DEPARTMENT OF COMPUTER SCIENCE

Protein Homology Without Gene Prediction Using the Apache Hadoop Framework

Benjamin Smithers

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering

May 2012 | CSMENG-12

Declaration

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Benjamin Samuel Smithers, May 2012

Acknowledgements

Firstly, I would like to thank Dr Julian Gough for his valuable guidance, feedback and patient explanations of biology throughout this project. In addition, thanks are owed to Matt Oates for his early input and advice on getting started with Hadoop.

Finally, I would like to thank HP Cloud Services, who accepted my application to their free-of-charge private beta test of their new cloud computing offering. This resource became very valuable to the project.

1 Summary

Protein homology is one of the key applications within bioinformatics. However, the rapid generation of DNA data enabled by Next Generation Sequencing means that new data analysis tools and algorithms are needed to keep pace. Increasingly, researchers are turning towards cloud computing technologies to help solve this problem.

This work introduces HadoopHmmer, which has two distinct features:

1. It allows HMMER, a popular protein homology search program based on Hidden Markov Models, to be distributed across a large number of nodes using the Apache Hadoop framework.
2. It implements a novel approach to the search for protein homologues, using DNA that has been directly translated to protein sequences without the need for gene predictions.

The performance of the Hadoop implementation was benchmarked and found to scale well with up to 19 compute nodes, which was 17.8 times faster than a single node. In addition, the new approach to protein homology was evaluated, correctly identifying 87.2% of the matches between sequences and protein models that were found by traditional methods, with a false positive rate of 7.3%.

1.1 Main Achievements

- I implemented and tested JavaHmmer, a JNI wrapper around HMMER to allow it to be run by Java programs.
- I developed and benchmarked HadoopHmmer, which allows HMMER to be distributed across multiple nodes using the Apache Hadoop framework.
- I researched, implemented and evaluated a novel approach to protein homology within HadoopHmmer, based on the merging of partial results obtained from running HMMER on a direct translation of DNA to protein sequences, without using gene prediction.
- I performed some benchmarks on the original HMMER code and used the results to guide the development and analysis of HadoopHmmer.
- I spent a large amount of time configuring and managing Hadoop clusters, first on top of the University's supercomputer, BlueCrystal, and then in a cloud computing environment.

Contents

2 Introduction and Motivation

Sequence homology is one of the core challenges in bioinformatics. It is the process of identifying biologically related DNA or protein sequences and has a variety of applications, including the determination of genetic function, protein structure and evolutionary history. For example, the function of a new protein sequence can be predicted by searching a database for similar proteins of known function.

At its core, it is string matching problem. But it is a string matching problem that must cope with the naturally occurring havoc of random mutations, natural selection and many other causes of sequence divergence [1]. A number of tools and algorithms for performing sequence homology have been developed over many years, the most effective of which use the statistical power of Hidden Markov Models.

The rest of this section discusses the current challenges for sequence homology, before explaining the objectives of this project.

2.1 Data Explosion

The time and cost associated with DNA sequencing have both fallen dramatically in recent years. This has been driven by a new wave of technologies - so called Next Generation Sequencing (NGS). Because of this, increasingly large volumes of data are being produced more and more quickly. One estimate suggests that NGS machines are producing data $500,000$ times faster than with older methods [2].

This data explosion is said to be outstripping Moore's Law, which means that the analysis of the data is, in relative terms, getting more difficult each day [3]. To give an example, Figure 1 shows how the cost of sequencing a human genome has fallen in recent years.

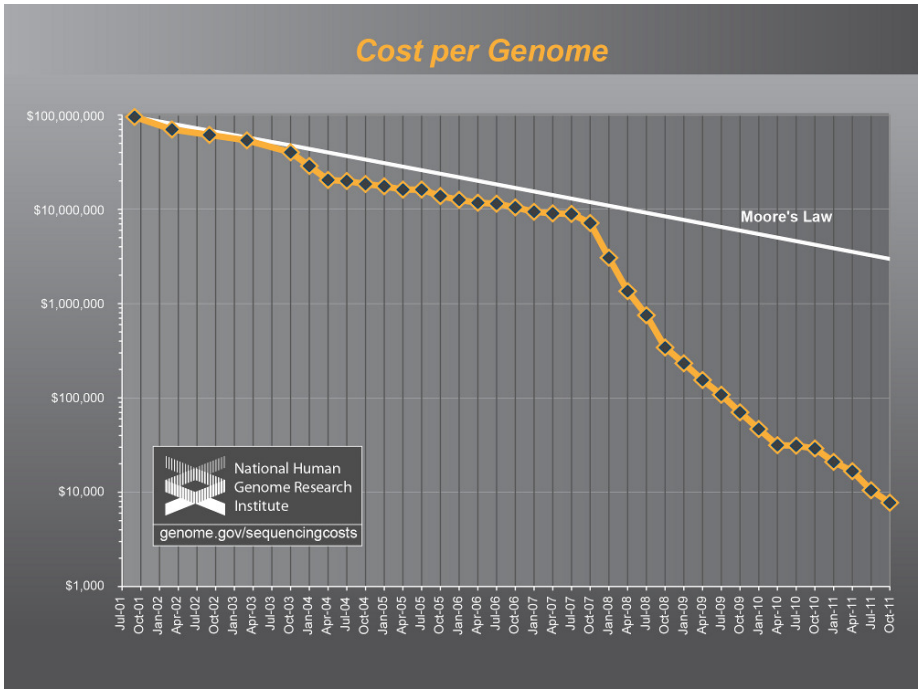


Figure 1: The rapid fall in the cost of sequencing a human genome. Reprinted from [4].

With DNA data being produced so quickly and cheaply, analysis tools need to be adjusted to handle huge volumes of data. Recently, many authors have discussed how bioinformatics tools may need to move into the cloud to cope with the continual increases in data [2, 3, 5].

2.2 Objectives of this Project

This project has two broad aims:

1. To take existing protein homology software, known as HMMER, and modify it so that it can be run on the Apache Hadoop framework, which will allow computation to be distributed over a large number of compute nodes.
2. To explore how homologous proteins can be found directly from DNA sequence data without using *gene prediction*. Gene prediction is a phase of analysis typically required before protein homology can be performed.

2.2.1 Choice of Platform

The Hadoop platform was chosen for a number of reasons. Firstly, it is designed for problems with large datasets, which this certainly is. For example, the human genome contains approximately three billion nucleotides. Second, it provides a high level abstraction for distributed programming using MapReduce. This not only means that distributing work should be easier and less error prone than with tools such as MPI, but also that fault tolerance and load balancing are more easily achieved. Third, it is well suited for use in the cloud, which, given the current drive towards cloud computing in bioinformatics, means this work is more likely to be useful for others. Finally, the Hadoop project is increasingly mature and actively supported by some of the world's largest companies. This means that more confidence can be placed in its reliability and performance and ensures that there is a wealth of information available online to help develop and debug applications written on top of Hadoop.

2.2.2 Benefits of Avoiding Gene Prediction

The primary benefit of a method of performing protein homology without gene prediction is that it reduces the number of different steps required to conduct analyses. This should help reduce the time, complexity and cost associated with sequence analysis. Secondly, gene prediction is not a perfect process and will produce errors; an alternative technique may therefore be useful. Finally, it would also allow protein homology to be used on the DNA of organisms for which there is either a non-existent or poor quality genome assembly or gene prediction. With Next Generation Sequencing machines producing DNA data faster than the it can be fully analysed, this is increasingly the case.

Of course, this method will also have its limitations. The most obvious of these is that homologous sequences will be found in sections of DNA that no longer code for a functional protein. In particular, they will be found in pseudogenes, which are genes that have mutated and deteriorated such that they are no longer functional. This is discussed more fully in section 4.3.

2.2.3 Summary of Objectives

To summarize, the main objectives of this project are to:

- Develop a method for distributing HMMER's workload using Apache Hadoop.
- Benchmark the two different methods HMMER provides for protein homology, using these results to guide the implementation.
- Evaluate the scalability of the Hadoop implementation.
- Investigate and implement a method for protein homology without gene predictions.
- Compare the results from this method with those obtained from traditional approaches.

3 Biological Background

This section gives a brief overview of the key areas of biology surrounding this work and is intended to make the rest of this paper easier to understand for readers who do not have a background in biology. The rest of this paper will try to explain most things in an abstract way, but a certain amount of terminology is inevitable. This section concludes with a short glossary of key terms.

DNA

DNA is a molecule that carries genetic information. It is used by almost all organisms to store information about their structure and function. DNA is constructed from a sequence of nucleotides. Each nucleotide contains one of four bases: A, C, T or G. The information in DNA is encoded in this four letter alphabet. DNA contains two strands, which are complementary: each base on one strand is represented with its base pair on the other strand. A pairs with T; C pairs with G.

In order to determine an organism's DNA, it must be sequenced and assembled. DNA sequencing machines determine the order of the DNA but are only able to produce relatively short strands of ordered data. DNA assembly constructs a single string of DNA from many of these short fragments.

Genes and Proteins

Genes are sections of DNA that encode proteins. Proteins are what make an organism; they provide a diverse range of structure and function, including the creation of tissue and bone, transportation of chemicals and the catalysis (speeding up) of many reactions. Proteins are constructed from a sequence of amino acids. There are 20 standard amino acids. Three bases of DNA, known as a codon, translate to one amino acid. With $4^3 = 64$ possible codons, this is not a one-to-one mapping - some amino acids are encoded by more than one codon.

Within a gene, there are regions known as introns and exons. Introns are said to be *non-coding* as they do not form the encoding of a protein. A protein is obtained by the concatenation of the exons within a gene. This is illustrated in Figure 2.

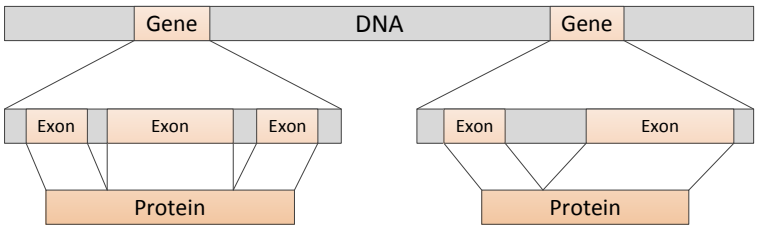


Figure 2: A diagram of the relationship between DNA, genes, exons and proteins. Introns are not shown, but they are the areas within the gene between exons.

3.1 Approximate Figures

To help give some intuition to the reader, some approximate sizes and lengths are listed here. These should only be treated as rough estimates.

The human genome has a length of just over 3 billion base pairs.

A typical protein is constructed from 100-1000 amino acids.

An average gene of a human is around 30,000 base pairs long.

The number of genes in the human genome is thought to be around 25,000.

3.2 Glossary

Amino Acid: A protein is constructed from a sequence of amino acids. There are 20 standard amino acids, and they can be thought of as the alphabet of protein sequences.

Base Pair: Each DNA base has a pair: A pairs with T; C pairs with G. Base pairs are also used as the unit of length for DNA sequences.

Codon: A triplet of DNA bases. There are 4 bases (A, C, T and G) so there are 64 different codons.

Chromosome: DNA is packaged into a number of different chromosomes for storage within a cell.

DNA: A molecule that caries and encodes genetic information.

Domain: A section of protein that has independent structure and function. Proteins are comprised of one or more domains.

Exon: A section of DNA within a gene that codes for a protein.

Gene: A section of DNA that encodes proteins.

Genome: An organism's fully sequenced and assembled DNA.

Intron: A section of DNA within a gene that does not code for a protein.

Nucleotide: An individual unit of DNA; DNA is constructed from a sequence of nucleotides.

Protein: A sequence of amino acids. Proteins provided a diverse range of structure and function to organisms.

Pseudogene: A deteriorated gene which is no longer functional; it may often "look like" a gene.

Stop Codon: A special codon that is not translated into an amino acid. Instead, it marks the end of a coding region of DNA.

4 Technical Background

In this section, we first give an overview of the technical background for the project, before discussing previous and related work.

4.1 Sequence Analysis

One of the core applications of bioinformatics is to determine if two sequences are related, or *homologous*, and this is one of the goals of sequence analysis. Sequences may be comprised of nucleotides or amino acids.

Although grounded in conventional computer science and statistics, algorithms for sequence analysis must also consider the importance of the underlying biology. In this section, a brief overview of early methods will be given, before introducing Hidden Markov Models, which are used by the HMMER package. For a more in-depth discussion, see Durbin et al. [1].

4.1.1 Sequence Alignment

Pairwise sequence alignment takes two sequences and attempts to arrange them such that characters in one sequence are aligned with the same characters in the other. Multiple sequence alignment is a generalization of this that compares more than two sequences, but is not something we consider here. Sequence alignment is a method of comparison that attempts to detect similarity whilst allowing for biological mutations: two sequences may be biologically related despite substitutions, where one characters is changed for another; insertions of characters; or deletions of characters [1]. Insertions and deletions are together known as gaps. As example of an alignment is given in Figure 3.

```
SequenceA: AGTAAGTAGCAG          AGTAAGTAGCAG
SequenceB: GAAGCAGCG             -G-AAGTAGC-G
```

Figure 3: On the left, two sequences. On the right, a possible alignment. Substituted characters are show in red, gaps are indicated by a - character.

Closely related to the Levenshtein distance, an early algorithm for sequence alignment is the Needleman-Wunsch algorithm, with a more efficient version later described by Gotoh [6, 7]. The algorithm requires a gap penalty, d, and *substitution matrix*, S. The gap penalty is a score given to a gap in the alignment. S is a two-dimensional matrix with $S(a, b)$ giving the cost associated with a substitution from a to b. It is this substitution matrix which allows the underlying biology to be considered. For example, certain amino acids are similar and a mutation between them may be reasonably likely, whilst others are dissimilar and a mutation is accordingly unlikely. Note that despite its name, the substitution matrix is used even if $a = b$, typically giving a high positive score. This allows the notion that some amino acids are more likely to be mutated than others to be captured.

The Needleman-Wunsch algorithm gives an optimal alignment using a dynamic programming approach. Given two sequences, A and B of length n and m respectively, there are three choices when aligning A_i to B_j: A_i may be aligned to B_j; A_i may be aligned to a gap; or B_j may be

aligned to a gap. Thus, if a matrix, M, is maintained where $M(i,j)$ is the maximum alignment score of $A_{0...i}$ to $B_{0...j}$, we have [1]:

$$M(i,j) = max \begin{cases} M(i-1, j-i) + S(A_i, B_j) & \text{i.e. } A_i \text{ is aligned to } B_j \\ M(i-1, j) - d & \text{i.e. } A_i \text{ is aligned to a gap} \\ M(i, j-i) - d & \text{i.e. } B_j \text{ is aligned to a gap} \end{cases} \quad (1)$$

The left column and first row of the matrix are initialized with $M(i,0) = di$ and $M(0,j) = dj$, corresponding to each position matching against a gap. The overall alignment score is then given by $M(n-1, m-i)$. Using a typical dynamic programming approach of maintaining a list of the path that was taken through the matrix, the actual alignment can be recovered [8].

The Needleman-Wunsch algorithm gives a global alignment, with one sequence fully aligned against the other. For biologically important reasons, it is often preferable to obtain an alignment between subsequences rather than the full sequence. This is known as a local alignment. The Smith-Waterman algorithm provides local alignments with a minor modification to Needleman-Wunsch: if the value of $M(i,j)$ in Equation 1 is negative, $M(i,j)$ is set to 0 instead [9]. This has the effect of starting a new alignment. The maximum value within the matrix can be used to determine the highest scoring local alignment or multiple local alignments can be extracted by examining any scores over a given threshold.

Both Needleman-Wunsch and Smith-Waterman give optimum solutions. Both also require $O(nm)$ time. With the large size and growth of protein databases[1] as well as the rapid generation of sequence data, this can often be too costly. Although recent work has shown Smith-Waterman to be very well suited to both SIMD parallelization with SSSE3 and GPU acceleration [10, 11], heuristic approaches which sacrifice the guarantee of optimality for large performance improvements have been the most widely deployed techniques.

The most commonly used heuristic approach is BLAST [12, 13]. BLAST attempts to cut the search space and avoid comparing entire sequences by first identifying particularly high scoring, but short, matches. Once identified, these short hits are extended to determine a full local alignment [1].

Finally, given an alignment and its score, it is important to be able to determine if the score is significant. For this, an expected value, or E-Value, is used. The E-Value gives the expected number of false positives: it is the expected number of alignments with the same score or higher if the database of sequences searched only contained random sequences [14]. For local alignment scores, E-values are calculated using an extreme value distribution.

4.1.2 Hidden Markov Models

Krogh et al. first introduced Hidden Markov Models (HMMs) as they are used today in sequence analysis with the concept of a profile HMM [15]. Proteins are grouped together into families, which share common structure and functionality. A profile HMM models a whole protein family and can be used to determine if a sequence is likely to belong to that family.

[1] The automatically annotated UniProt database currently lists over 21 million sequences; roughly double the number available two years ago. Statistics available at `http://www.uniprot.org/statistics/TrEMBL`. Accessed 06/05/2012.

Before discussing profile HMMs further, an overview of Hidden Markov Models and how they can be used for pairwise alignment is first given.

A Markov chain is given by a collection of states, with a transition probability between each state. Importantly, the probability of transitioning to a new state is governed only by the current state - any previous states are irrelevant [1]. As such, the probability of any path through a Markov chain can be calculated by simple multiplication of probabilities. An example of a Markov chain and path through it is given in Figure 4.

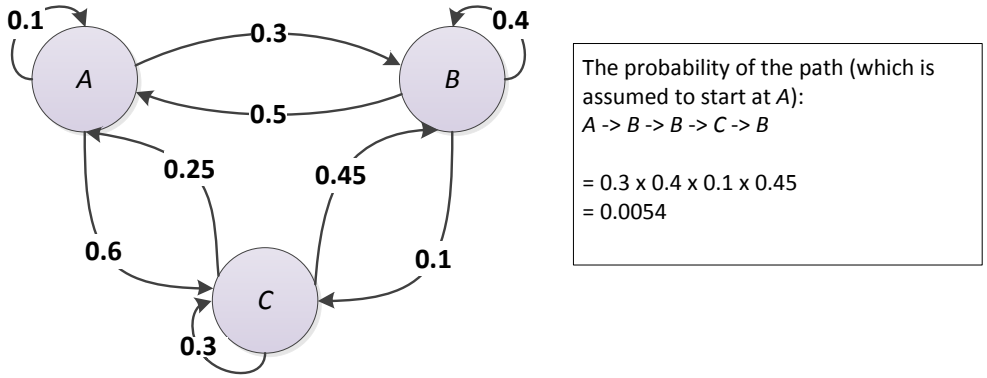
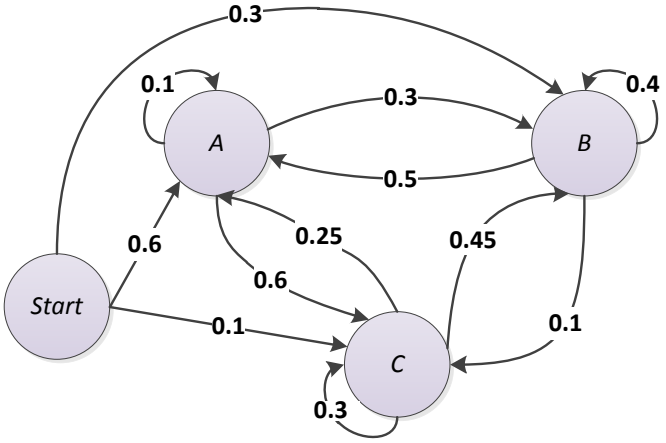


Figure 4: An example of a Markov chain with three states.

A Hidden Markov Model has a number of additional features. The first is that each state has a set of *emission probabilities*. On a path through the HMM, a symbol is *emitted* with a given probability at each state. In this way, HMMs can be viewed as sequence generators: they generate sequences with particular properties and probabilities. The Markov Model is "hidden" because the states of a path through the model are not observable; only the emitted symbols in the sequence are observed. An HMM may also contain start and end states, to model the start and end of a sequence. An example of an HMM is shown in Figure 5.



If the sequence α, γ, β is observed, we do not know which path was taken through the model, though some paths are more probable than others.

The probability of a path is dependent on both the transition probabilities and the emission probabilities at each state.

For example, the probability that the path *Start -> A -> C -> B* generated the sequence is:

0.6 x 0.7 x 0.6 x 0.6 x 0.45 x 0.5

Emission Probabilties:

	A	*B*	*C*
α	0.7	0.3	0.1
β	0.2	0.5	0.3
γ	0.1	0.2	0.6

Figure 5: An example of a Hidden Markov Model. Three states and a start state are shown, along with the emission probabilities for each symbol at each of the states. An end state is not shown.

As shown, there may be more than one path through the HMM that could generate a particular sequence. However, some paths are more likely than others. The Viterbi algorithm, originally given for the decoding of convolutional codes, can be used to determine the most probable path through an HMM [16, 1]. The Viterbi algorithm is another dynamic programming algorithm and operates as follows.

Let S be the set of all states and X be the set of all symbols. $E(s, x)$ is the emission probability of symbol $x \in X$ whilst in state $s \in S$. $T(s_1, s_2)$ is the transition probability between states s_1 and s_2. Given observed symbols $x_0, x_1 \ldots x_n$, we wish to calculate $V(s, i)$, the probability of the most probable path through the HMM that ends with symbol s after i observations.

$$V(s, i) = E(s, x_i) \max_{j \in S}(V(j, i-1)T(j, s)) \tag{2}$$

Therefore, if the probability of the most probable paths to all states at observation $i - 1$ is known, $V(s, i)$ can be calculated. Since the starting state is known, $V(s, 0) = 1$ if s is the *start* state, and 0 otherwise. This allows the dynamic programming matrix to be built, starting with the first observed symbol. In a similar manner as before, the actual path taken can be recovered by storing pointers to the previous state when building the dynamic programming matrix.

For each observed symbol, the Viterbi algorithm determines the probability of each state, taking into account transitions from all other states. The algorithm therefore has $O(n|S|^2)$ time complexity. If each state has a constant number of transitions rather than a transition to all other states, this is reduced to $O(n|S|)$ [17].

There are two further algorithms: the Forward algorithm and the Backward algorithm.

The Forward algorithm is used to determine the probability of the HMM generating a given sequence [1, 18]. Because there are potentially many paths through the HMM that could generate the sequence, they must all be considered. A simple modification to the Viterbi algorithm enables this to be determined without considering all of the paths independently, the number of which grows exponentially with sequence length. This modification is to replace the maximization in equation 2 with a summation.

The Backward algorithm is almost identical to the Forward algorithm, except that the dynamic programming matrix is built by starting at the end of sequence, instead of the start. By using the partial results contained within the dynamic programming matrices for both the Forward and Backward algorithms, the probability that a given symbol x_i was emitted from a specific state s can be obtained [1].

Returning to profile HMMs, an important question remains: how can Hidden Markov Models be used for sequence alignment?

An HMM can be built that models a number of related sequences, i.e. a family of proteins. This model can be used to solve the problem of aligning another sequence, a query sequence, to the family.

The HMM contains three states for each symbol in the model: a match state, an insert state and a delete state [17]. The match state corresponds to an alignment against the protein family. The emission probabilities of this state will be dependent on the distribution of the symbols found at this position within sequences contained in the family. An insert state corresponds to a substitution with emission probabilities based on the likelihood of particular mutations. A delete state corresponds to a gap in the alignment and does not emit a symbol. This is known as a silent state. An example of such a profile HMM is given in Figure 6.

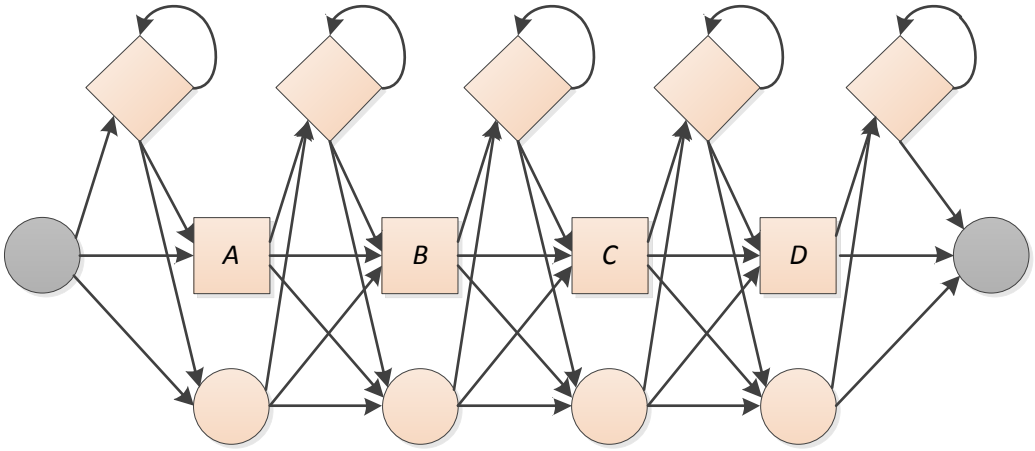


Figure 6: An example of a profile Hidden Markov Model. Match states are shown as labelled squares. Insert states are shown as diamonds, delete states as circles. The start and end state are shown in grey. Adapted from [17].

The query sequence is then compared to this model. The Forward algorithm gives an overall score and can be used to asses how likely it is that the query sequence is homologous to the

protein family [17]. Since the Viterbi algorithm gives the most probable path through the HMM and the states of the HMM indicate a match, insertion or deletion, the Viterbi algorithm gives the alignment to the model. Finally, the combination of Forward and Backward algorithms can be used to obtain the likelihood, or confidence value, for each of the aligned symbols [12].

The use of profile HMMs, and HMMs in general, offer a number of advantages. The first is increased sensitivity over tools such as BLAST [12]. This is due to the ability to more closely model biology. For example, some sections of a protein are more likely to be subject to insertions and deletions than others, whilst the distribution of inserted amino acids is position dependent [17]. Second is their ability to represent a family of proteins. A family of proteins typically share common functionality and a biologist may often be more interested in determining if a sequence is homologous to a given family than to an individual protein [1]. Finally, the statistical basis of HMMs allows formal probabilistic methods to be applied, unlike earlier profile-based approaches [17]. This allows E-Values to be calculated with the same meaning described at the end of section 4.1.1.

The major drawback of HMM-based methods has been performance, with programs typically running two to three orders of magnitude slower than BLAST [12]. The following section introduces HMMER and the heuristic approaches used to combat this problem. Previous attempts to improve performance through parallelism or distributed computing are discussed in section 4.5.

4.2 HMMER

HMMER is a suite of tools for sequence analysis using HMM methods [19]. The suite has existed for over 15 years, with the latest version being released in 2010. This work focuses on the two programs used to search protein sequences against HMM profiles, *hmmscan* and *hmmsearch*; these two programs are discussed in more detailed in section 4.2.3.

4.2.1 HMMER3 Statistics

One of the innovations in the latest version of HMMER concerns the statistical scoring system. Although Viterbi scores are analogous to alignment scores given by traditional methods such as BLAST, they are not the best scores to use [20]. The reason for this is that a Viterbi score considers only the optimum alignment. Since alignments are typically uncertain, they use less information than scores determined by the Forward algorithm, which calculates the probability of a sequence being generated by an HMM and therefore the likelihood of a given sequence being homologous to a protein family. It has been shown that Forward scores are more selective, i.e. better able to detect related sequences, than Viterbi scores [21]. However, the distribution of Forward scores has previous been unknown, which meant that rigorously establishing statistical significance of Forward scores was not possible.

The statistical distribution of the Forward scores was more recently conjectured and shown empirically to hold for a range of models and sequences. For full details, see [20]. Because of these developments, HMMER3 now uses Forward scores and has been shown to be more selective than the previous version of HMMER [12].

4.2.2 HMMER3 Heuristics

Despite HMM methods being more selective, their use has typically been limited because of their slow performance in comparison to tools such as BLAST [12]. Eddy argues that HMM methods are not inherently slower than traditional pairwise sequence alignments; they have simply lacked comparable heuristics to BLAST [14].

Because of this, HMMER3 introduces the Multiple Segment Viterbi (MSV) algorithm. In a similar manner to BLAST, the core idea to first identify particularly high scores. The MSV algorithm first simplifies the HMM by removing insert and delete states, which means only ungapped alignments are considered. A full dynamic programming algorithm is then used to calculate Viterbi scores on this simplified HMM. Any queries that to do not generate a score above a given threshold are not considered for further processing. The MSV algorithm is thus used as filter before running more time consuming algorithms.

Key to the speed of the MSV algorithm is that it can be implemented efficiently using SIMD vector instructions, such as SSE2 (x86 architectures) or AltiVec (PowerPC architectures). One of the reasons for this efficiency is that values are both scaled (resulting in loss of precision) and computed using saturating arithmetic. Since scores from this algorithm are used only as a filter, it is sufficient to know that a value is "large enough". These techniques allow 8-bit computation, resulting in 16 way SIMD parallelism in the 128-bit vector registers found in SSE2 and AltiVec.

The MSV algorithm is complemented by additional filters, including a standard Viterbi algorithm that uses reduced precision and SIMD parallelism, before the full Forward and Backward algorithms are used to calculate scores and statistical significance on sequences that have successfully passed through all the filters. It is important to remember that the filtering is a heuristic process, though a loss of just 0.3% of results is reported. In exchange, HMMER3 is between 100 and 1000 times faster than the previous version. This makes it comparable in speed to BLAST, whist remaining significantly more selective[12].

4.2.3 hmmscan and hmmsearch

HMMER offers two separate programs for searching one or more query sequences against one or more profile HMMs: hmmscan and hmmsearch. Although both of these programs essentially perform the same overall computation by comparing each profile to each sequence, they are structured differently and optimized for different tasks. hmmscan takes each sequence and compares it against a database of HMMs, whilst hmmsearch does the reverse taking each HMM and comparing it against a database of sequences. For each program, the database is read from disk for each query. Because of this, hmmscan is suited to cases with few sequences and lots of profile HMMs, whilst hmmsearch is suited the opposite case of few HMMs and a large sequence database.

When there are both a large number of sequences and a large number of HMMs, hmmsearch is recommended [2]. However, the point at which one is more efficient than the other is somewhat unclear and requires empirical analysis. This is investigated and discussed further in section 6.3.

[2]This issue does not appear to have been discussed formally, but was the subject of a post on the blog of the primary author of HMMER, Sean Eddy. `http://selab.janelia.org/people/eddys/blog/?p=424`. Accessed 07/05/2012.

4.3 Gene Predictions

Traditional methods of analysis follow a broadly similar pipeline. Firstly, an organism's genome is obtained through DNA sequencing and assembly. Next, gene prediction software is used to identify the genes contained in the genome. These genes are then read and translated into proteins. Finally, these proteins can be subject to homology searches.

Whilst the most accurate gene predictions are obtained by manual analysis, this is inherently a slow process [22]. Automatic methods of gene prediction are hence desired. There are two broad types of gene prediction tools: those that rely solely on DNA sequence and those that incorporate additional sources of information. Genscan is a popular example of the former, which uses HMMs to represent the many different markers of genes, including introns and exons [23]. An example of the latter is GeneWise, which predicts genes using the additional knowledge gained from protein homology searches [24].

In practice, gene predictions are usually obtained using a combination of many different methods. For example, the Ensembl gene prediction pipeline makes use of both Genscan and GeneWise, in addition to a raft of other tools [22].

4.3.1 Protein Homology Without Gene Predictions

One of the two core ideas in this work is to perform protein homology searches by translating DNA directly into protein sequences and skipping the gene prediction phase. To the author's knowledge, this is a novel approach that has not been explored before. Perhaps the closest work to this is in fact contained within gene prediction software, such as the aforementioned GeneWise, which uses protein homology searches of directly translated DNA in order to help define the locations of genes.

A possible reason why this has not been explored previously is the potential problem with pseudogenes. Pseudogenes are genes that have mutated and deteriorated to the point that they are no longer functional. However, because they retain a strong similarity to the original genes, significant matches between the protein sequence within the pseudogene and known proteins in database of HMMs will typically be found. Because the protein sequence forms part of a pseudogene, it is non-functional and so may be seen as a "false" result.

However, gene prediction is a difficult and time consuming process. In fact, the time required to perform gene prediction appears to be increasing. When the details of Ensembl's gene prediction pipeline were first published in 2004, it was expected that the process would take up to four weeks [25]. The Ensembl website now suggests that it typically takes over four *months*, with some genomes taking far longer [3]. Note that this is with the use of a large compute cluster - recent statistics do not appear to be available, but even in 2004, the Ensembl cluster contained over 1000 nodes [26]. Because of this, it could be argued that it would be more efficient to only run gene prediction tools on areas of the genome once it has already been established that there are "interesting" protein homologues within these sections.

[3] See `http://www.ensembl.org/info/docs/genebuild/genome_annotation.html`. The latest gene predictions for the Human genome took much longer: between 43 and 47 weeks - `http://www.ensembl.org/Homo_sapiens/2011_09_human_genebuild.pdf`. Both URLs accessed 15/05/2012.

4.4 Hadoop and MapReduce

Apache Hadoop is a collection of projects designed for distributed, scalable computing and includes the Hadoop Distributed File System (HDFS) and Hadoop MapReduce [27]. These provide an open-source implementation of two key technologies developed by Google: the Google File System (GFS) and MapReduce framework [28, 29].

4.4.1 HDFS

HDFS is a distributed file system, designed for fault-tolerance on commodity hardware. Like GFS, the innovation is not that it is distributed, but rather than specific assumptions and goals are made and designed for[28, 30]:

- Software should be resilient to hardware failure. Systems are typically built from cheap, commodity hardware and it is thus expected that, at any given time, some portion of the system will be unavailable.
- Files are typically large, rather than numerous. Large datasets consisting of files gigabytes or terabytes in size are expected and performance should be optimized for this.
- High bandwidth is more important than low latency.
- Files are typically used in write-once read-many situations - once written, they are rarely modified again.

These assumptions fit well for many large-scale data processing problems [27].

To give a brief overview of the architecture of HDFS, a cluster comprises a single NameNode and multiple DataNodes. These are analogous to GFS' master and chunkservers, respectively. Files in HDFS are stored in *blocks*, the size of which is customizable on a per-file basis with a 64MB default. Each block is stored on one more data nodes, depending on the *replication factor* and this replication plays a large part in the reliability and performance of a Hadoop cluster. The default replication factor is 3, though this is again customizable at the file level. The NameNode is responsible for all meta-data within the distributed file system as well as managing and coordinating the cluster. It maintains the file system's namespace and a mapping of filenames to blocks and their locations.

In order to minimize network traffic and help prevent the NameNode from becoming a bottleneck, the involvement of the NameNode in file operations is kept to a minimum. When reading or writing to a file, a client first queries the NameNode to open the file and determine the location of the file's blocks. From then on, clients interact only with the DataNode, with data being transfer directly between. In the case of a write, the DataNodes independently handle the updates to each replica. This is summarized in Figure 7.

In addition to the NameNode and DataNodes, there may also be either: a Secondary NameNode (now deprecated); one or more Checkpoint Nodes; or a single Backup Node[4]. Each of these helps to manage the NameNode's metadata (with the Backup node additionally being delegated the role of permanent storage of metadata) and, despite somewhat confusing nomenclature, do not provide any redundancy for the NameNode. Because of this, the NameNode is a single point of failure within an HDFS cluster - if the NameNode is unavailable, the entire cluster

[4]For small clusters, these services may actually be provided by the same machine that runs the NameNode.

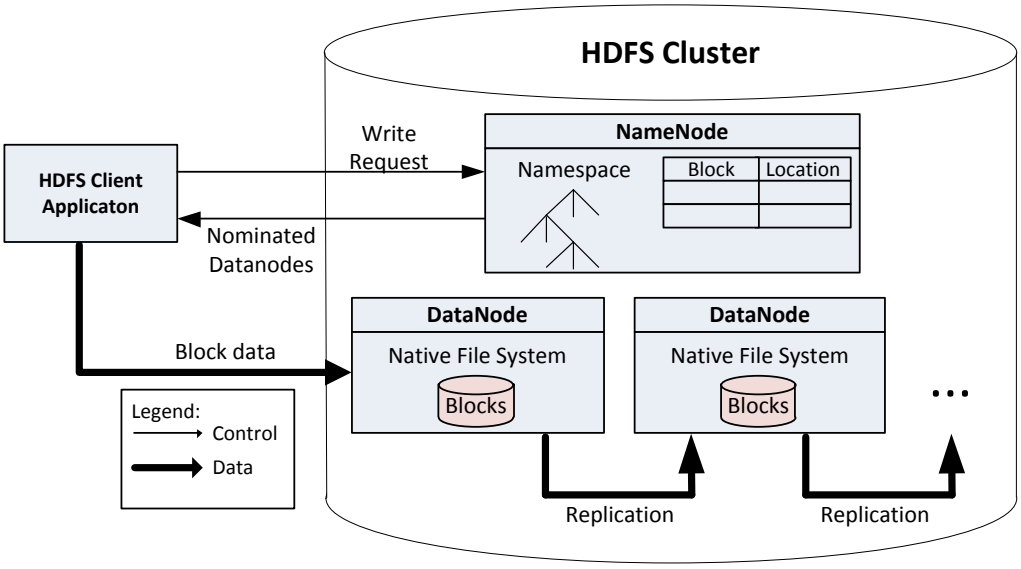


Figure 7: Simplified architecture and data flow diagram for HDFS. Adapted from [28] and [31].

can no longer operate. This limitation would clearly be a problem for a system delivering real-time results but is less of an issue for data-processing tasks. Additionally, the relatively light load that is placed on the master should help to reduce the likelihood of problems [27], a view supported by Facebook who claim to have experienced a single NameNode failure over a four year period with multiple production clusters [32].

4.4.2 MapReduce

MapReduce is a software framework for distributed computing, which attempts to abstract away the vast majority of the details in writing parallel software capable of running on hundreds or thousands of machines. Its core concept is that users provide a map function and a reduce function. A map function is one that takes input, specified as key/value pairs, and outputs zero or more intermediate key/value pairs. A reduce function receives input as a key and *all values* associated with that key, performs some computation on those values and produces zero or more outputs.

The canonical example for MapReduce is that of a program which counts the occurrences of each word in some text. Pseudocode for this is shown in Listing 1. The map function reads input and splits it up into words, before outputting each word as a key with value 1; i.e. each word occurred once. The reduce function receives all intermediate data with the same key and calculates the summation of the values, before outputting the word with the number of occurrences counted.

```
void map(String key, String value){
        for each word w in value{
                emit(w, 1);
        }
}

void reduce(String word, Iterator wordCounts){
        int sum = 0;
        for each count in wordCounts{
                sum = sum + count;
        }
        emit(word, wordCounts);
}
```

Listing 1: Pseudocode for the *Hello, World!* of MapReduce: the word counting program. The map function's input key is unused in this example, whilst its value is (some part of) a text file.

There are of course many more parts to the MapReduce framework. Figure 8 gives an overview of the major components of the Hadoop implementation of MapReduce, which closely follows Google's design.

A brief overview of the functionality of each of the major components of the system is now given.

InputFormat: The InputFormat is responsible for specifying *how* input should be read. This includes deciding if the input is splitable and, if so how to split it. This is a serial process run on one machine only so should typically not read the input. Splitting decisions are usually based on file size, HDFS chunk size and the number of map functions to be run. The InputFormat also specifies the type of RecordReader to use.

InputSplit: The InputFormat creates one or more InputSplits, which are chunks of input that can be processed independently.

RecordReader: Each instance of the RecordReader processes a complete InputSplit. The RecordReader provides a record oriented view by parsing the InputSplit and generating key/value pairs, which are the input to the Map function. An important aspect of RecordReaders is that they must typically cope with an InputSplit that starts and/or ends in the middle of a record, since InputSplits are often just a rough split of the data. This is typically handled by allowing a RecordReader to continue reading the last key/value pair from its split even if it extends past the end of that split. The RecorderReader for the next InputSplit will read and ignore all input until the start of a key is observed.

Mapper: A Mapper processes the key/value pairs provided by a RecordReader by running the map function on them.

Combiner: A Combiner is an optional part of the system and can optimize a MapReduce program by reducing data transfer. It does this by combining intermediate key/value pairs with the same key into a single intermediate pair. As such, it acts as a local Reducer, aggregating data from a single node only. Indeed, for some applications the Combiner and Reducer may be identical, offering an easy way to boost performance. The word counting program in listing 1 is an example where this is the case.

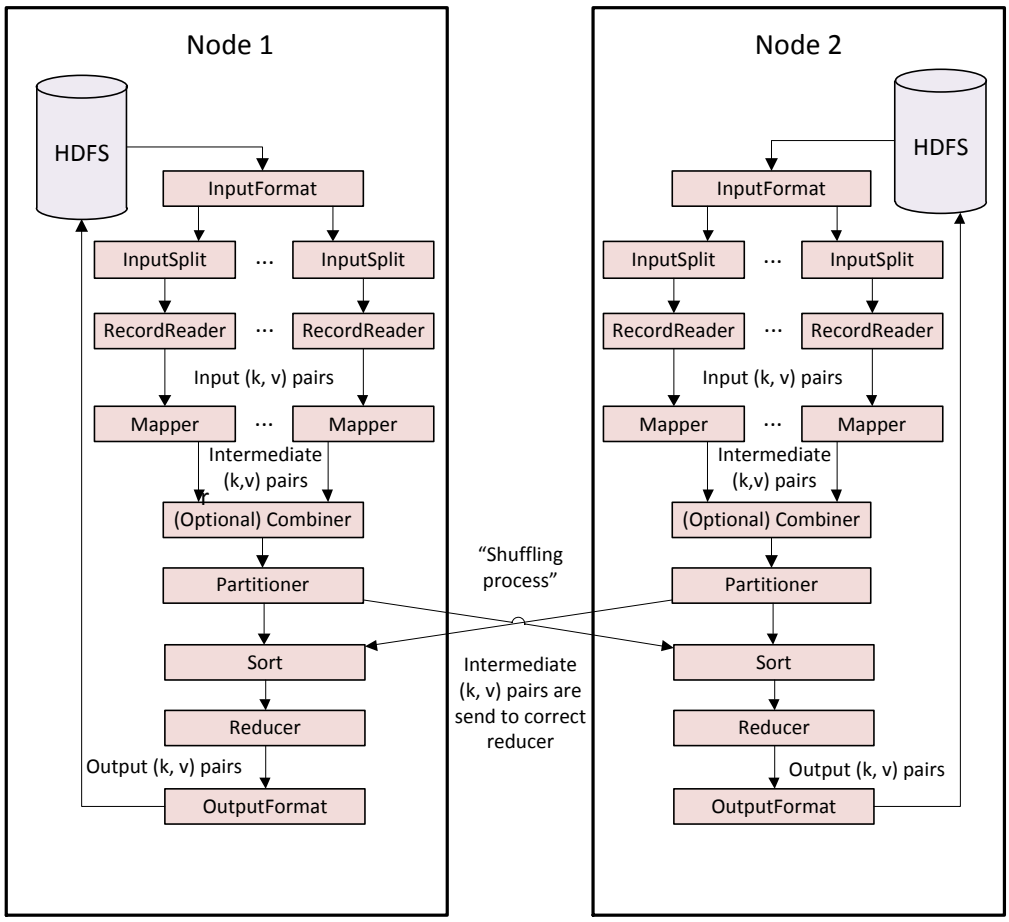


Figure 8: The major components within Hadoop MapReduce. Although only two nodes are shown, there may of course be many more. Adapted from [27] and [30].

Partitioner: The partitioner determines how data is exchanged so that all values for a given key arrive at the same Reducer. The default approach is to take the hash of the key modulo the number of reducers. This identifies the Reducer to send the data to. An even work load is only created if each key has a similar number of values [27].

Sort: Before the Reducer runs, the data it has received is sorted by the intermediate key, allowing received values to be grouped by key ready for the reduction. It is interesting to note that sorting is an expensive way of achieving this. The original Google whitepaper suggested that a sort was useful because it allowed the output file to be efficiently randomly accessed by key and was potentially more useful for end users [29]. Recently, there has been some movement towards grouping by hashing, for example in Google's Tenzing [33], which implements SQL in MapReduce. There has also been discussion of this in Hadoop's online issue-tracker[5].

Reducer: The Reducer receives all values for a given key and applies the reduce function, producing zero or more outputs.

[5]E.g.: `https://issues.apache.org/jira/browse/MAPREDUCE-3397` and `https://issues.apache.org/jira/browse/MAPREDUCE-1639`

OutputFormat: Analogous to the InputFormat, the OutputFormat specifies how the final data is written back, typically into HDFS. For example, the default OutputFormat writes each key/value pair on a separate line with a tab separating the key and value.

Like HDFS, the MapReduce framework runs with one master and many slaves. The master, or JobTracker, is responsible for scheduling MapReduce programs, known as jobs, and launching and monitoring tasks. A task may include a particular map or reduce or the setup of a given job. For small clusters, the JobTracker usually runs on the same node as the HDFS NameNode; for larger clusters they will typically run on separate machines [27]. The slaves are known as TaskTrackers and a TaskTracker should run on every DataNode of the HDFS cluster.

It is important that a TaskTracker runs on every DataNode for performance reasons. One of the assumptions in the design of Hadoop is that "*Moving Computation is Cheaper than Moving Data*"[34]. Because of this, Hadoop will attempt to improve data locality by scheduling tasks "near" to the data they require, ideally on the same node. For large clusters, the Hadoop system is *rack-aware*, helping to keep data transfer within the same rack where possible.

Errors and failures within a MapReduce program are handled by restarting tasks. The JobTracker communicates periodically with each TaskTracker to assess its status. If the TaskTracker does not respond or the running task has failed, the task will be rescheduled potentially on a different node. Since a given task forms only a small part of the whole program and individual tasks are isolated and independent from each other, only a portion of the whole workload must be recomputed. This policy allows both software and hardware errors to be handled and is central to the ability of MapReduce to run extremely large and long jobs in a fault tolerant manner.

Furthermore, this independence between tasks allows for a crucial performance improvement given by *speculative execution* [27]. Both errors and load imbalances can cause a given task to execute much more slowly on one node than the same task would on another. Because of this, the framework will automatically reschedule any remaining tasks on other TaskTrackers as the entire program nears completion. The results of that task will be taken from whichever TaskTracker finishes first. Google reported that this feature could improve performance by as much as 44% in their implementation of MapReduce [29], although the overall efficiency of this method has been questioned [35]. Within this work, gains in performance due to speculative execution were noted even on a small 10 node cluster, with load imbalances causing some tasks to run 3-4x slower on one node than another. These load imbalances are to be expected when running on cloud-based servers as resources are shared with other users.

Finally, another important feature is Hadoop's *DistributedCache*. The DistributedCache is a mechanism for providing files to all of the tasks within a given job [36]. At the beginning of a job, any files that are requested to be placed in the DistributedCache are copied to the local storage on all TaskTrackers. To reduce data transfer, files are kept in the DistributedCache between jobs and only copied again if the source file has been modified or the file was evicted from a nodes cache to make room for other files. This was used within this work both for distributing large data files and a shared library that were required by all map tasks.

4.5 Related Work

4.5.1 Parallelism and Distribution of HMMER

There have been a number of attempts to improve the performance of HMMER through parallelism, with both specialized hardware and software being investigated [12]. The implementation of HMMER's dynamic programming algorithms, outlined in section 4.1.2, on graphics processing units (GPUs) has frequently been explored. Walters et al. demonstrated a 12-38x performance increase on one of the core algorithms against an older version of HMMER [37]. More recently, a 10-15x speedup of the same algorithm against HMMER3 was demonstrated [38]. Field Programmable Gate Arrays (FPGAs) have also been targeted, with a 10x speedup of the whole HMMER3 program being reported [39].

Of more relevance here is the previous work which attempts to distribute HMMER across multiple nodes. The current HMMER3 release includes a "rudimentary implementation" of a distributed version of HMMER, using MPI (Message Passing Interface), which is said to scale poorly [12]. Improvements to MPI parallelism continues to be listed on HMMER3's "to-do list"[6]. By examining HMMER's source code, it seems reasonably apparent that there is frequent communication between MPI nodes, which is likely to contribute to the less-than-ideal scaling. It is also worth noting that the MPI implementation requires significant alternation to the HMMER code itself.

MPI-HMMER attempted to hide communication overhead by overlapping computation and communication in an MPI implementation of the previous version of HMMER [40]. This technique, along with bundling data to produce fewer but larger messages, achieved reasonable scaling on a modest number of nodes with 81% of the theoretical maximum speedup being observed when computation was spread amongst 16 machines.

SledgeHMMER appears to be the first attempt to distribute HMMER's work load to a larger number of nodes and reported linear speedups with up to 128 processors [41]. Although this work targeted the previous version of HMMER and the web server running SledgeHMMER is no longer responsive, some of the results are relevant to this work. SledgeHMMER used a file-locking approach to parallelism, rather than MPI, claiming that this offered better load balancing, reduced software requirements and the flexibility for workers to join or leave during the computation. The authors noted an advantage in splitting up the set of protein sequences rather than HMMs in order to distribute workload, which is the approach taken here as discussed in section 6.5.1. They also noted that dividing M sequences amongst N processors would result in a load-imbalance due to the varying sizes of the sequences. Borrowing from general loop optimization techniques [42], a "strip-mine" approach was used where the the number of sequences is first broken up into a number of "chunks", with these chunks being split equally amongst the N processors. In this work, the protein sequences were split into more than M/N pieces, with each mapper then processing a number of these to provide better load balancing.

A more recent MPI-implementation, HSP-HMMER, (although still targeting the previous version of HMMER) takes a similar approach to SledgeHmmer by dividing the protein sequences into chunks that can then be processed independently, reducing the need for communication[43]. Near linear speedups were demonstrated using their approach with up to 4096 processors. This demonstrates the potential for highly-scalable, distributed implementations of HMMER.

Despite now being released for more than two years, there appears to have been no work

[6]This is currently located on the blog of the primary author of HMMER, Sean Eddy. `http://selab.janelia.org/people/eddys/blog/?page_id=408`. Accessed 05/03/2012

attempting to distribute the latest version of HMMER other than the previously mentioned MPI implementation contained in the official release.

4.5.2 Cloud Computing in Other Areas of Bioinformatics

There do not appear to be any previous attempts to implement HMMER using the Hadoop framework, nor for HMMER's closest relative, SAM. However, the use of both Hadoop and cloud computing in general within the wider area of bioinformatics is increasing. Taylor gives an overview of these developments in [44].

Perhaps the most relevant to this work are the developments allowing BLAST to run on top of Hadoop [45, 46], with the latter demonstrating both good scaling up to 64 nodes and better performance than a leading MPI implementation. Other bioinformatics tools based on Hadoop or other cloud computing technologies include:

- Contrail, a DNA assembler running on Hadoop [44]
- CloudBurst, a tool for aligning raw DNA sequencing data to a reference genome [47]
- Crossbow, a tool for detecting variations in a single nucleotide (SNPs) [48]

In addition, there are a number of new companies specialising in cloud-based DNA analysis, including EasyGenomics and DNAnexus [7]. All of these developments suggest there is an increasing desire for algorithms and tools that support bioinformatics in cloud environments.

[7] EasyGenomics: `https://www.easygenomics.com`; DNAnexus: `https://dnanexus.com/`. Both accessed 14/05/2012

5 Overview of Software and Datasets

Software

The following pieces of third-party software were used in this work:

- HMMER3 was used to perform all protein homology searches. HMMER3 website: `http://hmmer.janelia.org/`
- Cloudera's distribution of Apache Hadoop was used to parallelize workload over a number of compute nodes. The Hadoop source code also proved useful for some of its less well documented aspects. Hadoop website: `https://hadoop.apache.org/`. Cloudera website: `http://www.cloudera.com/`
- An unpublished Perl script written by Julian Gough was used as a reference implementation for six-frame translation.

Datasets

Datasets were obtained from the following locations:

- SUPERFAMILY. The SUPERFAMILY database of Hidden Markov Models representing protein domains was used for protein homologue searches. Additionally, pseudogene data for the human genome was used, with additional information on the pseudogenes (their locations within the genome) obtained directly from SUPERFAMILY's authors. SUPERFAMILY website: `http://supfam.cs.bris.ac.uk/SUPERFAMILY/`
- Ensembl. DNA and protein sequences were obtained from Ensembl. In particular, release 67 of the Gorilla genome and protein sequences were used during development and release 58 of the Human genome and protein sequences were used for testing. Ensembl website: `http://www.ensembl.org`.

6 Implementation and Results

This project has two main aims. The first is to develop a method for running HMMER on top of the Hadoop framework to distribute its workload. The second aim is to search DNA data directly against protein HMMs without using gene prediction. This section discusses the design, implementation and results of software for both of these tasks.

The structure of this section is as follows. Section 6.1 gives an overview of the method to perform protein homology without gene prediction. Details on JavaHmmer, a wrapper for HMMER, are then given in section 6.2. Section 6.3 shows the results of benchmarks performed on HMMER, which are important for understanding later results. An explanation of the Hadoop cluster used for this work is given in section 6.4. Finally, section 6.5 introduces HadoopHmmer, which allows HMMER to be distributed with MapReduce and implements the method of searching against protein HMMs without gene prediction.

6.1 Method For Protein Homology Without Gene Predictions

As discussed in 4.3, a typical approach to analysing DNA data is to sequence and assemble DNA; use gene predictions to determine which sections of DNA code for proteins; translate those piece of DNA into proteins; and finally search for homologues of the proteins in a database.

Within this work, the assembled DNA is used directly. Because of this, protein sequences will be split by non-coding regions of DNA. When these protein sequences are compared against an HMM library, there may be many partial hits to one HMM, each of which are separated by these non-coding regions. The core idea of this work is to find these partial hits and merge them into a single, more complete match against the HMM. Partial hits may not be significant on their own, but when combined with other hits they become significant. A similar result to that obtained using traditional methods should then be given.

An illustration of the difference between a typical approach and the method used in this work is shown in Figure 9.

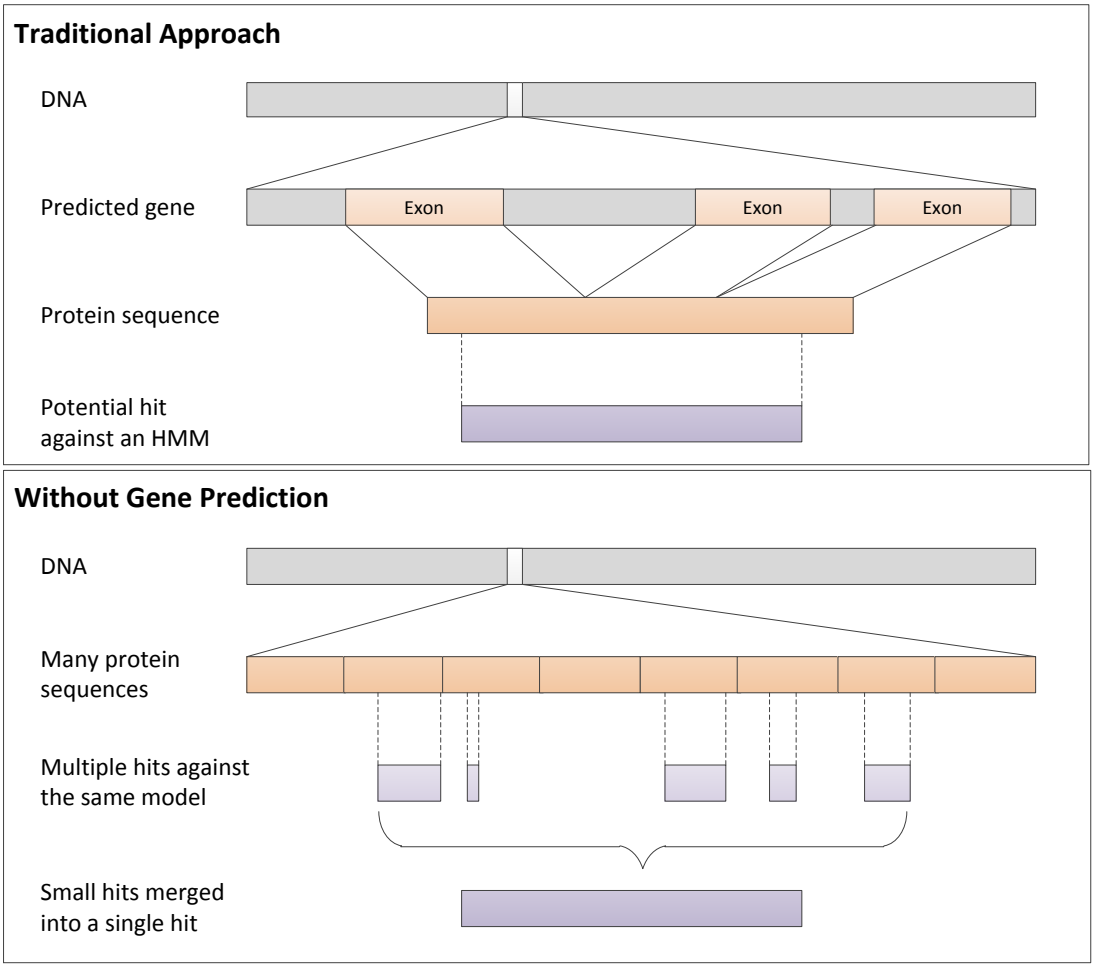


Figure 9: A comparison of a traditional approach to protein homology and the method used in this project. A traditional approach is to run gene prediction software on an organism's genome, which determine the sections of DNA that encode proteins. Homologues of these proteins can be then be searched for, which might generate a hit to a particular model. The method used in this work skips the gene prediction phase. Because of this, there are instead many partial hits to the model, separated by non-coding regions of DNA. These must then be merged together.

Full details of how this merging process is implemented are outlined in section 6.5.3.

6.2 JavaHmmer

Although Hadoop includes utilities to write MapReduce applications in any language[8], they do have some limitations; full access to all the feature of Hadoop is only available in Java programs. For this reason, all MapReduce programs in this work are written in Java.

[8]This language agnostic approach is achieved with Hadoop Streaming, which gives the input to mapper and reducer programs on standard input. The programs then write out intermediate or final key/value pairs on standard output. In addition, Hadoop Pipes provides a socked-based C++ interface to Hadoop.

Since HMMER is written in C, a Java interface was needed. This is known as JavaHmmer and the rest of this section discusses its implementation and evaluation.

6.2.1 Implementation

JavaHmmer provides a Java interface to HMMER using the Java Native Interface (JNI). JNI provides the ability for Java programs to both call, and be called by, code written in native languages including C and C++. One technique for interaction between Java and native programs is the use of *native methods*. Within a Java program, methods may be declared with the `native` modifier, which means that they will be implemented within a dynamically-linked native library (shared object) rather than Java code.

JNI provides a mapping between Java and C variable types. For example, a Java `int` is a C `jint`, whist something more complex such as a Java array of objects, such as strings, is a `jobjectArray` within C. JNI provides functions for native code to operate on a `jobjectArray` type, such as accessing an element, counting the number of elements in the array or converting between Java strings and native strings (i.e. a `char*`). For full details, see the JNI Programmer's Guide and Specification [49].

JavaHmmer uses native methods and there are thus two components of JavaHmmer: a shared object and a Java application. The shared object exposes two programs from HMMER, hmmscan and hmmsearch, which can then be run by a Java application that links with the object. See section 4.2.3 for a description of the differences between hmmscan and hmmsearch. To create the shared object, a number of modifications were made to the original HMMER source code:

- New methods, JNIHmmScan and JNIHmmSearch were added to hmmscan and hmmsearch respectively. These are the methods called by the Java program. Each of these functions accepts a `jobjectArray` as an argument, which is treated as an array of strings. These strings are the arguments to hmmscan or hmmsearch, which are normally passed on the command line. JNIHmmScan and JNIHmmSearch convert the `jobjectArray` to a `char**` before calling the original main function with the `char**` supplied as the command line arguments.
- In both hmmscan and hmmsearch, the core of the program compares one protein sequence against one HMM. After this comparison, the results are normally written out either to standard output or a file. This process is modified to pass the results back to the Java program that called HMMER. The Java program contains a class, `HmmerResult`, which acts as a simple container for all of scores, alignment information and statistics generated for each match. The C code creates a new `HmmerResult` object and populates its fields, using functions provided by JNI.
- Because results are passed back to Java, all other output is suppressed.

The Java program allows JavaHmmer to be run either from the command line or from other Java applications. Listing 2 shows the output from JavaHmmer when run from the command line with no arguments, which shows the parameters that can be passed to the program.

When JavaHmmer is run, it calls the appropriate HMMER program and passes the arguments `hmmfile`, `seqfile` and any additional arguments given after the `--hmmmer-options` flag[9]. This

[9]These can of course be set programmatically by other Java programs calling JavaHmmer too.

```
Usage: java JavaHmmer <hmmsearch or hmmscan> <hmmfile> <seqfile> [options]

Options:
        --out <file>: Print output to file rather than stdout
        --hmmmer-options: All further options passed directly to hmmer.
```

Listing 2: Output from JavaHmmer if run from the command line without any options

approach allows JavaHmmer to be compatible with all of the original arguments that HMMER accepts, without having to write any additional code for each of them. Whenever the C code passes back a result, one of two things happens. Either the results are written out (either to stdout or a file, depending on the options set) or they are passed to any registered listeners: other objects may receive results from HMMER by implementing the `HmmerResultListener` interface and registering themselves as a listener.

Finally, if results are printed out, they are printed using the same tabular format as the original HMMER code. This is important for two reasons: first, it provides a familiar format to users of the program; and second, it allows the results of JavaHmmer to be compared against those from the original program more easily.

6.2.2 Performance

Since JavaHmmer will be used by HadoopHmmer to run HMMER within the MapReduce framework, it is important that JavaHmmer doesn't add a significant overhead, as this would have an impact on the overall performance of the MapReduce program.

Program	Native C Runtime (seconds)	JavaHmmer Runtime (seconds)	JNI Performance Cost
Hmmscan	1419	1437	1.3%
Hmmsearch	4081	4188	2.6%

Table 1: A comparison of the run-time of JavaHmmer and the original HMMER programs, both running in serial environment. The times shown are the average of three trials. For both tests, the full SUPERFAMILY HMM database was used (15438 HMMs). The hmmsearch test used protein sequences from Ensembl's release 67 of the gorilla genome (50831 sequences). The hmmscan test used one 890 amino acid protein sequence duplicated 1250 times.

Table 1 shows how the performance of JavaHmmer compares with the original native C program for two different tests. The overall cost of running HMMER via JNI is relatively low, only increasing the run-time by a few percent.

6.2.3 Testing

Once again, with HadoopHmmer relying upon it, it was important to verify that JavaHmmer was functioning correctly and producing the same results at the unmodified HMMER programs. To facilitate this, a simple 'diff' tool was produced. Standard unix tools such as the diff utility were unfortunately unsuitable. Whilst C and Java's implementations of formatted output (e.g. `printf()` and `java.util.Formatter`, respectively) are mostly compatible, there are some

subtle differences that result in extra trailing zeros in Java that are not present in C[10]. This means that text-based diff tools will report that there is a difference. The diff tool produced overcomes this problem simply by comparing any numeric values as numbers.

Using this tool, the output of JavaHmmer was compared with the output of the unmodified HMMER code for a variety of sample inputs using both the hmmscan and hmmsearch programs. After the correction of some minor bugs, no differences were observed. It should be noted that identical results are only possible if HMMER uses the same seed for the random number generated each time it is run, which it does by default. This is because the internal HMM calculations model a complex probabilistic process.

6.2.4 Discussion

Although reasonably simple, JavaHmmer forms a critical part of the wider implementation and it was imperative to ensure that it functions and performs well. One of the key advantages of JavaHmmer is that it requires relatively few changes to the HMMER source code. The modifications essentially "patch" the entry and exit of HMMER, with its mature, tested internal pipeline left untouched. These small changes, which can be tested in isolation, open up HMMER to be run by a Java application on the MapReduce framework. In contrast, HMMER's MPI implementation contains significant changes to original code, making it harder to maintain and test.

Finally, although JavaHmmer already performs reasonably well, there are some JNI optimizations that could be explored. Before native code can call a Java method, it must look up its ID. This lookup is relatively expensive and it is suggested that the method ID should be cached if it is used frequently [49]. JavaHmmer does not currently do this. Additionally, since calling a Java method from native code has some overhead, it may be more efficient to return all of the separate scores, statistics and alignment information as arguments to one function, rather than returning each of them individually.

6.3 hmmscan vs hmmsearch

As discussed in section 4.2.3, HMMER provides two programs for comparing sequences to profile HMMs: hmmscan, which is efficient for lots of models and few sequences; and hmmsearch, which is efficient for lots of sequences and few models. When there are both a large number of models and a large number of sequences, hmmsearch is generally expected to be be faster but the exact trade-offs between the programs are unclear.

As section 6.5.1 explains, it is generally preferable for HadoopHmmer to divide the computation amongst nodes by splitting up the sequences, rather than the models. This suggests that hmmscan may be more appropriate, unless there are a large number of sequences distributed to each node. The performance trade-offs of unmodified hmmscan and hmmsearch programs was thus considered. Figure 10 shows how the run-time *per sequence* varies with the number of sequences, whist the number of models is fixed at $15,438$ (the full SUPERFAMILY library).

[10] An example of this occurs with the 'g' conversion character, which formats floating point numbers either as a standard decimal or in scientific notation depending on which representation is shorter. In C, trailing zeros are not printed when the scientific notation is used, whist Java does print them.

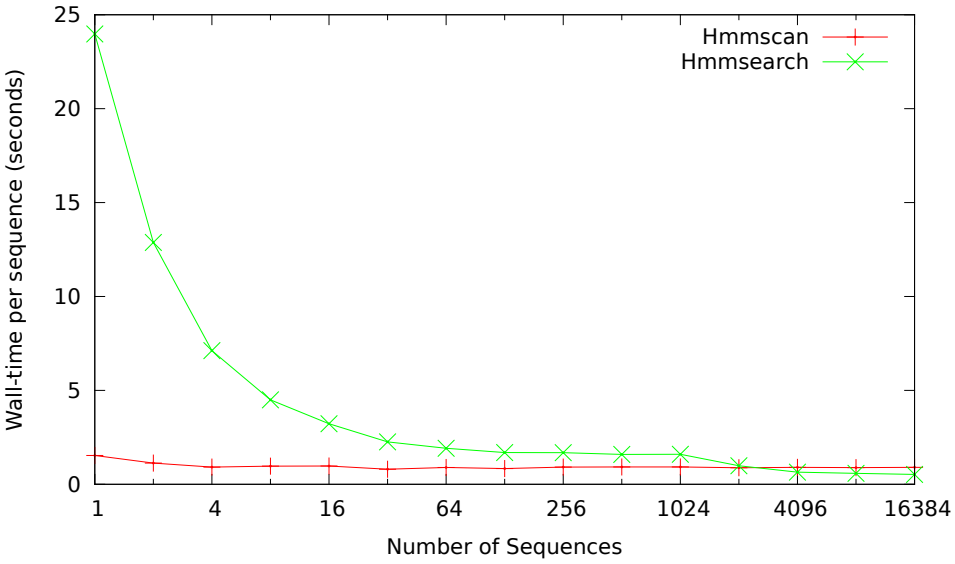


Figure 10: A graph of the run time per sequence as the number of sequences varies. The number of models was fixed, using all 15,438 HMMs in the SUPERFAMILY library. To generate the sequences, one protein containing 890 amino acids was duplicated. All times were obtained by averaging three trials.

As the graph shows, when using the full model library, hmmscan outperforms hmmsearch until there are around 2,000 - 4,000 sequences. This cross-over was found to be consistent on quite different platforms. Figure 10 was generated using data obtained on a cloud computing instance with 4 virtual cores; the same cross-over point was observed when running with 2 virtual cores and on an 8 core node of the University's supercomputer, BlueCrystal.

However, the cross-over point was found to be dependant on the length of each sequence. For hmmsearch, the database of sequences is re-read for each HMM so a change in the length of the sequences should have a big effect. For hmmscan on the other hand, the database is the profile HMMs so a change in sequence lengths will only effect the time needed to compare model and sequence. This is shown in Figure 11, which shows the results for two different-sized sequences, with the range of the axes set so the area of interest is more visible.

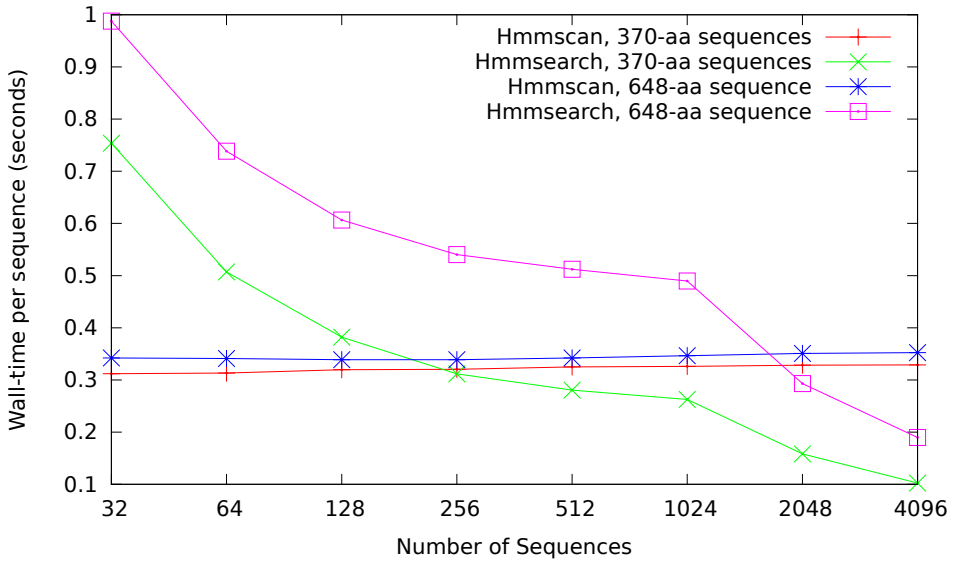


Figure 11: The same experiment as in Figure 10, but with two smaller sequences. The graph is zoomed to the area of interest.

Of course, there are additional metrics besides time to consider. Figure 12 shows the memory usage for both programs, using the same sequence data as in Figure 10. As can be seen, hmmscan has a much larger memory footprint. This to be expected as profile HMMs are much larger in size than protein sequences. The SUPERFAMILY HMM library contains 15,438 models, each representing an average of 175 amino acids, and is 489MB in size when stored in HMMER's binary format[11]. In contrast, the 16,384 copies of a protein sequence containing 890 amino acids used in Figures 10 and 12 is less than 17MBs in size, despite being stored in plain text.

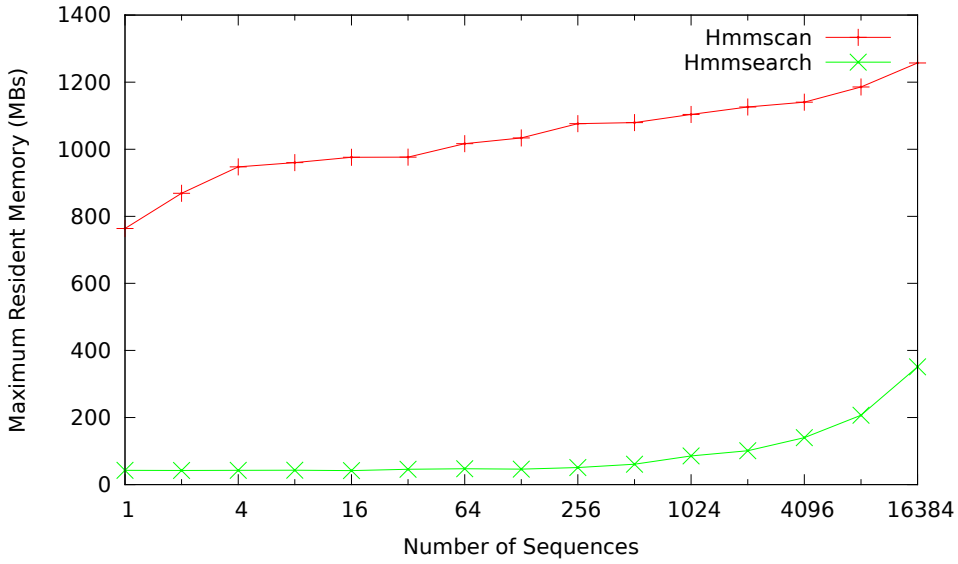


Figure 12: A graph of the maximum resident memory of hmmsearch and hmmscan as reported by the unix time utility. The same experimental data as in Figure 10 was used.

[11]HMMER also uses pre-computed data structures and indexes for models when using hmmscan. Although not exactly clear what is read into memory when, these also total over 600MBs for the SUPERFAMILY database.

Finally, the opposite case of a large, fixed number of sequences and varying numbers of profile HMMs was also tested. Although these results are of less direct interest to this work as HadoopHmmer does not currently divide up the HMM database, they are included in Appendix A for completeness.

6.4 Hadoop Cluster

After a successful application to a free-of-charge private beta test, the Hadoop cluster used in this project was setup on HP's new cloud computing service.

Under the terms of the beta test, the cluster was limited to a set of instances with a total of 20GB of RAM. The *Standard Small* instances were chosen for the Hadoop cluster, each of which provides 2GBs of RAM and 2 virtual cores [12]; therefore, the cluster was limited to 10 nodes[13].

Because the cluster was small, it was reasonable to allocate just one instance for the combined roles of NameNode, SecondaryNameNode and JobTracker. This left the other 9 instances available for use as DataNodes and TaskTrackers.

The cluster used Cloudera's distribution of Hadoop (version 3), primarily because it is easier to install than vanilla Hadoop and it had worked well running in *pseudo-distributed* mode, where all Hadoop services are provided on one machine, during development. Cloudera's documentation and guidance for the configuration of a Hadoop cluster also proved very valuable[14]. In general, the cluster used a standard configuration of Hadoop, though the maximum number of simultaneous map tasks per node was reduced to one. This was changed because HMMER is multi-threaded and able to fully utilize the node's resources; it was anticipated that running more than one instance of HMMER would reduce performance.

In the following sections, reference is made to the number of *compute nodes*. This should be taken to mean the number of DataNodes/TaskTrackers in the system as these are the nodes which perform the actual computation. The cluster will always contain the additional master node.

Evaluation of the Cluster

Performance of nodes in the cluster was found to be quite variable, with the same task often taking much longer on one node than another. This was probably due to the shared resources of the cloud environment causing variation in load. There were no reliability problems experienced with the Hadoop cluster, which ran for over a month (though not in active use for all of that time) without problems.

Finally, a current limitation of the HP Cloud is that there is no ability to use custom images when creating a virtual machine. This means that software has be downloaded, installed and configured each time a new node is added to a Hadoop cluster. Although this was worked-around by creating a collection of scripts to install the software and distribute the updated

[12] For details of the instance types available, see `https://www.hpcloud.com/products/cloud-compute`. Accessed 14/05/2012.

[13] As of 10/05/2012, HP Cloud Services has entered a phase of public beta testing and are currently charging for their services with 50% discount. At this point the limits on instance numbers were raised and a 20 node cluster was briefly tested. See section 6.5.6

[14] e.g. `https://ccp.cloudera.com/display/CDHDOC/CDH3+Deployment+on+a+Cluster`

configuration each time a new node was added, the whole process could be made easier if HP Cloud added support for this.

6.4.1 Hadoop On Demand

At the beginning of the project, Hadoop clusters running on top of the University's supercomputer, BlueCrystal, were used. This required the use of Hadoop On Demand (HOD). HOD works by submitting a job to the underlying queue system on the supercomputer. Once the job starts, HOD provisions a temporary Hadoop cluster using the resources it was allocated. A large amount of time was invested in finding out how to run HOD on BlueCrystal[15], including time spent working with one of the HPC System Administrators to debug problems on phase 2, partly due to a lack of documentation. To help future uses of HOD on BlueCrystal, some simple documentation was written and provided to the HPC support team.

Although HOD on BlueCrystal appeared to work well enough during development, when it came to running full tests and benchmarks the whole system was found to be quite unstable. This was particularly apparent on phase 2. The timely availability of the HP Cloud private beta meant that these problems were never fully diagnosed. It did, however, become obvious that HOD was quite sensitive to the versions of software installed. Instability also appeared to occur more frequently when BlueCrystal seemed to be under heavy load.

Additional problems with HOD

Aside from instability, usage of HOD was also quite cumbersome. Because the Hadoop cluster only exists for the duration of the job on the supercomputer, it is necessary to recreate the files and directories in HDFS each time a new HOD job is started and all any output files must be copied back from HDFS before the HOD job ends, otherwise they are lost. Furthermore, because HOD is treated as a normal job by the supercomputer's queuing system, it is necessary to wait for the HOD job to start, before any Hadoop program can be launched. This either requires additional scripts to be written or to be available to setup and launch a Hadoop program when the job starts. Finally, because HOD communicates with the supercomputer's task scheduler itself, it is not possible to specify certain options. These range from the slightly annoying, such as the inability to be emailed when the job starts, to the more critical, such as the inability to request all of the cores on one node. This problem was submitted to the Apache Hadoop issue tracker [16], though it has not generated any responses.

6.5 HadoopHmmer

HadoopHmmer provides the ability to run HMMER on the Hadoop framework. This is an extension to the way HMMER can be used, allowing it to be easily distributed across a large number of nodes. Results obtained from using HadoopHmmer in this way should be identical to those produced by the original HMMER programs.

Additionally, HadoopHmmer allows DNA data to be searched against a protein database without using gene predictions. This is a novel approach to protein homology. It is important to

[15] The time spent increased with the untimely shutdown of phase 1 of BlueCrystal part way through the project - HOD was originally only available on phase 1, not phase 2.

[16] `https://issues.apache.org/jira/browse/HADOOP-8081`. Issued submitted on the 16/02/2012/. URL last accessed 15/05/2012.

realise that this idea is in fact independent of Hadoop - it could be implemented in a serial environment. However, it is also well suited to Hadoop. A large number of partial hits are expected to be generated, which will then need merging based on which model the hit was against. Intuitively, the MapReduce paradigm is a good fit to this problem.

The rest of this section discusses the implementation and results of both of these features.

6.5.1 Splitting up the Data

In order to distribute the computation, the workload needs to be split. Since each profile HMM can be compared to each sequence independently, it is possible to split up the database of HMMs amongst the compute nodes, or to divide the sequences between nodes or both. However, because the database of HMMs is unlikely to be changed frequently, it is preferable to allocate the full database to each node once, where it can remain between Hadoop jobs without any further data transfer. The sequences on the other hand are likely to be different each time HadoopHmmer is used, so these will have to be re-distributed to the nodes each time HadoopHmmer is run regardless. Because of this, HadoopHmmer splits the workload by sequence only, distributing a portion of the sequences to each compute node. Each node then searches the sequences it received against the full HMM database.

This approach is supported by Hadoop's Distributed Cache (see page 25). Before launching the MapReduce tasks, HadoopHmmer places a zipped copy of the SUPERFAMILY HMM database in the Distributed Cache. This is copied to each node when the job is started and placed in the node's local filesystem, before being automatically unzipped by the Hadoop framework. It will remain in the node's cache until either: the cluster is restarted; the original zip file is modified; or the file is deleted to make space for others placed in the cache.

6.5.2 Six-Frame Translation

As explained in section 3, DNA contains two strands. Further, because codons are triplets of DNA bases, different codons, and hence amino acids, can be obtained from a sequence of DNA if reading starts at an offset of $0, 1$ or 2. This gives six different *reading frames*, which means that there are six different translations of a DNA sequence to a protein sequence. An illustrative example of six-frame translation is given in Appendix B. Additionally, we wish to divide up the protein sequence into chunks so that the sequence data, and hence workload, can be split amongst nodes.

Biology offers a suggestion of where to split sequences with *stop codons*. Stop codons mark the end of a coding region of DNA. However, it may not be ideal to split the protein sequence at every stop codon for two reasons. Firstly, very short sequences are unlikely to generate a reliable match to a profile HMM. Secondly, both mutations and errors in sequencing or assembly can cause stop codons to occur within a coding region; in general, not splitting at stop codons is a more conservative process as it adds less extra knowledge and places more emphasis on the statistical models. To provide a balance, within this work sequences are split at stop codons, but only once they reach sufficient length. This was chosen to be 500 amino acids; a fairy typical protein length. Additionally, if no stop codon has been found when the translated sequence reaches 1000 amino acids in length, the sequence is split anyway. A thorough investigation into the effect of varying these parameters remains an open problem, although it is briefly considered on page 46.

Currently, HadoopHmmer provides a utility to perform six-frame translation on DNA files as they are loaded into HDFS. This means that the translation is not distributed, with all computation occurring on the node that started the copy into HDFS. However, translation is a reasonably light-weight process and it could easily be parallelized by loading different files into HDFS from different nodes at the same time if this became necessary.

An alternative approach would be to store DNA data in HDFS and translate it in the RecordReader before it is presented to the rest of the MapReduce application. This has some difficulties due to InputSplits, however. As discussed on page 23, data is split without considering its format, so RecordReaders must handle records that straddle two InputSplits. Typically, a RecordReader reads past its InputSplit to read a complete record, whilst it skips the start of its InputSplit until the start of a new record. The input values are the protein sequences, which we want to split at stop codons, but only if the sequence is of a particular length. This means knowledge of what has come before in the entire sequence is needed to determine where the split should be. Simply splitting at the first stop codon after each InputSplit would be a reasonable approximation, with a few shorter sequences unlikely to have a big effect on results. This would be complicated slightly by the fact that each reading frame has stop codons at different locations. Overall, the current method is simple and effective for current needs. It also has the advantage that it can be easily changed without the need to modify any of the MapReduce code.

The translation performed by HadoopHmmer produces output with the six frames interleaved. As with the example at the end of Appendix B, many tools output all of the protein sequences in a reading frame sequentially (assuming the protein sequences are split in some way). Instead, HadoopHmmer will output protein sequences from one reading frame interleaved with the protein sequences from all other reading frames. An example is of this is given in Listing 3.

```
1st Sequence for forward frame 0
2nd Sequence for forward frame 0
1st Sequence for forward frame 1
2nd Sequence for forward frame 1
1st Sequence for forward frame 2
2nd Sequence for forward frame 2
1st Sequence for reverse frame 0
2nd Sequence for reverse frame 0
1st Sequence for reverse frame 1
2nd Sequence for reverse frame 1
1st Sequence for reverse frame 2
2nd Sequence for reverse frame 2
```

```
1st Sequence for forward frame 0
2nd Sequence for reverse frame 0
1st Sequence for forward frame 1
2nd Sequence for reverse frame 1
1st Sequence for forward frame 2
2nd Sequence for reverse frame 2
2nd Sequence for forward frame 0
1st Sequence for reverse frame 0
2nd Sequence for forward frame 1
1st Sequence for reverse frame 1
2nd Sequence for forward frame 2
1st Sequence for reverse frame 2
```

Listing 3: On the left, the order of sequences for a typical six-frame translation. On the right, a potential ordering produced by HadoopHmmer; the actual order depends on whichever frame reads a stop codon first. Note the second sequence of a reverse frame occurring before the first sequence. Each reading frame is shown in a different colour for clarity.

This has a big advantage as it allows all six frames to be produced in one pass of the DNA file, without high memory costs. Only the current split of protein sequence for each frame needs to be stored; when the split ends it is written out. Aside from the interleaving, which may

be less intuitive for a human reader of the file, there is another disadvantage to this approach. Because the DNA file is read in a forward direction only, protein sequence for reverse frames are output with chunks at end the of the reversed DNA sequence occurring first, as shown in Listing 3. Additionally, it is only possible to locate the reversed protein sequences and determine their frame offset relative to the end of the reversed DNA sequence, rather than the beginning. However, these are mostly internal concerns to HadoopHmmer and could easily be resolved during processing because the length of the sequence would then be known. HadoopHmmer does not currently handle the processing of matches to sequences in the reverse frames, though only minor changes would be needed to do so in the future.

Finally, a potential future optimization would be to write the translated data into HDFS in a Hadoop *SequenceFile*, which is a compressible, binary format. This would reduce the size of the files, resulting in less network traffic, and allow the use of the SequenceFile RecordReader and InputFormat that are part of the Hadoop API.

6.5.3 MapReduce Implementation

InputFormat and RecordReader

HadoopHmmer uses a custom InputFormat and RecordReader. The InputFormat specifies the RecordReader and defines the input to be splittable. The RecordReader reads records and converts them into key/value pairs. It also handles the case where an InputSplit starts in the middle of a record. Each record spans two lines of an input file. The first line contains a key, which is a sequence identifier. The second line contains a value, which is a protein sequence. These key/value pairs are written into HDFS in this format by the six-frame translator, discussed in section 6.5.2. The sequence identifier contains useful information, including the location of the sequence within the genome and which reading frame generated the sequence.

Mapper

The map function runs HMMER via JavaHmmer, using hmmscan with default parameters, on each sequence it receives as an input value. To do this, the key and value (i.e. the sequence identifier and sequence) are written to a file on the local file system. JavaHmmer is then run with this sequence file and the profile HMM database stored in the node's cache.

However, key/value pairs are actually processed in batches: each key/value pair is written to the file but JavaHmmer is only run after a number of inputs have been written. Currently, HadoopHmmer waits until $25,000$ amino acids of sequence data have been written, which means between 25 and 50 sequences based on the size of the sequences created by the six-frame translator. This batching is done to amortise the inevitable overhead associated with starting HMMER over a number of sequences.

A standard Mapper has its map function called for each input key/value pair but receives no notification that all key/value pairs have been processed. This causes a problem for this batching as once the final key/value pair has been written, the file must be processed. Because of this, HadoopHmmer implements the map function at a slightly lower level in the API, by implementing the `MapRunnable` interface. The default `MapRunnable` simply calls `next()` on the RecordReader and passes returned key/value pair to the map function. By controlling the calls to the RecordReader, HadoopHmmer's `MapRunnable` implementation can process the remaining

key/value pairs once RecordReader's `next()` method returns false, which indicates that there are no more inputs.

A map function outputs intermediate key/value pairs. For HadoopHmmer, an output is generated for each result returned by HMMER via JavaHmmer. Because HadoopHmmer will attempt to combine multiple matches to the same model into one larger match, all hits against one model must be sent to the same reducer. However, for biological reasons, only hits in one reading direction should be combined, although the combination of hits in different frame offsets is allowed[17]. Therefore, the model ID is concatenated with the frame direction to form the intermediate key. The model ID is one of the fields returned by HMMER, whilst the frame offset can be extracted from the sequence identifier, which is also returned by HMMER.

HadoopHmmer actually runs JavaHmmer in a separate thread. Whilst JavaHmmer is running, a second file on the local filesystem has the next batch of sequence data written to. Once JavaHmmer completes, it is re-run using this second file with sequence data then being written to the first file once again. This threaded approach ensures that there is little time lost in waiting for data. This could occur, for example, when the RecordReader returns its final key/value pair as this may extend past the InputSplit and has the potential to cause an unexpected fetch of data from another node in the cluster.

Reducer

The input key to each call to the Reducer is a model ID and reading direction; the input values are all matches to that model in that reading direction.

The simplest possible Reducer is known as an *identity reducer* and simply outputs all the intermediate key/value pairs it receives as final output without processing them. For HadoopHmmer, this means output is then equivalent to running the original HMMER programs. This is useful for two reasons. Firstly, it provides a way of simply distributing HMMER, without the additional work of combining hits. If the input files to HadoopHmmer are in fact protein sequences generated by the standard approach using gene predictions, HadoopHmmer could potentially be used as a drop-in replacement for HMMER. Secondly, it allows the performance of HadoopHmmer to be more fairly compared with the original program as they both perform the same task.

When HadoopHmmer is used on data that has not been through gene prediction, it should attempt to combine hits to models into a longer match. In this case, the Reduce function is more complex.

Firstly, HadoopHmmer discards any very low scoring hits. Currently, these are considered to be a hits with a bit score of less than two. The bit score is the score generated by the Forward algorithm (see section 4.2.1). Secondly, the matches are sorted by their start position, relative to the DNA sequence. To calculate this position, the query identifier contains the start of the sequence relative to the DNA (Figure 13, distance x) and this is added to the start of the alignment between sequence and model (Figure 13, distance y).

Next, the sorted list of hits is iterated over and hits are merged into a longer match where possible. In general, if a hit is aligned with some part of the HMM and a subsequent hit is

[17]This is because mutations may cause a *frame shift*, meaning the frame offset of the protein coding may change.

aligned with a latter part of the HMM, those hits may be merged. There are a number of parameters that control the merging process:

- Small overlaps in model alignment are allowed. This is because alignments are typically somewhat uncertain.
- Larger overlaps (> 5 nucleotides) are not allowed. If there is a large overlap, the hits will not be merged.
- If two hits are separated by a large amount of DNA (Figure 13, distance z), they will not be merged as they are considered unlikely to form the same protein sequence. This is currently set at $100,000$ base pairs.

The parameters used in the reducer were generally chosen by visual inspection of data, typically requiring scripts to be written to aggregate and sort data to help make an education decision. A number of other parameters were also considered, including the proportion of the HMM that had been matched against; the length of the model; and the reported accuracy of the alignment. However, no firm conclusions were drawn from the use of these.

Examples of hits that will and will not be merged are given in Figure 13.

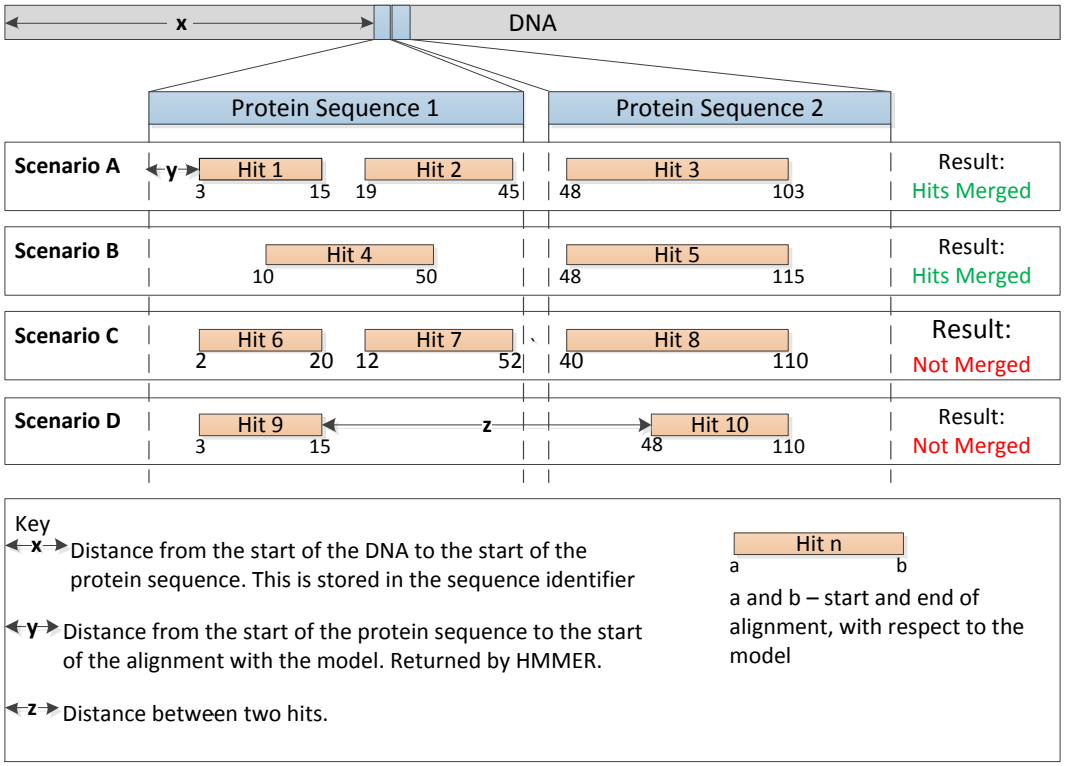


Figure 13: An illustration and explanation of some scenarios where hits will be merged together and some scenarios where they will not.

When hits are merged, their bit scores are added together as they represent a longer path through the profile HMM. Although the probability of a path through an HMM is calculated by multiplying individual probabilities, this process is carried out in log space to prevent problems with numerical underflow, which means the multiplication becomes an addition [1]. However, these added scores are undoubtedly an over-estimate as they do not include any form of gap penalty. This is a current limitation with this method.

Finally, the reducer outputs any hits (merged or otherwise) if they have a bit score above the required threshold. This is currently set at 10, though a higher value may be more appropriate as the next section will demonstrate.

6.5.4 Results

To test the quality of the results, they were compared against the results obtained from standard methods. To generate a target dataset to compare to, the following steps were taken:

- Obtain a dataset of protein sequences from predicted genes. For these experiments, protein sequences from Ensembl's release 58 of the Human genome were used. This older release was chosen as pseudogene data was available from the SUPERFAMILY website [18].
- To reduce the size of the dataset and allow for quicker processing, only protein sequences found on the forward strand of one chromosome, 22, were used.
- These proteins were searched against the full SUPERFAMILY HMM database using HMMER, generating a table of results
- Low scoring hits were filtered out from this table, with any result that had an E-Value of less than 0.0001 being discarded. This threshold was chosen as it is the threshold used by SUPERFAMILY.
- Any hits to the same model within the same gene were grouped together. The reason for this is that the gene may be read in multiple different ways potentially creating a number of similar proteins. Each of these proteins may include the same domain from the same location in the DNA. Hence there may be multiple proteins generating multiple hits to the domain, but there is only a single copy in the underlying DNA.

HadoopHmmer was then run using the raw sequenced DNA data for chromosome 22, with only hits on the forward strand being processed. The two datasets were then compared. Each hit h to model m within gene g in the target database was considered found if there were one or more hits to m in the Hadoop database within the bounds of g. Conversely, if there were no hits to model m within the gene g in the Hadoop dataset, then h was not found. A small tolerance of 100 base pairs is added to either end of the boundaries of g. This is because both alignments and gene boundaries will be somewhat uncertain. Additionally, the number of hits within the Hadoop dataset that do not match any hits in the target dataset are counted and considered as false positives.

Figure 14 shows how the number of hits found and missed and number of false positives varies as the final bit score threshold is varied. For this experiment, the target dataset does not include any hits contained within pseudogenes.

[18] A list of the domains found within the pseudogenes are available at `http://supfam.cs.bris.ac.uk/SUPERFAMILY/cgi-bin/gen_list.cgi?genome=ps6`. The locations of the pseudogenes themselves were obtained directly from the authors of SUPERFAMILY.

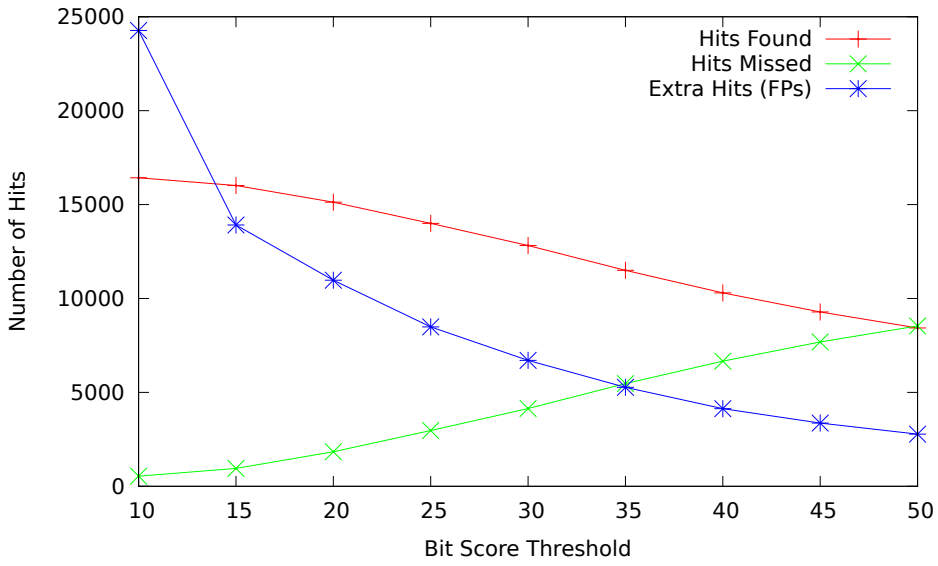


Figure 14: A graph comparing the results found by HadopHmmer to the results obtained from conventional methods. The final bit score threshold is varied to determine its effect. The target dataset is generated from Ensembl's 58th release of the human genome and does not include any pseudogene data. The number of results in the target data set is the number of hits found added to the number missed (16, 344). FP = False positive.

The graph suggests that a bit score of between 15 and 20 would be the optimum value and give the best results. The number of false positives is very high, however, being comparable to the number of hits found. This is perhaps to be expected without the inclusion of pseudogene data. Figure 15 shows the results obtained when the experiment is re-run with pseudogene data included in the target dataset. This increases the total number of target hits by 9, 169, making it 25, 531.

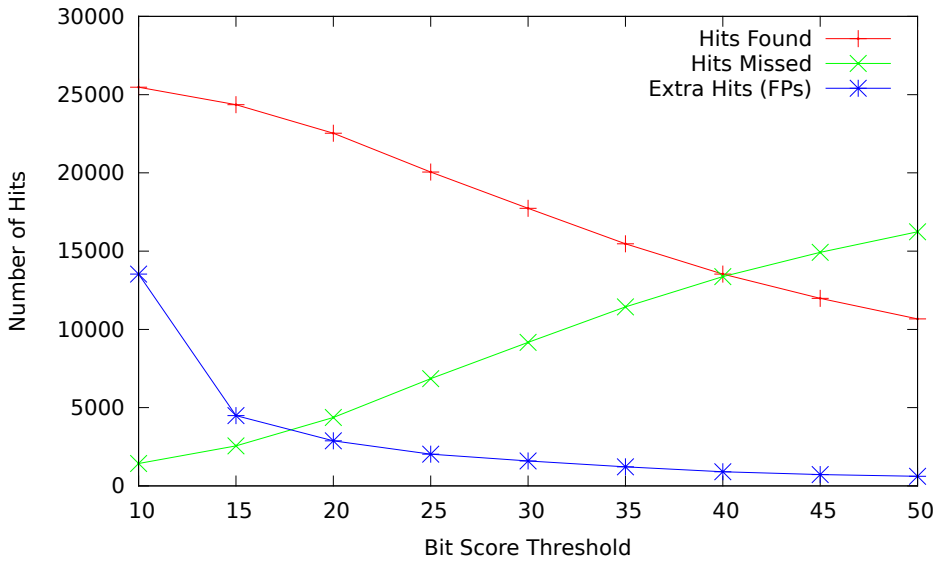


Figure 15: A rerun of the experiment in Figure 14 with pseudogene data included in the target dataset. This increases the number of target hits to 25, 531.

Once again, a bit score of between 15 and 20 appears to be optimal. With a bit score of 20, 87.2% of the target hits are found. Additionally, 7.3% of the hits reported by HadoopHmmer are false positives.

HadoopHmmer contains a number of parameters to be experimented with. Although a full and rigorous test of these remains as future work, two parameters were briefly considered. Figure 16 shows the results obtained when HMMER was run with a higher E-Value cut-off of 50. This means that roughly 50 false positives are expected per query, compared with a default value of 10 [50]. The rationale for this is that there may be hits that are insignificant on their own that are not reported by HMMER but, when merged with other results, they become more significant. However, very little difference was observed in the results. A possible explanation for this may be that these insignificant hits are filtered out in the first stage of HadoopHmmer's Reduce process. In future, the value used for this filter may need to be adjusted.

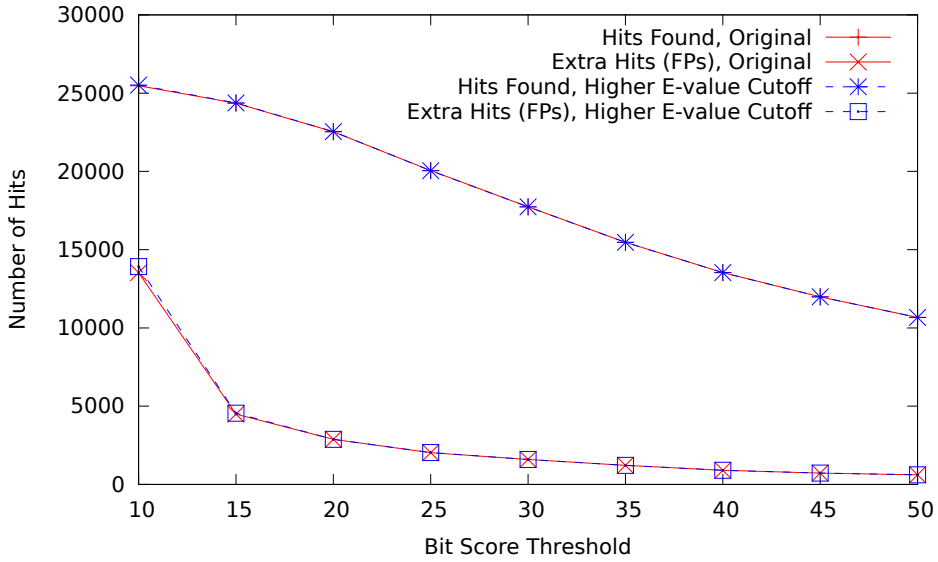


Figure 16: A repeat of the experiment in Figure 15 but with HMMER run with an E-value of 50 (`-E 50`). The results from Figure 15 are shown for comparison. For clarity, the number of Hits that were not found are not shown; any change in the number of his found produces a corresponding but opposite change in the number not found.

Figure 17 considers the effect of creating longer sequences in the six-frame translator . Longer sequences will mean fewer queries are made with HMMER, which should mean HMMER reports less false positives overall.

The results show a drop in both the number of hits found and the number of false positives, though for lower bit scores, the false positives are reduced more than the hits found. The reduction in the number of hits found could be due to there being less insignificant hits that are merged to create significant hits.

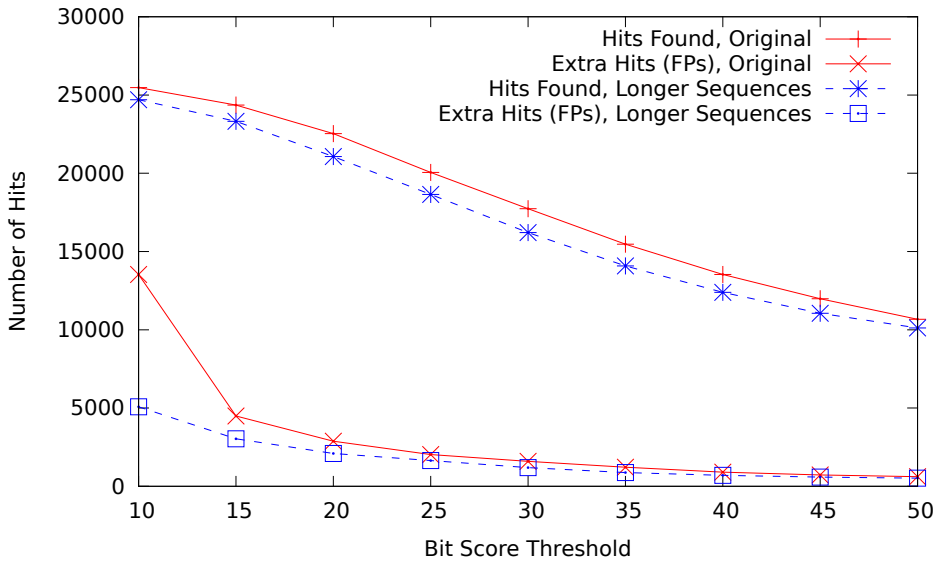


Figure 17: A repeat of the experiment in Figure 15 but with sequences between 5000 and 10000 amino acids created. The results from Figure 15 are shown for comparison. For clarity, the number of hits that were not found are not shown; any change in the number of his found produces a corresponding but opposite change in the number not found.

6.5.5 Scalability and Performance

In addition to working towards protein searching without gene prediction, the other aim of this work is to distribute HMMER using Hadoop. To determine how well this has worked, some small scaling tests were performed and the run-time compared with running JavaHmmer on one node using the same six-frame translated sequences. In these tests, an identity reducer was used, so both JavaHmmer and HadoopHmmer performed the same work. The results of this are shown in Figure 18.

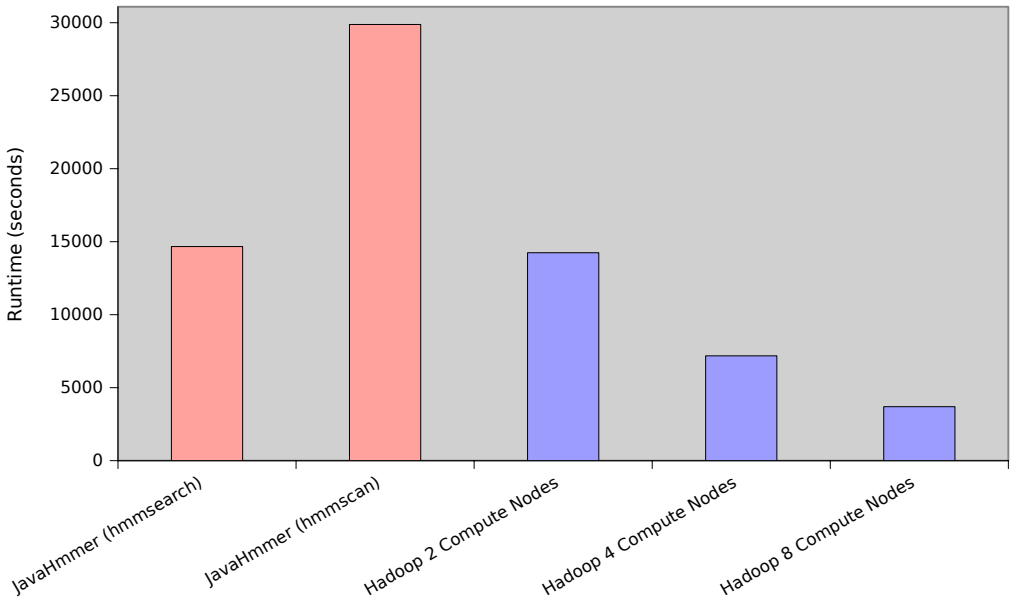


Figure 18: A comparison of the run-time of HadoopHmmer on different sized clusters and JavaHmmer running on one node. HadoopHmmer uses hmmscan in all cases.

This early result shows promising scaling, although it is not quite the perfect linear increase desired. For example, using 8 compute nodes is 3.85x faster than using 2 nodes. Additionally, much larger clusters would need to be tested to fully demonstrate scalability. One potential development for improving scaling would be to use smaller HDFS block sizes (see section 4.4.1). The sequence file used in these tests is 56MBs in size. Since the default block size of 64MBs was used, the file is stored in a single block. With the default replication of three, this means that only three DataNodes store any of the file. This mean that five nodes must transfer their InputSplit across the cluster. If a smaller block size was used and the file stored in multiple blocks, more nodes would have a part of the file stored locally. Hadoop's task scheduling would then attempt to perform computation on that part of the file on the same node, improving data locality and reducing the transfer between nodes. Additionally, a Combiner is not currently implemented, which could help to lower the data transfer between mappers and reducers.

The graph also shows that running hmmsearch is almost twice as fast as running hmmscan, when using JavaHmmer on one node. This is perhaps to be expected given the results shown in Figure 10, with hmmsearch being faster than hmmscan when there are a sufficient number of sequences. However, using hmmsearch with HadoopHmmer has some problems. Because the sequences are divided up amongst the compute nodes, each node runs hmmsearch with a smaller number of sequences, reducing the benefit of using hmmsearch. Figure 19 shows the the run-time of HadoopHmmer with hmmscan and hmmsearch as the number of map tasks is varied; the number of compute nodes is fixed at 8 in all cases.

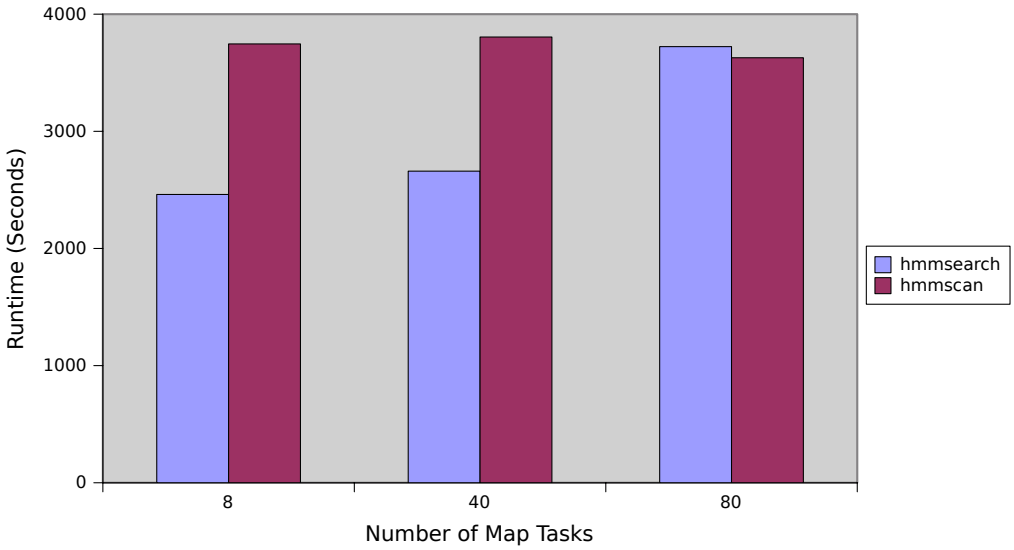


Figure 19: A graph showing the runtime for hmmscan and hmmsearch as the number of map tasks is varied. The number of compute nodes was 8 in all cases.

With hmmsearch, HadoopHmmer is fastest using the smallest number of map tasks. This is because hmmmsearch is run just 8 times, once per node, so each run has a large number of sequences [19]. As the number of map tasks increases, the number of sequences per run of hmmsearch falls, reducing performance. There is however a problem in using a very small number of map tasks. This is that the total run time becomes entirely dependant on the slowest node. Additionally, if one task failed for any reason (for example, hardware failure), a large portion of work would have to be repeated. By using a very small number of map tasks, the benefits of the Hadoop platform are reduced. Furthermore, even with just 8 map tasks, HadoopHmmer running hmmsearch on 8 nodes is only 5.96 times faster than hmmsearch on one node; well below the ideal speed-up of 8. Again, this is due to a smaller number of sequences being used. Using hmmscan does not have these problems because the run-time per sequence is independent of the number of sequences. This is shown with the run time for hmmsearch being almost identical regardless of the number of map tasks.

Using hmmsearch would in theory by preferable as the entire workload consists of lots of sequences and lots of models. To gain the best performance, each run of hmmsearch must use lots of sequences. The only way to achieve this would be to split the HMM database between nodes, distributing workload by splitting up the HMM database rather than the sequences. This is the opposite of the current approach outlined in section 6.5.1. A potential way forward may be to continue to use the Distributed Cache to distributed *all* of the profile HMMs to each node where they can remain between jobs but have each node only work on a subset of the models that they have. This is an approach which could be explored in the future.

[19] For these tests, all of the key/value pairs a mapper received were processed in one batch. See section 6.5.3 for details of batching.

6.5.6 Larger Scaling Tests

During the private beta of HP Cloud, the size of the Hadoop cluster was limited to 10 nodes. Very near the end of this project, HP Cloud entered a public beta phase. At this point, the limits on number of instances were increased, allowing for up to 20 nodes. Using the same experimental setup from Figure 18, an early result showed that 19 compute nodes (and one master node) were 17.8 times faster than hmmscan on a single node. Once again, this is near linear and a very good result, though there is room for improvement, as previously discussed. Finally, a request for this instance limit to be raised further was submitted. Unfortunately, this was not processed in time to generate any results for this work.

7 Evaluation and Conclusion

Quoting from section 2.2, the two broad aims for this project were:

1. To take existing protein homology software, known as HMMER, and modify it so that it can be run on the Apache Hadoop framework, which will allow computation to be distributed over a large number of compute nodes.
2. To explore how homologous proteins can be found directly from DNA sequence data without using *gene prediction*. Gene prediction is a phase of analysis typically required before protein homology can be performed.

Overall, both of these aims have been achieved with reasonable success. HadoopHmmer has provided a method for running HMMER on the Hadoop framework and has been shown to scale smoothly with up to 19 compute nodes. In addition, early results show promise for protein homology without gene prediction, with the matches found between sequences and models being relatively comparable to traditional methods.

In addition, section 6.3 provides some benchmarks of two of HMMER's programs, helping to quantify the trade-offs between hmmscan and hmmsearch. These results may be of general interest to those who user HMMER.

7.1 Discussion

Pseudogenes

A possible reason for the approach to protein homology developed in this paper not being employed before is due to pseudogenes, as they are typically thought of as something to avoid. However, although pseudogenes are non-functional, they still provide insights into evolutionary history and, as such, are still useful to include. Furthermore, if an analysis *is* only interested in protein sequences within true genes, gene prediction tools could be run after determining any homologous sequences. Gene prediction is difficult and time consuming, so it may be more efficient to only perform it on sections of DNA where there are known to be protein homologues. For example, Ensembl's gene prediction pipeline typically takes "at least four months" on a full genome[20].

E-Values

Currently, HadoopHmmer does not report E-values for the matches found, which means that it is difficult to assess the statistical significance of results. Although it is possible to convert the bit scores produced to E-values, doing so is currently somewhat meaningless. Without taking into account any gap penalty, E-values will suggest a falsely high level of significance. By considering the partial model alignments and their locations, it should be possible to compensate for any deleted amino acids - either with a constant gap penalty or by reading those used in the HMM. However, it is less clear how any inserted amino acids could be handled. This is because it is expected that there will be insertions between partial hits, due to the presence of introns and these should not be penalized. However, any insertions that are not due to an intron should be penalized.

[20] See `http://www.ensembl.org/info/docs/genebuild/genome_annotation.html`. Accessed 15/05/2012.

Free Parameters

Due to the exploratory nature of the project and the need for substantial research, HadoopHmmer has a number of free parameters that require further experimentation. These include the lengths of sequences generated by the six-frame translator, the bit score threshold used in the first stage of the reducer and the parameters controlling the merging of partial hits, i.e. the maximum overlap between model alignments and the maximum gap between the hits in the sequence. These have currently been set empirically. A more thorough investigation of the effects of these parameters may yield improved results.

Extensiveness of Testing

Although good early results were obtained both in terms of performance and quality of results, further testing is required. To properly demonstrate the linear scaling that should be possible with Hadoop, a larger number of nodes need to be tested. Unfortunately, this was not possible during this project due to limitations in hardware. Additionally, a variety of larger sequences - ideally a full genome - should be tested to compare the results from protein homology without gene prediction against traditional methods

7.2 Current Project Status

JavaHmmer

The implementation of JavaHmmer is generally complete, though there are options for potential future optimizations as discussed in section 6.2.4. With a little documentation, it could be used by others. Currently, building the shared library is a little awkward. The shared library must be compiled with `-fpic` to generate position independent code and also needs additional include paths set. It would be useful to determine how these additional build requirements can be integrated in HMMER's current build process, which uses GNU Autotools.

HadoopHmmer

HadoopHmmer's code base is not yet of production quality. For example, the reducer is not currently able to handle model hits that occur on the reverse strand of DNA, as discussed in section 6.5.2. Additionally, to support the use of HadoopHmmer as a drop-in replacement for HMMER (that is, to simply distribute HMMER, rather than perform the merging of hits) it would be useful for the InputFormat and RecordReader to support generic FASTA files (see Appendix B), which typically contain protein sequences spanning multiple lines; HadoopHmmer currently assumes the sequence is on a single line for simplicity.

Finally, there are a number of hard-coded aspects that should be configurable by command line arguments, such as the number of map tasks, hit merging parameters and arguments to JavaHmmer.

Future Release

Although not quite ready for release, all code produced in this project will eventually be released under a GNU General Public License (GPL) - the choice of license is dictated by the fact that

HMMER is under the GPL. In addition, images for Amazon EC2 instances will also be created and released, which will make it easier for others to quickly launch a Hadoop cluster in the cloud to run HadoopHmmer across as many nodes as they desire. This will enable the wide user base of HMMER to take advantage of the benefits offered by a cloud-based solution, which they are currently not able to do.

7.3 Future Work

There are a number of options to explore for future work:

- A full, systematic investigation into the various parameters used by HadoopHmmer.
- Larger scale testing, both of the scalability achieved and of the results compared to traditional methods. It would be beneficial to benchmark HadoopHmmer on clusters with hundreds of nodes using large datasets, such as the human genome.
- An extension to HadoopHmmer so that the HMM database can be split amongst the nodes allowing more efficient use of hmmsearch, which should be faster than hmmscan for large datasets
- Improving HadoopHmmer by documenting, tidying code and adding the features needed to publish a production level release. This would then be distributed to the academic community.

7.4 Final Conclusions

This work has been a success, both in terms of distributing HMMER using Hadoop and for performing protein homology without gene predictions. As discussed above, there are some limitations that remain for future work.

With HMMER being very widely used, the current explosion of DNA sequencing data and the drive towards analysis tools that run in the cloud, a Hadoop implementation of HMMER may be both timely and of interest to a large number of people within the academic community. Additionally, the novel approach to protein homology presented in this work is of increasing relevance. Next Generation Sequencing is allowing people to generate DNA data more quickly than ever before. As "desktop" sequencing machines become the norm, researchers will increasingly desire to use DNA sequencing simply as a means to and end, to answer their current hypothesis, rather than with the aim of producing a high-quality, fully sequenced, assembled and gene-predicted genome. To do so, analysis tools that do not rely on the availability of such a genome are needed.

The author will continue this project when starting PhD study later this year and work towards the publication of these results.

8 References

[1] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[2] M. Baker, "Next-generation sequencing: adjusting to data overload," *Nature Methods*, vol. 7, pp. 495–499, July 2010.

[3] M. C. Schatz, B. Langmead, and S. L. Salzberg, "Cloud computing and the DNA data race.," *Nature biotechnology*, vol. 28, pp. 691–3, July 2010.

[4] K. Wetterstrand, "DNA Sequencing Costs: Data from the NHGRI Large-Scale Genome Sequencing Program." `http://www.genome.gov/sequencingcosts`. Accessed 13/05/2012.

[5] L. D. Stein, "The case for cloud computing in genome informatics.," *Genome biology*, vol. 11, p. 207, Jan. 2010.

[6] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[7] O. Gotoh, "An improved algorithm for matching biological sequences.," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, vol. 42. The MIT Press, 3 ed., 2009.

[9] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[10] T. r. Rognes, "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation.," *BMC bioinformatics*, vol. 12, p. 221, Jan. 2011.

[11] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units.," *BMC research notes*, vol. 2, p. 73, Jan. 2009.

[12] S. R. Eddy, "Accelerated Profile HMM Searches.," *PLoS computational biology*, vol. 7, p. e1002195, Oct. 2011.

[13] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[14] S. R. Eddy, "A new generation of homology search tools based on probabilistic inference.," *Genome informatics International Conference on Genome Informatics*, vol. 23, no. 1, pp. 205–211, 2009.

[15] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler, "Hidden Markov Models in Computational Biology," *J of Molecular Biology*, vol. 235, pp. 1501–1531, 1994.

[16] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[17] S. R. Eddy, "Profile hidden Markov models," *Bioinformatics*, vol. 14, pp. 755–763, Oct. 1998.

[18] S. R. Eddy, "Hidden Markov models.," *Current Opinion in Structural Biology*, vol. 6, no. 3, pp. 361–365, 1996.

[19] R. D. Finn, J. Clements, and S. R. Eddy, "HMMER web server: interactive sequence similarity searching.," *Nucleic acids research*, vol. 39, pp. W29–37, July 2011.

[20] S. R. Eddy, "A Probabilistic Model of Local Sequence Alignment That Simplifies Statistical Significance Estimation," *PLoS Computational Biology*, vol. 4, no. 5, p. 14, 2008.

[21] S. Johnson, *Remote Protein Homology Detection Using Hidden Markov Models*. Phd thesis, Washington University School of Medicine, 2006.

[22] S. C. Potter, L. Clarke, V. Curwen, S. Keenan, E. Mongin, S. M. J. Searle, A. Stabenau, R. Storey, and M. Clamp, "The Ensembl Analysis Pipeline," *Genome Research*, vol. 14, no. 5, pp. 934–941, 2004.

[23] C. Burge and S. Karlin, "Prediction of complete gene structures in human genomic DNA.," *Journal of Molecular Biology*, vol. 268, no. 1, pp. 78–94, 1997.

[24] E. Birney, M. Clamp, and R. Durbin, "GeneWise and Genomewise.," *Genome Research*, vol. 14, no. 5, pp. 988–995, 2004.

[25] V. Curwen, E. Eyras, T. D. Andrews, L. Clarke, E. Mongin, S. M. J. Searle, and M. Clamp, "The Ensembl Automatic Gene Annotation System," *Genome Research*, vol. 14, no. 5, pp. 942–950, 2004.

[26] J. A. Cuff, G. M. P. Coates, T. J. R. Cutts, and M. Rae, "The Ensembl Computing Architecture," *Genome Research*, vol. 14, no. 5, pp. 971–975, 2004.

[27] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce (Synthesis Lectures on Human Language Technologies)*. Morgan and Claypool Publishers, 2010.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 29, 2003.

[29] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies MSST*, vol. 0, no. 5, pp. 1–10, 2010.

[31] Yahoo Inc!, "Hadoop Tutorial from Yahoo." `http://developer.yahoo.com/hadoop/tutorial/module4.html`.

[32] D. Borthakur, S. Rash, R. Schmidt, A. Aiyer, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, and A. Menon, "Apache hadoop goes realtime at Facebook," in *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, (New York, New York, USA), p. 1071, ACM Press, June 2011.

[33] L. Lin, V. Lychagina, and M. Wong, "Tenzing A SQL Implementation On The MapReduce Framework," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, pp. 1318–1327, 2011.

[34] D. Borthakur, "HDFS Architecture Guide." `http://hadoop.apache.org/common/docs/current/hdfs_design.html`.

[35] S. Groot, K. Goda, and M. Kitsuregawa, "A study on workload imbalance issues in data intensive distributed computing," in *Proceedings of the 6th international conference on Databases in Networked Information Systems*, DNIS'10, (Berlin, Heidelberg), pp. 27–32, Springer-Verlag, 2010.

[36] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media / Yahoo Press, 2010.

[37] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary, "Evaluating the use of GPUs in liver image segmentation and HMMER database searches," in *2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, IEEE, May 2009.

[38] S. Quirem, F. Ahmed, and B. K. Lee, "CUDA acceleration of P7Viterbi algorithm in HMMER 3.0," *IEEE International Performance Computing and Communications Conference*, vol. 0, pp. 1–2, 2011.

[39] N. Abbas, S. Derrien, S. Rajopadhye, and P. Quinton, "Accelerating HMMER on FPGA using parallel prefixes and reductions," in *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 37–44, 2010.

[40] J. P. Walters, B. Qudah, and V. Chaudhary, "Accelerating the HMMER sequence analysis suite using conventional processors," *20th International Conference on Advanced Information Networking and Applications Volume 1 AINA06*, vol. 1, pp. 289–294, 2006.

[41] G. Chukkapalli, C. Guda, and S. Subramaniam, "SledgeHMMER: a web server for batch searching the Pfam database.," *Nucleic acids research*, vol. 32, pp. W542–4, July 2004.

[42] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, pp. 345–420, Dec. 1994.

[43] B. Rekapalli, C. Halloy, and I. B. Zhulin, "HSP-HMMER," in *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*, (New York, New York, USA), p. 766, ACM Press, Mar. 2009.

[44] R. C. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics.," *BMC bioinformatics*, vol. 11 Suppl 1, p. S1, Jan. 2010.

[45] S. Leo, F. Santoni, and G. Zanetti, "Biodoop: Bioinformatics on Hadoop," *2009 International Conference on Parallel Processing Workshops*, vol. 0, pp. 415–422, 2009.

[46] A. Matsunaga, M. Tsugawa, and J. Fortes, "CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications," *2008 IEEE Fourth International Conference on eScience*, vol. 0, pp. 222–229, 2008.

[47] M. C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[48] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg, "Searching for SNPs with cloud computing," *Genome Biology*, vol. 10, no. 11, p. R134, 2009.

[49] S. Liang, *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall PTR, 1999.

[50] S. R. Eddy, "Hmmer user's guide." `ftp://selab.janelia.org/pub/software/hmmer3/3.0/Userguide.pdf`. Accessed 02/05/2012.

A Extra Graphs for Section 6.3

These graphs show the reverse of the situation focused on in section 6.3, with a large, fixed number of sequences and a varying number of profile HMMs.

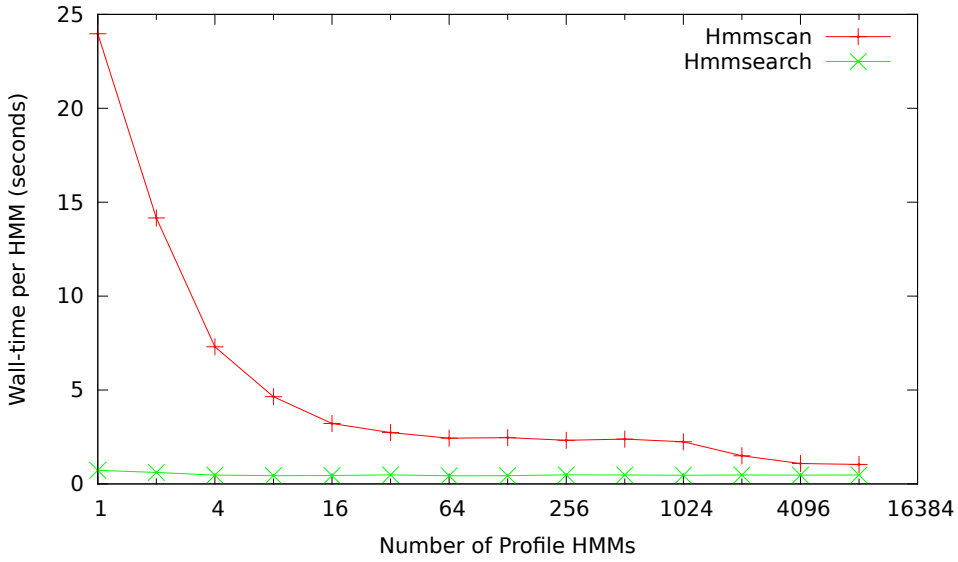


Figure 20: A graph showing the run time per model as the number of models is varied. The number of sequences was fixed, using all 27473 protein sequences from release 67 of Ensembl's Gorilla genome. To generate the profile HMMs, model 0034792 from SUPERFAMILY, representing 124 amino acids, was duplicated. All times were obtained by averaging three trials.

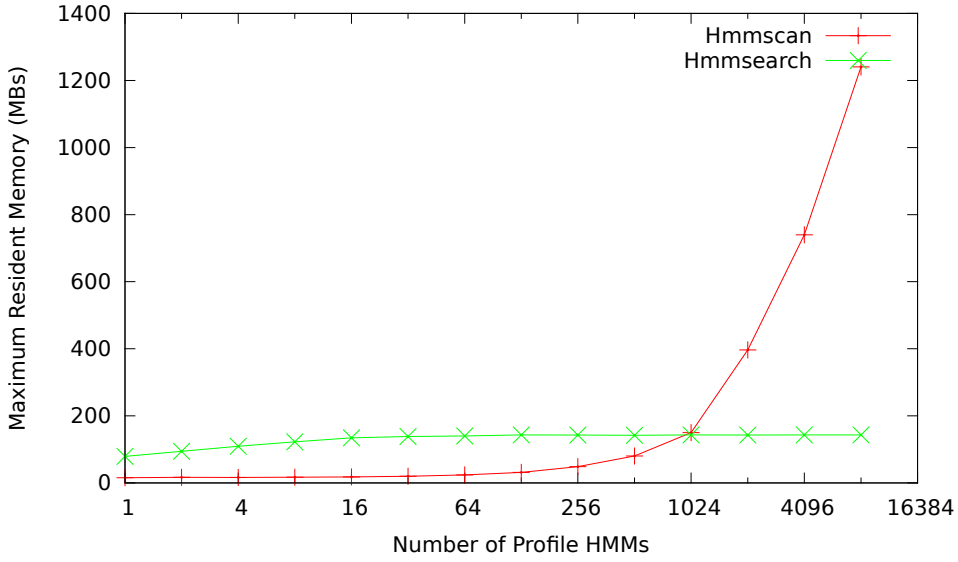


Figure 21: A graph of the maximum resident memory of hmmsearch and hmmscan as reported by the unix time utility. The same experimental data as in Figure 20 was used.

B Example Six-Frame Translation

To complement section 6.5.2, this Appendix gives a short example of six-frame translation.

FASTA files are a common file format for storing both DNA and protein sequences. They contain sequences separated by a one line comment or description associated with the sequence. An example is given below.

```
>Sequence Identifier
ACATAGACTACCCCATTGGCAGC
>Another Sequence. Comment line can include arbitrary data.
AATACGCGTAGCGATAGACTAGTACGTACTGCTGAAACGGCCGG
...
```

If we examine the first sequence, the forward DNA strand is:

```
ACATAGACTACCCCATTGGCAGC
```

The reverse strand is obtained both by reversing the string and by substituting each nucleotide for its base pair. A is paired with T and C is paired with G. The reverse strand is thus:

```
GCTGCCAATGGGGTAGTCTATGT
```

For each of the forward and reverse strands, there are three reading frames obtained by offsetting the start of the translation. The reading frames are shown below, with each three letter codon boxed.

```
Forward Frame 0: |ACA|TAG|ACT|ACC|CCA|TTG|GCA|GC
Forward Frame 1: A|CAT|AGA|CTA|CCC|CAT|TGG|CAG|C
Forward Frame 2: AC|ATA|GAC|TAC|CCC|ATT|GGC|AGC|

Reverse Frame 0: |GCT|GCC|AAT|GGG|GTA|GTC|TAT|GT
Reverse Frame 1: G|CTG|CCA|ATG|GGG|TAG|TCT|ATG|T
Reverse Frame 2: GC|TGC|CAA|TGG|GGT|AGT|CTA|TGT|
```

Applying a standard table of mappings between codons and amino acids [21], we obtain six protein sequences. These are shown below in FASTA format. The sequences have not been split at each stop codons; instead the stop codon is replaced by an X, which represents any protein.

```
>SequenceID_forward_frame:0
TXTTPLA
>SequenceID_forward_frame:1
HRLPHWQ
>SequenceID_forward_frame:2
IDYPIGS
>SequenceID_reverse_frame:0
AANGVVY
>SequenceID_reverse_frame:1
LPMGXSM
>SequenceID_reverse_frame:2
CQWGSLC
```

[21] This is not a one-to-one mapping. Some amino acids are encoded by more than one codon

If the sequences are instead split at any stop codon, the below FASTA file is generated. In this example, the sequence descriptions simply contain a numeric index indicating which split the sequence is associated with. In practice, it may be more useful to include the start of the sequence, relative to the original DNA.

```
>SequenceID_forward_frame:0_1
T
>SequenceID_forward_frame:0_2
TTPLA
>SequenceID_forward_frame:1
HRLPHWQ
>SequenceID_forward_frame:2
IDYPIGS
>SequenceID_reverse_frame:0
AANGVVY
>SequenceID_reverse_frame:1_1
LPMG
>SequenceID_reverse_frame:1_1
SM
>SequenceID_reverse_frame:2
CQWGSLC
```