D genes”. In: *Cell* 27.3 (1981), pp. 583–591. URL: `http://www.sciencedirect.com/science/article/pii/0092867481904001`.

[24] Yvan Royon and Stéphane Frénot. “A Survey of Unix Init Schemes”. In: *CoRR* abs/0706.2748 (2007). URL: `http://arxiv.org/abs/0706.2748`.

[25] Jack Shirazi. “Tool Report: JProfiler”. In: *Java Performance Tuning, Jun* (2002).

[26] T. Suganuma et al. “Overview of the IBM Java Just-in-Time Compiler”. In: *IBM Systems Journal* 39.1 (2000), pp. 175–193. ISSN: 0018-8670. DOI: `10.1147/sj.391.0175`.

[27] Herb Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[28] Andrew S. Tanenbaum and Herbert Bos. “A MEMORY ABSTRACTION: ADDRESS SPACES”. In: *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. Chap. 3.2. ISBN: 013359162X, 9780133591620.

[29] Andrew S. Tanenbaum and Herbert Bos. “INPUT/OUTPUT”. In: *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. Chap. 5. ISBN: 013359162X, 9780133591620.

[30] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. Chap. 16.4.6, 3. ISBN: 013359162X, 9780133591620.

[31] Andrew S. Tanenbaum and Herbert Bos. “SYSTEM CALLS”. In: *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. Chap. 1.6. ISBN: 013359162X, 9780133591620.

[32] Andrew S. Tanenbaum and Herbert Bos. “THREADS”. In: *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. Chap. 2.2. ISBN: 013359162X, 9780133591620.

[33] Terencehonles. *English: Hidden Markov Model*. Nov. 8, 2009. URL: `http://commons.wikimedia.org/wiki/File:HMMGraph.svg` (visited on 10/21/2014).

[34] Qt Trolltech. *4.1 Whitepaper*.