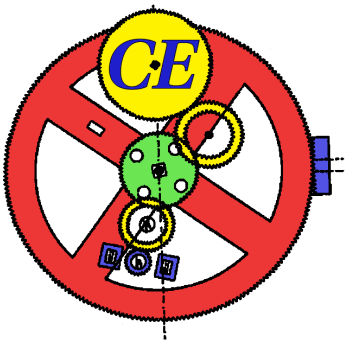


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Scalability of Bioinformatics Applications for Multicore Architectures

Ernst Joachim Houtgast
1056212

CE-MS-2009-30

Abstract

Exponential growth in biological sequence data combined with the computationally intensive nature of bioinformatics applications results in a continuously rising demand for processing power. Microprocessor complexity and, more importantly, computational capability increases as well, through transistor budgets that grow in line with Moore's law. However, limits in power consumption, frequency scaling and memory technology cause single threaded performance improvements to stagnate. The result is a paradigm shift to parallel architectures, an example being the state-of-the-art Cell Broadband Engine. In this thesis, suitability of this architecture is examined for HMMER, a bioinformatics application that identifies similarities between protein sequences and a protein family model. Qualitative and quantitative analysis is performed to reveal its scaling behavior and potential bottlenecks. Inspection of the program structure shows that the parallelization strategy imposes limits on scaling ability. Based on function profiling, a model for application performance is proposed that is accurate within 2%. From the model, the optimal PPE/SPE ratio is derived for different workloads. For typical workloads, the PPE can supply nine SPEs with jobs. The TaskSim simulator, whose phase-based simulations are accurate within 2%, is used to validate the model's predictions and to demonstrate that scaling behavior is mostly determined by the buffering of jobs.

Keywords: bioinformatics, Cell Broadband Engine, HMMER, many-core scaling, performance modeling, performance simulation, sequence analysis.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Scalability of Bioinformatics Applications for Multicore Architectures

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Ernst Joachim Houtgast
born in The Hague, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Scalability of Bioinformatics Applications for Multicore Architectures

by Ernst Joachim Houtgast

Abstract

Exponential growth in biological sequence data combined with the computationally intensive nature of bioinformatics applications results in a continuously rising demand for processing power. Microprocessor complexity and, more importantly, computational capability increases as well, through transistor budgets that grow in line with Moore's law. However, limits in power consumption, frequency scaling and memory technology cause single threaded performance improvements to stagnate. The result is a paradigm shift to parallel architectures, an example being the state-of-the-art Cell Broadband Engine. In this thesis, suitability of this architecture is examined for HMMER, a bioinformatics application that identifies similarities between protein sequences and a protein family model. Qualitative and quantitative analysis is performed to reveal its scaling behavior and potential bottlenecks. Inspection of the program structure shows that the parallelization strategy imposes limits on scaling ability. Based on function profiling, a model for application performance is proposed that is accurate within 2%. From the model, the optimal PPE/SPE ratio is derived for different workloads. For typical workloads, the PPE can supply nine SPEs with jobs. The TaskSim simulator, whose phase-based simulations are accurate within 2%, is used to validate the model's predictions and to demonstrate that scaling behavior is mostly determined by the buffering of jobs.

Keywords: bioinformatics, Cell Broadband Engine, HMMER, many-core scaling, performance modeling, performance simulation, sequence analysis.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-30

Committee Members :

Advisor:	G. N. Gaydadjiev, Assistant Prof, CE, TU Delft
Advisor:	S. Isaza Ramirez, PhD Student, CE, TU Delft
Chairperson:	K. L. M. Bertels, Associate Prof, CE, TU Delft
Member:	M. J. T. Reinders, Full Prof, ICT, TU Delft
Member:	B. H. H. Juurlink, Associate Prof, CE, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

The thesis resting in front of you is the product of many months of work. It forms the final element required for obtaining the Master of Science degree in Computer Engineering from the Delft University of Technology. This thesis would not exist if not for the support and guidance of many of my friends and family. To each of them I would like to express my gratitude. Furthermore, I would like to explicitly mention a select few, whose help has been invaluable to me:

- First of all, I would like to thank my parents and my family, for always having faith in me, for supporting me in whatever I do, and for being patient during the years of my study.
- I would like to thank my supervisors: Sebastian Isaza, for always making time for me whenever I had a question, for providing me with valuable feedback, and for the chats on a wide variety of non-thesis related subjects; and Georgi Gaydadjiev, for providing me with motivation, guidance and useful insights.
- I would like to thank the helpful people at the Barcelona Supercomputing Center, for allowing me to use their unreleased, experimental TaskSim simulator and their Cell blade environment, and for giving support whenever I encountered an issue.
- I would like to thank Jelle ten Hoeve, for the interesting discussions we had, for piquing my interest in bioinformatics, and for all the small talk we made during our many coffee breaks.
- And of course I would like to thank lovely Farnaz, for her love and support, for making these last few months at the TU the most enjoyable ones yet, and for always reminding me that '*someone has thesis*'!

Ernst Joachim Houtgast
Delft, The Netherlands
November 6, 2009

1 Introduction

The discovery of the DNA structure ushered in the widespread use of and interest in genetic data. Biological analysis at the genetic level brings innumerable possibilities. Its applications will influence the lives and welfare of millions of people: knowledge of the genetic system allows for development of drugs that target diseases in a much more specific manner, thus becoming more effective; key factors that induce illnesses such as Alzheimer's disease or cancer can be identified; even evolutionary links between different species can be established. Analysis of DNA and proteins, which form the blueprint and building blocks of life, allows biologists to gain novel insights into the development of organisms and into the evolution of life itself. The economic and scientific incentives such applications offer result in continued growth in attention for the field of genetics: research budgets are growing and more people become involved. As an industry, it generates a huge amount of data and the rate of accumulation will continue to accelerate. The result is an ever increasing demand in computation power required for processing.

In parallel to the developments in biology, computer technology experienced an evolution of its own. Starting out as a nascent industry in the previous century, it has grown to become a field that influences every part of modern life. Computing and communication technology have had a huge impact on lifestyle. Imagine a world without the communication capabilities of today, such as cellphones or the internet, or without contemporary processing capabilities. The commoditization of computing technology has led to its pervasive use. The advances in circuit density, as dictated by Moore's law, continue at an exponential rate. As a result, even a cheap contemporary mobile phone is many times more powerful than the huge mainframes of the last century, whose sheer size prohibited use outside of large laboratories.

Miniaturization proceeds apace and transistor budgets continue to grow at an exponential rate. However, three separate issues are troubling computer architects, resulting in the stagnation of improvement to single threaded performance: power consumption is soaring, resulting in a choice for more efficient techniques over techniques that are dedicated to raw performance; memory technology lags behind computational capabilities; and processor frequency scaling is peaking. These three 'walls' form the fundamental reason for the current paradigm shift towards parallel computing, a shift from single-core towards multi-core architectures. The switch from sequential to parallel computing attempts to ensure the expected growth in processing capability.

The exponential growth in computing power is a well-suited counterpart to the equivalent surge in the availability of biological data. This has resulted in the emergence of the bioinformatics field. Within this domain, computing technology is applied to solve biological problems to further the understanding about biological systems and processes. The available computation power is harnessed to manipulate and give meaning to data

sets whose sheer size makes analysis by hand unfeasible. In this thesis, the impact of multi-core computing on bioinformatics applications and the opportunities arising from it are investigated.

1.1 Motivation

The thesis project is one of the requirements of the Master of Science degree in Computer Engineering within the faculty of EEMCS of Delft University of Technology. This thesis has been performed within the Δ-ILIAD research theme of the Computer Engineering group, which concerns itself with research on novel computer architectural paradigms, investigating new and unconventional processor architectures. The research itself falls under the SARC project, an EU project focusing on long term research in computer architecture aimed at finding systematic scalable approaches to systems design.

Suitability and effectiveness of the multi-core paradigm on bioinformatics applications remains an open research question. No deep understanding exists of their behavior and scaling capability under many-core situations. Therefore, investigation of their structure and determination of their performance characteristics when scaling processor count is of great interest. This thesis contributes through the analysis of a particular bioinformatics application called HMMER. Part of the popular SPEC '06 processor benchmark, it is considered as representative for this class. Like most bioinformatics programs, HMMER exhibits parallelism at many levels. The execution time is dominated by a small, computationally intensive kernel which is applied to each element in the data set.

HMMER is used to compare protein sequences to a protein model in order to find homologues. These have many uses: for example, they give biologists valuable clues about overlap in functionality between proteins or they indicate the potential existence of a relationship between different species. Analyzing this data is computationally intensive. Hence, high performance computing platforms are often used for processing. However, as in general such computers are assembled using off-the-shelf processors, they do not provide a natural fit to the problem domain. As a result, power efficiency, performance, cost, or all of them suffer. An application-specific integrated circuit would offer optimal performance per watt, but they lack flexibility and their cost of development and production is prohibitive in this domain.

The Cell Broadband Engine is a state-of-the-art processor intended for high performance computing. Its architecture represents a trade-off between specialized and general purpose processing, as it contains both a general purpose core and eight cores specialized for streaming workloads. As such, its performance per watt is favorable for applications that fit the architecture well. The scaling behavior of HMMERCELL, a port of HMMER to this system, is analyzed. Qualitative analysis is performed by inspecting the program structure, explaining its behavior and revealing potential bottlenecks, such as the followed parallelization strategy and the choice for a modified Viterbi algorithm resulting from architectural limitations. Quantitative analysis is performed through profiling, modeling and simulations. The optimal ratio between PPEs and SPEs is deduced and bottlenecks in program structure and implementation architecture are measured to investigate their effect on scaling behavior. The results help creating guidelines for improved performance of bioinformatics applications on many-core architectures.

1.2 Problem Statement

The objective of this thesis is to contribute to the understanding of the interaction between bioinformatics applications and multi-core processing through determination of the performance characteristics and scaling behavior of HMMER, a biosequence analysis application. This results in the three project aims, summarized as follows:

1. Investigate the performance of bioinformatics applications, gaining a thorough understanding of their structure and behavior, through the analysis of a representative application called HMMERCELL.

2. Identify bottlenecks in program operation and system architecture by profiling, modeling and simulating the system's behavior.

3. Improve the understanding of scaling behavior for many-core computing platforms by evaluating the impact of the identified bottlenecks.

1.3 Thesis Overview

The work in this thesis is divided into three main parts:

1. A theoretical part, the result from a literature study into the various fields belonging to the problem domain. The overview provides the reader with a sufficient background and understanding of the relevant parts in biology, mathematics and computer architecture in order to be able to place the rest of the thesis in its proper context. In Chapter 2, a short introduction to the field of bioinformatics is given. Chapter 3 presents the fundamentals of Hidden Markov Models, the mathematical foundation on which HMMER is based. Chapter 4 describes the developments and trends in computer architecture and presents the Cell microprocessor, the implementation platform of this work.

2. An implementation and analysis part, resulting from the work with HMMER, with the implementation architecture, and with the various tools for profiling and simulation. In this part, the application itself is discussed. The program structure is investigated, its behavior analyzed, and scaling behavior simulated. Chapter 5 gives an overview of the internal working of HMMER and HMMERCELL. In Chapter 6, results from analyzing and profiling the application are presented and discussed. In Chapter 7, the results from simulations that measure the impact of bottlenecks to scaling capability are shown.

3. A final part with learnings from this thesis. Chapter 8 presents the conclusions that have been drawn. Recommendations are made indicating promising directions for future research.

PART I:
BACKGROUND

"The biologists thought that a database was an enzyme that acted on datab."
- WIREDWEIRD

2 Introduction to Bioinformatics

The discovery in 1953 of the DNA structure ushered in an explosion in the availability of genetic information. By now, biologists have sequenced the genetic information of many different species. Of such sequencing projects, the Human Genome Project is probably the most well-known. Started in 1990, it led to the first sequencing of an entire human genome in 2000. The databases where such sequences are stored contain huge amounts of data; for instance, the human genome contains approximately three billion base pairs. The sheer size of such data sets makes their analysis impossible by hand. The use of computers however has enabled research that would otherwise be unfeasible.

Bioinformatics is the discipline at the intersection of the fields of molecular biology and computing. The rapid growth in computational power allows for the development and application of new computationally intensive techniques. These techniques are used to increase the understanding of the biological processes that operate within cells.

Although bioinformatics is still in its infancy, the application of high performance computing to the massive data sets provided by biological research has already proven to be very successful. Nowadays, computers are utilized to solve a wide range of problems in bioinformatics. Analysis of biological sequences is one example; another is the field of recombinant DNA technology. Sequencing DNA strands sequentially is a time consuming process. Instead thousands of subsections are read and analyzed in parallel. Fitting the pieces back together requires the use of computers to explore each of the combinations that are possible. Yet another example is genome rearrangement, in which differences in the ordering of gene occurrence onto the DNA is investigated. Reversals, translocations, fusions or fissions cause organisms with similar genes to express wildly different phenotype. For example, the genomes of humans and mice differ by less than 250 rearrangements. One last domain is the field of systems biology modeling. Entire cells are simulated, modeling all the processes which occur within. Even with the computing resources available today, such techniques can only be applied to the simplest of organisms, such as E. Coli.

This chapter aims to provide the reader with a large enough background of bioinformatics to place the rest of the thesis in its proper context. Section 2.1 gives an introduction to molecular biology, explaining the relationship between and functionality of genes and proteins. Section 2.2 explains sequence analysis, the main subject of this thesis, in more detail. Section 2.3 explores some of the software tools that have been developed for biological sequence analysis. Section 2.4 discusses the enormous amount of online available sequence data. Section 2.5 concludes the chapter with a summary.

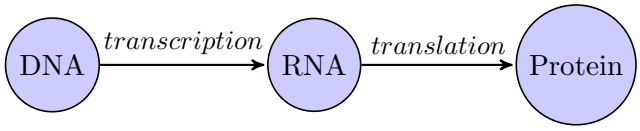


Figure 2.1: The flow of information within a cell.

2.1 Molecular Biology

Molecular biology is the part of biology that studies biological processes at the molecular level. It concerns itself with the processes within a cell and with interactions between cells. The central dogma of molecular biology as stated by Crick in 1956 [11] (later reformulated [12]) is as follows:

> "The central dogma of molecular biology deals with the detailed residue-by-residue transfer of sequential information. It states that information cannot be transferred back from protein to either protein or nucleic acid."

The central dogma describes the relationship between three important classes of molecules within a cell. These three are: DNA, RNA and proteins. Figure 2.1 provides a simplified view of the relationship between these three molecules. DNA can be seen as containing the blueprint of an organism. Proteins effectuate the realization of this blueprint; they are used as the building blocks of cells, used for signaling, used in the metabolism, and much more. RNA functions as a carrier of information: first, DNA is transcribed to RNA in the nucleus. Then, the RNA travels outside of the nucleus to the ribosomes. There, it is translated into proteins. For this thesis' purposes, genes and proteins are the most important. Hence, they will now be discussed in more detail.

The genome constitutes the genetic information of an organism, its complete set of DNA. Figure 2.2 shows the basic organizational structure of DNA and the relationship between DNA, genes and proteins. In essence, DNA can be seen as a long string of symbols, called base pairs or nucleotides. The four symbols in this alphabet are: Adenine, Guanine, Thymine, and Cytosine (inside RNA, Uracil is used instead of Thymine). Three base pairs together can be interpreted as a codon. Each codon codes for an amino acid. Table 2.1 gives the translation from codon to amino acid. A string of codons then codes for a sequence of amino acids. In the ribosomes, proteins are built based on such sequences. Genes are groupings of codons and they are the basic entity in the DNA that codes for proteins. They are considered to be the basic physical unit of heredity. The figure shows a gene that codes for a protein consisting of five amino acids.

The size of a genome varies per species. The genome of a human being consists of approximately three billion base pairs, although not all parts of its DNA contains useful information: large parts are duplicate, junk or control information. Parts of the DNA that code for proteins are called exons (for expressed region). Introns (for intragenic region) are the non-coding sections. About 95% of DNA is non-coding. Of the three billion base pairs a human genome contains, only 0,1% is different between two individual humans. Analysis of the human genome, after the Human Genome Project was completed, estimates that the human genome contains about 20,000 to 25,000 genes. The

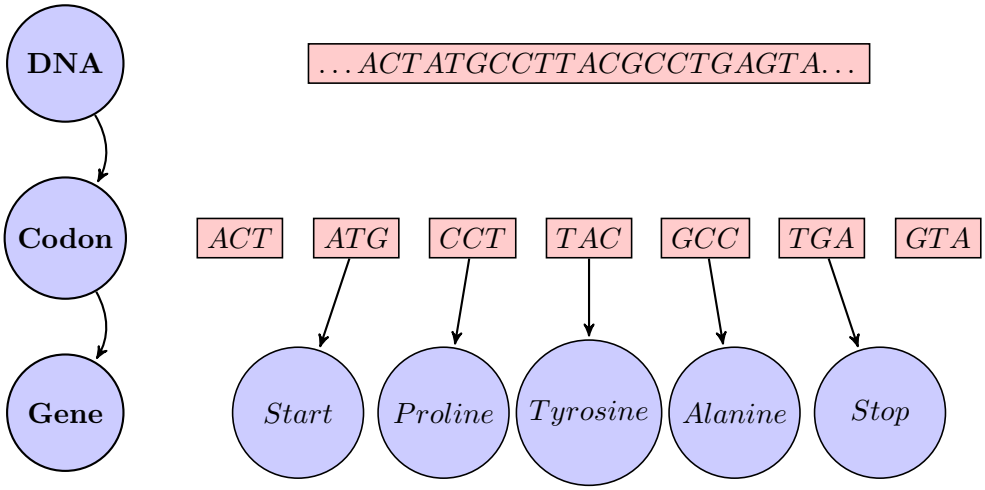


Figure 2.2: The organization of information inside DNA.

genome of Amoeba dubia, a single cell amoeba, is one of the largest known, containing about 670 billion base pairs.

One difficulty that arises when analyzing DNA fragments is the problem of deciding at what point and in which direction to start interpreting the DNA fragment as codons. Six open reading frames are possible: three reading the DNA string forward, three reading the DNA string backwards. A naive approach would be to simply use the length of genes in a frame as an indicator of its probability of correctness: there are twenty one codon types (twenty amino acids and the stop codons); hence, on average, one can expect every twenty-first codon to be a stop codon. Since genes are often longer, a lower length threshold could be used to prune unlikely candidates. One problem is that if this guideline is used, short genes cannot be found. A more elaborate method also takes into account the relative frequency in which codons are found in already discovered genes: some amino acids are encountered more often than others. An open reading frame is then more likely to be correct if it contains more probable codons. An even more elaborate method would also consider the frequency of pairs of consecutive codons.

Whereas genes contain information about an organism, proteins are the workhorses that actually realize an organism's functioning. Proteins are complex organic compounds that carry out many essential tasks: amongst others, activity of proteins controls signaling of functionality within and between cells; they also form the basis of and give structure to an organism's major components, such as hair or skin. As stated above, proteins consist of strings of amino acids, coded for by codons. Proteins are characterized by such sequences of symbols. Although a protein can be simplified to this one dimensional string, protein functionality is largely determined by the three dimensional structure this string folds into. Protein folding is therefore an important discipline within bioinformatics.

The discipline within bioinformatics that concerns itself with comparing sequences of DNA is called sequence analysis. Sequence analysis has many applications. As an example, genes can be compared to each other to discover similarities in functionality;

First Base	Second Base: T	C	A	G	Third Base
T	Phenylalanine	Serine	Tyrosine	Cysteine	**T**
	Phenylalanine	Serine	Tyrosine	Cysteine	**C**
	Leucine	Serine	Ochre(STOP)	Opal(STOP)	**A**
	Leucine	Serine	Amber(STOP)	Tryptophan	**G**
C	Leucine	Proline	Histidine	Arginine	**T**
	Leucine	Proline	Histidine	Arginine	**C**
	Leucine	Proline	Glutamine	Arginine	**A**
	Leucine	Proline	Glutamine	Arginine	**G**
A	Isoleucine	Threonine	Asparagine	Serine	**T**
	Isoleucine	Threonine	Asparagine	Serine	**C**
	Isoleucine	Threonine	Lysine	Arginine	**A**
	Methionine(START)	Threonine	Lysine	Arginine	**G**
G	Valine	Alanine	Aspartic acid	Glycine	**T**
	Valine	Alanine	Aspartic acid	Glycine	**C**
	Valine	Alanine	Glutamic acid	Glycine	**A**
	Valine	Alanine	Glutamic acid	Glycine	**G**

Table 2.1: A codon to amino acid translation table.

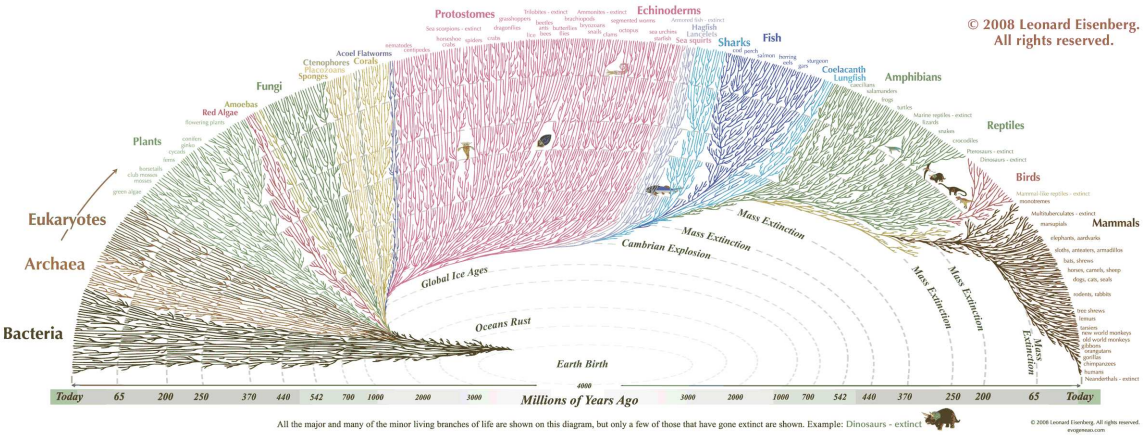


Figure 2.3: A phylogenetic tree (taken from [20]).

genomes of individuals within the same species can be compared to each other; similar genes can be combined into a profile, describing possible base pairs at each position of the gene and their likelihood. Such a profile can then be used to compare parts of one species' genome to other species' genomes, in order to estimate their evolutionary relatedness. This way, a phylogenetic tree can be composed that groups more related species closer to each other (see Figure 2.3).

2.2 Sequence Analysis

Sequence analysis, or sequence alignment, compares sequences to each other in order to discover similarities between them. Sequence analysis techniques can be applied to many different types of data: it can be used to compare literary works, to find similarities in musical compositions, to analyze financial data, or even to assist with speech recognition.

In bioinformatics, biosequence analysis is utilized to discover regions of similarity between strands of DNA or between the codings for proteins. For example, it can help establish an evolutionary or functional relationship between genes. Large similarity between genomes is thought of as close evolutionary relationship. More sensitive analysis techniques allow for the detection of more remote relationships.

Being the main topic of this thesis, sequence analysis is now discussed in more detail. A formal definition is given first; then, different types of sequence analysis are discussed along with techniques to obtain such alignments.

2.2.1 Formal Definition

Consider a sequence as consisting of a string of symbols from an alphabet. The objective is to find the best match between strings v and w, best being defined as having a highest possible score. When comparing symbols of each string, the following operations are available to create a best possible match: match the current symbols to each other (the scoring matrix s assigns a positive or negative score to the match, depending on the relatedness of the symbols), or delete either one of the current symbols (this assigns a negative score). Algorithm 2.1 shows the formulation of the sequence alignment problem by Smith and Waterman [39], allowing for arbitrary insertions and deletions (or indels).

Algorithm 2.1 The Smith-Waterman algorithm

INPUT:

strings v ($\|v\| = m$), w ($\|w\| = n$); scoring matrix s; gap penalties W.

RECURSION:

$$H_{i,0} = 0, 0 \leq i \leq m$$
$$H_{0,j} = 0, 0 \leq j \leq n$$
$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s_{v_i,w_j} & \text{if } v_i \text{ and } w_j \text{ are associated} \\ H_{i-k,j} - W_k & \text{if } v_i \text{ ends a deletion with length } k \\ H_{i,j-l} - W_l & \text{if } w_j \text{ ends a deletion with length } l \\ 0 & \text{no similarity up to } v_i \text{ and } w_j \end{cases}$$

OUTPUT:

an alignment of v, w with maximum score.

The algorithm finds segments with high similarity by filling matrix H. The value of $H_{i,j}$ can be interpreted as being the maximum similarity between two segments ending in v_i and w_j. The best alignment is then found by locating the highest value in H and

working backwards, each step selecting the biggest decrease until encountering an entry in H with a value of zero. Instead of assigning a fixed penalty for an insert or gap, W_k is used to determine the cost of an indel with length k, to account for affine indels. Gaps caused by an evolutionary mechanism are often longer than one base pair. Hence, the cost is split into a gap opening and gap extension penalty.

The scoring matrix s contains the score for a match or mismatch between symbols. In the case of protein sequences, the table contains the scores for substitutions between amino acids (also called residues). These scores are based on biological evidence: some mutations are more likely to occur than others. Harmful mutations are assigned a negative score; substitution between amino acids with similar biochemical properties are assigned a positive score. The choice of scoring in the similarity matrix determines the extent of substitutions the sequence search is likely to discover. As an example, Asparagine and Glutamine are alike, both are amino acids with acidic side chains and are very hydrophilic. Hence, they are likely to be substituted for each other. Alanine and Valine are both hydrophobic amino acids. Hence, these are unlikely to be substituted for Asparagine or Glutamine.

Two widely used scoring matrices are PAM (Point Accepted Mutation) [13] and BLOSUM (Blocks Substitution Matrix) [21]. BLOSUM uses observations of mutations in local alignments of highly conserved regions. For example, BLOSUM64 is based on sequences that are at least 64% identical. PAM is based on an explicit evolutionary model and considers replacements on the branches of a phylogenetic tree. PAM1 means that there is a 1% chance for each symbol in a sequence to have changed. Of course, symbols may have mutated several times or may not have changed at all. Hence, PAM100 does not imply a wholly different sequence. The two types of matrices are comparable: for example, BLOSUM64 is similar to PAM120.

2.2.2 Classification of Sequence Alignment

There are various ways in which sequence alignment can be performed. This section distinguishes between local and global sequence alignment and between pairwise and multiple sequence alignment (MSA).

Global sequence alignment aligns sequences over their entire length, whereas local sequence alignment finds the optimal alignment between parts of the two sequences. The Smith-Waterman algorithm, stated above in Algorithm 2.1, is a local alignment algorithm and is derived from the Needleman-Wunsch algorithm [32], which is used for global alignment. The two are similar, but by setting negative entries in the matrix H to zero, the Smith-Waterman algorithm allows alignments to start at arbitrary points in the sequence.

Local sequence alignment and global sequence alignment have different applications. If two protein segments are thought to be of similar length, global alignment should be used, since it will match them from end to end, even though parts might be dissimilar. However, since genes are often similar only over short sections, local alignment is mostly used.

Another categorization of sequence alignment is in the number of sequences that are to be aligned simultaneously. Pairwise alignment seeks to align two sequences together;

multiple sequence alignment attempts to align more than two sequences to each other. Naturally, the latter is far more computationally intensive. In order to find the optimal solution, every ordering of aligning sequences together should be investigated, leading to exponential growth in the amount of computations. The next section investigates both optimal and heuristic techniques for sequence alignment.

2.2.3 Sequence Alignment Techniques

The previous paragraph shows sequence alignment techniques can be classified in various ways. On the one hand, local and global alignment are closely related: usually variations of the same technique can be used to perform either kind of alignment. On the other hand, the difference between pairwise alignment and multiple alignment *is* significant. Pairwise alignment is a tractable problem. Therefore, obtaining the optimal solution is still feasible. Multiple sequence alignment, however, is intractable [25]. Therefore, for all but the most simple multiple alignments heuristic algorithms should be used.

Several solving methodologies exist for pairwise alignment. The Smith-Waterman and Needleman-Wunsch algorithms, discussed above, utilize a dynamic programming approach to calculate an optimal solution (according to their scoring parameters). Dynamic programming splits a complex problem into many less complex ones; results obtained by solving the sub-problems are reused, removing the need to recalculate certain steps multiple times. But even though such optimal methods can be used for pairwise alignment, in practice even here speed is preferred over accuracy. BLAST [8] and FASTA [36], two popular sequence alignment tools, both use heuristic methods to rapidly obtain alignments by using word based methods. In a word based method, from the initial sequence a list with words of length k is created. These words are sub-sequences of the original sequence. Then, all the words on the list are checked against the target sequence. Some words on the list will have a high scoring against the sequence. Then, it is tried to extend these hits to other hits by including residues before and after the segments. Another technique is based on Hidden Markov Models (HMMs). HMMs are explained in Chapter 3.

Multiple sequence alignments can be seen as an extension to pairwise alignments, though they are much more computationally intensive. The equivalent dynamic programming approach to algorithm 2.1 that creates an MSA would require an n-dimensional variant of matrix H to be filled. This would require exponential running time in the number of sequences to be compared. Problems that scale exponentially in input size are called NP-complete and are known to be solvable only for the most trivial of cases. Hence, the use of heuristic methods is mandatory.

One such heuristic method to produce MSAs is called progressive alignment. First, all sequences are compared to each other in pairs. Then, a guide tree is composed that indicates the order in which alignments are added to the MSA. Starting with the sequences that have the highest alignment score for each other, sequences are successively added to this MSA, until all sequences are contained in it.

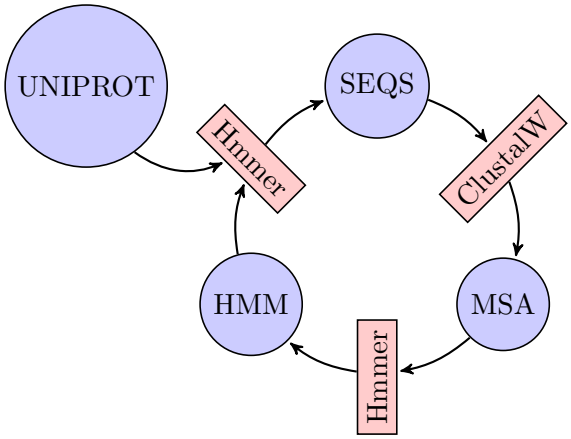


Figure 2.4: An example biosequence analysis workflow.

2.3 Biosequence Analysis Software

Over the years, many biosequence analysis tools have been developed, allowing biologists to take advantage of the computing power offered by modern day high performance computing. The scale in which sequence alignment and analysis is performed today would not have been possible without the availability of huge amounts of computation resources. Different tools have been developed, each specialized to their particular branch of sequence alignment.

Well-known tools are BLAST [8], ClustalW [41], FASTA [36] and HMMER [18]. Trading accuracy for speed, all of them are based upon heuristic alignment techniques. BLAST, FASTA and HMMER are pairwise sequence alignment tools. Two sequences are aligned to each other, producing a score that, if it exceeds a certain threshold, signifies a relationship between the two. Commonly, one protein sequence or model is compared against a database of sequences. BLAST and FASTA use word-based methods. Compared to FASTA, BLAST only evaluates the most significant word matches, rather than every word match. This makes it the fastest sequence alignment tool, being about fifty times faster than dynamic programming algorithms based on Smith-Waterman. HMMER, this thesis' main focus, is based on Hidden Markov Models. The HMM acts as a model for a family of proteins, and the basic idea is to find out if a certain protein sequence is related to this protein family. The following section describes HMMER in more detail. Finally, ClustalW is a tool used to create a multiple sequence alignment of DNA or protein sequences.

Figure 2.4 offers a scenario in which the above tools are used in conjunction. Starting at the top, from a set of related protein sequences, ClustalW is used to produce a multiple sequence alignment. Out of this MSA, HMMER creates a protein profile HMM. Finally, HMMER uses this profile to search through a protein database, in order to find related sequences. With this larger set of related protein sequences, the cycle can be started again.

2.3.1 HMMER

HMMER is a freely available open source family of tools often used in biosequence analysis, aimed specifically at protein sequence analysis. Included as a part of the SPEC CPU2006 benchmark [5], it is representative for a large group of biosequence applications. In contrast to BLAST and FASTA, HMMER uses profile Hidden Markov Models to analyze families of proteins. Its specific HMM architecture is explained in more detail in Section 3.3.1. In contrast to the Smith-Waterman and Needleman-Wunsch approaches, in the HMM approach the model instead of the algorithm itself determines whether local or global alignment is used, both with regard to the protein sequence as well as with the model. Also, the model determines if multiple domain hits per sequence are allowed. The main advantage of using HMMs over the dynamic programming methods is that gaps are modeled in a systematic way. For example, in algorithm 2.1 W_K is used to assign a score to gaps. A drawback all these techniques share is that they all assume that positions can be scored independently: higher-order correlations between consecutive residue positions in the protein sequences cannot be modeled.

Some of the programs in the HMMER family are: hmmsearch, which aligns sequences from a database with an HMM profile using the Viterbi algorithm in order to find related sequences; hmmpfam, which compares a single sequence to an HMM database; hmmbuild, which builds a profile HMM from a multiple sequence alignment; and hmmcalibrate, which calibrates a profile HMM for more accurate E-values. E-values estimate how many hits can be expected by chance when comparing a profile to a database of a certain size filled with random sequences by performing Monte Carlo simulations. The fundamentals on which HMMER is based are explained in Chapter 3.

2.4 Biological Sequence Databases

Biologists around the world, from small universities to large research centers, are performing biosequencing of DNA and proteins. The more information that becomes available, the better patterns between proteins and protein families become visible. These large quantities of sequence information are stored in freely accessible databases. For this thesis' purposes, two classes of databases are of importance: databases containing protein sequences and databases containing models of protein families.

Sequence databases contain vast amounts of protein or DNA sequences. Moreover, the size of these databases is growing exponentially. Figure 2.5 shows the growth of the European Molecular Biology Laboratory (EMBL) nucleotide database. There are two reasons for this growth: ever growing interest in the field of biosequencing and bioinformatics, and the fact that the sequencing process itself is becoming ever faster. This makes the development of newer and faster methods for analyzing the data in these databases crucial.

UniProt [27], the Universal Protein Database, is an international effort to create a centralized freely accessible database on proteins. It is the largest protein database and is part of the EMBL. The EMBL exchange data between the DNA DataBank of Japan and GenBank at the National Center for Biotechnology Information on a daily basis, under the umbrella of the International Nucleotide Sequence Database Collaboration.

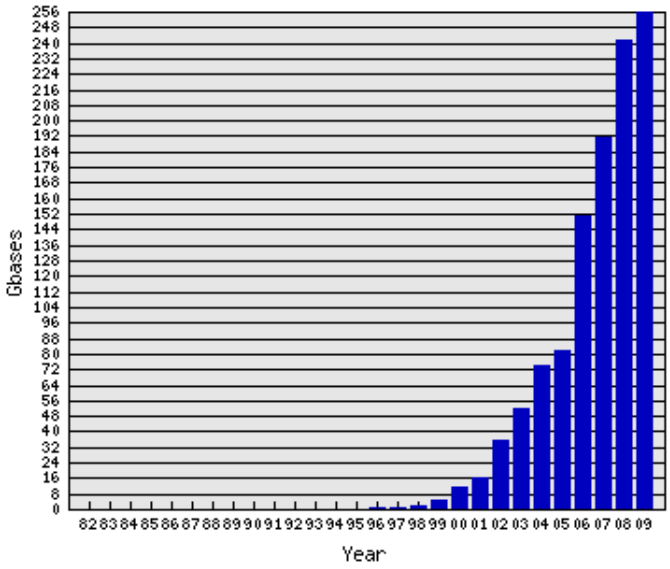


Figure 2.5: EMBL database growth (taken from [2]).

The UniProt Knowledge Base is divided into two parts: UniProtKB/TrEMBL, which contains unreviewed protein sequences and UniProtKB/Swiss-Prot, which contains manually annotated proteins.

The other database type is the protein *structure* database, with descriptions of protein models. These models can then be used to search through protein sequence databases in order to find related sequences. Pfam [40] is a database containing protein descriptions based on Hidden Markov Models. These models can be used with HMMER. Pfam-A is the high quality, manually curated part of Pfam. Pfam-B is the larger automatically generated section, containing potentially less accurate models. This database can be used when no matches against Pfam-A are found, to increase coverage.

2.5 Summary

This chapter provided the reader with an introduction to the field of bioinformatics and to biological sequence analysis in particular. The central dogma of microbiology shows the relation between DNA and proteins. Sequence analysis compares fragments of DNA or protein sequences to each other. Alignment can be categorized according to locality, global or local, and multiplicity, pairwise or multiple sequence alignment. Optimal methods based on dynamic programming exist, but for speed reasons, heuristic methods are usually preferred. Out of all the existing biosequence tools, FASTA, BLAST (based on k-word methods), and HMMER (based on Hidden Markov models) are the most popular ones for pairwise alignment. ClustalW is a tool for MSA. Sequence databases containing DNA or protein sequences such as UniProt are growing at an exponential pace. As a result, analysis of the resulting data becomes an ever greater challenge.

More information on bioinformatics can be found in the excellent introductory book on the subject by Jones and Pevzner [24].

3 Profile Hidden Markov Models

Chapter 2 introduced a biosequence analysis tool called HMMER, which is the main focus of this thesis. HMMER's search mechanism is based on probabilistic theory and uses Hidden Markov Models to represent protein models, against which sequences are aligned. To this end, it utilizes a specific architecture, called the Plan7 Profile Hidden Markov Model. This chapter presents the theoretical background behind the model used in HMMER, studying the mathematical fundaments on which the application is based. The development and use of such pattern recognition models originates from research in domains such as speech recognition [38].

Section 3.1 provides a short introduction to Markov models. Section 3.2 explains the concept of Hidden Markov Models and the Viterbi algorithm. Section 3.3 discusses profile Hidden Markov Models, as used in bioinformatics applications and discusses the specific architecture used by HMMER, the Plan7 Profile Hidden Markov Model. Then, E-values and bit scores are discussed, which aid in the interpretation of the significance of an alignment. Section 3.4 concludes the chapter with a summary.

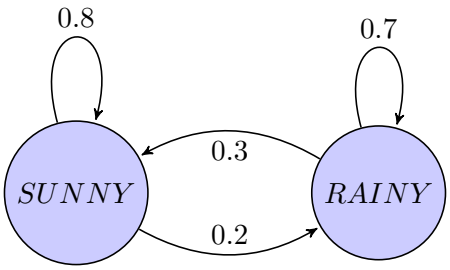
3.1 Markov Models

One way of categorizing models of systems is to distinguish between deterministic systems and probabilistic systems. In a deterministic system, a given starting condition will always result in the same outcome. This outcome is defined solely by the system inputs and parameters. Conversely, in a statistical system identical starting conditions do not necessarily result in the same outcome: outcomes are inherently uncertain. A priori, the outcome of a given starting condition is unknown, although assumptions about the likelihood of certain outcomes can usually be made.

A Markov model then, being a simple statistical model, is probabilistic in nature. It is defined by a set of states and a set of probabilities for transitioning from one state to another. At any given time, the system is in one of its states. For a discrete Markov model, every time step the system transitions from the current state to the next state (to another one or possibly to the current one), according to a set of transition probabilities A. A Markov model is characterized by its n states S_n and its $n \times n$ transition probability matrix A governing transitions between these states. Hence, observing a Markov model over time shows a sequence of observed states O, for example:

$$O = \ldots, S_1, S_5, S_2, S_2, S_3, \ldots \tag{3.1}$$

The defining characteristic of a Markov model is that it exhibits the Markov property: transitions from the current state to the next state are dependent only on a fixed number



Set of States:

$$S = \begin{pmatrix} Sunny \\ Rainy \end{pmatrix}$$

Transition Probabilities:

$$A = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix}$$

Figure 3.1: A simple Markov model of the weather.

of previous states. In a first order Markov model, the next state Q_{t+1} is dependent only on the current state Q_t. The following equation states that the chance for a certain transition $Q_{t,t+1}$ is only dependent on the previous state in a more formal manner:

$$P(Q_{t+1} = S_i | Q_t = S_j, Q_{t-1} = S_k, \dots) = P(Q_{t+1} = S_i | Q_t = S_j) \tag{3.2}$$

For static Markov models, the transition probabilities from each state to each other state are given by the $n \times n$ transition matrix A, where:

$$a_{j,i} = P(Q_{t+1} = S_i | Q_t = S_j) \tag{3.3}$$

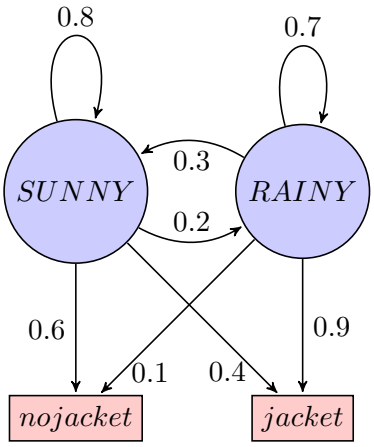
$$\text{with} \qquad a_{j,i} \geq 0 \quad \text{and} \quad \forall_j : \sum_{i=1}^{n} a_{j,i} = 1$$

As an illustration, consider Figure 3.1, which models the weather of consecutive days in a very simple manner. The model consists of just two states: a day is either rainy or sunny. Moreover, the weather of any given day is assumed to be dependent only on the weather of the previous day. Hence, this system can be modeled as a first order Markov model. The labels of the arrows between states show the transition probabilities. From the model, it follows that the weather is stable: if it is sunny on a certain day, it is 80% likely to be sunny the next day as well; if it rains, it is 70% likely to rain the day after as well.

3.2 Hidden Markov Models

Hidden Markov Models (HMMs) form a specialized class of Markov models. In a HMM, the state of the model cannot be observed directly. Instead, states emit symbols and those emitted symbols are observed. Based on this sequence of emitted symbols and the parameters of the model, it is possible to make assumptions about the probable sequence of states that occurred. Thus, in contrast to a normal Markov model, the observed sequence O is not a sequence of states, but a sequence of emitted symbols.

An HMM can be thought of as to act as a generator for an infinite number of sequences. The model acts as a probability distribution over this infinite number of



Emission Alphabet:

$$V = \begin{pmatrix} jacket \\ nojacket \end{pmatrix}$$

Emission Probabilities:

$$B = \begin{pmatrix} 0.6 & 0.4 \\ 0.1 & 0.9 \end{pmatrix}$$

Figure 3.2: A Hidden Markov Model of the weather.

sequences. Since the total probability of all sequences sums to one, increasing the likelihood of some sequences must result in a corresponding decrease in likelihood of other sequences. Starting from an initial state, a state is chosen to transition towards, according to the transition probabilities A. Then, based on that state's emission probabilities E, a symbol is emitted. These steps are repeated until an end state is encountered.

Formally, the HMM is characterized by the original set of Markov parameters - the n states S_n and the $n \times n$ transition probabilities matrix A -, extended with the alphabet of emitted symbols V_m and the $n \times m$ probability distribution matrix of emitted symbols per state B.

As an illustration, consider the model shown in Figure 3.2. Again, a system of the weather for consecutive days is modeled. However, since in this example the model is a HMM, emission variables and probabilities are added. These can be interpreted as follows: person X will either wear a jacket or not, depending on the weather. If it rains, X will almost certainly wear one (a 90% emission chance). If it is sunny, X usually does not (only a 40% emission chance). Let person Y be unable to observe the weather directly (the HMM premise), but let him be able to observe whether X is wearing a jacket or not. Also, Y knows the likelihood that X wears a jacket based on the weather conditions and he is familiar with the general behavior of the weather. In other words, he knows the system's parameters. Then, based on his observations of X, he will be able to make some assumptions about the weather conditions.

Three canonical problems are associated with HMMs:

- Given are an HMM with known parameters S, A, V, B and a sequence of emitted symbols O. What is the probability of the occurrence of output sequence O? Formally: compute $P(O|S, A, V, B)$. The forward/backward algorithm is a method to solve this problem [38]. In biosequence analysis terms, this problem produces an estimate of how well sequence O matches the protein model.

- Given are an HMM with known parameters S, A, V, B and a sequence of emitted symbols O. What is the most likely sequence of states leading to this output? Formally: compute a sequence of states $S_a, S_b, S_c, \ldots$ that maximizes $P(O|S, A, V, B)$.

This sequence can be obtained using the Viterbi algorithm [42]. This is precisely what HMMER does: it produces the most likely alignment of states of the protein model to the protein sequence. The Viterbi algorithm will be discussed in more detail in Section 3.2.1.

- Given is an HMM with states S and emitted variables V; also given are k output sequences O_k. Find the parameters A, B of the HMM that maximize the likelihood of the model producing sequences O_k. This problem can be solved using the Baum-Welch or Baldi-Chauvin algorithm. In bioinformatics terms, this can be used to create a protein family model based on a set of related sequences.

3.2.1 The Viterbi Algorithm

The Viterbi algorithm can be used to find a sequence of states that maximizes the chance for a given sequence of output symbols. Since HMMER utilizes the Viterbi algorithm to align sequences to HMMs, the Viterbi algorithm is now discussed in more detail. The exact model HMMER uses will be explained in Section 3.3.1; this section describes how the Viterbi algorithm obtains the most likely sequence of states for a given HMM and observed sequence.

Algorithm 3.1 The Viterbi algorithm

INPUT:
HMM with parameters S, A, V, B,
observed sequence O,
initial state distribution P.

INITIALIZATION:
$k = 0$,
$\forall_{s \in S} : x_s^0 = S_s$,
$\forall_{s \in S} : p_s^0 = P_s$.

WHILE O_k:
$\forall_{i \in S} : p_i^{k+1} = \max(\forall_{j \in S} : p_j^k \cdot A_{j,i} \cdot B_{j,O_k})$,
add corresponding state to survivor path x_i^{k+1}.

OUTPUT:
sequence x of states $S_a, S_b, \ldots, S_z$, maximizing $P(O|x)$.

Algorithm 3.1 shows the basic operation of the Viterbi algorithm. The algorithm is recursively defined, based on the following idea: a state sequence x with length k can end up in any of the n states in the state space. For each of these n states, there is one state sequence that maximizes $P(O|x)$, the survivor path. Let x_i^k denote the survivor path to state i with length k. Any longer optimal state sequence *must* go through one of these survivor paths. Hence, the optimal sequence with length $k+1$ can be determined by solely considering each of the survivor paths with length k and the corresponding transition

$k \rightarrow k+1$. Recall that $A_{j,i}$ is the state transition probabilities matrix $P(S_i|S_j)$ and that B_{j,O_k} is the emission probabilities matrix $P(O_k|S_j)$. Therefore, it follows that the algorithm just has to keep track of the survivor path to each state S_n and the probability of said path. This probability is denoted by p_i^k. When the entire observed sequence is processed, the state sequence that maximizes $P(O|x)$ is given by the survivor path with the highest probability.

Analyzing the algorithm to determine time and space complexity gives the following results: the time the algorithm takes is linear in the length of the observed sequence ($\|O\|$), since the algorithm steps over each symbol. Every step, for each state the new survivor path has to be calculated, by considering the path from every other state. Hence, in the worst case when the states are fully connected, the algorithm scales quadratically in state count. Thus, time complexity is of order $O(\|O\| \cdot n^2)$. With regard to space complexity, for each state the survivor path is tracked, and the corresponding probability. This leads to $O(\|O\| \cdot n)$ space complexity.

Two concluding remarks: since the probabilities are multiplied with each other, they quickly become very small. In order to prevent round-off errors, it is custom to store them in logarithmic format (log-odds). Also, the forward/backward algorithm, which computes the likelihood of a given output sequence (recall Section 3.2), is very similar to the Viterbi algorithm. It can be obtained by taking the sum instead of the maximum during computation of the survivor paths. Hence, both are usually calculated at the same time.

3.3 Profile Hidden Markov Models

Proteins are an important part of molecular biology, as they constitute the elements that perform a cell's functioning. Although proteins have a three-dimensional shape, their basic structure can be described by its constituent string of amino acids. The idea that similarly structured proteins have similar function has led to the comparison of protein descriptions to discover related families. In Section 2.2.3, various techniques for performing sequence analysis have been discussed. One probabilistic method is based on Hidden Markov Models. In bioinformatics, such models are often called profile Hidden Markov Models, or pHMMs. This nomenclature stems from the fact that a HMM is used as a representation of the profile of a protein.

Aligning a protein sequence to a profile model amounts to matching the sequence of amino acids in the protein sequence to a sequence of emitted symbols, and thus to a sequence of states in the profile model. Compared to dynamic programming or k-word techniques, a large advantage of using a pHMM for sequence analysis is that it allows for the natural and systematical modeling of gaps. Other forms of sequence alignment require manual setting of these parameters. Because the profile is trained on a multiple sequence alignment, every residue position contains its own scores for substitution, insertion and deletion. To prevent over-fitting of these parameters to a specific MSA, pseudo-counts with observed amino acid frequencies are often combined with the parameters.

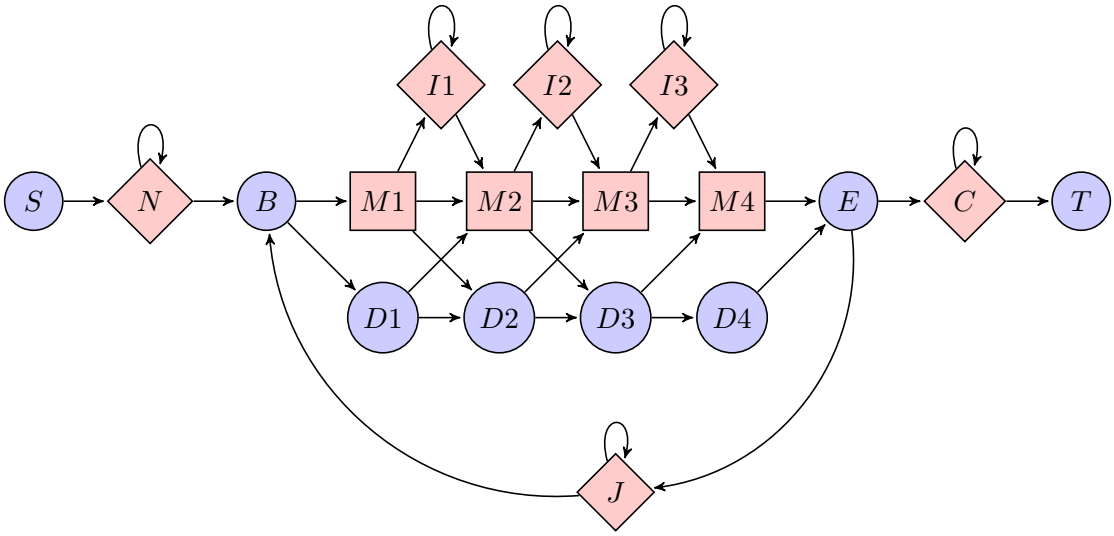


Figure 3.3: The Plan7 Profile HMM Architecture.

3.3.1 The Plan7 Profile HMM Architecture

The Plan7 Profile HMM Architecture is the model used by HMMER to describe protein models [18]. Figure 3.3 shows the basic layout of the model, using a length-4 motif as an example. The diamond and square states are emitting states, the circles non-emitting.

The main part of the model consists of Match, Delete and Insert states. A M/D/I-triple is called a node. Match and Insert states are able to emit amino acid symbols, Delete states emit no symbols. A Match state signifies a match between a particular position in the protein sequence and the HMM. Insert states are used to allow for gaps in the protein sequence as a result of evolution. Delete states let the protein sequence skip over some part of the HMM. Together, these states constitute the data dependent part of the model.

The Plan7 Architecture also contains extra states, which control the way in which alignment is performed. Global and local alignment is supported, as are multiple hits against the profile in one sequence. The N-state allows for random symbols in the protein sequence to precede the profile, the C-state for random symbols afterwards. If the loop back from the E-state to the B-state via the J-state is allowed, multiple hits are possible. The J-state allows for symbols in between hits. Finally, the S-state is the entry point of the HMM, the T-state the end.

Section 3.2.1 discussed the Viterbi algorithm. The time complexity of the algorithm was determined to be $O(O \cdot n^2)$. The algorithm is linear in the sequence size and quadratic in the model size, since for every symbol in the sequence, the path from every state to every other state needs to be taken into account. However, this worst case time complexity only occurs if the graph is fully connected. The specific form of the Plan7 architecture shows that the progression through the graph follows a linear path through the nodes, from left to right. Hence, in this case, time complexity of the Viterbi algorithm is just on the order of $O(O \cdot n)$.

3.3.2 Bit Score and E-value

Creation of an alignment between a protein model and a sequence by HMMER produces three results: the actual alignment between the sequence and the model, an E-value and a bit score. These last two help interpret the statistical significance of the alignment. Good homologues will have a low E-value and a high bit score.

The bit score measures how well the sequence matches the model. It is a log-odds score based on the probability the sequence matches the model compared to how well it matches the null model, which is a model based on random sequences. A positive log-odds value indicates that the HMM is a better fit than the null model. The bit score S is given by the following equation:

$$S = log_2 \frac{P(seq|HMM)}{P(seq|null)} \tag{3.4}$$

When comparing a model to a random sequence database with a certain size, purely by chance a number of hits will occur, each with a certain bit score. The E-value indicates this number of false positives. It is calculated based on the bit score. Hence, a lower E-value is better, as it means that the sequence is unlikely to be caused by chance alone. Empirically, hits with an E-value of 0.1 can be trusted to be significant hits. Also, note that the E-value is dependent on database size! The bigger the database grows, the higher the bit score of a sequence must be to be counted as a significant hit.

A more detailed explanation on E-scores and bit values can be found in [17].

3.4 Summary

This chapter discusses the mathematical fundaments of the HMMER application, which is based on Hidden Markov Models. Markov models are simple probabilistic models. Their fundamental characteristic is the Markov property, which states that state transitions are only based on the previous x states. In a Hidden Markov Model, the actual state is invisible; only emitted variables can be observed. The Viterbi algorithm is used to discover the most likely path that generates a specific sequence of emitted variables. In bioinformatics, Markov models are usually called profile Hidden Markov Models, since they are used to model the profile of a protein sequence. The specific architecture of HMMER is called the Plan7 Profile HMM architecture. The data dependent part consists of M/I/D states, other states in the model control its alignment behavior, allowing for global or local alignment and multiple hits. An alignment between a sequence and an HMM has a bit score, which measures how well the sequence matches the model, and an E-value, which indicates how many hits with identical bit score are expected to be generated by chance alone for a random test set of similar size.

Computer Architecture Trends 4

Over the last few years a trend has become apparent: even though Moore's law still holds, providing computer architects with continuously increasing transistor budgets (albeit at a slower pace), single threaded performance growth appears to have come to a near standstill. Instead of improvements to single threaded performance, the additional transistor budget has been spent on putting multiple cores on a single chip. This chapter explains the causes and effects of this trend. Section 4.1 describes the reasons for and implications of the paradigm shift from single-core to multi- and many-core computing. Section 4.2 briefly discusses different types of parallelism. In Section 4.3, issues with parallelism are discussed. Section 4.4 illustrates how computer architects are coping with the aforementioned issues by discussing the Cell Broadband Architecture, a state-of-the-art microprocessor and our implementation architecture. Section 4.5 concludes the chapter with a summary.

4.1 Paradigm Shift to Parallel Architectures

In the recent years, parallelism seems to have taken flight. All major semiconductor companies have embraced the practice of adding additional cores to their processors as a means to increase performance. Traditional semiconductor companies are scaling up the core count of their processor design. Intel, the world's largest semiconductor company, ships more multi-core than single-core processors since the third quarter of 2006. Furthermore, the number of processors on a single chip is expected to increase ever farther, resulting in processors with thousands of cores on a chip, or *many-core* processors. An example is Intel's Terascale initiative [4]. Graphics processor design follows a similar trend. These traditionally very parallel but fixed function designs are becoming increasingly more general purpose, an example is Nvidia's CUDA, which attempts to unlock the vast floating point capabilities of their GPUs to software engineers. [33]. The convergence of these trends seems to lead to processor designs with very many simpler cores.

What is the underlying cause for this shift? Parallel computing itself is not new: High Performance Computing is a longstanding research discipline. To obtain the highest performance possible, early supercomputers already contained dozens or more processors. The reasons for parallel computing to become a mainstream phenomenon are different. There are many reasons for this shift from single-core to multi-core processors and for the trend towards parallel architectures in general. Many problems are easy to state and execute in a parallel manner (although many also are not). Moreover, there will always be need for additional computation resources and parallel systems will naturally always be faster than single processor systems. However, the main argument in favor of

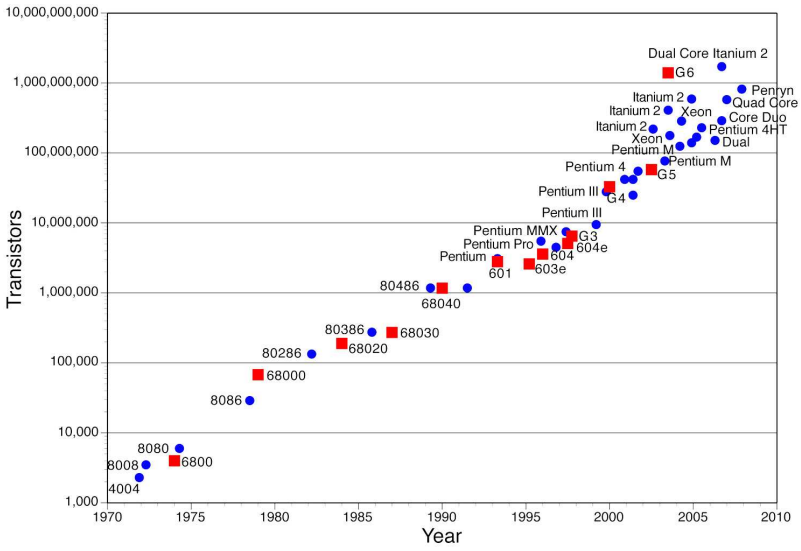


Figure 4.1: Chip transistor count over time (taken from [34]).

multi-core architectures is the fact that increasing sequential computing performance is becoming ever more difficult: the shift to parallelism is more of a necessity.

Moore's law [31] states that the number of transistors on a chip roughly doubles every two years. Figure 4.1 displays the exponential growth in transistor count of modern processors. A relationship exists between transistor count and processor performance: microprocessor performance is determined by the product of clock rate and number of instructions executed each clock cycle (IPC). In the past, this growing transistor budget was usually aimed at raising IPC through a variety of techniques, such as branch prediction, superscalar execution, out of order execution and large on-chip caches. However nowadays, three issues are preventing much further scaling of single-threaded performance [23]: frequency scaling limitations, power consumption constraints, and memory technology not keeping pace with the increase in computational capabilities. Therefore, a trend towards parallel architectures is emerging.

4.1.1 The Frequency Wall

Processor frequency is an important determinant to overall microprocessor performance. The clock rate of a chip is limited by the distance signals are required to travel each clock cycle. Hence, clock rate can be raised by decreasing the amount of work per clock cycle; i.e. the number of gates between buffers. One way to realize this is by increasing pipeline depth. However, although longer pipelines allow a higher clock frequency, they also cause detrimental side-effects to performance: instruction dependencies cause longer stalls since instructions must be processed by more pipeline stages, and pipeline flushes as a result of mispredicted branches become costlier. Current pipeline depths represent a fair balance, increasing them further leads to diminishing returns. As a result, microprocessor frequency is reaching a plateau: the maximum clock rate of modern mainstream processors has been hovering at the same level for a few years now.

4.1.2 The Power Wall

Besides processor frequency, the other determinant to microprocessor performance is its IPC. For a long time, the mantra in processor design was maximize performance at all costs. This resulted in the development and implementation of techniques such as branch prediction, superscalar execution, out of order execution and the inclusion of large on-chip caches. These techniques raise IPC considerably, but, when comparing their performance benefits to increases in power consumption, it becomes obvious that they often cost a disproportionate amount of transistors and power. High power consumption is becoming problematic in many situations. Desktop computers require large cooling solutions to keep processors from overheating. The widespread adoption of mobile devices, where battery longevity is an important factor, shift the focus more and more from absolute performance towards power consumption and performance per watt. And an even more important factor is the increasingly large cost of running data centers. To illustrate of the magnitude of this problem: in 2006, energy use of data centers amounted to 1.5% of total energy consumption in the U.S. [7].

Besides limiting IPC, power consumption also limits clock frequency scaling. One rule of thumb states that a 1% increase in clock rate causes a 3% increase in power use. Therefore, a processor consisting of multiple cores at a lower clock rate will have a higher performance per watt than a processor with a single higher clocked core, provided applications are able to take advantage of parallel processing (see Section 4.3).

4.1.3 The Memory Wall

A final bottleneck to processor performance is the fact that memory technology is unable to scale as fast as processor speed. Although memory bandwidth has risen considerably, latency has not been lowered significantly. Hence, from the perspective of the processor, access times to memory are becoming longer: the observed latency is increasing. To lessen the impact, processors are often equipped with elaborate memory hierarchies with multiple cache levels, or execution proceeds speculatively. However, such mechanisms take up a large number of transistors and consume much power. An alternative is to execute multiple threads simultaneously. Then, when one execution thread is stalled while data is fetched from main memory, another thread is allowed to proceed with execution.

4.2 Types of Parallelism

The above gives some insight in the difficulties with improving single threaded performance, making a case for more parallelism in computing. The different levels at which parallelism can be exploited [15] will now be discussed.

Lower forms of parallelism are handled within the microprocessor itself. These are transparent to the software engineer. This form is called hardware parallelism. Examples are bit-level and instruction-level parallelism. Bit-level parallelism increases the amount of information a processor can manipulate in a single clock cycle and is improved by enlarging the computer word size. Instruction level parallelism (ILP) operates on the

instruction stream, where instead of performing instructions one by one, multiple instructions of the program stream are executed simultaneously by reordering this stream and using multiple functional units inside the processor. Of course, such parallelism is limited by dependencies within the instruction stream.

Higher level forms of parallelism include data parallelism and task parallelism. Data parallelism divides a domain of data, on which certain calculations have to be performed, in smaller regions and executes calculations on those regions in parallel. An example: the parallel addition of individual elements in two arrays. Task parallelism can be utilized when multiple independent tasks are executed concurrently. These higher level types of parallelism, also called software parallelism, are visible to the programmer and must be manually exploited in order to take advantage of the offered parallelism.

An important aspect limiting the possible gains from parallelism is the granularity of the parallelism. Granularity defines the amount of communication and synchronization that is needed between tasks. Fine-grain parallelism requires continuous communication, coarse-grain almost none. Very fine grain parallelism includes instruction level parallelism; fine-grain parallelism includes data parallelism and parallelism within loops; medium-grain parallelism resides on the control level, such as parallel function calls; coarse-grained parallelism takes place on the task level. Finally, "embarrassingly" parallel tasks require no communication between processes and hence are the easiest to parallelize.

4.3 Issues with Parallelism

The previous sections demonstrate the reasons underlying the trend to more parallel architectures and have shown the various forms in which parallelism can manifest. However, widespread adoption of parallelism in applications still lags these developments in hardware. This section will illustrate some of the difficulties while writing parallel programs. First of all, there are limits on the gains to be expected from parallelizing software. Secondly, software engineering issues hamper widespread adoption.

4.3.1 Limits on Performance Gains

Amdahl's law describes the maximum speedup that can be obtained by the parallelization of a program [9]. It gives an upper bound for the maximum obtainable speedup of a program. The following equations show its basic form:

$$Speedup = \frac{(S+P)}{(S+\frac{P}{N})} = \frac{1}{(S+\frac{P}{N})} \tag{4.1}$$

$$Speedup_{N\to\infty} = \lim_{N\to\infty} \frac{1}{(S+\frac{P}{N})} = \frac{1}{S} \tag{4.2}$$

$$\text{with} \qquad S+P=1$$

Consider a program as consisting of a fraction P that is amenable to parallelization and a fraction S that is not. The program is executed on a system with N processors.

Then, the maximum realizable speedup from parallelism will eventually be limited by the non-parallelizable fraction: if the program were to be executed on a system with infinitely many processors, the execution time would be equal to the time required for the execution of just the serial fraction. Hence, the maximum speedup is $\frac{1}{S}$.

Amdahl's law implies that in order to make optimal use of many-core architectures, algorithms and programs have to be written in a manner amenable to parallelization. But it also implies that fast single-core execution capability is important for those parts of the program that cannot be parallelized. Therefore, a heterogeneous processor design with fast cores for non-parallelizable parts and more efficient cores for parallelizable parts of the code would represent a good trade-off between power and performance.

4.3.2 Parallel Software Engineering Difficulties

New processor designs with higher clock frequency and IPC allow for faster execution of existing software, without any intervention by the software developer. In contrast, processors with additional execution cores require software that is specifically written to take advantage of this. Part of the burden of extracting more performance from the hardware is shifted from the hardware engineer to the software engineer. Several problems make parallel software design more difficult than sequential software design. These are explained below.

In general, writing parallel software requires more effort than writing sequential programs. The reasons for this are readily apparent: in order to parallelize a program, additional steps must be undertaken compared to a single threaded program: parallelizable parts of the program have to be identified; these parts need to be broken down into smaller tasks, the tasks need to be assigned to individual processors and then communication and synchronization between tasks has to be implemented. Note that some programs might lack any such parts, making parallelization impossible!

Secondly, reasoning about the execution of a parallel program is more complex than reasoning about a sequential one: the state space of a running program is much larger. Whereas in a sequential program instruction are executed from a single program stream, parallel programs execute multiple streams simultaneously. External factors influence the order in which instructions from different streams are executed, making program order non-deterministic. This makes parallel software development more bug prone and bugs can be difficult to reproduce, especially if they only occur in exotic circumstances. As an example, consider when multiple processes require access to the same memory location. Simultaneous reading poses no problem. But when one or both of the processes need to write to the memory, the order in which events take place determine the program semantics. Hence, synchronization is required to ensure proper behavior, otherwise such race conditions can lead to inconsistent program states. However, synchronization schemes need to be implemented with care. Otherwise they can lead to deadlocks.

Portability is often another problem for parallel software. If a program is written with a certain architecture in mind, - for example requiring a fixed number of processors on which to execute or requiring a specific memory subsystem -, it cannot readily be executed on another architecture. Moreover, not all programs lend themselves to parallelization: if some section of a program must execute before another section, those two

tasks cannot be run in parallel. In such a case, a complete rewrite of the program using a different algorithm more amenable to parallelization might be the only option possible.

4.4 The Cell Broadband Engine

To illustrate the issues mentioned in this chapter, the Cell microprocessor is now discussed. The Cell is the implementation architecture for this thesis. It represents a radical paradigm shift in processor design. Developed by Sony, Toshiba and IBM, its design addresses the three main difficulties facing microprocessor engineers: the power wall, the memory wall and the frequency wall. This section presents a brief overview of the Cell's design and how its architecture deals with the aforementioned problems. Afterwards, peculiarities about development on Cell are discussed and a roadmap showing future iterations of the platform is shown.

4.4.1 Architecture Overview

The Cell Broadband Engine architecture is a radical departure from traditional modern microprocessors design. The floor plan shown in Figure 4.2 shows some of its important characteristics, the most obvious being its heterogeneous nature. The Cell contains nine cores: one traditional PowerPC core (the PPE) and eight Synergistic Processing Engines (SPEs). Table 4.1 lists some of the differences between both types of cores.

The PPE is a general purpose core, fully backwards compatible with the PowerPC instruction set extended with special VMX SIMD instructions. The PPE allows the Cell to execute all standard PowerPC software (albeit slowly). The PPE accesses main storage in a traditional manner, using load/store instructions to transfer data between the register file and the cached main storage. It is intended to run the operating system, coordinating the SPEs, and other general purpose tasks.

The SPEs are specialized processor cores, designed for streaming workloads such as games, multimedia applications or high performance computing. The SPEs have a very simple architecture, divided into two parts: the Synergistic Processing Unit and the Memory Flow Controller. These units operate in parallel, allowing for parallel computation and transfer of data. The SPU features a custom SIMD instruction set that is processed in-order with dual issue capability. The absence of branch prediction or caches makes its behavior deterministic. The MFC represents a radical break with conventional architecture models, because it explicitly parallelizes computation and the transfers of data and instructions. SPEs cannot access main memory directly, code and data must be transferred to the Local Store through DMA. This also implies that both code and data must fit in the 256kB local store.

Other notable features of the Cell are the high speed Element Interconnect Bus (EIB), which allows for high bandwidth, low latency communication between PPE, SPE, main memory and I/O, and the Memory Interface Controller (MIC) with a peak data rate of 25.6 GBps.

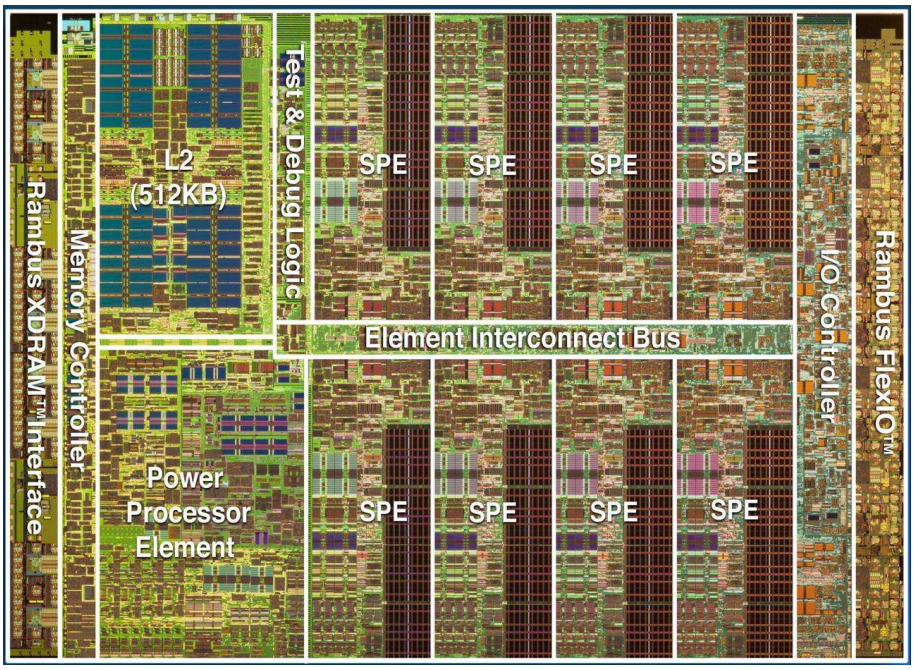


Figure 4.2: Cell Broadband Engine architecture (taken from [6]).

	PPE	SPE
Core Count	1	8
Function	General purpose tasks	Streaming workloads
ISA	PowerPC + VMX	Custom SIMD
Type	In-order dual issue	In-order dual issue
Registers	32x 64b (GPR) 32x 64b (FPR) 32x 128b (VMX)	128x 128b
Memory Access	Direct (cached)	Through DMA
On-chip Memory	32kB L1 + 256kB L2	256kB Local Store
Branch Prediction	Yes	No
Other	2-way SMT	

Table 4.1: PPE and SPE comparison.

4.4.2 Addressing the Three Walls

In the previous section the architecture of the Cell processor has been shown. Its design addresses the frequency, power and memory barriers, in order to attain high performance at relatively low power [26]. Its heterogeneous nature allows both types of processor core to be optimized for their respective use, allowing for a low power, high frequency design. The PPE is intended as a control-plane processor, the SPEs as data-plane processors. Figure 4.2 shows that the SPEs are, compared to the PPE, relatively small. But for certain workloads they are extremely fast. At 3.2 GHz, Cell has a peak performance

of 204.8 GFLOPS. In order to reach this peak rate, parallelism must be utilized on all levels. The SIMD-nature of both processor types allows for fine grained parallelism; coarse grained parallelism is achieved by running multiple threads on the various cores.

In order to address the memory wall problem, memory management has been made explicit. The SPE's Local Store reduces the dependence on low latency main memory access. Parallel to computations, the MFC should be used to stream in new code or data in the background. In effect, this setup trades latency for bandwidth. The combination of MFCs, EIB and MIC allow for very many concurrent memory accesses.

To illustrate the level of performance the Cell processor is able to attain, consider the Stanford Folding@Home project (FAH). FAH is a distributed computing project, where volunteers donate computing time to aid with protein folding calculations. The combined computation rate has peaked at a record five PetaFLOPS earlier this year. The computations the FAH kernel performs are very well suited to the streaming nature of the Cell. The Folding@Home client for the PLAYSTATION3, which contains the Cell processor, is an order of magnitude faster than a 2.66GHz Xeon processor [30]. About one tenth of the more than 330000 active clients constitute PLAYSTATION3 clients, however they contribute about a quarter of the total available TFLOPS [3].

4.4.3 Cell Development Issues

In order to obtain high performance out of the Cell processor, programs are required to have both good thread and data parallelism. Programs that contain both fine and coarse grain parallelism are able to fully utilize the SIMD nature of all nine cores. This fact, combined with what was shown in Section 4.3, shows that not all workloads are equally suited to the Cell: at the very least, many programs will require extensive rewrites.

The exotic nature of the SPE memory architecture, with its explicit nature and small Local Store, is another issue making software development more difficult. Software kernels must be split in small chunks in order to fit into this tiny amount of memory. Compare this to standard program development, where code and data size is no issue at all. It is much more similar to the programming of embedded devices. Some program types are suited well to the SPE's nature, others, such as branch-heavy or pointer-chasing applications, are not. These are best run on the PPE.

To simplify development on Cell, different programming models have been developed for Cell. These include the PPU centric and SPE centric model, the function offloading model, the device extension model, the computational acceleration model, streaming models, and the asymmetric thread runtime model.

4.4.4 Cell Platform Roadmap

The flexibility of the Cell platform is expressed in the roadmap of Figure 4.3. The heterogeneous multi-core nature allows for natural upwards as well as downwards scaling. Low power designs can consists of fewer SPEs; high performance computing oriented designs can utilize more. For example, the PowerXCell contains special SPEs that are natively able to work with double precision floating point operations. As a result, its double precision throughput is much higher, which is important for scientific workloads. Also shown is a higher clocked future Cell variant consisting of 2 PPEs and 32 SPEs.

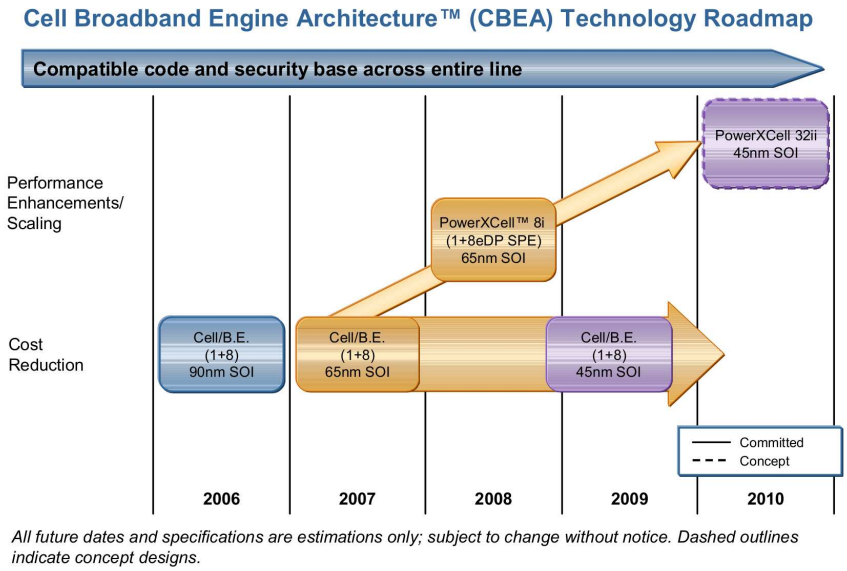


Figure 4.3: Cell technology roadmap (taken from [22]).

Other ways in which the architecture could evolve include:

- Better performance per SPE by virtue of new instructions.
- Larger Local Stores per SPE.
- Lower main memory latency and higher bandwidth.

4.5 Summary

Continuously increasing transistor budgets as predicted by Moore's law allow for increasingly complex microprocessors. Three reasons underlie the current trend towards multicore architectures: frequency scaling is becoming more difficult, power consumption is becoming more important, and memory technology scaling lags behind the increases in microprocessor computational capabilities. Hence, parallel architectures are favored over sequential architectures.

Parallelism can be exploited on many levels: from instruction level parallelism to thread level parallelism. Parallelism exposed to the software engineer places additional burden on software design. Moreover, based on Amdahl's law, limits exist on the speed-up gained from parallelism, especially when parallelizing existing software.

The Cell Broadband Engine, the implementation architecture of this thesis, is a state-of-the-art microprocessor aimed at providing high performance at acceptable power consumption levels by extracting parallelism at all levels. Its heterogeneous nature and exposure of memory parallelism to the software engineer represents a radical break with traditional microprocessor design.

PART II:
IMPLEMENTATION AND ANALYSIS

"I do not fear computers. I fear the lack of them."
- Isaac Asimov

5 HMMER and HMMERCELL

This thesis contributes to the understanding of the behavior of bioinformatics applications on many-core architectures by investigating HMMER, a representative biosequence analysis application and part of SPEC CPU2006 [5], a popular benchmark for computer performance. The Cell Broadband Engine architecture is chosen as the implementation architecture, being a state-of-the-art heterogeneous multi-core architecture. The thesis is centered around the Cell variant of HMMER, aptly called HMMERCELL. The functionality and mathematical fundaments of this application have already been covered in Chapters 2 and 3 respectively; the Cell architecture was discussed in Chapter 4. In this chapter, HMMER's internal functioning and the peculiarities of the Cell version are shown.

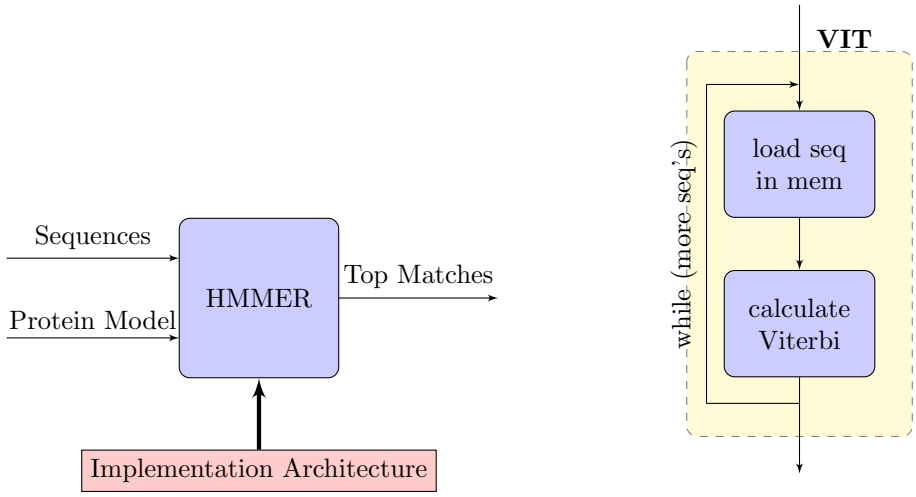
In Section 5.1, the standard version of HMMER is discussed. In Section 5.2, relevant aspects of the port of HMMER to the Cell architecture are described: the parallelization strategy that was followed, implementation details and its limitations. In Section 5.3, related work is discussed. The chapter concludes with a summary.

5.1 HMMER

HMMER is a freely available open source family of tools often used in biosequence analysis by Eddy et al [14]. It is aimed specifically at protein sequence analysis. Groups of protein sequences thought of as belonging to the same family are modeled with profile Hidden Markov Models. HMMER has a specific profile model format, called the Plan7 Profile HMM Architecture (see Section 3.3.1 for more details). Residue positions in the protein model correspond to states in the profile. Certain states have emission probabilities, these represent the chance that an amino acid occurs at a given residue position. In this manner, differences between sequences in the family, caused by for example evolutionary mechanisms, can be modeled. The official website of the HMMER project [16] contains the following description of the software's function:

> "Profile hidden Markov models (profile HMMs) can be used to do sensitive database searching using statistical descriptions of a sequence family's consensus. HMMER is a freely distributable implementation of profile HMM software for protein sequence analysis."

This thesis focuses on one tool within the HMMER suite in particular: *hmmsearch*. This program is used to compare a protein profile, or consensus, to one or more protein sequences. The model is usually compared to entries in a sequence database (see Section 2.4). The Viterbi algorithm is used to generate a score showing the quality of the fit between the model and a sequence (see Section 3.2.1). Before taking an in-depth look at



(a) Black box view. (b) Internal view.

Figure 5.1: Functioning of *hmmsearch*.

the implementation, the program can be analyzed as a black box to make assumptions on expected behavior. After obtaining empirical results, these can be compared to the expectations. Peculiarities are then explained by inspecting the program structure.

Figure 5.1 shows a simplified view of *hmmsearch* functionality. Figure 5.1(a) summarizes *hmmsearch*'s operation on some implementation architecture: based on a profile HMM and one or more protein sequences, the output is generated by aligning each sequence to the profile. Each alignment results in a score. Alignments whose score exceeds a certain threshold - i.e. sequences that appear to match the profile - are displayed. Generally speaking, the execution time of *hmmsearch* is dominated by the time spent on Viterbi decoding, which is performed once for each sequence in the sequence database. Profiling shows that for all but the simplest workloads, Viterbi decoding accounts for 98+% of total *hmmsearch* running time. This is exemplified in Figure 5.1(b).

The time required to compute these results depends on a multitude of factors, amongst others the size of the HMM, the amount of sequences in the test set and their length, and the implementation architecture. The impact of these parameters on performance will be more thoroughly investigated in Chapter 6. Since biosequence data sets are growing at an ever increasing pace, minimizing the computation time is of utmost importance.

5.2 HMMERCELL

HMMERCELL is a port of *hmmsearch* to the Cell micro-architecture by Lu et al [29]. The next section describes the parallelization strategy they followed when HMMER was ported to the Cell architecture. It shows how the strengths of the Cell architecture were exploited and its weaknesses circumvented. Then, limitations that the Cell architecture imposed on the program are clarified.

5.2.1 Parallelization Strategy

Cell's exotic nature makes optimal utilization difficult. Careful planning is required to exploit its strengths while at the same time circumventing its weaknesses (recall Section 4.4). It offers parallelism at multiple levels: Cell contains nine computation cores, a PPE and eight SPEs, which ideally should be utilized simultaneously at all times; moreover, in order to benefit from Cell's very strong SIMD performance, vectorized code that exhibits data parallelism is essential.

Like many bioinformatics applications, *hmmsearch* includes a computationally intensive kernel that is repeated many times in succession; in the case of HMMER, the Viterbi algorithm. As the most computationally expensive part, it is a natural starting point for the exploitation of parallelism. Thus, the parallelization strategy followed by HMMERCELL is to parallelize the Viterbi algorithm as thoroughly as possible. The Viterbi algorithm contains both coarse grain and fine grain parallelism, forming a good match for the Cell architecture. Processing sequences is an inherently parallel task, as each sequence alignment can be determined independently and hence simultaneously. In accordance with the manager/worker paradigm, these coarse grained jobs are distributed over the processors. The PPE prepares jobs, which are then processed by the SPEs. A job consists of aligning a single sequence to the profile HMM by performing Viterbi decoding. The Viterbi algorithm itself is also amenable to fine grain parallelization. Very fast SIMDized versions of the algorithm exist; the implementation in HMMERCELL is based on the highly efficient Altivec implementation by Lindahl [28].

Another important aspect of the Cell architecture is the explicit memory management (again, recall Section 4.4). SPEs are unable to address main memory directly; instead, each SPE has its own local store (LS) of 256 KB. Code and data must be transferred to the LS via DMA. This has two important consequences for HMMERCELL.

First of all, the small size of the LS imposes limitations on the size of the data structures the Viterbi algorithm can use. The algorithm requires memory space linear in sequence length and HMM size, or $O(n \cdot m)$ (see Section 3.2.1). Combined with the small LS size, this has as result that the algorithm's data structures only fit in memory for small HMMs in conjunction with short sequences. To allow HMMERCELL to function for common HMM sizes and sequence lengths, a modified version of the Viterbi algorithm with a smaller memory footprint is used. This algorithm only outputs a score, it does not store the alignment to which the score pertains. In this case, keeping the survivor path to each state in memory is no longer necessary. Just an intermediate score is kept for each state; hence, not $O(n \cdot m)$, but only $O(n + m)$ space is required. In effect, the SPEs are used to indicate which of the sequences in the test set are of interest. A sequence that scores above a certain threshold is deemed a potential match. For those high scoring sequences, the PPE obtains the alignment by performing traceback, i.e. running the full Viterbi algorithm. The idea behind this division is that only a small percentage of sequences actually scores above the threshold, hence few PPE traceback is required. In the next chapter, traceback behavior is analyzed in more detail.

Secondly, the SPE code uses double buffering. Each SPE processes a large number of sequences, one by one. To hide the latency of the memory operations, computation and communication operations are overlapped: while the SPU is busy performing the

Viterbi algorithm for one sequence, the MFC is already transferring the next sequence (if available) to the LS. This is allowed by the explicit control over both units granted to the programmer.

5.2.2 Implementation Details

HMMERCELL's parallelization is organized according to the manager/worker pattern. The PPE creates jobs, whereas the SPEs consume them. Figure 5.2 shows the basic idea of the implementation. The Viterbi function consumes most of the time in standard HMMER. Therefore, the various functions displayed in the figure all relate to the parallelization of the Viterbi processing phase in HMMER. The figure is divided in three parts: the PPE, the SPE and main memory. Individual sequences are processed in three stages: first, PPU_BUF, then SPU_VIT, and finally PPU_TB (the corresponding states are UNPROCESSED, UNCHECKED and FINISHED). In the figure, data movements corresponding to each function are annotated with one, two and three, respectively. The individual stages are now discussed in more detail.

The PPE performs two important functions: buffering and traceback. First, all sequences are buffered, loading them one-by-one from storage into aligned memory. For each sequence, an appropriate entry into the administrative structure is created, containing, amongst others, the length of the sequence, its location in memory, and a status entry that signifies the state of the sequence (PPU_BUF). Initially, this status is set to UNPROCESSED. When all sequences are buffered, the PPE proceeds with scanning through the status entries in the administrative structure to check their results. For each unchecked sequence, the following procedure is performed: if the sequence has a high enough score to count as a significant match to the profile, the PPE computes the Viterbi algorithm to produce the alignment to the model. Then, regardless of score, the status entry is updated to FINISHED (PPU_TB).

SPE execution follows a simple routine. When the SPE is first started, the HMM against which all sequences need to be aligned is copied to the LS (not shown in the figure). Then, as long as there are still sequences to be processed, SPEs wait for a sequence to become available. When a job becomes available, its administrative entry and sequence are transferred to the LS, the reduced Viterbi algorithm is performed, and the entry of the job in the administrative structure is updated with status UNCHECKED and its resulting score (SPU_VIT). When all sequences have been processed, the SPEs quit.

In the figure, only one SPE is displayed. However, as the processing of sequences is independent, the number of SPEs that could be put to use is restricted just by the number of sequences in the test set. Normal workloads contain tens of thousands of sequences or more. Hence, in practice, there is no limit on the SPE count. The figure also shows that most communication between PPE and SPEs is performed through the data structures in main memory. Synchronization by means of mutexes ensures exclusive access to these data structures. Not shown in the simplified overview is the use of prebuffering of sequences at the PPE-side. Before the SPEs are started, an initial batch of jobs is created for the SPEs to ensure that they do not immediately stall and wait until the first jobs become available. When this first batch is created, the PPE starts the SPEs

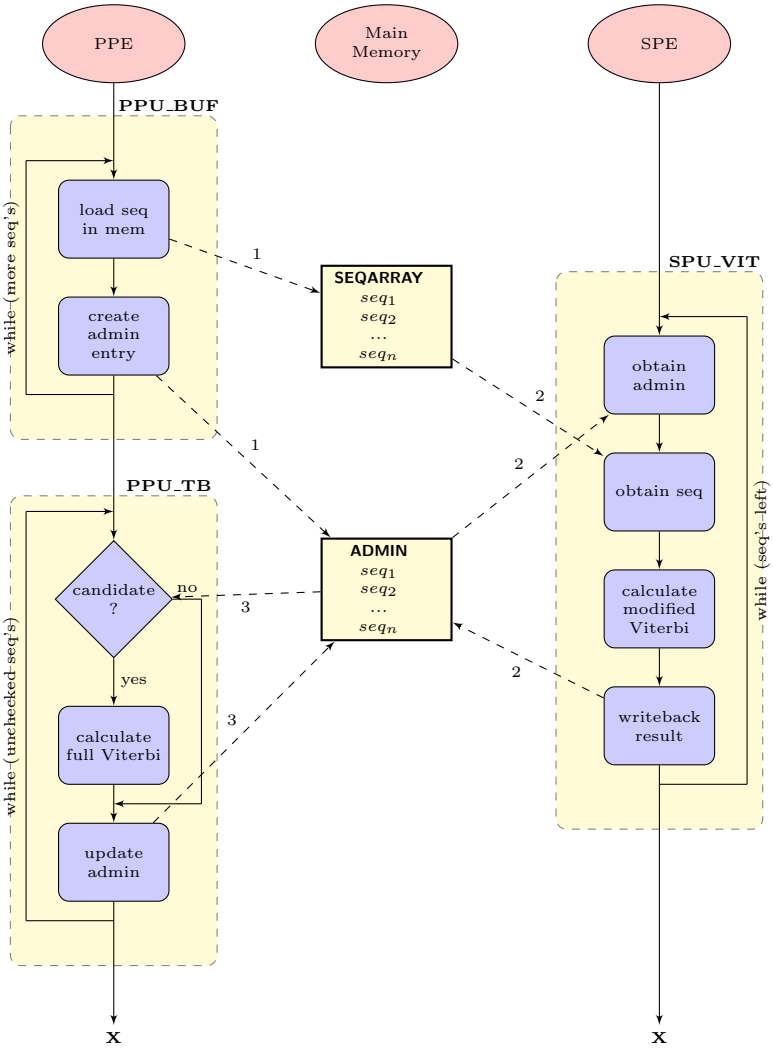


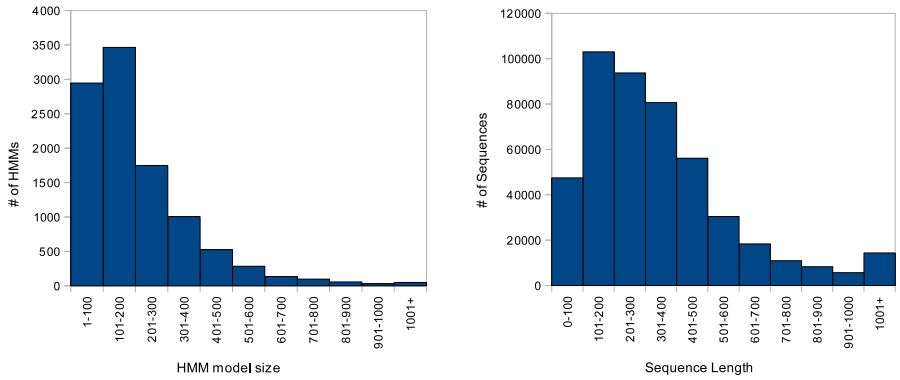
Figure 5.2: HMMERCELL internal functioning.

and then continues buffering more jobs. Finally, as explained earlier, double buffering is used at the SPE-side, to overlap the transfer of a new sequence with computations on another sequence.

Two characteristics of the current program structure, the manager/worker pattern and the requirement for full traceback, create a potential bottleneck: the PPE is put under pressure as it has two separate functions: ensuring that enough jobs are available for the SPEs and that traceback calculations are performed. In the next chapter, the implications of these issues are investigated.

5.2.3 Limitations

Certain characteristics of the Cell architecture make it well-suited to bioinformatics applications. The parallelism naturally available in such applications is a good fit to



(a) Pfam database (b) UniProt database

Figure 5.3: Distribution of protein database element sizes.

the parallel nature of the Cell. However, its architecture also imposes some limitations. As shown earlier, the small LS size requires the use of a modified version of the Viterbi algorithm. Even when using this variant of the algorithm, hard limits on the size of the profile HMM and the sequences are imposed. This section investigates these limitations in more detail.

The size of the SPE LS is 256 KB. HMMERCELL allocates this available memory as follows: 150 KB is reserved for the profile HMM, 10 KB is reserved for the sequence to be processed. But as double buffering is used, 2x 10 KB are required. The rest of the LS is filled with code and temporary data. The 150 KB reserved for the HMM corresponds to models with a length of at most 500 residue positions. Each residue position in the model requires the following data: emission probabilities and scores for every amino acid and transmission probabilities to the follow-up states. The 10 KB reserved for sequences allows for sequences with a length of at most 10000 symbols, as amino acids can be represented by a single byte.

The limitations on the size of the HMM models and on the length of the sequences that can be used somewhat restrict the applicability of HMMERCELL. However, as will be shown, these restrictions do not severely reduce the number of queries that can be performed. In order to show the impact, the following figures depict the distribution of HMM model size and sequence length in two popular bioinformatics databases. In Figure 5.3(a), the distribution of model sizes in the Pfam database is shown. The histogram shows for each profile size bin how many models the bin contains. Just 651 out of 10340 profile HMMs contain more than 500 residue elements; 93% of the models are shorter. Moreover, only a few models are much larger: 99.5% of the models contain a thousand elements or less. Consequently, a somewhat larger LS, where twice as much memory could be allocated to the HMM, would allow practically every HMM to be used. Figure 5.3(b) shows the distribution of sequence lengths in the UniProt database. The histogram shows the number of sequences ordered by sequence length. Just thirteen out of 468851 protein sequences are larger than the imposed limit of 10000 symbols. The average sequence contains only 354 symbols. Hence, this limitation is not very restrictive.

5.3 Related Work

This section reviews various work related to HMMER. This helps placing HMMERCELL into its proper context. First, performance of HMMERCELL is compared to versions of HMMER that run on other processor architectures, which allows a better appreciation of the performance characteristics of Cell for suitable applications. Then, results from FPGA and MPI versions of HMMER are discussed. Also shown is a first look at HMMER3.

The performance of HMMERCELL as compared to commodity x86 architectures is shown in Figure 5.4 [29]. In this figure, results from alignment to a four hundred length HMM are shown, with performance normalized to a regular eight SPE Cell. Cell is compared against the AMD Opteron platform (2.8 GHz, 1-4 cores used) and against the Intel Woodcrest platform (3.0 GHz, 1-4 cores used). From the graph, the exceptional computational capabilities of the Cell architecture become obvious. A single Cell is thirty times faster than a single-core Intel or AMD processor. However, note that the SIMD capabilities are left unused on the x86 platforms, which would grant them a 4-7x speedup. But even then, Cell compares favorably.

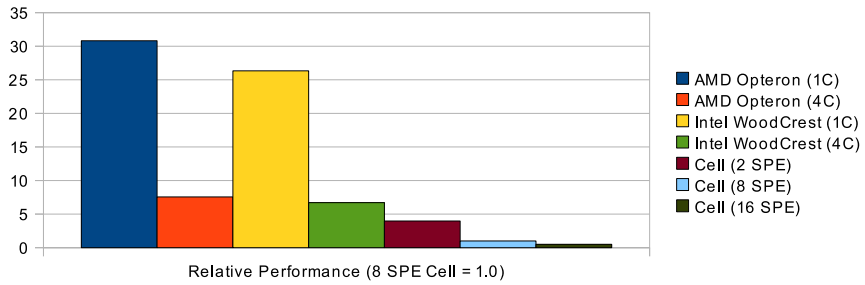


Figure 5.4: HMMERCELL Performance Comparison.

HMMER has been ported to many different platforms and architectures. Field Programmable Gate Arrays (FPGAs) represent a cost-effective solution allowing potentially large speed-up, if the application can be mapped well to them. In [35], suitability of FPGAs to HMMER is investigated. As in HMMERCELL, the computationally intensive kernel of the Viterbi algorithm is mapped onto the FPGA. Similar to HMMERCELL, the FPGA is used as a filtering mechanism: sequences with a promising score require reprocessing on the host machine. A thirty fold speed-up over an AMD Athlon64 3500+ is reported, which is comparable to results from HMMERCELL.

The MPI-based MPI-HMMER was introduced to take advantage of computer clusters [44]. Similar to HMMERCELL, one node is assigned a manager-role, the rest of the machines are workers over which the workload is distributed. To cope with overhead from message passing, sequences are grouped in larger bundles and sent as one message. Through double buffering, communication latency is minimized. An eleven-fold speed-up is reported when using sixteen machines. In [43], MPI-HMMER is analyzed and the manager node is demonstrated to be a bottleneck. For MPI-HMMER, scaling is effective up to 32-64 nodes, depending on workload. PIO-HMMER is introduced, addressing I/O-related bottlenecks through use of parallel I/O and optimized post-processing. The

manager distributes an offset file with sequences to each node, worker nodes read the sequences from their local database. Furthermore, nodes only report significant results back to the manager. The resulting scaling capability is much improved, as up to 256 machines can be used effectively.

HMMERCELL is based on the most recent version of HMMER. A new version, HMMER3, will be released in the near future [16]. Its most promising feature is that it addresses the slow speed of profile HMM-based methods, which forms their main drawback. In contrast, HMMER3 is as fast as BLAST. This makes HMMER even more relevant, as the trade-off between sensitivity (choosing HMMER) or throughput (choosing BLAST) is no longer required. The improvement is mainly the result of pervasive use of vector instructions in combination with the processing of sequences through an increasingly more sensitive and slow filtering pipeline [19]. The results from this thesis are not directly applicable to HMMER3, as the new algorithms will have a profound effect on its behavior. However, the general methodology described remains valid.

5.4 Summary

In this chapter, the results from qualitative analysis of HMMER, a representative bioinformatics application, are presented. The *Hmmsearch* tool in the HMMER package is used to align a set of sequences to a protein model, which is represented by a profile HMM. The alignment is obtained by performing the Viterbi algorithm for each sequence in the data set. HMMER's execution time is almost exclusively determined by this function, making it a good candidate as starting point for parallelization.

HMMERCELL is the port of *hmmsearch* to the Cell architecture by Lu et al [29]. Their implementation was inspected to discover the program structure and its suitability to the Cell architecture. The Viterbi algorithm is parallelized using the manager/worker pattern. It maps well on the Cell architecture, as both coarse grained parallelism (processing sequences in parallel) as well as fine grained parallelism (using a SIMDized version of the algorithm) can be utilized. The Cell architecture also presents some challenges: in order to fit the algorithm into the SPE's local store, a modified version of the algorithm with a smaller memory footprint is used. This version only produces a score, which indicates how well a sequence matches the model. When a sequence scores above a certain threshold, the PPE has to perform the full Viterbi algorithm in order to produce the alignment between the sequence and the model. Additionally, the small size of the local store places restrictions on the size of the HMM and of the sequences that can be processed. The manager/worker parallelization strategy and the traceback mechanism both introduce potential bottlenecks. Their effects will be investigated in the next chapters.

Cell appears to be an excellent implementation platform for HMMER, as HMMERCELL outperforms x86-based implementations by up to thirty times. FPGA implementations are able to achieve similar speed-up. HMMER can also take advantage of large computer clusters, MPI-based version are scalable up to 256 machines. A new version of HMMER will soon be introduced. This makes the use of profile HMM-based methods even more attractive, as HMMER3 addresses their main drawback: processing speed.

6 Performance Analysis

In the previous chapter, the internal operation of *hmmsearch* and the Cell specific version called HMMERCELL was clarified. An abstract view of the program's functionality was presented, the parallelization strategy being followed was discussed, and limitations of the port were shown. In this chapter, the performance of HMMERCELL is analyzed. The parameters and the functions that are relevant to the application's performance are identified and the influence of these parameters is investigated. Implications of the peculiarities of the HMMERCELL port and the Cell architecture are scrutinized. Based on the results, an analytical model to predict the performance of HMMERCELL is proposed. Applications of this model include estimation of the required computation time for a test set or determination of the number of SPEs that can be effectively used to minimize execution time for a given workload.

In Section 6.1, HMMERCELL behavior is discussed and the test sets and test environment are described. In Section 6.2, profiling results are shown. In Section 6.3, an analytical model is formulated based on the results. The model is validated and used to identify potential bottlenecks to performance. The chapter is concluded with a summary.

6.1 Inspecting HMMERCELL Behavior

Before presenting the profiling results, relevant parameters to HMMERCELL performance are discussed first. A short summary of the parallelization strategy is given to help obtain a clear view of the important parts of the program structure. The input and output variables are discussed next. Then, the test sets that have been used and a motivation for their structure is given. Finally, the test environment from which the results have been obtained, will be discussed.

Execution time in *hmmsearch* is spent almost exclusively in the Viterbi function, which computes an alignment between an HMM and a sequence. HMMERCELL's parallelization strategy attempts to distribute this workload evenly over the available processors by utilizing the manager/worker pattern: the PPE creates jobs (PPE_BUF) which are consumed by the available SPEs (SPE_VIT). The small size of the SPE's local store requires the use of a modified Viterbi algorithm with a smaller memory footprint. This modified algorithm stores only the resulting alignment score; the survivor paths are not stored in their entirety. Therefore, sequences with a sufficiently high probability of being a match to the model require recalculation of the alignment by the PPE (PPE_TB). Three aspects deserve further investigation: the effects of the introduction of this manager/worker split, the number of SPEs that can be put to effective use, and the influence of traceback on performance.

The input parameters for *hmmsearch* are a profile HMM and a set of sequences that

Input Parameters	
Profile HMM	Model length
Sequences	Number of sequences
	Distribution of sequence lengths
Output Parameters	
High Scoring Sequences	Bit score and E-value
	Alignment
Execution Time	Time in milliseconds
Architecture Characteristics	
Cell Broadband Engine	Number of SPEs in use

Table 6.1: Relevant *hmmsearch* parameters.

will be matched to the profile. The output consists of a set of high scoring alignments. A particular combination of profile HMM and set of sequences should always yield the same resulting alignments. Hence, only quantitative measures are of interest. Execution time of the algorithm is therefore taken as the measure of application performance.

Table 6.1 summarizes relevant details of the input and output variables *hmmsearch* uses. With regard to input parameters, for the profile HMM only the number of residue positions in the model is relevant. Important characteristics of the sequence test set are the number of sequences the set contains and the distribution of lengths of sequences in the set. The influence of the number of SPEs on performance is investigated to find out what the maximum number of processor cores is that can be effectively used.

One aspect deserving further explanation is the following. The resulting alignments and alignment scores are of course dependent on the particular combination of HMM and sequence set: some sequence sets will match the model better than others, as they contain more homologues. Also, some HMMs model proteins formed out of more common substructures than other more exotic proteins, resulting in more tracebacks. As a result, the number of sequences that require traceback will vary accordingly. Hence, the requirement for traceback depending on alignment score introduces some inherent uncertainty, as will be shown later on.

As execution time is dominated by the Viterbi calculations, the expected result of altering input parameters can be determined based on the behavior of this algorithm. Processing time should scale linearly in the size of the profile HMM, since the Viterbi algorithm is linearly time dependent on the number of states in the profile HMM. The composition of the set of sequences (their number and size), has the following effect on execution time: the Viterbi algorithm is linearly time dependent on the length of a sequence. Moreover, since sequences are processed independently of one another, execution time should scale linearly in the number of sequences.

Before running actual tests, it is useful to think about what ideal scaling behavior would be for this application. An increase in the number of SPEs should result in a linear

decrease in processing time, since the workload is evenly spread over all processors. Of course, this assumes perfect load balancing, i.e. the PPE and SPEs are all busy and finish at exactly the same time. Since workloads have a discrete size, this is unlikely to be the case. Moreover, the current parallelization pattern makes the PPE responsible for two tasks: managing the SPE job creation and performing traceback for promising sequences. Hence, for every workload there exists an SPE count that will saturate the PPE: either when job creation proceeds at a slower pace than SPE job consumption, or when PPE job creation and traceback together takes more time than the SPE Viterbi calculations. As next generation CELL processors will likely tilt the balance between PPEs and SPEs towards more SPEs (see Section 4.4.4), it is interesting to find out how many SPEs can be used effectively for different workloads.

6.1.1 Test Sets

HMMERCELL has been subjected to a variety of tests in order to determine the application's behavior. In a typical HMMER use case, a protein model is compared against all entries in the UniProt database (see Section 2.4). Constraints on the size of the data set led to the use of a smaller representative subset of this database. With this data set, performance of a typical use scenario is simulated and analyzed. To investigate scaling behavior, the different functions of importance are investigated separately. The influence of three aspects is measured: the characteristics of the profile HMM to search with, the composition of the sequence test set to compare against and the number of SPEs the application is allowed to use. Table 6.2 gives an overview of the different conditions that have been tested.

To determine the impact of the HMM model size, five HMM lengths are used; their sizes evenly distributed up to the maximum allowed size by HMMERCELL. They have been selected at random out of the Pfam database. The table indicates the specific HMMs that have been used. During most tests, only one HMM of each size is used. For the test aimed at investigating traceback behavior, the HMMs listed in parenthesis are used. As the number of tracebacks for a given combination of HMM and test set is dependent on the biological match between protein sequences and protein model (recall Section 3.3.2), multiple HMMs are used in order to produce an average traceback score.

The sequence test sets consist of randomly selected sequences from the UniProt database. The following sequence sets have been created:

"Representative" To investigate typical HMMER use, a set of sequences has been made whose distribution in length is representative to the occurrence of sequences in the UniProt database (see Section 2.4). The number of sequences in the test set, 20000, is chosen sufficiently large, making overhead from e.g. initialization negligible.

"Length" To investigate scaling behavior in sequence length, a number of sequence sets with a thousand sequences each have been created. Each set contains successively larger sequences: the first contains sequences ranging from 1-120 symbols; the second sequences with 121-240 symbols; et cetera. The largest set contains sequences of up to 1560 symbols.

Profile HMM Length and Description		
100	COQ7	(SRI_2, Syntaxin-6_N, Transposase_9, tRNA_synt_1c_R2, T_Ag_DNA_bind)
200	CNTF	(HutD, ICAP-1_inte_bdg, NanE, RIO1, R_equi_Vir)
300	CitG	(ketoacylsynt, Lipoprotein_1, Ndr, Nuc_sug_transp, Peptidase_U4)
400	Acyl_transf_1	(DUF819, DUF898, Herpes_U34, IpaC_SipC, RNA_pol_Rpc82)
500	ADP_PFK_GK	(ADP_PFK_GK, Arabinose_Isome, Sulfatase, GDE_C, Molybdopterin)
Sequence Test Set Composition		
"Representative"		20000 sequences with representative* length (see text)
"Length"		1000 entries each, sequence sizes range from 1-120, 121-240, ..., up to sequences of 1560 symbols.
"Test Set Size"		Sets containing 100, 1000 and 10000 sequences.
Number of SPEs Used		
1, 2, 4, 8, 16		

Table 6.2: Test set overview.

"Test Set Size" Traceback behavior is also dependent on test set size (again, recall Section 3.3.2). Hence, test sets containing different numbers of sequences (100, 1000 and 10000 sequences) have been created. Of each size, five test sets have been produced by randomly selecting UniProt sequences (see Section 2.4).

To investigate scaling behavior of the application in the number of SPE cores that are used, tests have been performed with 1, 2, 4, 8 and 16 SPEs. Although a Cell processor contains only eight SPEs, the test systems contain two Cell processors each. Hence, sixteen SPEs can be utilized (but in this case, the second PPE is not used).

6.1.2 Test Environment

HMMERCELL is a port of HMMER v2.3.2 to the Cell architecture. Tests are performed on the Cell cluster located at the Barcelona Supercomputing Center [1]. The BSC is an important partner in research and development for the Cell architecture. Dedicated test facilities are available on which jobs are scheduled. These jobs are granted exclusive access, preventing external factors from influencing the results. Each machine contains two Cell processors and hence two PPEs and sixteen SPEs.

In order to analyze HMMERCELL, traces that record information about runtime behavior have been created by executing an instrumented version of the application. These traces are reviewed to investigate performance. HMMERCELL source code has been instrumented with calls to the MPItrace tracing library [10]. Calls to this library

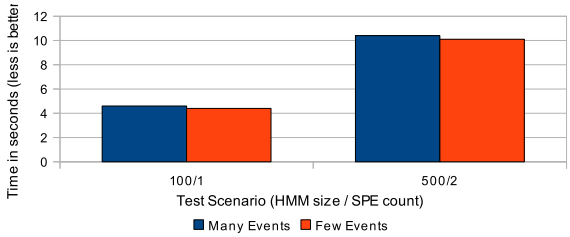


Figure 6.1: Overhead resulting from trace instrumentation.

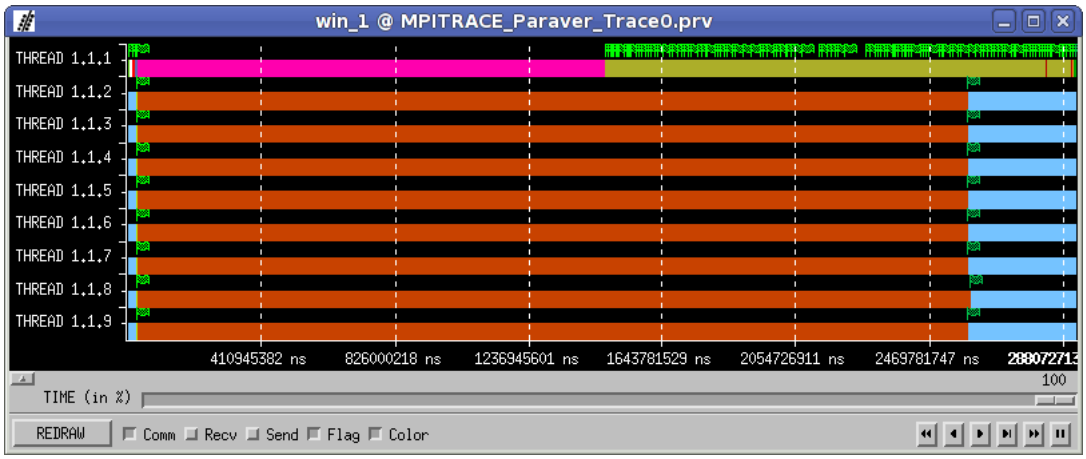


Figure 6.2: Paraver trace file visualization.

are manually inserted at critical parts of the code, so that the order and timing of important events that occur during runtime can be analyzed later on. Impact of the tracing instrumentation calls on performance is shown in Figure 6.1. Overhead has been investigated under two scenarios (both using the representative test set): one with an HMM of length 100 using one SPE; the other with an HMM of length 500 using two SPEs. During one setup traces with only a few events are generated, resulting in a trace file of a few KB. Another setup, tracing a great number of events, produces trace files in the order of a few MB in size. From the figure, it is obvious that the resulting overhead is small to negligible as the difference in execution time is marginal.

The generated traces are inspected with the Paraver tool [37], a visualization environment for trace files. Traces can be displayed in various ways: for example, a graphical overview can be shown or detailed numerical statistics can be produced. The trace file can be filtered to prune irrelevant events, in order to focus on specific details. A semantic module can be used that interprets the events, giving meaning to the data set. An example visualization is presented in Figure 6.2. It shows the result from a test with one PPE and eight SPEs. The phases PPU_BUF (pink), PPU_TB (green) and SPU_VIT (orange) are clearly distinguishable. The light blue color signifies inactivity.

MPItrace and Paraver have both been developed by the BSC. When used in conjunction, they offer the user powerful tools of analyzing program performance and behavior.

6.2 Profiling Results

In this section, the results obtained from profiling HMMERCELL are presented. First, an overview of the application's performance profile is given. Then, the scaling behavior of the various functions that constitute overall performance (PPU_BUF, SPU_VIT and PPU_TB) is investigated in more detail. To this end, traces have been generated running the application using various input parameters: test sets consisting of sequences of varying length, different HMM sizes and a varying number of SPEs.

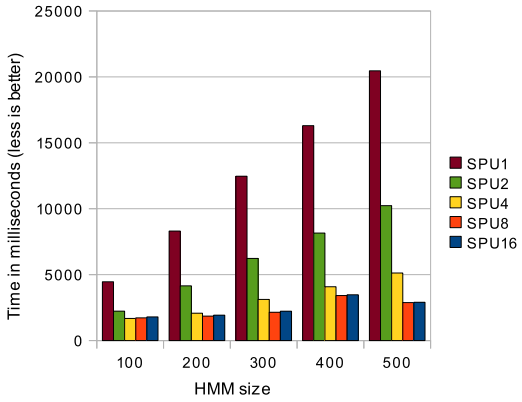


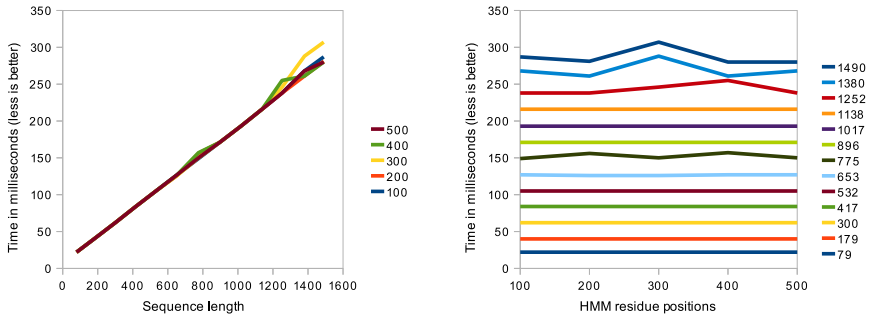
Figure 6.3: HMMERCELL execution time overview.

Figure 6.3 shows execution time when performing sequence alignment using the representative sequence test set for different HMM lengths and SPE count, giving a general idea of run-time performance. The overview immediately reveals a few trends: larger HMMs require correspondingly longer execution time; in general, using additional SPEs leads to shorter execution times; only a certain number of SPEs can be used effectively, depending on the workload. Due to overhead, using more SPEs results in identical or deteriorated performance. For example, using sixteen SPEs is slightly slower than eight, as off-chip communication is required since the Cell contains just eight SPEs. The effective SPE count can be calculated using the model proposed later on.

The results have been investigated in more detail by inspecting the scaling behavior of the functions that constitute HMMERCELL performance. For each function, two graphs are shown: one in which HMM size is varied, the other with varying sequence length. This facilitates the recognition of scaling behavior. Behavior at the functional level is irrespective of SPE count. Hence, results are shown with just one SPE.

6.2.1 Scaling of PPE Buffering Function

Figure 6.4 shows the scaling behavior of the PPU_BUF function. This function loads sequences from disk into main memory and creates corresponding entries into the administrative structure. In both figures, the vertical axes represent execution time. Figure 6.4(a) shows scaling results in sequence length. In this figure, the horizontal axis stands for the sequence length, ranging from very short to medium length sequences.



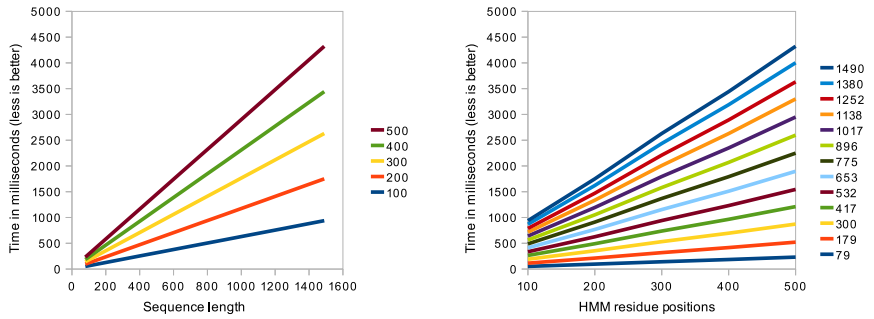
(a) vs sequence length (b) vs HMM size

Figure 6.4: PPU buffering function behavior.

The different lines show the results for the different HMM sizes. Figure 6.4(b) presents scaling results in HMM size. In this figure, on the horizontal axis HMM size is given and the lines stand for the various sequence sets. In the legend, the average sequence length per test set is shown.

From the figure, it becomes clear that PPU_BUF computation time scales linearly in the sequence length and it is irrespective of HMM size. This is in line with expectations: loading a sequence from disk by the PPE has no relation whatsoever with the HMM size, it is dependent solely on the sequence's length.

6.2.2 Scaling of SPE Viterbi Function

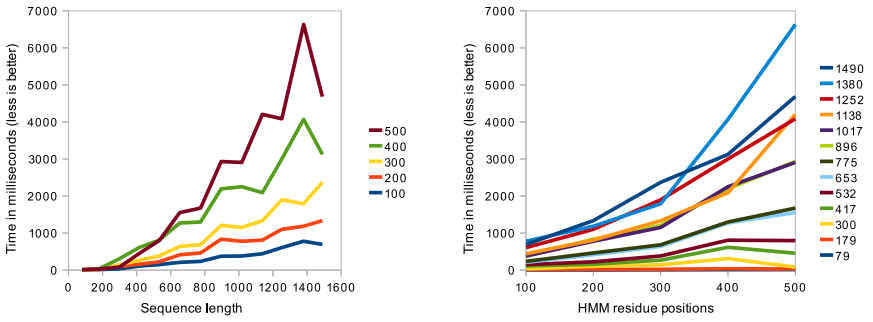


(a) vs sequence length (b) vs HMM size

Figure 6.5: SPU Viterbi function behavior.

Figure 6.5 shows the scaling behavior for the SPU_VIT function, which performs the Viterbi calculations on the SPE. As before, the vertical axis in both graphs displays execution time. The left figure shows scaling in sequence length, the right scaling in HMM length. From the figures, it is obvious that SPU_VIT computation time scales linearly both in the length of the sequence and in the size of the HMM profile. Again, this confirms expectations, as in Section 3.3.1 it was already shown that the Viterbi algorithm scales linearly in sequence length and linear for models cast in HMM form.

6.2.3 Scaling of PPE Traceback Function



(a) vs sequence length (b) vs HMM size

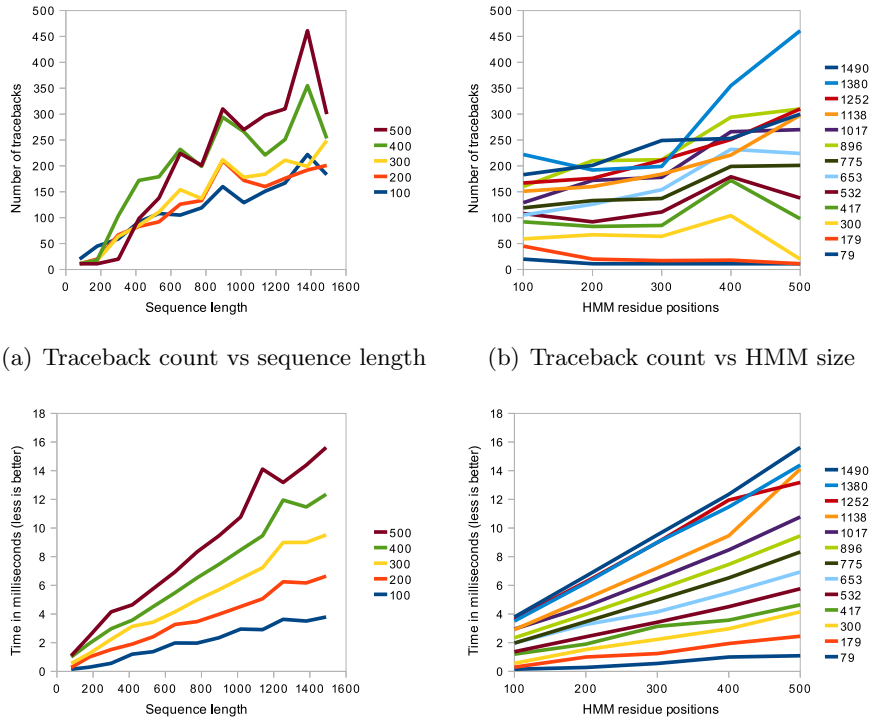
Figure 6.6: PPU Viterbi traceback function behavior.

Figure 6.6 shows the scaling behavior of the PPU_TB function, which performs the full Viterbi algorithm on the PPE to produce the alignment for those sequences that are a potential match to the model. Compared to previous graphs, PPU_TB behaves less regular. Whereas PPU_BUF and SPU_VIT are performed for each sequence in the test set, the Viterbi calculations in PPU_TB are only performed for a subset of sequences: those whose score exceeds a certain threshold. The actual number depends on how well sequences in the test set correspond to the model and varies accordingly. Performance of PPU_TB is now analyzed further by breaking it down in two components: the number of times traceback is performed and the time required for an individual traceback.

Figures 6.7(a) and 6.7(b) show the number of tracebacks performed given different sequence and HMM combinations; Figures 6.7(c) and 6.7(d) show the average time for a single traceback. The number of tracebacks multiplied by the time per traceback produces the total time spent in the traceback function, as given in Figure 6.6.

The time per individual traceback behaves as expected: execution time scales more or less linearly in both sequence length and HMM size. The full Viterbi algorithm requires a large data structure in memory. Traversing the memory hierarchy causes the observed staggered scaling. Fluctuations in the number of tracebacks are the main reason for the erratic results in total traceback time. Some correlation between length and traceback count does exist: generally speaking, longer sequences lead to more hits, since local alignment is performed. Subsections of a sequence are allowed to form a match to the model, hence the longer the sequence the larger the probability of a matching subsection. HMM size has no clear effect on performance.

The number of tracebacks is affected by two factors (recall Section 3.3.2): the particular combination of HMM and sequence set, i.e. their biological match; and the number of sequences in the test set, as the E-value depends on test set size. Table 6.3 contains results from further traceback count analysis, showing the number of tracebacks required for different test set sizes and HMMs. Five test sets of each size were generated (listed in the columns). Also, five different HMMs of each HMM size were used (listed in the rows). The results show large variation in the number of tracebacks between HMMs, even for those having identical length. Differences between the test sets of equal size are



(a) Traceback count vs sequence length (b) Traceback count vs HMM size

(c) Time per traceback vs sequence length (d) Time per traceback vs HMM size

Figure 6.7: Traceback in-depth analysis.

much less pronounced. Larger numbers of sequences only require a few more tracebacks, as a weak logarithmic relationship between test set size and traceback count is present (again, recall Section 3.3.2).

Number of Sequences in Test Set

HMM length	100					1000					10000				
100	29	34	29	28	32	47	47	43	51	58	47	58	49	60	62
	36	37	38	31	33	93	89	106	100	95	202	194	200	180	198
	42	38	45	43	43	92	90	97	94	103	155	154	155	149	149
	30	33	34	29	32	69	75	74	63	66	132	125	108	113	133
	31	29	33	25	21	46	50	43	60	44	65	63	60	63	65
200	40	39	43	31	38	99	101	81	80	90	135	142	156	152	131
	21	25	28	22	23	35	39	41	43	36	42	49	56	43	47
	44	42	41	41	41	115	102	108	114	107	226	189	221	197	196
	39	46	44	39	37	81	81	74	75	78	116	122	122	134	125
	35	34	43	30	33	86	69	63	60	83	93	97	92	106	116
300	38	37	41	30	37	110	104	103	97	102	215	215	223	232	242
	25	23	26	26	24	55	55	50	60	63	96	95	97	94	103
	27	26	24	26	28	44	46	47	54	49	69	59	56	67	65
	27	29	28	27	31	73	69	77	73	76	176	161	195	161	166
	26	25	29	26	26	57	60	73	66	69	106	82	100	90	108
400	32	29	33	28	34	105	84	104	98	102	351	296	322	301	321
	31	29	28	29	29	116	89	106	107	118	535	436	517	479	524
	22	25	26	21	22	37	41	37	35	29	50	48	39	48	51
	21	27	28	24	26	64	63	71	74	82	162	163	177	150	165
	28	27	31	27	27	65	56	77	79	75	122	119	119	116	145
500	25	25	31	25	25	66	61	66	75	80	93	106	96	109	125
	39	40	38	38	33	94	76	75	79	83	136	108	111	118	140
	23	26	25	26	28	43	49	53	50	49	75	81	63	77	80
	32	28	29	26	27	54	49	57	71	54	83	72	77	89	88
	28	25	31	26	27	55	52	67	62	61	99	105	89	104	108

Table 6.3: Traceback count vs HMM (row) and vs number of sequences (column).

	HMM Size	Sequence Size	Test Set Size
PPU_BUF	independent	linear	linear
SPU_VIT	linear	linear	linear
PPU_TB(# of TB's)	independent	* (see text)	*(see text)
PPU_TB(time per TB)	linear	linear	independent

Table 6.4: Scaling behavior summary.

6.3 Analytical Model

As discussed earlier, the function that performs Viterbi computations dominates HMMER execution time. In HMMERCELL, this function was decomposed into three subfunctions: buffering of sequences (PPU_BUF), reduced Viterbi calculations (SPU_VIT) and full Viterbi calculations (PPU_TB). Based on the findings from the previous section, models can be formulated to predict the execution time for each of these functions. Together, they can be combined to form a model for HMMERCELL's performance. Table 6.4 summarizes the scaling behavior of the different functions of interest (PPU_TB has been split up in number of tracebacks required and in time per individual traceback).

The input parameters mentioned in Section 6.1 are variables in the model. They are parameterized as follows:

- the profile HMM H has a model length m,
- the test set S contains n sequences: $s_1, s_2, \ldots, s_n$; each with length l_x,
- the Cell processor contains one PPE and q SPEs.

As per table 6.4, linear equations have been formulated for the scaling behavior of each function. Equation 6.1, 6.2 and 6.3 model the required execution time for processing an individual sequence. Function $I_{TB(H,s_i,n)}$ is an indicator function, returning 1 when the alignment between sequence s_i and model H is significant for a test set of size n; otherwise it returns 0.

$$t_{PPU\_BUF(s_i)} = \alpha \cdot l_i + C_\alpha \tag{6.1}$$

$$t_{SPU\_VIT(s_i)} = \beta \cdot m \cdot l_i + C_\beta \tag{6.2}$$

$$t_{PPU\_TB(s_i)} = (\gamma \cdot m \cdot l_i + C_\gamma) \cdot I_{TB(H,s_i,n)} \tag{6.3}$$

Aggregating the equations for individual sequences to the entire test set results in Equation 6.4, 6.5 and 6.6. Notice that the indicator function I_{TB} has been replaced by a generalized probability P_{TB} for a sequence in test set S to require traceback. Predicting the result of indicator function I_{TB} is difficult, as it requires full knowledge of the biological match between the protein model and sequence. On the other hand, probability P_{TB} can be more easily estimated based on the traceback count results from the previous section. Also, note that SPU_VIT is time required for the Viterbi computations on all the SPEs combined.

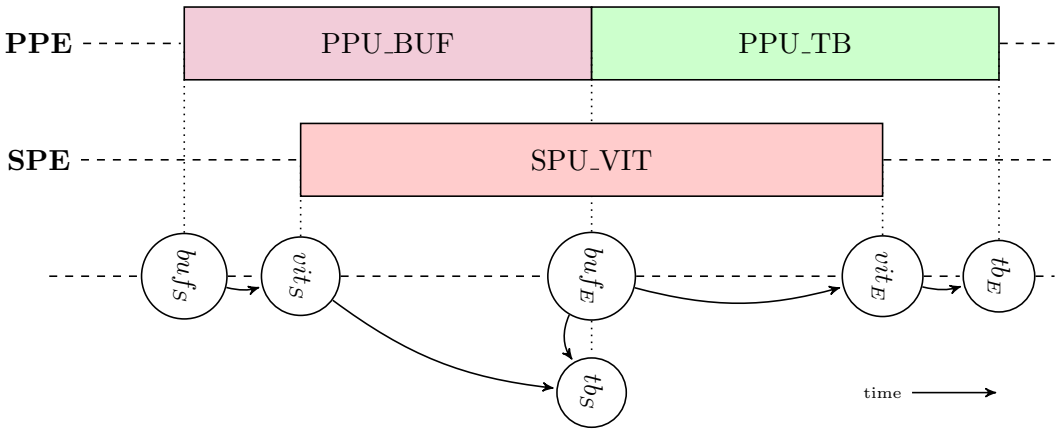


Figure 6.8: Schematic overview of dependencies between functions.

$$
\begin{aligned}
T_{PPU\_BUF(S)} &= \alpha \cdot \sum^{n} l_i + n \cdot C_\alpha \\
&= n \cdot (\alpha \cdot \bar{l} + C_\alpha) && (6.4) \\
T_{SPU\_VIT(S)} &= \beta \cdot m \cdot \sum^{n} l_i + n \cdot C_\beta \\
&= n \cdot (\beta \cdot m \cdot \bar{l} + C_\beta) && (6.5) \\
T_{PPU\_TB(S)} &= (\gamma \cdot m \cdot \sum^{n} l_i + n \cdot C_\gamma) \cdot P_{TB(H,S)} \\
&= n \cdot (\gamma \cdot m \cdot \bar{l} + C_\gamma) \cdot P_{TB(H,S)} && (6.6)
\end{aligned}
$$

$$
\text{with} \quad P_{TB(H,S)} = \frac{\delta \cdot \ln n + C_\delta}{n}
$$

In order to obtain an integrated model for HMMERCELL performance, the interrelation between the different functions needs to be taken into account. The dependencies between the three functions have been discussed in Chapter 5 and are visualized in Figure 6.8. Function SPU_VIT starts after function PPU_BUF starts, as at least one sequence has to be buffered before processing by the SPEs can commence. Function SPU_VIT ends after function PPU_BUF ends, as the last sequence to be buffered must be processed as well. Function PPU_TB starts when function PPU_BUF ends, as the PPE is first occupied with the buffering of sequences, after which it commences with traceback. Function PPU_TB ends after function SPU_VIT ends, as the last processed sequence must be checked by the PPE as well. Depending on the workload, a large part of PPU_TB can consist of idle time, where the PPE waits for an SPE to finish processing a sequence.

In general, a test set contains thousands of sequences. Hence, the processing time of any individual sequence is insignificant when compared to total execution time. For example, take a test set consisting of just a thousand sequences. Even when using ten

SPEs, on average, processing time for a single sequence would be only 1% of total execution time. This observation allows for two simplifications: first, the above dependencies between the functions can be approximated as follows: PPU_BUF and SPU_VIT start at the same time; PPU_TB starts when PPU_BUF completes; and PPU_TB must finish after SPU_VIT. Secondly, load balancing between SPEs is assumed to be perfect, as all processes will finish at approximately the same time.

Then, the accuracy of the model relies on the assumption that the test set contains a large number of sequences, so that the granularity of individual sequence processing becomes very small. Otherwise, the dependencies between processing stages become a non-negligible factor to performance. However, as relevant workloads consist of many thousands of sequences, this assumption seems valid. With these simplified rules, execution time per processor can be summarized as shown in Equation 6.7 and 6.8. Equation 6.9 shows the total execution time:

$$T_{PPE} = T_{PPU\_BUF} + T_{PPU\_TB} \tag{6.7}$$
$$T_{SPE} = (T_{SPU\_VIT})/q \tag{6.8}$$

$$T_{total} = \max(T_{PPE}, T_{SPE}) \tag{6.9}$$

6.3.1 Parameter Estimation

The model as proposed in the previous section contains a few unknowns: parameters α to δ, C_α to C_δ from the linear equations and function $P_{TB(H,S)}$. Their values are specific to the implementation of HMMERCELL on the Cell architecture. To estimate their values, the results from Section 6.2 have been used to parameterize Equation 6.4, 6.5 and 6.6, resulting in Equation 6.10, 6.11 and 6.12:

$$T_{PPU\_BUF(S)} = n \cdot (\frac{0.19}{10^3} \cdot \bar{l} + \frac{5.52}{10^3}) \tag{6.10}$$
$$T_{SPU\_VIT(S)} = n \cdot (\frac{0.59}{10^3} \cdot \frac{m}{10^2} \cdot \bar{l} + \frac{0.88}{10^3}) \tag{6.11}$$
$$T_{PPU\_TB(S)} = n \cdot (\frac{2.25}{10^3} \cdot \frac{m}{10^2} \cdot \bar{l} + \frac{35.7}{10^3}) \cdot P_{TB(H,S)} \tag{6.12}$$

$$\text{with} \quad P_{TB(H,S)} = \frac{21.9 \cdot \ln n - 73.2}{n}$$

From these equations a few immediate observations can be made by comparing the constants in the different equations. First, recall from Section 2.4 that the typical size of an HMM (parameter m) is in the order of a few hundred symbols and the average length of sequences in the UniProt database is 355 (parameter $\bar{l}$). Then, filling in these parameters in the equations, it follows that PPE buffering takes about an order of magnitude less time than the Viterbi calculations on the SPE. Another observation is that an SPE is about four times as fast at Viterbi calculations than the PPE, as this

Test	Empirical Result			Expected Result			Diff	
$(n/m/q)$	BUF	VIT	TB	BUF	VIT	TB	BUF	VIT
20k, 150a, 1	1498	6349	581	1459	6301	176	-2.6%	-0.8%
20k, 150a, 8	1492	797	581	1459	788	176	-2.2%	-1.2%
20k, 150b, 1	1442	6345	336	1459	6301	176	1.2%	-0.7%
20k, 150b, 8	1492	797	335	1459	788	176	-2.2%	-1.2%
20k, 450a, 1	1441	18440	777	1459	18868	519	1.3%	2.3%
20k, 450a, 8	1448	2303	776	1459	2359	519	0.8%	2.4%
20k, 450b, 1	1441	18436	1032	1459	18868	519	1.3%	2.3%
20k, 450b, 8	1446	2305	1031	1459	2359	519	0.9%	2.3%
40k, 150a, 1	3071	12747	1031	2934	12673	196	-4.7%	-0.6%
40k, 150a, 8	3026	1605	714	2934	1584	196	-3.1%	-1.3%
40k, 150b, 1	2927	12748	427	2934	12673	196	0.2%	-0.6%
40k, 150b, 8	3026	1603	427	2934	1584	196	-3.1%	-1.2%
40k, 450a, 1	2925	37021	509	2934	37949	577	0.3%	2.4%
40k, 450a, 8	2931	4627	507	2934	4744	577	0.1%	2.5%
40k, 450b, 1	2924	37021	1531	2934	37949	577	0.3%	2.4%
40k, 450b, 8	2929	4629	1551	2934	4744	577	0.2%	2.4%

Table 6.5: Model validation results (results in ms).

type of calculations is well-suited to their architecture. Also, note that the test set should contain at least 29 sequences for P_{TB} to give a valid result. Finally, for larger test sets, P_{TB} becomes very small. Table 6.3 shows that whereas for a test set of a hundred sequences, on average 31% of the sequences requires traceback; for a test set of ten thousand sequences, only 1.4% is in need of traceback. More observations and implications will be made in Section 6.3.3.

Finally, substituting Equation 6.10, 6.11, and 6.12 into Equation 6.7, 6.8, and 6.9 results in Equation 6.13, which gives an estimate for the total execution time:

$$T_{total} = \max \begin{cases} T_{PPE} \Rightarrow n \cdot [(\frac{0{,}19}{10^3} + \frac{2{,}25}{10^3} \cdot \frac{m}{10^2} \cdot P_{TB}) \cdot \bar{l} + (\frac{5{,}52}{10^3} + \frac{35{,}7}{10^3} \cdot P_{TB})] \\ T_{SPE} \Rightarrow n \cdot (\frac{0{,}59}{10^3} \cdot \frac{m}{10^2} \cdot \bar{l} + \frac{0{,}88}{10^3})/q \end{cases} \tag{6.13}$$

6.3.2 Model Validation

The validity of the model as proposed in the previous section has been investigated by performing additional tests. This way, the accuracy of the various functions can be estimated. Table 6.5 summarizes the findings for the various test conditions. Again, representative sequence sets have been created, as such, average sequence length is about 355 symbols (see Section 2.4). The table shows the expected result as predicted by the model for each function as well as the actual result.

From the table, it is obvious that the execution time for PPU_BUF and SPU_VIT can be accurately estimated. The average difference between result and expectation for PPU_BUF is 1.5%, for SPU_VIT 1.7%. However, estimation for PPU_TB is unreliable for two reasons: first, estimating the number of required tracebacks is difficult as it is dependent on the biological fit between data set and model. Second, in contrast to the SPE Viterbi code, the time per traceback on the PPE varies considerably.

The inaccuracy in PPU_TB estimation only affects overall performance estimation when two conditions are fulfilled: first of all, performance must be constrained by the PPE in order for the inaccuracy in PPU_TB to be reflected in the results. Secondly, PPU_TB must form a substantial part of total PPE execution time. This occurs when a high fraction of sequences in the test set requires traceback. However, recall that traceback count scales logarithmically in test set size. For common scenario's, such as a comparison against the entire UniProt database with its hundreds of thousands of sequences, the percentage of tracebacks P_{TB} is marginal. In such cases, PPU_TB will contribute little to total execution time and its influence on uncertainty in the model is small. Due to the practical reason of test execution time and due to limitations in available storage space, relatively small test set sizes have been used, where the impact of traceback remains of influence. Furthermore, one should realize that traceback counts are dependent on biological semantics. As such, their estimation is inherently uncertain. For example, a data set solely containing homologues to the model would result in a traceback percentage of 100%!

6.3.3 Potential Bottlenecks

Based on the profiling results and the model shown in this chapter, some observations on potential bottlenecks can be made. The program structure and implementation architecture both have an effect on the specific performance characteristics. Two issues will be discussed here: on the one hand the structural decision in HMMERCELL to use the manager/worker parallelization strategy; on the other hand the architectural choice in the Cell for a certain SPE count and the choice for a local store, which is directly responsible for the introduction of the traceback mechanism.

An important structural characteristic of HMMERCELL is the manner in which Lu et al [29] parallelized the most time-consuming portion of HMMER, the phase in which Viterbi calculations are performed. The manager/worker pattern was followed, where the PPE creates Viterbi jobs, which are processed on the SPEs. The Viterbi calculations lend themselves well to the nature of the SPEs, as the workload is very regular and computation to communication ratio is very high. In theory, as many SPEs can be used as there are sequences. Hence, for a typical test set consisting of many thousands of sequences, there is no practical limit on the number of SPEs, given that the PPE is able to supply them with jobs. However, job creation forms a bottleneck and the use of this pattern implies that for any workload a certain SPE count will saturate the PPE.

A notable characteristic of the Cell architecture is the fact that each SPE contains its own limited size local memory. This requires the use of the reduced Viterbi algorithm on SPEs, which leads to the introduction of the traceback phase on the PPE. Traceback forms another bottleneck, as this PPE function is not parallelized. Additionally, the traceback mechanism introduces inherent uncertainty, as discussed before. Other mechanisms could offer more predictable time estimation for workloads, especially for smaller test sets. However, for large test sets, such as comparing a model against the entire UniProt database, only a negligible fraction of sequences require traceback; PPE time is dominated by the buffering phase. Thus, uncertainty in this function does not adversely affect overall inaccuracy of the model. Moreover, in buffering constrained cases,

HMM Length	100	200	300	400	500
q (max SPE count)	3	6	9	12	15

Table 6.6: Maximum effectively usable SPEs.

the number of SPEs that saturate the PPE for a given workload can be estimated by assuming that PPE time consists solely of buffering time and then solving for q when setting T_{PPU_BUF} equal to T_{SPE}, which leads to the following equation:

$$q \approx \frac{T_{PPU\_BUF}}{T_{SPE}} = \frac{n \cdot (\frac{0,19}{10^3} \cdot \bar{l} + \frac{5,52}{10^3})}{n \cdot (\frac{0,59}{10^3} \cdot \frac{m}{10^2} \cdot \bar{l} + \frac{0,88}{10^3})} = \frac{0,59 \cdot \frac{m}{10^2} \cdot \bar{l} + 0,88}{0,19 \cdot \bar{l} + 5,52} \tag{6.14}$$

From Equation 6.14 it follows that the number of usable SPEs is independent of the test set size, apart from the requirement of a large enough test set to make PPU_TB irrelevant. Moreover, it is only marginally affected by the sequence length. The most influential factor is the HMM model size. Table 6.6 gives the maximum number of usable SPEs for various HMM sizes when using sequences with typical length. For example, when an HMM of length 200 would be aligned to the UniProt database, six SPEs would be enough to saturate the buffering capabilities of the PPE. This formula allows for determination of the optimal ratio between PPEs and SPEs.

6.4 Summary

In this chapter, results from analysis of the different functions that constitute HMMER-CELL performance were discussed. The effects of the input parameters - the HMM, the set of sequences and the number of SPEs - on performance have been investigated. Scaling results were as follows: PPE buffering scales linearly in the total length of all the sequences in the test set; the Viterbi calculations on the SPEs require time linear in the length of the sequences and linear in the length of the HMM; the traceback Viterbi calculations on the PPEs are more difficult to estimate, as the traceback count is data dependent.

A model to estimate the required processing time for a workload was proposed and validated. The models for the buffering and SPE Viterbi functions are accurate within two percent; the model for the traceback function is, as expected, much less exact. However, for typical test sets, such as comparing a model against the UniProt database, the fraction of execution time resulting from traceback calculations is negligible. In general, the model predicts execution time accurately. Consequences of the manager/worker parallelization structure imply that for any given workload, a certain SPE count will saturate the PPE. This number is mostly dependent on the length of the protein model. For a typical model of length three hundred, nine SPEs can be put to effective use.

7 Simulation Results

Earlier chapters showed the result from investigation of the behavior of HMMER and HMMERCELL on an actual available system, the Cell platform. Through qualitative analysis the internal structure was determined, which aids in the understanding of its behavior. Quantitative analysis, through profiling of the implementation and through accurate modeling of the relevant functions, revealed its performance characteristics. Both types of analysis help to gain insight in the manner the application reacts to different workloads. Bottlenecks and limitations were found in the capability of the application to scale in the number of SPEs. The importance of each bottleneck shifts depending on the workload. For example, when aligning sequences against a short HMM, PPE buffering quickly becomes the bottleneck to performance as the SPE's capability to perform Viterbi decoding exceeds the PPE's buffering ability. Conversely, for longer HMMs, Viterbi processing on the SPEs becomes the dominant factor of execution time. When the number of sequences in the input is small, traceback calculations on the PPE require a large amount of time relatively.

In this chapter, simulation is used to gauge performance of the application. Changes to the implementation structure and hardware architecture can be investigated to discover promising methods to improve the fit between the application and implementation architecture. The analysis is performed for hypothetical variants of the Cell platform. As shown earlier, the Cell platform roadmap indicates that future iterations of the architecture will possess a greater number of PPEs and SPEs. It is therefore interesting to analyze the effects of such changes on the existing bottlenecks and to deduce the optimal clustering ratio between PPEs and SPEs for a given workload. Results are obtained through the use of TaskSim, an experimental and as of yet unreleased Cell simulator.

In Section 7.1, simulation of the Cell architecture is discussed and the simulation environment is introduced. In Section 7.2, the concept of scaling behavior is explained and in Section 7.3 the effects of the various functions on scaling behavior is analyzed by inspecting their limiting behavior. In Section 7.4, the results are related to the findings from Chapter 5 and 6. Section 7.5 concludes the chapter with a summary.

7.1 Simulating the Cell Architecture

As discussed in Section 4.4.4, the Cell Broadband Engine is the first iteration of a future platform of Cell processors. It contains one core aimed at general purpose tasks combined with eight cores specialized for streaming workloads. Next generations of the platform can evolve in multiple directions: architectural improvements can be implemented, such as native support for double precision floating point operations (as already planned for the PowerXCell 8i), enlargement of the SPE local store, enlargement of the cache, or

improvements to the communication channel; another probable direction is the addition of more cores to the processor die. The current ratio between PPEs and SPEs does not have to be maintained; it is probable that, based on developer experience and an improved software infrastructure better able to take advantage of the SPEs, this ratio will be skewed towards more SPEs per PPE. As an illustration, one planned extension of the current layout is the PowerXCell 32ii with two PPEs and thirty two SPEs.

In this chapter, the impact of such developments is assessed by simulating the effects of such improvements. Simulators model program behavior according to a hardware description of the architecture. By modifying this description, the characteristics of the architecture can be changed. The TaskSim tool, which is used to perform the simulations, is introduced first. Then, the accuracy of this tool is investigated, in order to discover whether its results conform to observations.

Simulating a hardware architecture has several benefits. During the design stages of a microprocessor, preliminary simulations can give the developer valuable feedback, which can help during design space exploration to make decisions about trade-offs. Simulators can also be used to start software developments before actual hardware is available. Another important use, especially for this thesis, is the ability of a simulator to simulate modified designs of a processor, for example simulating a machine description with more or faster execution cores.

Disadvantages of simulation include the fact that simulation is time-consuming. Depending on the detail level that the simulation is performed at, speed or accuracy can be impaired. High-level simulations can be completed relatively swift, as they only model a few basic characteristics. Cycle-accurate simulations on the other hand require a large amount of processing time, but then provide more accurate details. As always, the best type of simulation depends on its purpose.

7.1.1 The TaskSim Simulator

TaskSim is a high-level, trace-driven simulator. Trace-driven simulators use an input trace, which is a pre-determined path through the program, and a machine description to generate an output trace with the resulting simulated behavior. In contrast, execution-driven simulators simulate the actual program, and hence they require both the program and input data. Then, based on this input data different paths through the program might be taken. TaskSim uses its own TaskSim Trace Format (TTF) for traces, but tools are provided to convert other trace formats into TTF. The behavioral description it simulates is composed of three aspects: processing phases, DMA data transfers, and synchronization dependencies between threads. The CPU is simulated as performing abstract bursts of computation, with duration obtained from the traces. Data transfers are simulated cycle-accurate (amongst others the DMA controller, caches, interconnect, memory controller, and DRAM). Ratio files can manipulate the duration of program phases. TaskSim is in development at the BSC [1].

> "TaskSim is a computer architecture simulator that allows several detail levels from programming-model to micro-architecture level. The application can be split up in phases, each one being simulated at a different detail level. This allows speeding-up simulation while avoiding accuracy penalties."

Simulation Trace Characteristics	
"Complex"	Aligning 40k sequences against length 500 HMM (87 tracebacks).
"Typical"	Aligning 40k sequences against length 300 HMM (108 tracebacks).
"Simple"	Aligning 40k sequences against length 100 HMM (120 tracebacks).
Number of SPEs Used	
1, 2, 4, 8, 16, 32*, 64* (see text)	

Table 7.1: Simulation trace overview.

To obtain TaskSim traces, the HMMERCELL source code is instrumented further to generate traces that record execution phase duration, DMA transfers and synchronization between threads. The important phases are: PPU_BUF, SPU_VIT with corresponding idle phase SPU_WAIT_FOR_JOB (when an SPE is waiting for a job), and PPU_TB with corresponding idle phase PPU_WAIT_FOR_TB (when the PPE is waiting on SPEs to complete their job). Dependencies that ensure correct program order are inserted between buffering a sequence and its processing on the SPE, and between processing a sequence and its traceback on the PPE. Recording when a phase is idle is important, as differentiating between computation phases and time spent waiting on other threads allows for the elimination of such idle time if dependencies are fulfilled earlier (an example is an SPE waiting for a job to become available). The trace is then converted into TTF format and fed to TaskSim for simulation, according to the configurable machine description. The resulting trace is inspected using Paraver [37].

Traces have been created for three different trace scenarios, listed in Table 7.1. Each trace covers the alignment of forty thousand sequences with typical sequence length. Varying HMM sizes are used to highlight the impact of their length on scaling behavior. The number of threads in the trace is determined by the number of used SPEs. Traces are obtained with one, two, four, eight and sixteen SPEs. The results for thirty two and sixty four SPEs are obtained by using the sixteen SPE trace in combination with a modified ratio file, in which the SPE execution is sped up two-fold and four-fold respectively. The accuracy of this approximation is reviewed in the next section.

7.1.2 TaskSim Validation

In order to verify the accuracy of the simulations performed by TaskSim, results from profiling the actual application are compared to the traces generated by TaskSim. For these tests, TaskSim simulates the traces according to a standard Cell machine description, with execution times for idle states set to zero. This is achieved through the use of TaskSim ratio files. As the Cell blades contain two Cell processors with in total sixteen SPEs, validation can only be performed for simulations with up to sixteen SPEs. Also shown are results when using one SPE in combination with a ratio file that speeds up its execution in order to emulate more SPEs. Validation tests have been run for each of the three different trace scenarios.

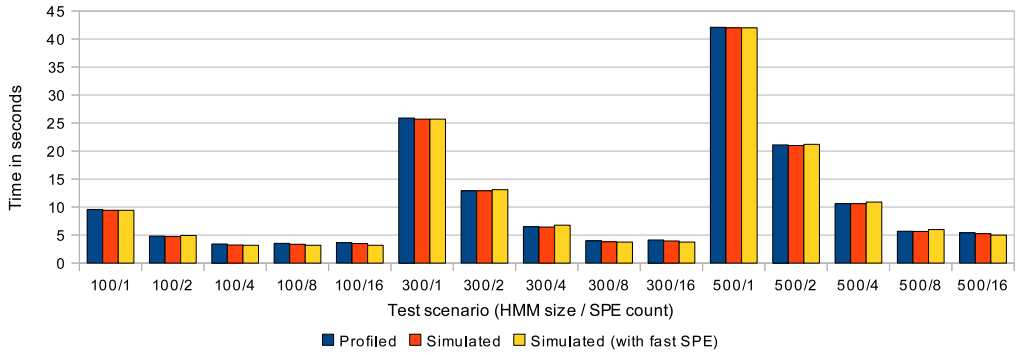


Figure 7.1: TaskSim validation results.

Figure 7.1 shows the results. Simulated and actual execution time are comparable, the largest difference being 5%. These differences result from the fact that TaskSim does not simulate waiting states, such as looping through the administrative array during the PPE traceback phase to find newly processed jobs. The results from simulation using multiple SPEs and a sped up SPE are also similar. They differ on average by 3%, with an outlier of 9% for the 100HMM/16SPE case. The inaccuracy results from a combination of less initialization overhead and less efficient bus utilization.

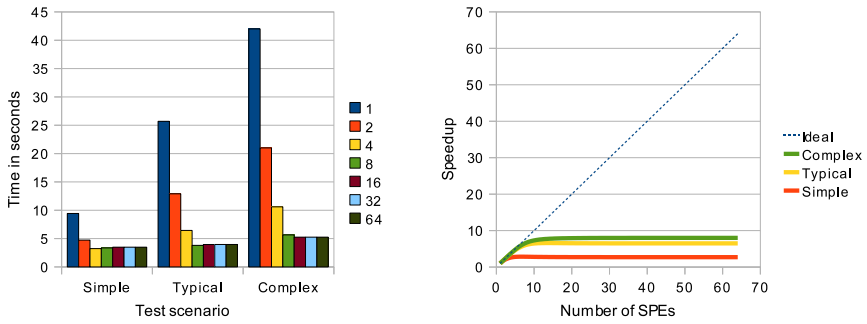
The results prove that even though TaskSim performs its simulations at a high-level, enough characteristics are modeled for the simulation to be very accurate. Ratio files also prove to be a useful tool for simulation. In the case of HMMERCELL, trace-driven simulation is suitable, as the execution time is largely independent of underlying data. For similar sized inputs, similar instruction streams are executed.

7.2 Improvements to Scaling Behavior

The growing importance and interest in genetics leads to continuously increasing amounts of biosequence data. Consequently, manipulation of these data sets necessitates continuously larger processing capabilities. As seen in Chapter 4, a clear trend in microprocessor design exists towards multi-core architectures. Hence, it is important for bioinformatics applications to be able to utilize such parallelism and to have favorable scaling metrics. In the remainder of this chapter, the effects of the various functions are investigated on the scaling behavior of HMMERCELL.

A characteristic feature of bioinformatics applications is that they are often amenable to various forms of parallelization. Coarse-grained parallelism is available in the form of computations that must be applied to all elements in a data set. Fine-grained parallelism is available as the computations themselves usually contain data parallelism. In the case of HMMER, these types correspond to the alignment of sequences to an HMM through the application of the Viterbi algorithm. The abundance of sequences in the data set makes the coarse-grained parallelism infinite for practical purposes. In the limiting case, each of the thousands of sequences could be processed by its own processor.

HMMERCELL has been subjected to different forms of analysis, resulting in the



(a) Simulated execution time (b) Speedup from SPE scaling

Figure 7.2: Standard simulation results.

identification of certain bottlenecks to scaling behavior. These findings can be used as a starting point for improvements. From Chapter 5, in which the program structure was analyzed, it followed that bottlenecks arise from the design decision to have one processor responsible for both job creation and the processing of the results. In Chapter 6, in which profiling results were shown and a model for system behavior was proposed, the existence of these bottlenecks was confirmed. The model can be used to estimate the optimal clustering size between managers and workers, or in the case of the Cell processor, PPEs and SPEs. Preferably, a situation would be reached where all parts of the architecture's computational and communication abilities are fully utilized.

Amdahl's law (see Section 4.3.1) describes the maximum speedup that can be obtained by the parallelization of a program, giving an upper bound for the maximum obtainable speedup of a program. At some point, the non-parallelizable part of the application will prevent further speed-up from using additional processing cores. This is illustrated in Figure 7.2, which shows simulated scaling capabilities of HMMERCELL for the different test sets. The graph on the left shows execution time for varying SPE count; the graph on the right displays the resulting speedup from increased SPE count. A fully parallelizable application would show scaling behavior that follows the ideal line: an increase in the number of processors would lead to a corresponding decrease in execution time. However, from the figure it is obvious that scaling is only effective up until a certain SPE count, depending on the scenario. The serial nature of buffering and traceback limits further scaling. In Section 6.3.3, limits were given for the number of SPEs that the PPE could supply with jobs, based on the proposed model. Due to traceback requirements and other overhead, these findings do not fully agree. However, as will be shown in Section 7.3.2, when traceback is disregarded the results do confirm the predictions given by the model.

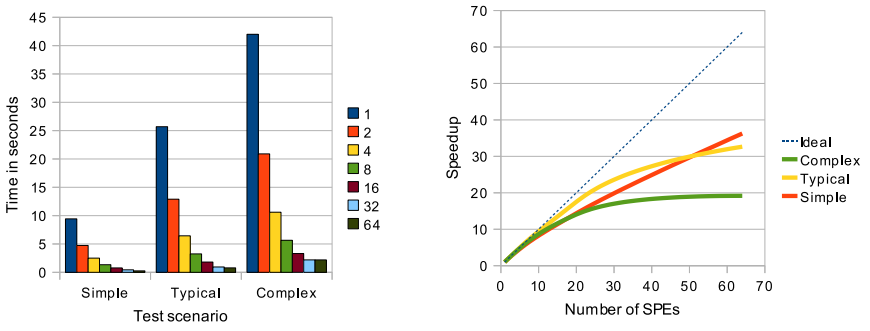
7.3 Limiting Behavior Analysis

To determine promising directions to improve the scaling capability, the limiting behavior of the various functions of importance in the application (PPE sequence buffering, SPE Viterbi calculations and PPE traceback) is investigated. Simulations are run with the

processing component of each of these functions set as to require zero time. This is achieved through the use of TaskSim ratio files. These limiting behavior tests provide valuable information on the existence of bottlenecks in HMMERCELL.

In the following subsections, the results from these simulations are presented. For each of the tests, the simulation results are presented in two different graphs. In the first graph, the execution time is given for different SPE counts. In the second graph, the speedup gained from utilizing more than one SPE is shown. The results are explained and possible methods to reach such limiting situations are mentioned, through improvements in application structure or improvements in hardware architecture.

7.3.1 Fast PPE Buffering



(a) Simulated execution time

(b) Speedup from SPE scaling

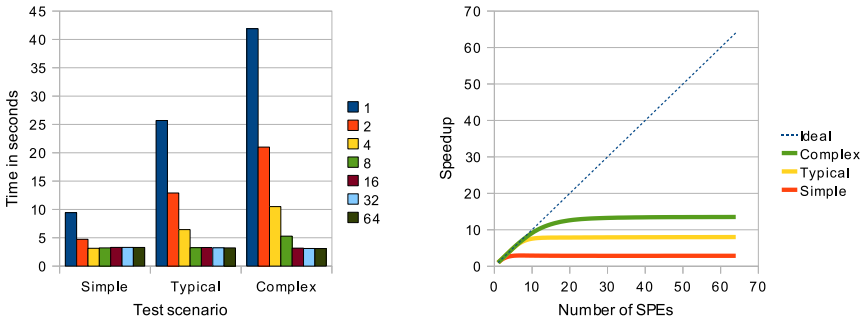
Figure 7.3: Swift buffering simulation results.

In this simulation, the time required for sequence buffering is set to zero through the use of a ratio file. The increased buffering capability leads to behavior as shown in Figure 7.3. Compared to normal HMMERCELL scaling behavior (see Figure 7.2), scaling is much closer to the ideal speed-up line, i.e. scaling behavior is much improved.

The limiting behavior is as expected. As the buffering function corresponds to the manager role in the manager/worker pattern, it is a major limitation in scaling capability. SPE Viterbi function scaling is almost ideal (recall Section 6.2.2). In this simulation, scaling is ultimately bottlenecked by the traceback component, as this function is performed sequentially on the PPE as well. Then, the PPE determines the execution time. As traceback is dependent on HMM length, the complex test simulation (the green line) is quickly bottlenecked by traceback. The typical scenario can be scaled to more SPEs and the simple scenario scales the best. More complex scenarios are relatively less affected by overhead, as it becomes a relatively smaller portion of execution time. This explains why the typical scenario temporarily outperforms the simple scenario.

From the figure it is obvious that PPE buffer time has a large effect on scaling capability. The speed of buffering determines for a large part the optimal ratio between managers and workers. As such, an optimal clustering ratio between PPEs and SPEs can be determined, as shown in Section 6.3.3. A faster PPE allows more SPEs to be used per PPE.

7.3.2 Fast PPE Traceback



(a) Simulated execution time (b) Speedup from SPE scaling

Figure 7.4: Swift traceback simulation results.

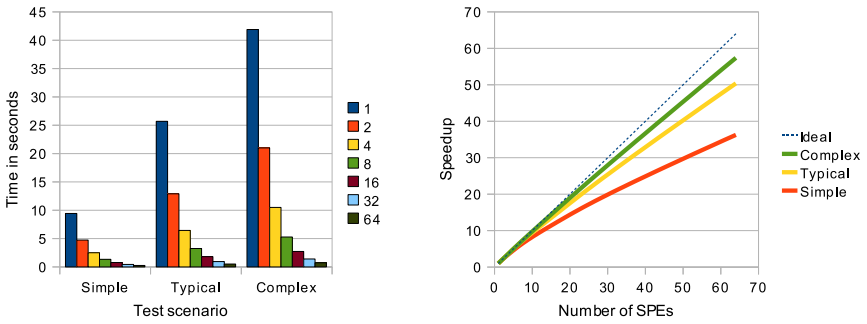
In this simulation, traceback on the PPE is practically eliminated by setting the required traceback time to zero through the use of a ratio file. The results are shown in Figure 7.4. Compared to normal HMMERCELL scaling behavior (see Figure 7.2), the limiting behavior stabilizes at a higher SPE count. In other words, more SPEs can be utilized effectively. The results also prove that traceback indeed forms a bottleneck, although its importance decreases for larger test sets, as less sequences require traceback.

In this situation, scaling characteristics are mostly determined by buffering speed. The results are therefore in accordance with Table 6.6, which lists the number of SPEs that can be used effectively for different HMM sizes, confirming the predictions made using the analytical model. From this simulation, the effective number of SPEs appears to be three, eight and fourteen for models of length 100, 300 and 500 respectively; the model predicted three, nine and fifteen. The difference is caused by overhead from non-parallelizable parts in the application.

The traceback bottleneck can be reduced through the use of a faster PPE or by parallelizing traceback to multiple PPEs. Parallelization is feasible, as tracebacks can be performed independent of each other, and the number of tracebacks, although relatively small compared to the total test set size, is still large enough to benefit from more processors. The scaling bottleneck from traceback could also be removed entirely by either parallelizing traceback on the SPEs through a Viterbi algorithm that uses streaming to circumvent the small local store size. Another option is to perform the full streaming Viterbi algorithm on the SPEs for all sequences. However, this last option might adversely affect total execution time as the number of tracebacks is insignificant compared to the total amount of sequences. Moreover, such techniques are only useful when execution time is not dominated by SPU Viterbi calculation time or PPE buffering time.

7.3.3 Fast PPE Buffering and Traceback

In this simulation, both bottlenecks on the PPE-side are eliminated. The results are shown in Figure 7.5. Compared to normal HMMERCELL scaling behavior (see Fig-



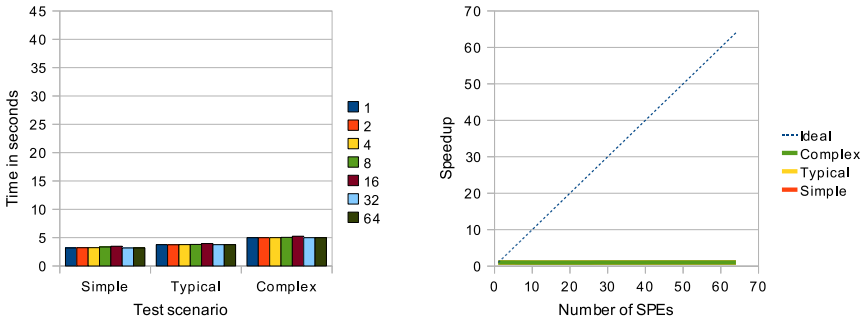
(a) Simulated execution time (b) Speedup from SPE scaling

Figure 7.5: Swift buffering and traceback simulation results.

ure 7.2), the scaling behavior is much more favorable. Basically, it shows that the resulting application is very parallelizable, as the manager role in the manager/worker pattern is removed as a bottleneck. Overhead from non-parallelizable parts in the application causes the deviation from the ideal line. The more complex the scenario is, the less it is affected by this overhead. It is also interesting to compare this figure to Figure 7.3. This comparison reveals the impact traceback has on scaling behavior.

Removing these bottlenecks can be achieved by following the directions from both earlier sections.

7.3.4 Fast SPE Viterbi Calculations



(a) Simulated execution time (b) Speedup from SPE scaling

Figure 7.6: Swift Viterbi simulation results.

In this simulation, the SPEs are removed as a practical bottleneck by reducing the time required for Viterbi processing to zero. The results are shown in Figure 7.6. It is obvious that the results are independent of SPE count, as the parallelizable part of the application is eliminated. This way, HMMERCELL is reduced to a non-parallelized application. In terms of Amdahl's law, it is the situation where the parallelizable part is infinitely sped up and P goes to zero (see Section 4.3.1).

The remaining execution time is the combined overhead from buffering and traceback on the PPE. As buffering requires the same time for each scenario (only the HMM differs), the variations are caused by differences in traceback time resulting from HMM length and traceback count. More complex HMM models result in longer traceback time. Also noticeable is that traceback time compared to buffer time is quite small.

This situation can be reached through the inclusion of more SPE cores, or using faster SPEs. More SPEs and faster SPEs basically results in the same outcome, since the Viterbi function scales very well. Latency per individual sequence is somewhat higher when using multiple SPEs, but throughput is unaffected. Usually, large numbers of sequences are processed, thus the most efficient option of the two should be used. This is also confirmed by the tests with one sped up SPE as compared to many SPEs (see Section 7.1.2). However, most workloads will be bottlenecked by buffering and/or traceback. Hence, faster SPE processing is mostly useful in situations where large HMMs are used.

7.4 Discussion

In order to investigate the performance characteristics of HMMERCELL and to be able to improve its fit to the implementation architecture, the application has been subjected to various forms of qualitative and quantitative analysis: program inspection, performance profiling, behavioral modeling and simulation. In this section, the findings from each form of analysis are reviewed and lessons to improve performance and scaling behavior are drawn.

In Chapter 5, the results from program inspection were discussed, revealing details of its internal structure. Interesting features of the application include: the specific algorithm used and how it reacts to the workload (sequence length, sequence count and HMM size); the parallelization strategy that was followed to parallelize this algorithm (the manager/worker pattern); and the necessity for traceback, as an algorithm with smaller memory footprint was used on the SPEs to let it fit into the small local store size. All of these influence performance and form potential bottlenecks to scaling behavior.

In Chapter 6, the actual effects of these bottlenecks were investigated through the profiling of each function. The scaling characteristics of each function agreed with expectations set by the used algorithm, and the relative importance of each function was determined. The buffering function on the PPE and the Viterbi calculations on the SPE were concluded to be most influential for performance, as they are linearly dependent on the number of sequences in the workload, whereas traceback reduces in importance for larger workloads. Based on these results, a model to predict the execution time was proposed and verified to be accurate within 2%. Traceback is excluded from the model, as it is assumed to be negligible. Using this model, the optimal PPE-to-SPE cluster ratio was derived for various HMM lengths. For an HMM with a typical length of three hundred positions, nine SPEs should be able to be effectively used.

In this chapter, simulations were used to predict the behavior of HMMERCELL for future iterations of the Cell architecture. Even though TaskSim simulates the system at a high-level, its phase-based simulations are accurate within 2%. The impact of each of the identified bottlenecks on the SPE scaling ability of HMMERCELL was determined.

The simulations supported the predictions made by the model for such situations. The impact of each function on scaling behavior was investigated under different workloads. Buffering is shown to be the most important determinant to scaling potential, which agrees with expectations set by the use of the manager/worker pattern. Traceback mostly affects smaller workloads consisting of just a few sequences. Simulations with the traceback component of the application eliminated confirmed predictions from the model, as the SPE scaling behavior acted as was described. These findings can be used to discover the most efficient ways of improving performance.

From a software perspective, the impact of the software design decisions on the program's execution time is of importance. The manager/worker split, which is used to parallelize the most time consuming portion of HMMER to the SPEs, is directly responsible for creating a bottleneck at the PPE, determining the effective number of SPEs that can be utilized. The traceback mechanism, used to circumvent the small local store size, has been shown to work well as its detrimental influence to performance and scaling behavior is reduced for larger and thus more relevant workloads. However, as the PPE often already forms a bottleneck, parallelizing traceback to SPEs is a valid software-based method to further improve the scaling behavior, especially for smaller workloads where traceback time is significant. The effects of this improvement can be seen by comparing the chart with standard behavior to the chart with fast traceback behavior (Figure 7.2 and 7.4), or by comparing the chart with fast buffering behavior to the chart with fast buffering and traceback behavior (Figure 7.3 and 7.5).

When viewed from a hardware perspective, a few observations can be made. First of all, the Cell appears to be well-suited to HMMER. An abundance of both coarse and fine-grained parallelism is available to keep all parts of the processor occupied. Also, the Viterbi algorithm is a natural fit to the SPEs and they perform very fast when compared to the PPE. An important indicator to track from a hardware perspective is the utilization of the hardware. Ideally, the PPE and the SPEs are kept busy during all stages of program execution. As shown by the model, the relative weight of each function depends on the workload, but is mostly dependent on the HMM model length. Hence, the optimal PPE/SPE ratio can be determined for typical model sizes. This is useful when designing improvements to the hardware to ensure that the resulting architecture is balanced.

In the case of the Cell Broadband Engine with its eight SPEs, sequence alignment against an HMM model of around three hundred residue positions results in optimal PPE/SPE utilization. In that case, neither the SPEs are constrained by the job creation rate, nor the PPE is forced to idle during traceback while waiting for jobs to be processed by the SPEs. One approach to improve efficiency between program and hardware would be to create multiple Cell derivatives, each with a different PPE/SPE ratio. Then, a task could be executed on the specific machine for which its model length is optimal: workloads requiring alignment against a short HMM would be processed on a Cell with just a few SPEs; workloads requiring alignment against a longer HMM would be processed on a Cell with a great number of SPEs. Such a setup would ensure optimal utilization of resources.

7.5 Summary

In this chapter, simulation results generated by TaskSim, a trace-driven simulator, were shown. Its high-level, phase-based simulations reflect actual program execution with an accuracy within 2%. To determine limits in scaling behavior, simulations were performed with Cell-like machine descriptions with SPE count ranging from one to sixty four SPEs. The impact of HMMERCELL's important functions was investigated by setting the execution time for each of these functions to zero through the use of TaskSim ratio files.

From the simulations where execution time per function is set to zero, the impact on execution time can be determined. The buffering portion appears to be the most important, confirming findings from the earlier chapters. As the PPE determines the maximum number of effectively usable SPEs, parallelizing traceback to the SPEs is a software-based method that can result in improved scaling behavior, even though its importance diminishes for larger workloads. The model and the findings can also be used to design Cell variants that are optimally suited to the HMMER workload. As the effective PPE/SPE ratio is dependent on the length of the HMM model, using multiple Cell variants with different ratios ensures optimal utilization under each workload.

PART III:
LEARNINGS

Hofstadter's Law:
"It always takes longer than you expect, even when you take into account Hofstadter's Law."
- Gödel, Escher, Bach, Douglas Hofstadter

Conclusions and Recommendations

8

Bioinformatics applications such as HMMER are recognized as an important class of programs. Their uses are manifold and they will allow numerous novel methods to improve quality of life around the world. Their often computationally intensive nature combined with the exploding growth of biological data sets creates a continuously increasing demand for processing capability. Multi-core architectures form a new development in computer architecture, as a mechanism to avoid the power, memory and frequency scaling constraints of traditional microprocessor design, in an attempt to ensure the continuity of processing power increases. However, the suitability and effectiveness of the multi-core paradigm on bioinformatics applications remains an open research question. To further the understanding of their structure and scaling capability, this thesis contributes by examining the behavior of a biosequence analysis application called HMMER on the Cell architecture, a state-of-the-art heterogeneous multi-core processor.

As stated in the introduction, the three research goals for this thesis were as follows:

1. Investigate the performance of bioinformatics applications, gaining a thorough understanding of their structure and behavior, through the analysis of a representative application called HMMERCELL.

2. Identify bottlenecks in program operation and system architecture by profiling, modeling and simulating the system's behavior.

3. Improve the understanding of scaling behavior for many-core computing platforms by evaluating the impact of the identified bottlenecks.

Chapter 2, 3 and 4 contain a short introduction on bioinformatics, Hidden Markov Models and computer architecture respectively. In Chapter 5, the program structure and behavior of HMMER and HMMERCELL was presented, resulting from qualitative analysis. Chapter 6 showed results obtained from application profiling and a model for run-time behavior was proposed. In Chapter 7, simulation results on Cell-like machine architectures were given. These results were related to the findings from earlier chapters. In Section 8.1, the conclusions drawn from this work are presented. The thesis is closed by Section 8.2, in which promising directions for future research are indicated.

8.1 Conclusions

To investigate the suitability of HMMER to the Cell microprocessor and to be able to judge the effectiveness of improvements to it, knowledge of all facets of the application and the implementation architecture is required. Therefore, various forms of analysis

have been used to provide insight in how HMMER and HMMERCELL are structured, and how the port plays into the strengths and weaknesses of the Cell architecture, and how bottlenecks affect performance. The different forms of analysis all provide different clues about application behavior, and each complements the findings of the others.

The following was learned from program inspection: like most bioinformatics applications, HMMER contains a small but computationally intensive kernel that is applied to each element of a huge data set. This implies that both coarse-grained as well as fine-grained parallelism is available in abundance, which is a requirement for optimal performance on the Cell architecture. In the case of HMMER, this amounts to aligning protein sequences to a Hidden Markov Model using the Viterbi algorithm. For each alignment, a score shows the likelihood of the two being related. HMMER's execution time is determined almost exclusively by this function. Lu et al [29] parallelized the Viterbi algorithm according to the manager/worker pattern. One core buffers jobs into memory, these jobs are then consumed by the other cores; the results are processed by a traceback procedure. Thus, one core is responsible for supplying the other cores with jobs, which forms a potential bottleneck in scaling behavior. A Cell-specific design choice is the fact that the workers perform an indicator function: a modified Viterbi algorithm with a smaller memory footprint (necessary to make it fit into the small accessible memory of most of the Cell's cores) is used to designate high scoring sequences. For those sequences, the requested alignment is generated by performing the full Viterbi algorithm. This traceback procedure forms another potential bottleneck.

The following was learned from quantitative analysis: the existence were verified to exist and their effects quantified through application profiling, behavioral modeling and simulation. Profiling results confirmed that the scaling behavior of each function was in line with expectations set by the use of the Viterbi algorithm. The analysis also demonstrated the relative importance of the functions. Buffering is regarded as being the most important determinant to scaling behavior. Traceback was shown to be of relative minor importance: it scales far slower in the number of elements in the input than other functions. A model to predict execution time for the important functions was proposed and shown to be accurate within 2%. An important use of the model is to derive the optimal ratio between manager and worker cores for different workloads. For the Cell processor, this optimal ratio is determined to be one manager to nine workers for an HMM with three hundred residue positions. The TaskSim simulator was used to measure the impact of the bottlenecks on HMMERCELL's ability to scale in the number of usable worker cores. Its high-level, phase-based simulations are demonstrated to be a fast and effective means of simulating behavior: simulation results are within 2% of actual results. Moreover, results are in agreement with predictions made by the model, in particular when the traceback phase of the program is removed. Scaling capability is mostly determined by the speed at which jobs are able to be generated.

Software and hardware-based design choices both affect scaling behavior. The use of the manager/worker pattern directly determines the number of workers that can be utilized effectively. The traceback mechanism works well, as its negative influence on performance diminishes with larger test sets. As the manager core is often a bottleneck, parallelizing traceback to the other cores is an valid software-based method to further improve scaling behavior. From a hardware perspective, the Cell appears well-suited

to HMMER and bioinformatics applications, as there is an abundance of parallelism available. The Viterbi algorithm itself maps well to the SPEs and their SIMD nature. The model can be used for guidance during the design stages of hardware development, as the optimal ratio between manager and worker cores can be estimated to optimize utilization. The limited size of the local store restrict the range of sequence lengths and especially HMM models that can be used. Enlarging the local store would allow for a greater percentage of models to be used.

The aforementioned ideas are also relevant for other bioinformatics applications. First, as bioinformatics applications usually contain an abundance of coarse-grained parallelism, the manger/worker pattern is an useful strategy to divide the work over multiple cores. Care has to be taken however that the manager core is able to supply the worker cores with enough jobs. Therefore, in order to attain optimal scaling behavior, this core should be relieved of as many other tasks as possible. Second, modeling the behavioral characteristics of a program has multiple uses: it is a valuable aid for decision-making during design space exploration; it can point out interesting places to start with performance improvements; or it can help to optimize run-time scheduling of workloads if machines with variable performance characteristics are available. Computationally intensive workloads can be scheduled on machines with many worker cores, lighter tasks on machines with less worker cores. Third, phase-based simulation appears to be an accurate and relatively fast approach for the simulation of bioinformatics workloads. The data-independent nature of their calculations is captured well by such simulations.

The approach this thesis takes to capture the essential nature of bioinformatics applications seems valid. The different analysis techniques - inspection, profiling, modeling and simulation - complement each other well and their combined use helps to create insight in the relevant performance characteristics of the application.

8.2 Recommendations

Genetics and computer architecture are fields both subject to constant change. As such, they contain many open research questions. A relevant question in the context of this thesis is the suitability of multi-core architecture to bioinformatics applications. This thesis contributes by gaining understanding in the scaling behavior of one such application, HMMER. Other applications would benefit from similar research. An example that immediately comes to mind is analysis of HMMER3, when it becomes available.

The thesis itself also offers a few interesting directions for future research. In the simulations section, a number of software-based methods were mentioned that could lead to improved scaling behavior. The implementation of such changes is outside the scope of this thesis. However, for example parallelizing the traceback phase to the SPEs using a streaming algorithm and comparing the resulting behavior with expectations set in this thesis would be of interest.

Simulations were used to forecast performance of Cell-like derivatives with more SPEs than the Cell Broadband Engine contains. When a Cell platform becomes available offering such SPE counts, it would be interesting to compare the simulated scaling behavior to results obtained on actual hardware. An ideal candidate would be the announced PowerXCell 32ii, which will contain thirty two SPEs.

Bibliography

[1] *Barcelona Supercomputing Center website*, http://www.bsc.es.

[2] *The EMBL nucleotide sequence database, statistics*, http://www.ebi.ac.uk/embl/Services/DBStats/.

[3] *Folding@Home client statistics by OS*, http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats.

[4] *Intel Tera-scale Computing Research Program website*, http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[5] *SPEC CPU2006 benchmark suite*, http://www.spec.org/cpu2006/.

[6] *The Cell project at IBM Research*, http://www.research.ibm.com/cell/.

[7] U.S. Environmental Protection Agency, *Report to congress on server and data center energy efficiency.*

[8] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, *Basic local alignment search tool.*, J Mol Biol **215** (1990), no. 3, 403–410.

[9] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, (2000), 79–81.

[10] Barcelona Supercomputing Center, *MPI trace tool user's guide*, 2000, http://www.bsc.es/media/1379.pdf.

[11] F. H. C. Crick, *On protein synthesis*, Symp. Soc. Exp. Biol. **12** (1958), 138–163.

[12] ———, *Central dogma of molecular biology*, Nature **227** (1970), no. 5258, 561–563.

[13] M. O. Dayhoff, R. Dayhoff, M.O., Schwartz, and B.C. Orcutt, *Atlas of protein sequence and structure*, volume 5, supplement 3 ed., pp. 345–358, Nat. Biomed. Res. Found., 1978.

[14] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison, *Biological sequence analysis : Probabilistic models of proteins and nucleic acids*, Cambridge University Press, July 1999.

[15] Lus Moura e Silva and Rajkumar Buyya, *Parallel programming models and paradigms*, 1999.

[16] S. Eddy, *HMMER: biosequence analysis using profile Hidden Markov Models*, http://hmmer.janelia.org/.

[17] ———, *HMMER User's Guide*, Howard Hughes Medical Institute and Dept. of Genetics.

[18] S. R. Eddy, *Profile Hidden Markov Models*, Bioinformatics **14** (1998), no. 9, 755–763.

[19] Sean Eddy, *The need for (vector) speed.*, Cryptogenomicon; The Eddy lab: genome sequence analysis (2009), http://selab.janelia.org/people/eddys/blog/?p=66#more-66.

[20] Leonard Eisenberg, *A phylogenetic tree*, http://www.evogeneao.com/.

[21] S. Henikoff and J G Henikoff, *Amino acid substitution matrices from protein blocks.*, Proceedings of the National Academy of Sciences of the United States of America **89** (1992), no. 22, 1091510919, PMC50453.

[22] IBM, *Cell broadband engine architecture computing platforms from ibm: Quick reference guide*, http://www-03.ibm.com/technology/cell/index.html.

[23] J. Bautista J. Held and S. Koehl, *From a few cores to many: A Tera-scale Computing Research overview*, Tech. report.

[24] Neil C. Jones and Pavel A. Pevzner, *An introduction to bioinformatics algorithms (computational molecular biology)*, The MIT Press, August 2004.

[25] W. Just, *Computational complexity of multiple sequence alignment with SP-score.*, Journal of computational biology : a journal of computational molecular cell biology **8** (2001), no. 6, 615–623.

[26] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, *Introduction to the Cell multiprocessor*, IBM J. Res. Dev. **49** (2005), no. 4/5, 589–604.

[27] European Bioinformatics Institute (European Molecular Biology Laboratory), Protein Information Resource, and Swiss Institute of Bioinformatics, *Uniprot press release - establishing a universal knowledgebase of proteins*, http://pir.georgetown.edu/pirwww/otherinfo/102502uniprot.pdf.

[28] E. Lindahl, *Altivec-accelerated HMM algorithms*, http://lindahl.sbc.su.se/.

[29] Jizhu Lu, Michael Perrone, Kursad Albayraktaroglu, and Manoj Franklin, *HMMer-Cell: High performance protein profile searching on the Cell/B.E. processor*, ISPASS '08: Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software (Washington, DC, USA), IEEE Computer Society, 2008, pp. 223–232.

[30] Edgar Luttmann, Daniel L. Ensign, Vishal Vaidyanathan, Mike Houston, Noam Rimon, Jeppe Øland, Guha Jayachandran, Mark Friedrichs, and Vijay S. Pande, *Accelerating molecular dynamic simulation on the Cell processor and Playstation 3*, Journal of Computational Chemistry **9999** (2008), no. 9999, NA+.

[31] G. E. Moore, *Cramming more components onto integrated circuits*, Proceedings of the IEEE **86** (1998), no. 1, 82–85.

[32] S. B. Needleman and C. D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins.*, J Mol Biol **48** (1970), no. 3, 443–453.

[33] Nvidia, *CUDA*, http://www.nvidia.com/cuda.

[34] University of Wisconsin Madison Materials Research Science and Engineering Center, *Slides on computer architecture*, http://mrsec.wisc.edu/Edetc/SlideShow/slides/computer/Moores_Law.html.

[35] T. Oliver, L. Yeow, and B. Schmidt, *Integrating FPGA acceleration into HMMER*, Parallel Computing **34** (2008), no. 11, 681–691.

[36] W. R. Pearson, *Rapid and sensitive sequence comparison with FASTP and FASTA.*, Methods in enzymology **183** (1990), 63–98.

[37] Vincent Pillet, Jess Labarta, Toni Cortes, Sergi Girona, and Departament D'arquitectura De Computadors, *PARAVER: A tool to visualize and analyze parallel code*, Tech. report, In WoTUG-18, 1995.

[38] Lawrence R. Rabiner, *A tutorial on hidden markov models and selected applications in speech recognition*, Proceedings of the IEEE, 1989, pp. 257–286.

[39] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences*, Journal of Molecular Biology **147** (1981), no. 1, 195–197.

[40] E. L. Sonnhammer, S. R. Eddy, and R. Durbin, *Pfam: a comprehensive database of protein domain families based on seed alignments.*, Proteins **28** (1997), no. 3, 405–420.

[41] J. D. Thompson, D. G. Higgins, and T. J. Gibson, *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice.*, Nucleic Acids Res **22** (1994), no. 22, 4673–4680.

[42] Andrew J. Viterbi, *Error bounds for convolutional codes and an asymptotically optimum decoding algorithm*, IEEE Transactions on Information Theory (1967), 260–269.

[43] John Paul Walters, Rohan Darole, and Vipin Chaudhary, *Improving MPI-HMMER's scalability with parallel I/O*, IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (Washington, DC, USA), IEEE Computer Society, 2009, pp. 1–11.

[44] John Paul Walters, Bashar Qudah, and Vipin Chaudhary, *Accelerating the HMMER sequence analysis suite using conventional processors*, AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications (Washington, DC, USA), IEEE Computer Society, 2006, pp. 289–294.

PART IV:
APPENDICES

"'The road is clear,' said Galt. 'We are going back to the world.'"
- Atlas Shrugged, Ayn Rand

List of Terms

A

Some of the terms and expressions in this thesis are very specific to the respective domains of bioinformatics and computer architecture. As such, the reader might be unfamiliar with their meaning. This appendix presents an alphabetical list of commonly used terms with their explanation, in order to aid in the understanding of this thesis.

A.1 List of Terms

Bioinformatics Discipline that attempts to increase the understanding of molecular biology through the development of novel algorithms and techniques, making use of the exponential growth in computing power.

Bit Score Log-odds measure for the fit between sequence and protein model.

Buffering (HMMERCELL program phase) The buffering stage creates jobs that are ready to be processed by the SPEs. Loads sequences into memory and creates administrative entry.

Cell Broadband Engine State-of-the-art heterogeneous multi-core processor, developed by IBM, Sony and Toshiba, consisting of nine cores: a general purpose oriented PPE, and eight SPEs aimed at streaming workloads. Implementation platform of this thesis.

DNA String of base pairs containing the entire genetic code of an organism.

E-Value Number of expected false positives for a certain bit score and database size.

Gene A sub-sequence of the DNA that codes for a single protein.

Genome The combined total of an organism's genetic data.

Hidden Markov Model Markov Model in which the model state is hidden. States output emission variables, based on which model state can be recovered.

HMM See Hidden Markov Model.

HMMER Computer program to align protein sequences to a protein family model (based on HMMs). Resulting alignment has E-value and bit score that indicates likelihood of match.

HMMERCELL Port of HMMER to the Cell architecture.

Homologue Similarity between DNA or protein sequences resulting from common ancestry.

Manager/Worker Pattern A design pattern used for the parallelization of software. One processor produces jobs that are consumed by other processors.

Markov Model A simple state-based probabilistic model. State transitions only depend on the previous x states.

Paraver Trace visualization environment. Used in this thesis to interpret generated traces.

Pattern (Computer Science term) A generalized solution for commonly encountered problems.

Pfam Database containing protein descriptions based on HMMs.

Plan7 Profile HMM Architecture The specific profile HMM structure as used by HMMER.

PPE Power Processing Element. Cell processor core intended for general purpose processing.

Profile Hidden Markov Model HMM as used in bioinformatics.

Protein Basic building block of life. Many uses, amongst others: forms major components of organism's body, controls signaling within a cell.

Sequence A string of symbols; in this context usually a list of symbols that code for a protein.

Sequence Alignment Arranging sequences to each other and scoring the resulting alignment. Often, the highest scoring alignment needs to be found.

Sequence Analysis Comparing sequences to each other to find similarities.

SPE Synergistic Processing Element. Cell processor core intended for streaming workloads.

TaskSim High-level phase-based simulator for computer architectures. Used in this thesis to simulate Cell-like derivatives.

Traceback (HMMERCELL program phase) Produces alignment of a sequence to an HMM. Traceback is performed only for sequences scoring above a certain threshold.

UniProt The Universal Protein Database, a database containing protein sequences.

Viterbi Algorithm Finds the most likely sequence of states through an HMM to generate a given sequence of output symbols.