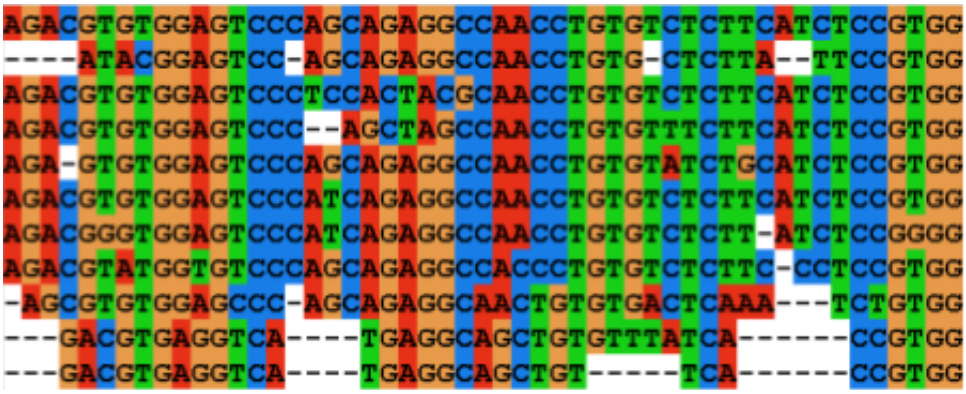
TÉCNICO LISBOA



SIMD Parallelization of Profile HMMs

Miguel Antunes Mendes Ferreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Supervisor:	Prof. Luis Manuel Silveira Russo
Supervisor:	Prof. Nuno Filipe Valentim Roma
Members of Committee:	Prof. José Carlos Alves Pereira Monteiro
	Prof.ª Sara Alexandra Cordeiro Madeira

October 2013

Acknowledgments

First, I would like to thank my two advisors, Professor Nuno Roma and Professor Luis Russo, for their support and patience throughout this thesis, as well as their valuable insights.

I want to thank my family for their unwavering support, without which I would not be here today.

Finally I want to thank the authors of HMMER, Professors Sean Eddy and Elena Rivas, whose software made this thesis possible and gave me many headaches.

Abstract

Sequence comparison is a crucial task in bioinformatics, to evaluate the similarity (homology) between biological sequence regions. Homology search involves either alignment of sequences, or probabilistic models such as Hidden Markov Models (HMMs), both of which use Dynamic Programming algorithms.

Given the enormous size of the sequences databases, it is essential to parallelize the searches. The parallelization strategies can be divided between intra-task (only one task is parallelized) and inter-task (multiple tasks are conducted in parallel). One successful strategy for Intra-task parallelism is the striped method of Farrar, which is employed by HMMER, a popular HMM tool, using SIMD vector units of commercial CPUs (e.g., x86's SSE)

In this thesis, an alternative Inter-task solution for HMMER was developed, based on the 2011 work of Rognes, also on SSE. This work, named COPS, solved some of Rognes' problems, and deployed a Cache-oblivious technique to process HMMs of arbitrary lengths. The results are largely positive: it is faster than HMMER's version in all tests, and reaches a maximum speedup of 2x against HMMER. The potential for an additional intra-task multi-threading using a wave-front model was also explored, with positive results.

With this work, new avenues for parallelization of HMMs are opened. COPS has shown to be strongly improve on the current HMMER's Farrar implementation, specially when using longer models. The same methods can be applied to other areas, such as speech recognition. COPS can also be extended to AVX2, the 256-bit successor of SSE.

Keywords

Alignment, parallelization, SSE, Hidden Markov Models, HMMER, Viterbi

Resumo

A comparação de sequências é uma tarefa crucial em bioinformática, para avaliar a semelhança (homologia) entre regiões de sequências biológicas. Pesquisa por Homólogos envolve ou alinhamento, ou modelos probabilísticos como os Modelos Ocultos de Markov (HMMs), e ambos usam algoritmos de Programação Dinâmica

Dado as enormes bases de dados, é essencial paralelizar as pesquisas. As estratégias de paralelização podem ser divididas entre Intra-tarefa (apenas uma tarefa paralelizada) e Inter-tarefa (várias tarefas realizadas em paralelo). Uma estratégia Intra-tarefa eficiente é o método intercalado do Farrar, usado pelo HMMER em SIMD de processadores comuns (por exemplo, SSE do x86).

Foi desenvolvida uma solução Inter-tarefa alternativa para o HMMER, baseada na abordagem do Rognes de 2011, também em SSE. Esta abordagem, denominada COPS, resolveu alguns problemas do Rognes, e implementou uma técnica *Cache-Oblivious* para processar HMMs de comprimentos arbitrários. Os resultados são fortemente positivos: mais rápido que o HMMER para todos os testes realizados, e com um *speedup* máximo de 2x contra o HMMER. Também foi explorado o potencial de uma paralelização adicional Intra-tarefa com *threads*, através de um modelo frente-de-onda, obtendo resultados positivos.

Assim, são abertos novos caminhos para paralelização de HMMs. Demonstrou-se ser uma alternativa mais eficiente face à implementação Farrar do HMMER, especialmente para modelos longos. Os métodos estudados aqui podem ser aplicados a outras áreas, tais como reconhecimento de voz. Também podem-se estender a AVX2, o sucessor do SSE, de 256-bit.

Palavras Chave

Alinhamento, paralelização, SSE, Modelos Ocultos de Markov, HMMER, Viterbi

Contents

List of Figures

List of Tables

1

Introduction

Contents

Nucleic Acids (nuclear DNA, mitochondrial m-DNA, RNA), and the proteins that they code, suffer sporadic mutations when they replicate, because their different strains recombine during sexual reproduction. The macroscopic result of such mutations is the species' evolution, which historically was the first sign of such microscopic forces at work. In molecular biology, among other things, it is studied these mutations, the reasons behind them, and the phylogenetic (evolutionary) trees that connect different nucleic acids, proteins and organisms. By analyzing the differences or similarities between two sequences, it is possible to infer about their probable homology (evolutionary relatedness). This can then be extended to other molecular biology tasks, such as genome sequencing through comparison with similar known genomes.

Nowadays Bioinformatics techniques and applications play an essential role on molecular biology and related fields. Sequence alignment algorithms in particular, and probabilistic models which mimic alignment algorithms, are the preferred methods used to search for similarity between biological sequences or parts of such sequences. Probabilistic models such as Hidden Markov Models (HMMs) are specially useful to represent a family of closely related sequences, which has been previously aligned or clustered together. It is then faster, and more reliable, to search against a HMM that models a sequence family, instead of searching and aligning against each individual sequence separately.

The amount of data involved rapidly becomes daunting: a DNA genome can have up to 150×10^9 basepairs (the human genome has 3×10^9 bps), and proteins' size ranges from just a few dozen amino-acids (called residues in a protein), to almost 30.000 in the case of tinins [8]. Widely used databases like Swissprot and TrEMBL have millions of sequences, so a search of a single sequence against such databases, with the fastest tools available, can take considerable time. In many cases, the space and time cost of running a complete optimal alignment is prohibitive. Therefore, optimized and non-optimal approaches have been the focus of much study to surpass these problems.

1.1 Motivation

The importance of alignment applications can be seen by the large databases of biological sequences now in use (SwissProt, UniProt, NCBI, EMDBL, GenBank, etc). Moreover, these databases double in size each year or so [5], due to faster, better and ever more numerous sequencing technologies.

This exponential growth can rapidly render any algorithm unusable, especially since the known algorithms that are guaranteed to find the optimal alignment have at best quadratic complexity. These are Dynamic Programming (DP) algorithms, which fill in a complete $N \times M$ sub-problem matrix in order to evaluate all the possible paths for the perfect alignment, and compute the respective score.

To circumvent such heavy algorithms, heuristic approaches have been proposed, that greatly reduce the problem complexity. In fact, the most used homology search tools in the past decades have been heuristic tools. Some particularly successful ones are Blast [2] and Fasta [41], which were the earlier ones developed, and have spawned a few improved variants. Others with similar functionality are MUMmer [11] and BLAT [29]. These tools use a variety of methods to search for a 'good alignment', that is not guaranteed to be the best.

For Markov Models, one of the most widely used tools is HMMER ([14]), which is now in its third version. HMMER also employs heuristic methods, in a succession of heuristic filtering algorithms, which prune most of the sequences to search before reaching the slowest, most accurate, HMM algorithms.

A very efficient alternative strategy to speedup homology search is to parallelize DP algorithms on modern processors. Database search of a query can be trivially parallelized by running each alignment on a separate process/thread, and can be easily mapped and spread out across a cluster of machines (usually referred to as 'Inter-task Parallelization').

Parallelization of a single alignment (Intra-task parallelization) is a more challenging goal, to which end many different approaches have been proposed. Specialized hardware solutions on FPGAs and others have obtained good results [39], [57], but are naturally expensive, and cannot be easily reconfigurable or integrated.

Software solutions can be divided in two levels: 'coarse-grained' functional parallelism on MIMD architectures, in which the DP matrix is divided in 'chunks' to be handled separately by each processor; and 'fine-grained' data parallelism, on vectorized processors implementing a SIMD architectural model.

In vector processing, multiple cells are computed in parallel by the same single operation. Note that DP algorithms are intrinsically vectorial: the computation of a cell follows the same exact function as every other cell, with the sole problem of solving data dependencies. As such, vector processing has yielded excellent results, as is one of the most hopeful and historically pursued for the past decades. Many implementations have been developed on numerous specific vector-processing architectures: Cell Broadband and PS3 [51], [47], [18], GPUs [36], [35]; as well as vector units available on general purpose scalar CPUs: Intel x86' MMX and SSE [46] [17], AMD's 3DNow!, SPARC's VIS [56]. Speedup factors in the order of 30x and higher, on a single parallelization approach, have been reported, such as in the interesting work of Rognes [45].

1.2 Objectives

Given the huge interest that biological alignment algorithms and applications have garnered over the past decades, varied multiple approaches have been proposed to tackle their efficiency problem.

After studying some promising alignment tools, the focus of this thesis fell on the widely used HMMER suite, which has already been vectorized using Farrar's striped method [17] of Intra-task parallelism. This thesis endeavored to develop a novel alternative approach to vectorize the DP algorithms used by HMMER, based on the work of Rognes [45] with Inter-task parallelism on SSE. As of now, this approach has not been applied before to HMM algorithms.

1.3 Dissertation outline

This document is structured in 6 chapters. The current chapter, Chapter 1, introduces the developed work.

Chapter 2 surveys the state-of-the-art of alignment algorithms and sequence search with Markov Models.

Chapter 3 reviews the related work on parallelization of alignment and HMM algorithms, which given their similarity, employ the same techniques.

Chapter 4 describes the proposed solution, its implementation, the problems found and how they were solved. Some novel improvements on Rognes strategy are presented in this chapter, including a new multi-threaded parallelization level.

Chapter 5 presents and discuss the evaluation results for the main contributions of this work.

The final Chapter 6 concludes the thesis with an analysis of the parallelization approaches studied and developed, their comparative advantages and drawbacks, and possible avenues for future study and extension.

2

Sequence Homology Search

Contents

This chapter will present two of the most widely used methods to search for homolog sequences: Alignment Algorithms described in the first section; and probabilistic models known as Markov Models, presented in the second section.

2.1 Alignment Algorithms

Sequence alignment is a very simple technique: taking two sequences, aligning its symbols side by side, *in order*, and inserting gaps to fill the places of inserted or deleted symbols (a deletion in one sequence is equivalent to an insertion in the other). An alignment of two sequences can be easily extracted from an edit distance matrix, and it shows in a very readable form the divergences between the two sequences, as well as the matches between them (known in molecular biology as 'conserved regions').

As such, the alignment of two protein or DNA strings is a useful and appealing way of highlighting their similarity. More than that, the aligning procedure serves the goal of evaluating and quantifying how interesting is the alignment found; or in other words, how related or distant are the sequences. For these reasons, it became the main technique used to study protein and DNA mutations.

The alignment can be global, in which the whole sequences are aligned, from start to finish; or it can be local, aligning only a certain, partial region of each sequence. The algorithms described in the next sections are 'optimal' algorithms - they are guaranteed to find the best possible alignment according to a specific distance (homology) metric. Later on, some non-optimal (i.e. heuristic) alignment algorithms will be briefly presented.

To measure the performance of alignment programs, the CUP measure is the most widely used. CUP stands for 'Cell Update per Second', though a more suitable name would be 'Cell Computation per Second'. The final CUP score for one single alignment is given by $\frac{M \times N}{t}$, where M and N are the sequences' lengths, and t is time in seconds. Since the CUP scores in modern processors and implementations are very high, MCPs (mega CUPs) and GCUPs (giga CUPs) are more common.

2.1.1 Edit Distance of Levenshtein

A good starting point is the popular 'Edit Distance', first formulated by Levenshtein in 1966 [31] for string matching. The edit distance of two sequences is defined as the minimum number of edit operations necessary to transform one sequence into the other. An edit operation may be a substitution, an insertion of a symbol, or a deletion of a symbol (the insertion of a symbol from sequence A in sequence B can always be replaced by the deletion of that symbol in sequence A, hence these last two operations are symmetrical). Insertions and deletions (indels) are usually referred to as 'insertions of gaps' in the context of biological alignments.

The levenshtein edit distance if usually computed with resort to an also well-known Dynamic Programming algorithm, first defined by Wagner and Fischer in 1974 [53]. It uses a typical DP matrix to map the three dependencies of the algorithm: match of the two symbols or substitution of one symbol for the other, deletion of a symbol from a sequence, or insertion into that sequence. In matches and substitutions, both symbols are consumed and the algorithm moves to the next cell in the diagonal. In insertions and deletions, only one symbol is consumed from one of sequences. Each cell computes a function given by the Maximum or Minimum of its dependencies' costs (diagonal, horizontal and vertical), and chooses the origin (dependency) that has the best cost and so yields the best result for that cell. This algorithm, and all similar ones, have quadratic complexity $O(n \times m)$, since it requires the computation of all $N \times M$ matrix cells.

The edit distance recursive relation can be formulated as follows:

$$D_{m,n} = Min \ \ (D_{m-1,n-1} + SubstScore, \ \ D_{m,n-1} + GapCost, \ \ D_{m-1,n} + GapCost)$$

where the value of SubstScore depends on whether it is a match or mismatch. An example is presented in Figure 2.1 and Figure 2.2, with a match score of 0 and mismatch of 1.

The optimal alignment of the two sequences can be easily extracted from the Edit Distance matrix, using a tracebacking mechanism. The procedure starts from the last cell, and traces each choice of the algorithm (i.e. the path chosen for each cell) until the starting point. In order to do this, each cell must keep the dependency (direction) used to compute its own value.

	_	A	L	I	G	N	I	N	G
_	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
L	2	1	0	1	2	3	4	5	6
G	3	2	1	1	1	2	3	4	5
O	4	3	2	2	2	2	3	4	5
R	5	4	3	3	3	3	3	4	5
I	6	5	4	3	4	4	3	4	5
T	7	6	5	4	4	5	4	4	5
H	8	7	6	5	5	5	5	5	5
M	9	8	7	6	6	6	6	6	6

Figure 2.1: Edit Distance matrix for sequences 'algorithm' and 'aligning', using unitary costs. The first dependencies are marked. The non-picked directions have dotted blue lines, picked ones are red and straight, and the optimal path is in green bold.

1st Part	*2nd Part*
ALIGN_	I_NG
AL_GOR	ITHM
ALIG_N	IN_G
AL_GOR	ITHM
ALIGN	ING_
ALGOR	ITHM

Figure 2.2: Resulting 9 optimal alignments - each part from the left concatenated with a right part. All the alignments have the last I's aligned, and there are 3 possible variations before and after the I's.

2.1.2 NW Algorithm for Global Alignment

In 1970 [38] Needleman and Wunsch adapted the Levenshtein metric to use in the field of biological alignment. The NW algorithm is mostly similar to the later Wagner and Fischer algorithm, with two differences introduced to improve its biological meaning:

- Gaps are considered a single mutational event. A single ”gap opening” penalty is weighted each time the alignment may evolve from a continuous region to a gap;

- Each substitution may have a different real-world occurrence probability, and hence it should have a different weight. These probabilities were later measured and tabulated (see Section 2.1.6).

The resulting score of a global alignment, like that of a levenshtein edit distance, is given by the score of the last cell (which is the score of the complete path between the start of both sequences and their end). The function used for each cell may be the maximum or the minimum of previous cells.

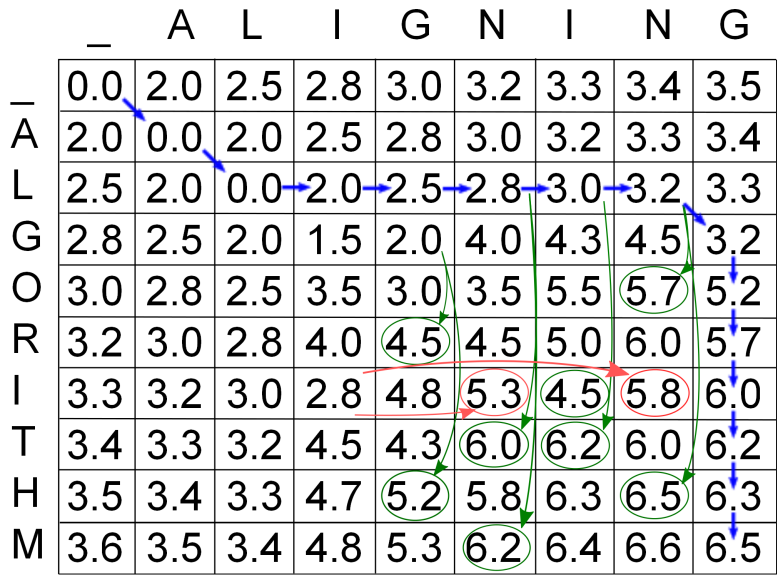
Needleman and Wunsch's original metric for single gaps, and Sellers' similar method [50], were later extended by Waterman in 1976 [55] to support an arbitrary number of deletions/insertions and more complex gap cost models. Since then, all global alignment metrics and algorithms based on edit distances used for biological alignment are referred to as Needleman-Wunsch's.

Any type of function can be used for gap scoring, tough the function used has a great impact on the algorithm's complexity. If a linear cost function is used, each cell's value can be computed by inspecting only its direct ascendants (horizontal, vertical and diagonal). However, when using a non-linear cost function, such as an affine function with an initial gap opening constant penalty, this is not the case. Given the complex behavior of these functions, each value cannot be completely inferred from the previous discrete value. The contribution of a gap sequence may 'skip' some cells (usually the first) because the diagonal value presents a better option; and still be chosen for later cells, due to the function's amortized behavior (see Figure 2.3). This means that, in order to compute each cell, the contributions of all the previous cells have to be considered and computed, and hence the algorithm's complexity rises from $O(n \times m)$ to $O(n \times m \times max(n, m))$.

2.1.3 SW Algorithm for Local Alignment

Despite the good results of the NW algorithm, most of the time molecular biologists do not want a global alignment. A global alignment forces all the sequences to be aligned - one way or the other. This means that, if there are regions with high similarity and regions with high divergence, the overall result would be average, or even bad. The interesting regions cannot be identified and extracted from the complete alignment. Moreover, global alignments always force the inclusion of gaps on the alignment fringes.

To tackle these limitations of global alignment, Smith and Waterman in 1981 [49] extended the previous algorithms with the possibility of ”alignment restart”, thus creating the first algorithm to



	_	A	L	I	G	N	I	N	G
_	0.0	2.0	2.5	2.8	3.0	3.2	3.3	3.4	3.5
A	2.0	0.0	2.0	2.5	2.8	3.0	3.2	3.3	3.4
L	2.5	2.0	0.0	2.0	2.5	2.8	3.0	3.2	3.3
G	2.8	2.5	2.0	1.5	2.0	4.0	4.3	4.5	3.2
O	3.0	2.8	2.5	3.5	3.0	3.5	5.5	5.7	5.2
R	3.2	3.0	2.8	4.0	4.5	4.5	5.0	6.0	5.7
I	3.3	3.2	3.0	2.8	4.8	5.3	4.5	5.8	6.0
T	3.4	3.3	3.2	4.5	4.3	6.0	6.2	6.0	6.2
H	3.5	3.4	3.3	4.7	5.2	5.8	6.3	6.5	6.3
M	3.6	3.5	3.4	4.8	5.3	6.2	6.4	6.6	6.5

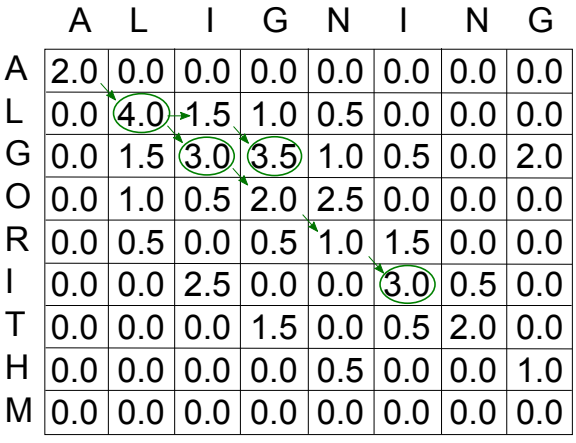
Figure 2.3: Needleman-Wunsch matrix with a logarithmic gap model ($GapCost = 2 + 0.5 \times log_2(\#gaps)$, Match 0, Mismatch 1.5). Red and green arrows show the chosen non-continuous gaps (i.e. that are only chosen in later cells). The optimal alignment is traced in blue bold lines.

compute local alignments (in Figure 2.4). The SW algorithm is similar to NW's and Sellers', with the differences:

- Each cell value is maximized with 0. This mechanism causes a 'restarting' in the alignment path. Naturally, this requires the penalties to be negative and the match scores positive.

- The maximum score may be obtained from any cell of of the matrix, instead of only the last one. Each cell holds the score for the optimal local alignment that ends in that cell.

- To compute the gap cost, only those previous cells before the first zero (when followed backwards from the current cell) need to be inspected. All more distant previous cells cannot lead to a positive gap score.



	A	L	I	G	N	I	N	G
A	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
L	0.0	4.0	1.5	1.0	0.5	0.0	0.0	0.0
G	0.0	1.5	3.0	3.5	1.0	0.5	0.0	2.0
O	0.0	1.0	0.5	2.0	2.5	0.0	0.0	0.0
R	0.0	0.5	0.0	0.5	1.0	1.5	0.0	0.0
I	0.0	0.0	2.5	0.0	0.0	3.0	0.5	0.0
T	0.0	0.0	0.0	1.5	0.0	0.5	2.0	0.0
H	0.0	0.0	0.0	0.0	0.5	0.0	0.0	1.0
M	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 2.4: Smith-waterman algorithm. $GapCost = -2 - 0.5 \times (\#gaps)$, Match +2, Mismatch -1. The best alignments are shown and traced, with the maximum values circled.

2.1.4 Gotoh Algorithm

These DP algorithms are overly simple, and only a few optimizations have been found to reduce the required time and space lower bound. The quadratic time barrier cannot be lowered since they require that every cell in the $N \times M$ be computed. For arbitrary cost models, the complexity is even worse ($O(n^3)$). However, for some restricted cost models, a few improvements are possible.

Gotoh in 1982 [20] proposed a very important optimization for general DP algorithms that use affine scoring functions. Such functions are actually one of the most suitable to biological sequences. Each gap opening is highly penalized, and all successive gaps have a reduced, linear cost, which models quite well the real-world occurrence of single mutational events, frequently responsible for inserting/deleting many elements at once.

Gotoh's algorithm is based upon the idea that the gap costs can also be modeled and computed in parallel, using Dynamic Programming, in separate matrices for vertical and horizontal gaps. Each cell in these two matrices hold the cost for a gap that ends in that cell.

These two matrices are computed based on their own dependencies and on the respective value in the main distance matrix (choosing the current main score 'restarts' the gap sequence, opening a new gap from the current cell). For the main matrix, the match/mismatch value is taken from its previous diagonal cell, and the gap contributions come from the same cell in the two twin matrices (i.e. the cost of a horizontal or vertical gap ending in that cell).

The gap matrices are usually named Q and P (after Gotoh), or E (horizontal) and F (vertical), while the main matrix is called D (distance matrix). The recursion relation can be formulated as follows:

$$D_{m,n} = OP(D_{m-1,n-1} + d(a_m, b_n), E_{m,n}, F_{m,n})$$

Where OP may be Maximum or Minimum, and $d(a_m, b_n)$ is the match/mismatch score for the pair of elements (a_m, b_n). E and F are both given by the same identical relation:

$$Em, n = OP(D_{m-1,n} + W_1, E_{m-1,n} + W_{ext}) \qquad Fm, n = OP(D_{m,n-1} + W_1, F_{m,n-1} + W_{ext})$$

W_1 is the gap cost function value for one gap, and W_{ext} is the gap extension penalty, which is the slope of the affine cost function W. $W_i = W_{open} + i \times W_{ext}$. An example is presented in the associated figures.

Besides the reduced number of previous values that need be seen to compute each cell, Gotoh's improvement has another precious effect: the whole matrices are themselves not needed. They have to be computed but do not need to be stored, since only the last line column of diagonal of each one is used. As such, the algorithm can be very efficiently implemented on a computer, using only three of two arrays to store the intermediate cells. Gotoh's algorithm, therefore, reduces the time complexity to $O(N \times M)$ and the space complexity to O(min(n,m)).

One important note though, about reduced space requirements: gotoh's algorithm computes only the optimal score (or a set of scores, if they are saved separately). The alignment pattern itself cannot be extracted when this method is used, since no tracebacking data is stored.

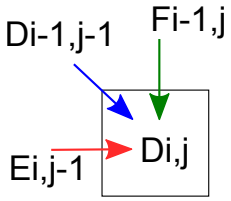


Figure 2.5: Dependencies for each cell of Gotoh's algorithm. Diagonal, horizontal and vertical.

	_	A	L	I	G	N	I	N	G
_	0.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0
A	2.5	0.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5
L	3.0	2.5	0.0	2.5	3.0	3.5	4.0	4.5	5.0
G	3.5	3.0	2.5	1.4	2.5	4.4	4.9	5.4	4.5
O	4.0	3.5	3.0	3.9	2.8	3.9	5.8	6.3	6.8
R	4.5	4.0	3.5	4.4	5.3	4.2	5.3	7.2	7.5
I	5.0	4.5	4.0	3.5	5.8	6.5	4.2	6.7	7.2
T	5.5	5.0	4.5	5.4	4.9	7.2	6.7	5.6	8.1
H	6.0	5.5	5.0	5.9	6.8	6.3	7.2	8.1	7.0
M	6.5	6.0	5.5	6.4	7.3	8.2	7.7	8.6	9.5

Figure 2.6: Gotoh's NW algorithm D main matrix, with $GapCost = 2 + 0.5 \times (\#gaps)$, Match 0, Mismatch 1.4. Rightwards values in red come from the E matrix, and downwards green values from the F matrix.

	_	A	L	I	G	N	I	N	G
_	-	-	-	-	-	-	-	-	-
A	oo	5.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5
L	oo	5.5	5.0	2.5	3.0	3.5	4.0	4.5	5.0
G	oo	6.0	5.5	5.0	3.9	4.4	4.9	5.4	5.9
O	oo	6.5	6.0	5.5	6.0	5.3	5.8	6.3	6.8
R	oo	7.0	6.5	6.0	6.5	7.0	6.7	7.2	7.7
I	oo	7.5	7.0	6.5	6.0	6.5	7.0	6.7	7.2
T	oo	8.0	7.5	7.0	7.5	7.4	7.9	8.4	8.1
H	oo	8.5	8.0	7.5	8.0	8.5	8.8	9.3	9.8
M	oo	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0

Figure 2.7: Gotoh's left gaps (E) matrix. The values chosen for the main matrix are marked.

	_	A	L	I	G	N	I	N	G
_	-	oo	oo	oo	oo	oo	oo	oo	oo
A	-	5.0	5.5	6.0	6.5	7.0	7.5	8.0	8.5
L	-	2.5	5.0	5.5	6.0	6.5	7.0	7.5	8.0
G	-	3.0	2.5	5.0	5.5	6.0	6.5	7.0	7.5
O	-	3.5	3.0	3.9	5.0	6.5	7.0	7.5	7.0
R	-	4.0	3.5	4.4	5.3	6.4	7.5	8.0	7.5
I	-	4.5	4.0	4.9	5.8	6.7	7.8	8.5	8.0
T	-	5.0	4.5	5.4	6.3	7.2	6.7	9.0	8.5
H	-	5.5	5.0	5.9	6.8	7.7	7.2	8.1	9.0
M	-	6.0	5.5	6.4	7.3	8.2	7.7	8.6	9.5

Figure 2.8: Gotoh's vertical gaps (F) matrix. The values chosen for the main matrix are marked.

2.1.5 SWAT optimization of the Smith-Waterman Algorithm

For most sequences and typical scores, the D value in the Smith-Waterman algorithm remains at 0, or close by, in the majority of matrix cells (see Figure 2.4). The E (horizontal) and F (vertical) dependencies can only affect D when the D value is higher (i.e. worse) than the weight of a single gap ($W_1 = W_{open} + 1 \times W_{ext}$). Such a penalty is usually chosen to be considerable costly, sometimes double the cost of a mismatch. Owing to that, only in relatively few cells are the E and

F vectors able to affect the D cell value.

The SWAT optimization ([21],[40]) exploits this tendency. It consists of a relaxation in the required cells that need to be seen to evaluate the gap dependencies: all F and E cells lower than W_1 cannot contribute to the alignment, and neither can all cells before those, because they would offer a worse gap starting point. Therefore the previously presented zero-barrier may be extended to a W_1-barrier.

Using the original zero-barrier and this optimization, the cubic complexity of the SW algorithm can be lowered in practice to $O(a \times n \times m)$, where 'a' is some empirical constant. The Gotoh variant can also be greatly improved with these barriers, by ignoring the E and F values below the threshold.

2.1.6 Substitution Scoring matrices

Alignment algorithms operate on the special context of biological systems. Each sequence base (or residue) has a specific and distinct nature. Some are more frequent than others, some degenerate and mutate more easily, some are found more frequently in some regions. In particular, each symbol has a different probability of mutation into each other specific symbol. In order to accurately study and predict these dynamic biological events, the scoring system used to evaluate each sequence pair must take all this into account, and it must be constructed and trained with real biological data (i.e. mutations empirically observed in proteins and nucleic acids). These values were measured by Dayhoff in 1978 [9] to create PAM matrices, and by Henikoff in 1992 [23] in the form of BLOSUM matrices, using a logarithmic scale. The two matrix types use opposite scoring schemes: PAMs were constructed from the observed mutations along a phylogenetic tree, whereas BLOSUMs measure the conserved regions. A lower PAM (fewer mutations) corresponds to a higher BLOSUM (more conservation), and vice versa.

An extension of the concept of substitution scores matrix is the Position Specific Scoring Matrices (PSSMs), also known as (Position Specific) Weight matrices, Frequency matrices or Profile matrices. These matrices count the number of occurrences of each symbol (line), for each sequence position (row). Employing PSSMs instead of simple substitution matrices has been proven to better model the sequences' homology. [22] [24] [48]

2.2 Homology Search with Markov Models

In this section, a different approach to sequence homology search will be described, an approach based on Hidden Markov Models. Despite the differen formulation of the problem, the computation involved is very similar, and thus the algorithms and parallelization methods previusly analyzed, for the most part, can equally be used with Markov Models.

2.2.1 Markov Models

In this section, we will first introduce very briefly what are Markov Models. A Markov Model is a probabilistic model of a system which enjoys the Markov property: the future states of the system depend entirely on the present state, and not on any previous states. A Hidden Markov Model (abbreviated HMM) is a Markov Model with unobserved states, so called 'hidden states'. In a HMM, the states may emit more than one possible token, according to a probability distribution. A HMM is therefore characterized by a number of unobserved states Q; a number of possible emission tokens T; a probability distribution of transitioning between the states Ft(q1,q2); and a probability distribution of emitting each token in each state Fe(q,w). The probability of some specific sequence of tokens being generated by a specific path of states π is then given by:

$$P(x, \pi) = t_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) \times t_{\pi_i \pi_{i+1}}$$

When using HMM, there are two tasks that are particularly important:

- Decoding: Which path of states is the more likely to have generated a given token sequence, and what is the corresponding probability? This task is computed by the Viterbi Algorithm [52], which progressively computes the most likely state to generate each new output symbol (i.e. token) , from the begin pseudo-state until reaching the end pseudo-state. For each sucessive sequence symbol x_i, the Viterbi algorithm computes the recursive relations:

  $$V_l(i) = e_l(x_i) \times max_k(V_k(i-1) \times t_{kl})$$

  $$Ptr_i(l) = argmax_k(V_k(i-1) \times t_{kl})$$

  with $V_l(i)$ being the probability of being in state l, and emitting symbol x_i, and $Ptr_i(l)$ being the pointer to the chosen previous state. To recover the most likely path, one must only traverse the computed pointers, $Ptr_i(l)$, in the reverse direction, from the end-state to begin-state.

- Generation: What is the likelihood of a given sequence being generated by the model?

  To compute the likelihood probability, we need to sum the probabilities of all the possible generation paths. Since the number of possible paths is exponential, a more efficient method must be used. A solution is given by the Forward algorithm, which is similar to the Viterbi algorithm: a progressive sum of the probabilities of all previous state paths for each new token.

  For each sucessive sequence symbol x_i, the Forward procedure computes the following relation:

  $$F_l(i) = e_l(x_i) \sum_k F_k(i-1) \times t_{kl}$$

The final probability is calculated by summing the final probabilities of all the states:

$$F(x) = \sum_k F_k(L) \times t_{k0}$$

These two procedures can encounter value representation problems when implemented on a computer. Since they involve sucessive products of small probabilities, especially the Viterbi algorithm, the computed probalities quickly exceed the floating-point representation range of modern computers and go into underflow. This can be avoided by working in log-space, with logarithmic probabilities in every calculation, and the products turned into sums.

The Logarithmic transformation poses a problem in the Forward algorithm: the log of a sum of two arguments is not easily computed from the logs of the arguments. The exact value is given by $\tilde{r} = log(exp(\tilde{a_1}) + exp(\tilde{a_2}))$, which can be written as $\tilde{r} = \tilde{a_1} + log(1 + exp(\tilde{a_2} - \tilde{a_1}))$. However, the function $log(1 + exp(\tilde{a_2} - \tilde{a_1}))$ can be approximated by interpolation from a table with a reasonably small size. [13].

HMM have been used quite successfully for the past decades for various applications of statistical modeling and machine learning [43].

2.2.2 Alignment Profiles

Until now, we have been looking at the homology search problem through the process of single alignment of one sequence against another. In real applications however, namely for searching homologs (i.e. similar sequences) in a databse, we are looking for sequences that belong to a certain family or have a high similarity to a given group of sequences. Therefore, it is oftentimes more useful to search and compare the database against a group of sequences instead of a single query.

Table 2.1: Example of a Consensus Profile derived from a multiple alignment of a family of similar sequences.

Alignment		A	T	C	C	A	G	C	T
		G	G	G	C	A	A	C	T
		A	T	G	G	A	T	C	T
		A	A	G	C	A	A	C	C
		A	T	G	C	C	A	T	T
		A	T	G	G	C	A	C	T
Profile	**A**	5	1	0	0	5	5	0	0
	C	0	0	1	4	2	0	6	1
	G	1	1	6	3	0	1	0	0
	T	1	5	0	0	0	1	1	6
Consensus		A	T	G	C	A	A	C	T

Profiles provide a more flexible way to identify homologs of a certain family, by highlighting the family's common features, and downplaying the divergences between the family's sequences. There are many ways to generate profiles, but most involve an initial multiple alignment of the sequences, followed by a probabilistic breakdown of the elements (residues or base-pairs) present in each position.

Profiles can thus effectively model a whole sequence family. This allows for a more efficient homology search: instead of comparing a query to each sequence in the family, the query can be compared to the Profile alone, greatly reducing the involved computational cost. Besides, a Profile gives a more correct representation of the defining characteristics of a family, by weighing the elements in proportion to their actual frequency (and thus importance) in the underlying family.

2.2.3 Profile Markov Models

A promising and widely used type of Profile constructs are Hidden Markov Models [42].

Hidden Markov Models can be used to statistically model the distribution of sequence elements in a Profile, capturing the probability of each element in each position as the emission probability of tokens in each state. In such a model, each model state represents one position of the family consensus. These Profile HMMs are then used to search a sequence database, by computing the probability of each database sequence being generated by the query model.

HMMs may also be used to find distant homologs by iteratively building and refining a model that describes them (such as in the SAM tool, [28]). In such an application, we start with an empty model. Iteratively, each new sequence is aligned against the model, and that alignment is used to add the sequence to the multiple alignment of sequences that underlie the model. Finally the model is re-parametrized with the new multiple alignment.

Krogh and Haussler in 1994 [30] developed a straightforward generalized profile Hidden Markov Model for homology searches, that emulate the results of a optimal alignment algorithm. The model is composed by three different types of states, respectively for matches/mismatches, insertions and deletions.

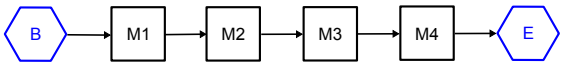


Figure 2.9: Example of a HMM composed solely of match states, allowing for ungapped global alignment.

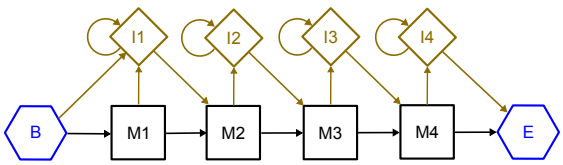


Figure 2.10: Example of a HMM that allows arbitrary insertions.

The construction of the model is intuitive. First, a simple ungapped global alignment is modeled by a succession of match states, with a single string of transitions, as seen in Figure 2.9. These match states emit the alignment symbols, so they have a corresponding emission probability distribution, which is derived from the relative frequencies of symbols in the family's sequences.

Since the symbols' frequencies vary in each position of the family's Profile, the emission distribution of the model, $e_{M_i}(x)$, will also be position-specific (similar to a PSSM, see Section 2.1.6).

Our objective is to determine if a given sequence belongs to the model's family, so we have to discriminate the positive probability (sequence X generated by that model) against the negative probability (sequence X generated by some other random model). The negative probability is the background probability distribution, q_x., i.e., the probability of belonging to the standard random model. This is evaluated for the emission probabilities, in the form of log-odds ratios:

$S(x) = log\frac{P(x|model)}{P(x|random)} = \sum_{i=-1}^{L} log\frac{e_i x_i}{q_{x_i}}$

After having a model of match states for ungapped global alignment, we proceed to deal with gaps. It is possible to handle insertions (i.e., sections of the sequence x that do not match the model) with an extra layer of states - insert states. An inserted region can occur at any point in the model, so we need to include an Insert state between each pair of match states. These new insert states must have a loop to allow for arbitrarily long inserted regions. A sketch of the resulting model with match and insert states is shown in Figure 2.10.

These insert states also have an emission probability distribution $e_{I_i}(x)$, that represents the inserted symbols. This distribution however is arbitrary, it does not depend on the model, since these inserted elements are unknown to the model. To simplify the calculations, the insert emissions $e_{I_i}(x)$ are usually set to the background distribution q_x, resulting in a null odds-ratio contribution. The I $\rightarrow$ I model the gap-extend costs for non-matched symbols in the query sequence.

Lastly, deletions (portions of the profile not matched by the sequence) are added to the model, as forward jumps from each match state to every other suceeding match state. Each of the N match states will have one average $\frac{N}{2}$ forward transitions. The basic idea is shown in Figure 2.11. However, a topology with these forward jumps faces a cumbersome problem: the number of added transitions, $\frac{N^2}{2}$), is quite large, and can significantly slow down any algorithm.

To circumvent the $\frac{N^2}{2}$) transitions, it is possible to develop an alternative topology that makes use of a third sucessive layer of silent 'jump states'. Silent states are those that do not emit any output symbol, but they may serve an important purpose in simplyfing a model. A forward jump is then achieved by sucessive transitions along the 'jump states' from one match state to the other, instead of a single transition between the two match states. A topology with jump states has the major advantage of using only a linear number of transitions ($N \times 2$ new transitions) and new states (N new states). An example of this topology is shown in Figure 2.12.

We can therefore use the 'jump states' as Delete states, where in each forward jump represents a region deleted from the profile model in the alignment. The D $\rightarrow$ D transitions thus correspond to gap-extend costs. Finally, the new layer of Delete states is added our previous model, which only had two layers, namely matches and inserts. Additional transitions from insert states to delete states, and vice-versa, can be included for the sake of correctness, although these transitions are usually very improbable and have a negligible effect. The final model for

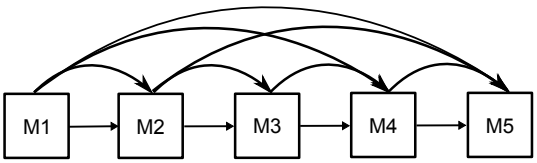


Figure 2.11: Example of a HMM with a continuous sequence of states, and arbitrary jumps ahead.

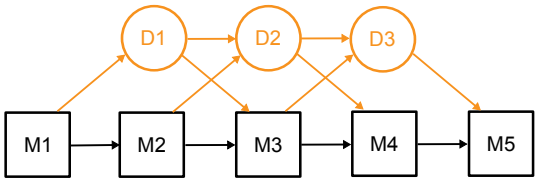


Figure 2.12: Example of the previous HMM with the forward jump transitions converted to a layer of silent 'jump states'.

gapped global alignment, as proposed by Krogh and Haussler [30], can be seen in Figure 2.13. Krogh-Haussler's model is quite simple and intuitive, and has become the blueprint for later Profile HMMs [28].

After having decided the structure of the HMM, it is necessary to parametrize it with the given family of sequences, namely to calculate from the data all the probability distributions required for the HMM. The parameterization of HMMs is a complex subject that falls outside the scope of this work (for this subject, see [13]). The Viterbi algorithm is also used for this purpose.

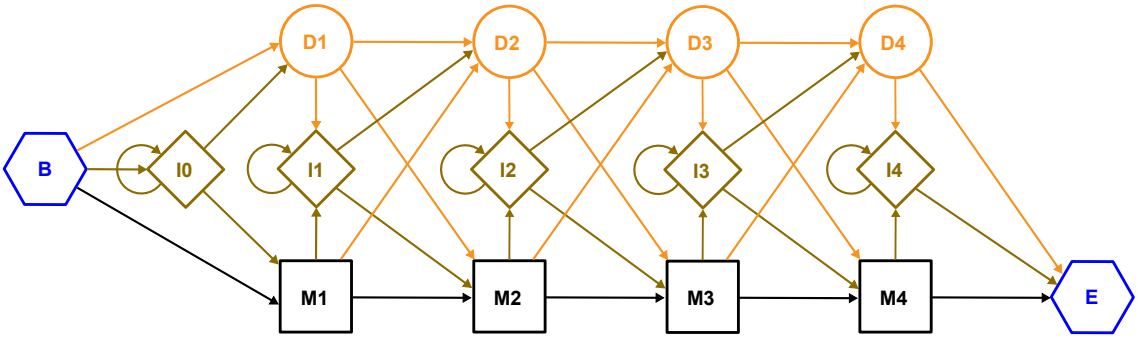


Figure 2.13: Krogh and Haussler' Hidden Markov Model for optimal gapped global alignment.

2.2.4 Algorithms for global alignment Profile HMMs

In a Profile HMM, the Viterbi algorithm gives the most likely path of states to generate the sequence, and the corresponding probability. It is therefore suitable for computing the alignment of the query sequence against the already multiple-aligned family's sequences (represented by the HMM).

The Forward algorithm computes the overall likelihood of the sequence being generated by that model through any path, as opposed to a random model, by summing the probabilities of all possible paths. It is thus more suitable as a general similarity measure, indicating the likelihood of the query sequence belonging to the family.

For a global alignment Krogh-Haussler model, working in log-space, the Viterbi algorithm is

expressed by the following DP recurrence relations:

$$V_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + Max\begin{cases} V_{j-1}^M(i-1) + log\ t_{M_{j-1}M_j} \\ V_{j-1}^I(i-1) + log\ t_{I_{j-1}M_j} \\ V_{j-1}^D(i-1) + log\ t_{D_{j-1}M_j} \end{cases}$$

$$V_j^I(i) = log\frac{e_{Ij}(x_i)}{q_{xi}} + Max\begin{cases} V_j^M(i-1) + log\ t_{M_jI_j} \\ V_j^I(i-1) + log\ t_{I_jI_j} \\ V_j^D(i-1) + log\ t_{D_jI_j} \end{cases}$$

$$V_j^D(i) = Max\begin{cases} V_{j-1}^M(i) + log\ t_{M_{j-1}D_j} \\ V_{j-1}^I(i) + log\ t_{I_{j-1}D_j} \\ V_{j-1}^D(i) + log\ t_{D_{j-1}D_j} \end{cases}$$

In regards to the Forward algorithms, for a global alignment model, working in log-space, we need to deal with following equations:

$$\begin{aligned} F_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + log\ [\ & t_{M_{j-1}M_j} \times exp\ (F_{j-1}^M(i-1)) \\ + & \ t_{I_{j-1}M_j} \times exp\ (F_{j-1}^I(i-1)) \\ + & \ t_{D_{j-1}M_j} \times exp\ (F_{j-1}^D(i-1))\ ] \end{aligned}$$

$$\begin{aligned} F_j^I(i) = log\frac{e_{Ij}(x_i)}{q_{xi}} + log\ [\ & t_{M_jI_j} \times exp\ (F_j^M(i-1)) \\ + & \ t_{I_jI_j} \times exp\ (F_j^I(i-1)) \\ + & \ t_{D_jI_j} \times exp\ (F_j^D(i-1))\ ] \end{aligned}$$

$$\begin{aligned} F_j^D(i) = log\ [\ & t_{M_jD_j} \times exp\ (F_{j-1}^M(i)) \\ + & \ t_{I_jD_j} \times exp\ (F_{j-1}^I(i)) \\ + & \ t_{D_jD_j} \times exp\ (F_{j_1}^D(i))\ ] \end{aligned}$$

These equations are expensive to compute, given the logarithms and exponentials involved. An approximation can be devised through the use of interpolation, with lookup on a pre-computed $TabLog$ table (as previously explained in Section 2.2.1), which yields the following pseudo-code:

$$\begin{aligned} F_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + TabLog\ [\ & log\ t_{M_{j-1}M_j} + F_{j-1}^M(i-1), \\ & TabLog\ [\ log\ t_{I_{j-1}M_j} + F_{j-1}^I(i-1)), \\ & \quad log\ t_{D_{j-1}M_j} + F_{j-1}^D(i-1)\ ]\ ] \end{aligned}$$

$$\begin{aligned} F_j^I(i) = log\frac{e_{Ij}(x_i)}{q_{xi}} + TabLog\ [\ & log\ t_{M_jM_j} + F_j^M(i-1), \\ & TabLog\ [\ log\ t_{I_jM_j} + F_j^I(i-1)), \\ & \quad log\ t_{D_jM_j} + F_j^D(i-1)\ ]\ ] \end{aligned}$$

$$F_j^D(i) = TabLog\ [\ log\ t_{M_j D_j} + F_{j-1}^M(i),$$
$$TabLog\ [\ log\ t_{I_j D_j} + F_{j_1}^I(i),\ log\ t_{D_j D_j} + F_{j_1}^D(i))\ ]\ ]$$

The transition scores used are all in log-odds (i.e. $log\ t_{X_j Y_i}$).

2.2.5 Simplification of the general Global Alignment HMM

Some of terms may be removed from the Model. The D $\rightarrow$ I and I $\rightarrow$ D transitions have a negligible impact on the alignment, and therefore can also be removed. If the emission probabilities $e_{Ij}(x_i)$ are set to the background distribution, then the term $log\frac{e_{Ij}(x_i)}{q_{xi}}$ becomes null.

The simplified Viterbi equations in log-space are thus:

$$V_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + Max\begin{cases} V_{j-1}^M(i-1) + log\ t_{M_{j-1}M_j} \\ V_{j-1}^I(i-1) + log\ t_{I_{j-1}M_j} \\ V_{j-1}^D(i-1) + log\ t_{D_{j-1}M_j} \end{cases}$$

$$V_j^I(i) = Max\begin{cases} V_j^M(i-1) + log\ t_{M_j I_j} \\ V_j^I(i-1) + log\ t_{I_j I_j} \end{cases}$$

$$V_j^D(i) = Max\begin{cases} V_{j-1}^M(i) + log\ t_{M_{j-1}D_j} \\ V_{j-1}^D(i) + log\ t_{D_{j-1}D_j} \end{cases}$$

And the Forward equations in log-space, with the Interpolation approximation, become:

$$F_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + TabLog\ [\ log\ t_{M_{j-1}M_j} + F_{j-1}^M(i-1),$$
$$TabLog\ [\ log\ t_{I_{j-1}M_j} + F_{j-1}^I(i-1)),$$
$$log\ t_{D_{j-1}M_j} + F_{j-1}^D(i-1)\ ]\ ]$$

$$F_j^I(i) = TabLog\ [\ log\ t_{M_j I_j} + F_j^M(i-1),\ \ log\ t_{I_j I_j} + F_j^I(i-1)]$$

$$F_j^D(i) = TabLog\ [\ log\ t_{M_j D_j} + F_{j-1}^M(i),\ \ log\ t_{D_j D_j} + F_{j_1}^D(i)]$$

2.2.6 Extension of Profile HMMs to Local Alignment

A Profile HMM for global alignment may be converted to support local alignment as well, with the inclusion of some additional states. In essence, a local alignment constitutes a subregion of the query sequence that is aligned against a subregion of the model, with a significant score, flanked by two unmatched regions that include the rest of the query sequence and the rest of the model.

To capture a local alignment with a Profile HMM, one needs only to add these two 'flanking regions', for instance as sell-looping states with transitions from and to each match state ([13]). These 'flanking states' also emit tokens with a probability distribution, which we can set to the background random distribution to remove it from the computations.

Therefore, the following specia-states are included:

- two flanking states connecting all normal states are added, the 'B' state before and the 'E' state after. These states have corresponding probability distributions for the array of possible transitions to or from normal states.
- two self-looping flanking states, 'N' before and 'C' after. The self-looping states are characterized simply by a loop probability and a 'jump' probability out of the state.

An example of a Profile HMM to evaluate a single local alignment is shown in Figure 2.14.

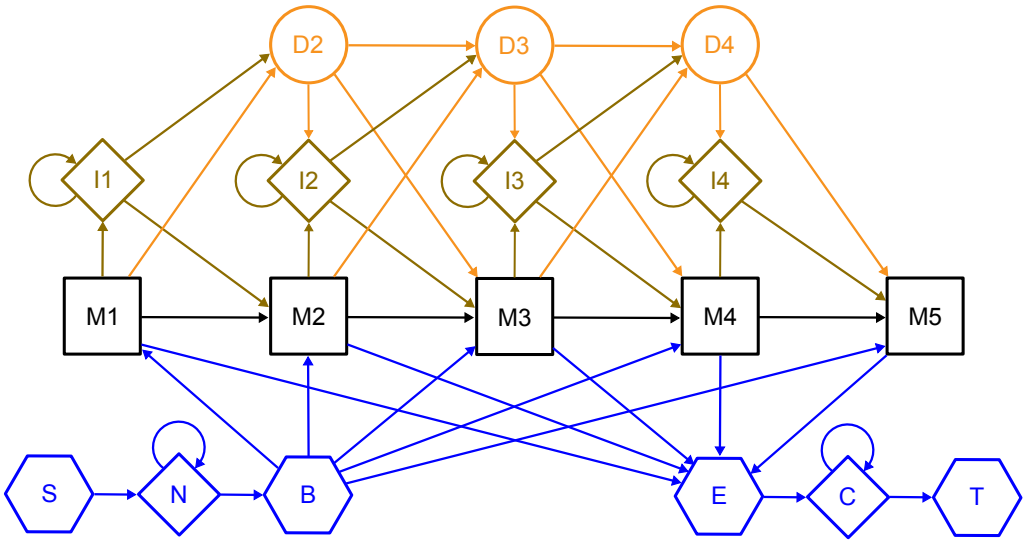


Figure 2.14: HMM for unihit local alignment (i.e. a single aligned region).

2.2.7 Extension of Profile HMMs to Multihit Alignments

Up until now, it has only been discussed the case of *unihit* evaluation with HMMs: only a single unbroken region, either the whole sequence and model (when in global alignment) or a subregion of the sequence and model (in the case of local alignment). It is also possible to use HMMs to match multiple regions, of both the sequence and the model. This is known as *multihit* alignment mode, and it more closely resembles the classical alignment algorithms like Smith-Waterman.

The previous HMMs can be extended to allow multihit alignments, by introducing another special state ('J'), connecting the B and E flanking states. This J state requires only two parameters as well, a loop probability and a jump probability.

A multihit modeling capacity can be added to both global and local alignment HMMs. The Figure 2.15 represents a HMM for multihit global alignment, while the more complex example for local alignment can be seen in Figure 2.16.

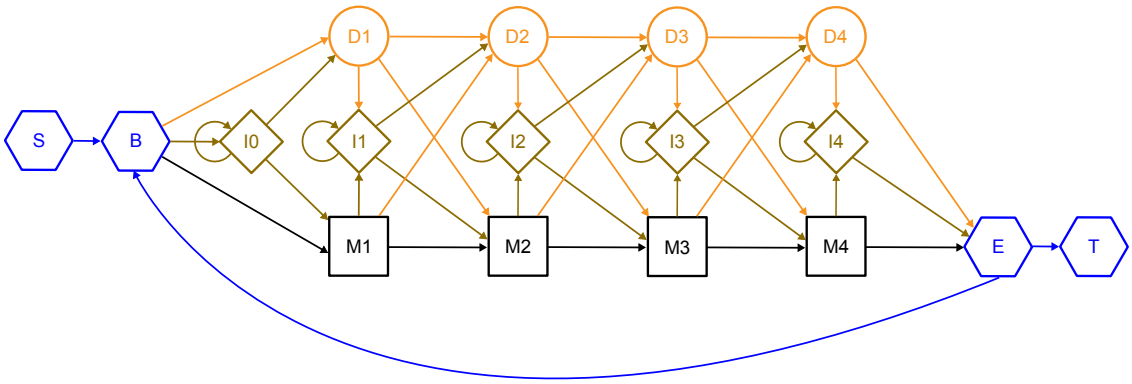


Figure 2.15: HMM for multihit global alignment

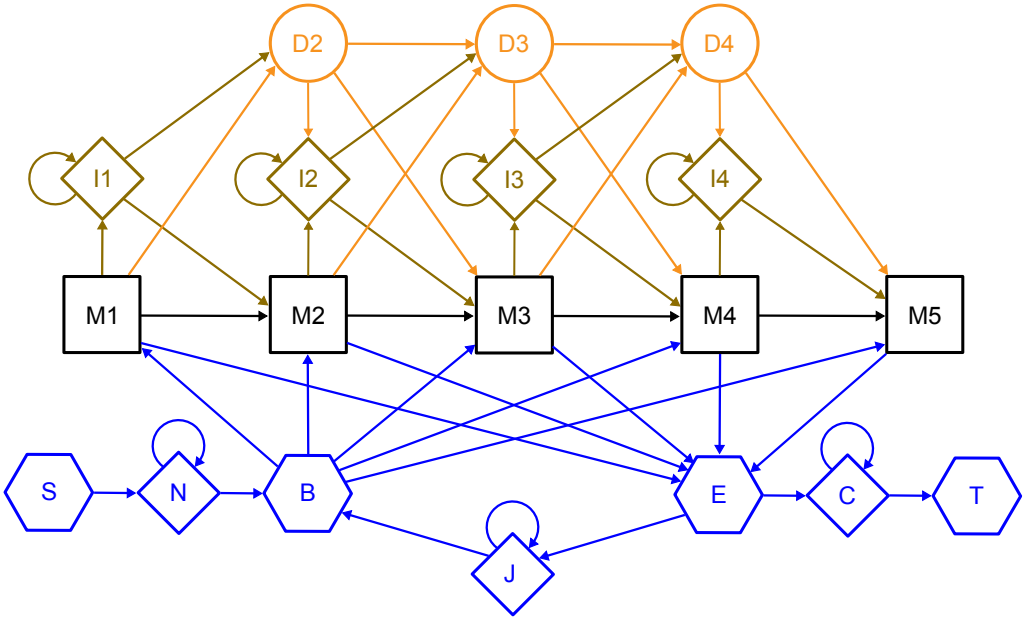


Figure 2.16: HMM for multihit local alignment

These multihit HMMs differ from the classical algorithms like Smith-Waterman and Needleman-Wunsch because the whole model is re-aligned against a new subregion of the sequence in each whole model loop. As a result, the same model region may be aligned multiple times to different sequence regions.

As for the algorithms to compute these multihit models, only the more interesting case of local alignment will be presented here. The Viterbi algorithm for multihit local alignment can be computed by the pseudo-code in Listing A.1. The Forward algorithm for multihit local alignment

model, with the additional special-states, is given by the pseudo-code in Listing A.2.

3

Parallelization of Homology Search

Contents

This chapter will review the most promising and well-studied techniques for parallelization of Homology Search, both of classical Alignment algorithms and Profile HMMs algorithms. The first section will deal with the parallelization of Alignment algorithms, and the later section will focus on the parallelization of Profile HMMs, which employ the same strategies, as it will be explained.

Parallelization approaches can be divided in Intra-task parallelism, wherein each alignment task is itself parallelized; and Inter-task parallelism, which consists of running multiple tasks (in this case, multiple alignments) in parallel.

3.1 Parallelization of Alignment Algorithms

3.1.1 Instruction-level Parallelism and Code Optimization

Scalar instruction-level parallelism in sequence alignment has been described by various authors such as Alpern [1], Szalkowski in SWPS3 ([51]), Farrar [18], Rudnicki [47], [45] and others. It is usually achieved by carefully re-arranging the instructions, normally done by the compiler or processor itself, in order to exploit the available processing pipelines on modern superscalar architectures. Special care must be taken on the implementation of very tight loops to allow the processor and compiler to better re-organize and parallelize the sequential code. The main concern is the removal of data dependencies and divergent execution paths (conditional branches).

Moreover, modern processors also employ many speculative optimization techniques, such as out-of-order execution and branch prediction. All these depend on the parallelization potential of sequential code, and as such, any un-parallelizable code section leads to a drastic performance cut. In particular, branch mis-predictions cause the whole pipeline to be flushed out, and all instructions already speculatively executed are discarded.

Another concern is the cache use: data locality is of the utmost importance, as well as the cache size itself, which determines how much of the inner variables and data can fit into cache. The cache size available, and the conscious and explicit optimization for such size, has a dramatic influence on the overall performance. A program with optimal cache utilization may come near a 100% cache hit rate on the innermost loops that can fit into cache, and hence achieve an enormous speedup on the global runtime.

Many strategies have been described on this level to carefully hand-tune the most critical assembly code, such as:

- loop unrolling of the innermost loop [1], [45], so that it processes two or more iterations sequentially at a time. This reduces the number of inner branches, and hints the processor to parallelize the two iterations on the available concurrent execution pipelines;

- tiling of loops (strip-mining) to improve cache reuse [1]. This technique consists of transforming a single loop into two nested loops, thus reducing the inner-loop length, to ensure that the inner-loop data stays in cache;

- reduce and remove any possible conditional branch, especially in the inner loops. In many cases, they can be moved out to the outer loops, initialization sections, or external same-level loops. This comes at the cost of some repeated or redundant work, but it is a small price to pay.

3.1.2 Fine-grained Parallelism using SIMD units

Fine-grained parallelism is implemented on the lowest instruction level. Besides the mechanisms described in the previous section, it can also be achieved through vector (SIMD) processing.

By now, SIMD extensions have already a reasonable long history of use and success. They implement a Single-Instruction/Multiple-Data parallel model on scalar CPUs, through the use of specialized vector processing units. Several of these instruction set extensions have enjoyed a considerable success, such as MMX/SSE on Intel and AMD's 3Dnow. One of most widely available is the SSE extensions on Intel processors. Each SSE register has 128-bits, capable of storing four 32-bit integers (rarely needed for alignment), eight 16-bit short integers (these can be used for alignments that reach high scores - such as very long sequences, very similar sequences, or very high base scores/penalties), or sixteen 8-bit bytes (the common case, allowing scores up to 255). Each stored element/value defines a single sequential 'execution channel'.

The first use of SIMD extensions for alignment tasks in general-purpose processors was employed by Alpern [1] in 1995, using 64bit registers to simulate SIMD vectors. Others followed quickly, using different parallelization strategies and decomposition patterns, on an increasing variety of different architectures (VAX mini-computers, Intel Pentiums, Cell BE, etc).

3.1.2.1 Decomposition Patterns in Intra-task Parallelism

When employing SIMD vector processing, using each matrix cell computation as a *primitive task*, the direction in which the cells are processed is of paramount importance. It determines how dependencies are parallelized, how many can be parallelized, and how costly it is to resolve the serial (non-parallelized) dependencies. When using SIMD units, the greatest obstacle is the dependencies within each SIMD vector (i.e. between the N parallel elements in the same N-channel SIMD unit vector). It is the most decisive factor in the overall performance.

Three main parallel decomposition patterns have been studied over the last decades to tackle the difficult problem of parallelizing data dependencies in intra-task parallelism:

Decomposition in a Diagonal Pattern

The most natural vectorial pattern to process the alignment matrix is by following the direction of the data dependencies, which is a diagonal direction 3.1(a). Processing each anti-diagonal in parallel is therefore a good method that was first proposed by Wozniak [56] in 1997 and often adopted ever since. One major advantage of this pattern is that it has no conditional branches in the inner loop. Still, it has one serious drawback: the heavy cost of the basic operations,

required to diagonally access and process the matrix. Moreover, the diagonal pattern restricts the applicability of some precious optimizations, such as the the one proposed by Green in SWAT (see Section 2.1.5).

Decomposition in a Vertical (or Horizontal) Continuous Pattern

Processing each column (or line) of the matrix in parallel is the other natural approach, perhaps the simplest and most intuitive 3.1(b). This pattern has also been thoroughly employed, with some very optimized and efficient implementations, starting with Rognes [46] in 2000 . The main problem of this strategy is the vertical (or horizontal, when parallelizing by lines) dependencies, which cannot be parallelized - they must be resolved sequentially, in a scalar way, amidst the SSE code, which is very inefficient.

Decomposition in a Vertical (or Horizontal) Striped Pattern

Building on Rognes solution, Farrar [17] in 2007 devised a striped pattern decomposition to tackle the problem of vertical dependencies 3.1(c). Through the use of a striped pattern, beside the parallelization of horizontal and diagonal dependencies, it is also possible to parallelize the vertical dependencies: each element j of the i-th segment will influence the same element j of the $(i + 1)$-th segment. The biggest remaining problem is the dependencies across 'segment sections' (continuous sections) - in Farrar's algorithm, these are processed later, in a second inner loop (*Lazy-F loop*), and following a SWAT-like optimization (i.e. the F values may only have any effect if they are higher than W_1). Despite the striped decomposition's better results, it is also less flexible and less prone to extension and customization (for instance, to run banded alignments). Moreover, for very short sequences, the striped pattern has an overhead (mostly from the Lazy-F loop) that is not easily amortized.

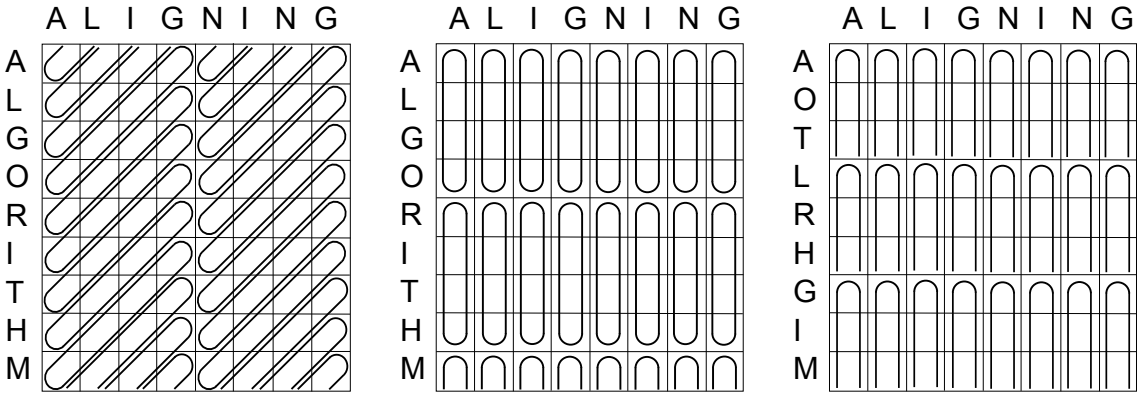
3.1.2.2 Main Problems of SIMD Intra-task parallelism

There are several issues afflicting SIMD algorithms, the most serious being:

Limited Range of the parallel SIMD elements

To maximize the algorithm's throughput, it is naturally desired to process as many elements in parallel as possible. This leads to fields of 8bits (1byte) for each element, which can have a maximum range of merely 255 (assuming only positive scores and penalties, or convert them to be such). For short sequences and reasonable typical scoring values, this is usually enough, but not always. Very similar sequences can easily overflow the 255-range, which invalidates the final score obtained, and forces alignment to be repeated with a higher range.

Scoring matrices



(a) Decomposition in anti-diagonals. The lower, upper, and right sides are padded.

(b) Decomposition in continous lines. The lower side (last segment per column) is padded.

(c) Decomposition in striped lines. Each segment is padded.

Figure 3.1: Decomposition patterns for intra-task parallelism

It is costly to access scoring matrices. Each access requires two indirections, one to find the sequence residue, and the other when the residue is used to access the matrix of scores. When using SIMD instructions, this problem increases tenfold - each vector element requires a different score, but those are scattered over the matrix (which is indexed by residue value, both in lines and columns).

Divergent execution paths

Divergent execution paths are introduced by conditional branches in the inner loops, which in practice turn the theoretical parallel code into scalar code. Furthermore, these hamper the processor's pipeline and branch prediction. One example is the required branches when processing the vertical dependencies within a SIMD vector.

Scalar sequential code

Some inherently scalar sequential code cannot simply be parallelized. For instance, the vertical dependencies cannot be fully parallelized, as well as the initialization scalar values. These singular cases require manipulation of the internal SSE elements with cumbersome vector instructions, such as shifts, packs and shuffles, to simulate simple scalar operations.

Cache Storage

A fundamental issue is the cache availability to store the arrays of SSE vectors used in the innermost loops: one E array, one of two H array(s), the F value, several auxiliary variables and constant values (variable names taken from Gotoh's equations, see Section 2.1.4). A lazy tuning of the inner loops data to the cache storage size may considerably impair the overall system

performance.

Banded Alignment

Banded alignment brings new challenges to vectorial parallelism. Farrar's striped pattern works very well for long runs, but the Lazy-F loop overhead becomes a problem in very small runs (a band width of 30 bases can fit entirely in two SSE vectors). Another problem is the frequent inner-loop re-initializations. Regarding the query profiles, their loading onto the SSE registers also induces an additional problem, since the memory becomes unaligned when processing each new column starting one cell later.

3.1.2.3 Improvements for SIMD Algorithms

To solve these main problems and improve the SIMD implementations, a few clever optimizations have been proposed:

Unsigned and Saturated Arithmetic

In order to increase the SIMD elements available range, unsigned arithmetic can be used, thus doubling the maximum range.

To deal with negative values, all scores and matrix cells are biased by a fixed small amount that guarantees no negative scores. Saturated arithmetic is then used to prevent values from dropping below 0 (and underflowing), as well as capping the values at the maximal value, preventing overflows. The minimum operation of the SW algorithm is also spared, being replaced by the implicit sature-to-zero operation. Underflows correspond to the maximum operation with 0, and are part of the algorithm. Overflows, on the contrary, are an error introduced by the limited range used, and are used as a sign that the whole procedure needs to be repeated with a higher range. As such, the test for the correctness of the alignment needs only to be done in the end, and it is simply '*final score* $>$*MaxValue - Bias*'.

There are cases however, when this mechanism cannot be used, as necessary architectural support is lacking. For instance, SSE only supports signed arithmetic for short values (16bit). To overcome this obstacle, some emulations of unsigned saturated arithmetic were devised. Farrar biased all values with -32k (the short min value), and then used saturated signed arithmetic to solve the problem. Rognes [45] in 2011 used a similar approach in the 7bit range, with both saturated arithmetic and unsaturated maximum, to avoid the Add instruction needed to unbias the scores.

For architectures without any support for saturation, Farrar [18] proposed a different biasing procedure. The idea is to add a small bias to every value, maximize each computed value with this bias (preventing it from dropping below 0), and testing for overflow with '*value* $>$*MaxValue - Bias*'.

Query-specific scoring profile

Packing each element at a time into the required SSE vector would be a considerable bottleneck. To avoid this, the substitution scores must be stored sequentially (or striped in Farrar's version) in memory, according to the respective sequence elements, so that a single load instruction can fetch the whole SSE vector from the substitution matrix in memory onto a register. The result is a new scoring matrix adapted to the query sequence, called a query-specific scoring profile.

The storage required for such a profile can also easily grow out of hand, especially for very large queries. This is a serious problem in computers with very small on-chip memory (GPUs and Cell's synergistic processors, for instance). The mechanism described in the next paragraph attempts to minimize this problem.

Compacting and expanding scoring profiles

A possible method to compact the scoring profile was proposed in [51]. The idea is to use the minimum required memory to keep the substitution scores (normally only 1byte per value), and expand them to larger sizes using SSE's unpack instruction, when executing with 16bit of higher elements' ranges. The sequences themselves can also be compacted with 2bits per base.

Optimizations of the Lazy-F loop

Some optimizations have been studied for Farrar's Lazy-F loop [51] [18], and achieved good results. One of the improvements consists mainly on relaxing the loop termination condition, since the original Lazy-F loop has in the worst case N (the complete column length) iterations. The other idea is moving some of the conditionals to a more external (and hence, less executed) position.

3.1.2.4 SIMD Implementation of Global Alignment

As it has been described in Section 2.1.3, global alignment (in particular the NW algorithm) differs from local alignment (SW) in that the alignment must cover the whole sequences instead of just partial, positive-scoring, subsequences. Global alignment can be regarded as subproblem of local alignment when the start and end point of the alignment are fixed. The NW algorithm, being a somehow reduced form of the SW algorithm, is also and more efficient and simpler to implement. The main differences in the implementation are the following:

- No need to keep the best score found. The resulting alignment score is always the score of the last (rightmost) cell. A maximum operation in the inner loops is thus spared

- No need to use a zero lower bound - the maximization with 0 is also avoided.

- No alignment re-runs due to overflowing: Considering that, increasing scores can be used instead of decreasing, the values' range will always be positive; hence there are no underflows and lower-bound saturation is not needed. Higher-bound saturation is still needed to avoid wrap-

around's. However, the overflown cells no longer constitute a problem since they have an extremely poor score (the higher the cell value, the worse it is). They are merely an innocuous side-effect of the procedure. In principle, they will never be chosen by the recursion's rule, and thus can be safely ignored. Actually, this may not be always the case, if the used arithmetic range is very ill-adjusted to the sequences. For correctly parametrized alignments, this technique can generally be safely used.

- Given that the start and endpoints of the alignment are known, some tuning of the algorithm can be done (for instance, heuristic filters and metrics).

3.1.3 Intra-task Parallelism on Multiple Cores

Middle-level parallelism is achieved by dividing the DP matrix in 'chunks' that can then be processed in parallel by the different cores on a single machine. This can also be extended to a grid of compurers, with each node processing a chunk, when the sequences are quite large. Several recent applications have been developed using this paradigm ([33], [37], [10], [26]).

This middle-level approach is much easier to implement than the fine-grained SSE: this one follows a simple MIMD model, in which each single processing datapath can diverge without any penalty, contrary to the SIMD model of vector processing that binds all the datapaths together.

The first issue to consider in this level is the order in which the chunks are processed. Which ones can be processed in parallel? How many at the same time? The strategy that seems to work best is by parallelizing along each anti-diagonal (similar to Wozniak's fine-grained pattern [56]), which is the natural direction of the algorithm's dependencies. This leads to a so-called "tiled-chunk pattern" (each diagonal of chunks being seen as a tile). Authors in [7], [6] and [26] use this pattern. The order in which the chunks are processed is not very strict: they can advance vertically or horizontally. The only restriction is that each chunk above and to the left must be processed before.

To minimize the communication cost of exchanging values between cores, each one should process a chunk as large as possible in each iteration. The best layout is to use a single chunk per process per level (communication round). Each chunk should have roughly close to $\frac{N}{P}$ columns (or rows vice versa, N columns, P processors) , so that the computational work is balanced among all cores. As for rows, each chunk can have any number of them. In particular they can have only one row, allowing for a finer-grain parallel decomposition [7] (especially useful when the communication cost is low, such as when using shared symmetric memory multiprocessors). Note that the score matrix is an abstract concept: in practice the Gotoh linear-space algorithm is always employed, and only a couple of arrays are needed.

When using a granularity of one for columns and rows (1 cell for each chunk), each diagonal only depends on the past two diagonals (the previous one for vertical and horizontal dependencies, and the other for the diagonal dependencies). When using larger chunks, the data

requirements are much better: each diagonal front needs only a 'stair-shaped' previous diagonal chunk-frontier (with $M + N$ cells, M columns, N rows). Since each diagonal has $a \times N$ cells (arbitrary granularity, a), the needed data-to-computation ratio is quite good. On the other hand, the higher the granularity is, the slower becomes the initial start-up time when some cores are idle waiting for available chunks.

Several alternative algorithmic strategies have been proposed to decompose the matrix between workers (such as ParAlign [44] and GCM-BSP [3]). These use exploitable properties of the algorithms to split the computing space among parallel nodes, and later merge the results, usually with some restriction or accuracy loss. Most of these algorithms also only work for global alignment, since the 'restarting' step of local alignment is much more unpredictable.

3.1.4 Inter-task Parallelism

Besides parallelizing a single alignment task, it is also possible to process multiple alignment matrices in parallel too (a single cell, or chunk, of each matrix at a time). This alternative method has also been much studied in the last decade.

3.1.4.1 SIMD extensions

Instead of using a SIMD vector to process multiple cells of the same alignment matrix, we could feed a single cell from multiple matrices to each one of the independent parallel channels. This approach was first proposed by Alpern [1] in 1995, and later extended and improved by Rognes in 2011 [45], on SSE.

Rognes thought of using vector processing such as SSE to implement inter-task parallelism (i.e. running many alignment tasks in parallel), using a lock-step processing model. Each vector is loaded with N different sequences, one in each vector element (or channel), and the algorithm aligns them concurrently against a target sequence, using the N vector channels to hold the independent computed values.

The main advantage of using parallel channels is the complete elimination of all data dependencies among SIMD elements. The alignment tasks are completely independent.

The drawbacks of this strategy are its restrictive applications and limitations, deriving from the fact that the N alignments proceed on step from begining to end. Any divergence on the program flow carries a prohibitive performance penalty, either as stoppage time or as wasted computing potential (for instance, empty padded cells).

One problem is the necessary pre-loading and arrangement of the per-residue emission scores. The emission scores to use depend on the searched sequences, and thus cannot be predicted, pre-computed and memorized before knowing those sequences. Each new batch of N sequences to search requires the loading of new emissions scores.

Rognes solution is to load the emission scores for the N different residues from the N database

sequences, each from its own emission scores' array, before the inner loop through the query. The emission arrays are thus loaded for the N new residues to process, and transposed from the original continuous pattern into a striped pattern, using the SSE operations unpack and shuffle. After the transposition, each new SSE vector then holds N single emission scores from the different N residues, all of which corresponding to a single query residue.

Another problem is the different lengths of the N concurrent database sequences. Rognes' solution is to stop the algorithm whenever one of the sequences ends, and replace it with another in the same channel. An alternative way would be to pre-sort the whole database by sequence length, and thus minimize the length differences in each run of the algorithm. The total cumulative length differences could be so small as to become negligeble, and so the wasted computed cells could be easily ignored.

3.1.4.2 Multiprocessors

Using a MIMD model for inter-task parallelization is much simpler, and more flexible. The sequences to align are divided by the different scalar processing units, which can be threads on a multicore or processes in separate machines. Intuitively, for smaller sequences, it is more convenient and efficient to use a single multicore processor to reduce communication costs, while for larger alignments, a grid of machines offers more available memory and resources. When compared to SIMD inter-task parallelism, this strategy is much less constrained, and supports larger sequences.

Load balancing is also an important issue: the overall sequence load for each node should be roughly identical. For this purpose, the sequences may be first sorted by length, and then 'horizontally' divided among the nodes.

3.2 Parallelization of Profile Hidden Markov Models

Here, the most studied and promising parallelization strategies for Profile HMMs will be discussed, building upon the methods for classical alignment algorithms, already described in section Section 3.1. As has been shown in the last section, the HMMs algorithms (Viterbi and Forward) have a similar structure and data access pattern as the classical single alignment algorithms (Smith-Waterman, Needleman-Wunsch etc). As would be expected, this similarity in the algorithms makes the parallelization strategies of single alignment to be very fitting for HMMs as well.

3.2.1 Comparison between Profile HMMs and Single Alignment algorithms

The algorithms used for Profile HMMs (Viterbi and Forward) are largely similar to the single alignment algorithms that were presented first (Smith-Waterman and Needleman-Wunsch with the Gotoh Optimization, see Section 2.1.4). They both are Dynamic Programming algorithms with

three components to model Matches/Mismatches, Insertions and Deletions. They both have the same recursive dependencies, such as, horizontal, vertical and diagonal.

To compare them, these are the Gotoh Recursions for the Smith-Waterman single-alignment algorithm:

$$\text{match } M(i,j) = Max \begin{cases} 0 \\ M(i-1,j-1) + weight(a_i; b_j) \\ I(i,j) \\ D(i,j) \end{cases}$$

$$\text{insert } I(i,j) = Max \begin{cases} M(i-1,j) + Wext + Wopen \\ I(i-1,j) + Wext \end{cases}$$

$$\text{delete } D(i,j) = Max \begin{cases} M(i,j-1) + Wext + Wopen \\ D(i,j-1) + Wext \end{cases}$$

And these are the three main recursions of the Viterbi algorithm for a Profile HMM that suppors local alignment, in log-space, using a similar notation and ignoring the other special states for now:

$$\text{match } M(i,j) = log\ e_{Mj}(x_i) + Max \begin{cases} B(i-1) + log\ t_{B_{j-1}M_j} \\ M(i-1,j-1) + log\ t_{M_{j-1}M_j} \\ I(i-1,j-1) + log\ t_{I_{j-1}M_j} \\ D(i-1,j-1) + log\ t_{D_{j-1}M_j} \end{cases}$$

$$\text{insert } I(i,j) = Max \begin{cases} M(i-1,j) + log\ t_{M_jI_j} \\ I(i-1,j) + log\ t_{I_jI_j} \end{cases}$$

$$\text{delete } D(i,j) = Max \begin{cases} M(i,j-1) + log\ t_{M_{j-1}D_j} \\ D(i,j-1) + log\ t_{D_{j-1}D_j} \end{cases}$$

It should be remembered that the transitions and emissions scores ($log\ t_{X'_jY_j}$ and $log\ e_Mj(x_i)$) are pre-computed scores, which depend solely on the sequence symbol (in the case of emissions) and the transition states (in both cases). It is thus clear that these two algorithms have are very much alike, and can be computed with roughly the same methods. The same holds true for the Forward algorithm, although with the particular table lookups.

Now, it is relevant to consider and contrast the differences between the two types of algorithms:

1. Gap Scores: While Smith-Waterman uses two single constant values, respectively for opening and extending a gap; Viterbi uses a series of constant values dependent on the position-specific transition states. This series of values introduces a necessary lookup in an array, and thus an unavoidable delay.

2. Match scores: Smith-Waterman uses a Weighting matrix, also known as a 'scoring matrix'

(such as BLOSUM [23] and PAM [9]), accessed by a combination of the target symbol and query symbol. The weighting matrix can be optimized into a query-specific Scoring Profile (see Section 3.1.2.3), yielding a single continuous array of scores for each inner-loop iteration. The rough equivalent in the Viterbi case are the Match emission scores, which vary according to the current Match State M_j and sequence symbol. Thus they are already in a 'model-specific profile', and can be re-used between sequences, essentially like a Query-Specific Profile for the Smith-Waterman.

3. Data dependencies:

   (a) In the SW algorithm, the I values depend solely on the last column, the D values on the last line, and the M values on the current I and D values, plus the previous diagonal M value. The computations can be re-ordered to use a single array for the I and M values, and a single cell for the D values (as per the Gotoh optimization). Although most authors use two switching arrays for the M values (for the current and last columns), it is possible to dispense one of them by doing delayed stores, i.e., loading the next $M(j)$ before storing the current $M(j)$. The re-ordered computations follow the scheme in Listing 3.1.

Listing 3.1 Pseudo-code of the Smith-Waterman algorithm with delayed Match writes

for $i \leftarrow 1$ to $TargetLength$ **do**
  $D \leftarrow Mprevious \leftarrow 0$
  for $j \leftarrow 1$ to $QueryLength$ **do**
    // Use of delayed *H(i-1,j-1)* value, i.e., *Mprevious*; and update in-place of *D*

$$D \leftarrow Max \begin{cases} Mprevious + Wext + Wopen \\ D + Wext \end{cases}$$

    // Update in-place using the values of *I(j)* and *M(j)* from the *i-1* outer-loop iteration

$$I(j) \leftarrow Max \begin{cases} M(j) + Wext + Wopen \\ I(j) + Wext \end{cases}$$

    // Use of delayed *H(i-1,j-1)* value, i.e., *Mprevious*

$$Mnew \leftarrow Max \begin{cases} 0 \\ Mprevious + weight(Target_i; Query_j) \\ I(j) \\ D \end{cases}$$

    $Mprevious \leftarrow M(j)$ // Save *H(i-1,j)* for next iteration
    $M(j) \leftarrow Hnew$ // Delayed store of *H(j)*
  end for
end for

(b) In the Viterbi (or Forward) algorithm, the data dependencies are more complex but they can also be implemented with single I and M arrays, using delayed load/store operations. As for the D dependencies, they now require a whole array instead of a single variable, since the previous D column ($D(i-1, j-1)$) has to be saved for the computation of the match states ('M'). The resulting re-ordered computations are shown in Listing 3.2 (all scores are in log-odds).

Listing 3.2 Pseudo-code of the Viterbi algorithm with delayed writes of the Match values. All scores are represented in log-odds.

for $i \leftarrow 1$ to $TargetLength$ **do**
  $D \leftarrow Mprevious \leftarrow 0$
  for $j \leftarrow 1$ to $QueryLength$ **do**
    // Use of delayed values from last iteration
    $$Mnew \leftarrow e'_{Mj}(x_i) + Max \begin{cases} B + a'_{B_{j-1}M_j} \\ Mprevious + a'_{M_{j-1}M_j} \\ Iprevious + a'_{I_{j-1}M_j} \\ Dprevious + a'_{D_{j-1}M_j} \end{cases}$$
    // Preemptive Load of previous values, from the *i-1* outer-loop iteration
    $Mprevious \leftarrow M(j)$
    $Dprevious \leftarrow D(j)$
    $Iprevious \leftarrow I(j)$
    // Delayed Store of new values
    $M(j) \leftarrow Mnew$
    $D(j) \leftarrow Dnew$
    $$I(j) \leftarrow Max \begin{cases} Mprevious + a'_{M_j I_j} \\ Iprevious + a'_{I_j I_j} \end{cases}$$
    // Preemptive Computation of *D(i,j+1)*, to be stored in the next iteration
    $$Dnew \leftarrow Max \begin{cases} Mnew + a'_{M_{j-1}D_j} \\ Dnew + a'_{M_{j-1}D_j} \end{cases}$$
  end for
end for

So, to surmise, the Viterbi algorithm requires more complex dependencies, with more delayed load/stores, and using more memory (state-specific transitions scores and a 'D' array).

3.2.2 Intra-task parallelization of Profile HMMs

Since the Viterbi and Forward algorithms for Profile HMMs are so similar to the Smith-Waterman algorithm, it can be used the same parallelization strategies described before for Single Alignment. These can be divided in intra-task (parallelizing each task in itself, each execution of the algorithm)

and inter-task approaches, also known as 'data-parallelism' (running multiple tasks concurrently, i.e. executing the algorithm concurrently on different data sequences).

Intra-task parallelism has been the most used on SIMD architectures (such as SSE). Lindahl in 2005 [34] used a continuous pattern (as in Section 3.1.2.1) on the AltiVec SIMD instruction set. Lindahl also unrolled the inner loop, producing 24 intermediate values to make better use of the available AltiVec registers, interleaved computations with memory fetches to hide some memory latency.

The most successful decomposition pattern for intra-task parallelism is Farrar's striped (Section 3.1.2.1), which was implemented in the HMMER tool ([16]). The other data decomposition patterns for SIMD intra-task parallelism, analyzed in Section 3.1.2.1, can equally work for Profile HMMs, but Farrar's approach has been the most adopted due to its better performance.

The Farrar method for Viterbi Decoding of Profile HMMs mostly uses the re-ordered sequence of operations shown in the previous section, with a data decomposition for SSE units following the striped approach. The main differences between a striped parallelization for Viterbi, and the original for Smith-Waterman, are in the treatment of the Delete values.

Each ith Match state of the Profile HMMs has an incoming transition from the *previous* $(i-1)th$ Delete State, i.e., the Delete state of the previous column, whereas in the Smith-Waterman, the Match values depended on the Delete values *of the same column*. As a result of this difference in the Dynamic Programming relations:

1. The whole column of D values, i.e., the whole set of Delete states for each input symbol, has to be saved in order to be used for the calculation of the M values of the next input symbol.

2. The M values computed for the $(i-1)th$ input symbol do not require the D values that are computed in that same iteration (since these will be only used later). This 'de-coupling' of the calculations allows the Lazy-F loop to correct *only* the D values of the current iteration, and leave the M and I values unchanged, since these were computed with D values of the previous iteration. The Lazy-F loop is thus quite simplified, when compared to the Smith-Waterman case, in which any change to a D value had also to be propagated to the M and I values that had been computed with the Ds of the same iteration.

3. In the main inner-loop, over the model states, only the M $\rightarrow$ D are used to calculate the temporary D values. The D $\rightarrow$ D transitions can all moved down to the Lazy-F loop, which will factor them in as required. Again, this is possible because the D values are not immediately used to calculate the M values.

Another important issue in the parallelization of Profile HMMs algorithm is the necessary memory, larger than for the SW algorithm (due to the transitions scores and the new D array).

3.2.3 Inter-task parallelization of Profile HMMs

An alternative solution is to use inter-task parallelism (i.e., running multiple tasks in parallel) instead of intra-task. Inter-task parallelism is generally implemented on a multi-threading or grid environment, wherein each task runs isolate on its own processor core. This approach was pursued by [54], which scaled HMMER on a computing grid with MPI. The latest release of HMMER has been parallelized for a multi-threading environment, and distributed with MPI.

Inter-task parallelism can also be achieved with a SIMD model (i.e., vectorization), such as GPUs and SS (the same technology used for intra-task parallelism). ClawHMMER [25] implemented the Viterbi and Forward algorithms on a GPU, by vectorizing multiple database sequences against the same model. The same could be explored with SSE, in a way similar to what Rognes did for the Smith-Waterman algorithm (see Section 3.1.4.1). There is however a dearth of work devoted to inter-task vectorization of HMMs algorithms in SSE, a dearth which will be addressed in the next chapter.

3.2.4 HMMER

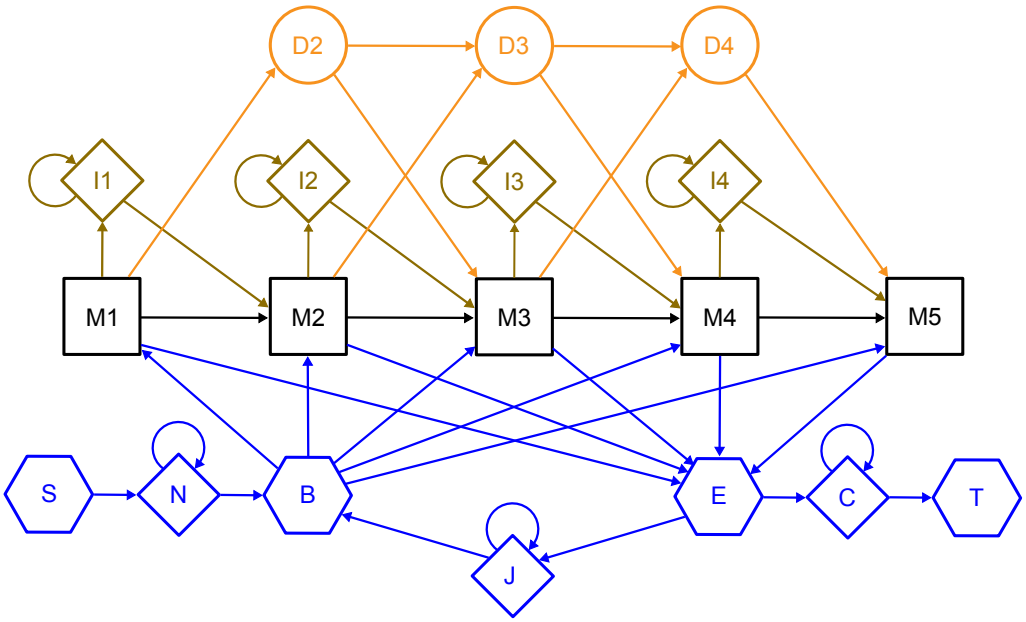


Figure 3.2: Complete model used by HMMER 3.0. Allows for multiple hits (multihit mode) in either local or global alignment mode.

HMMER is a tool developed by Eddy [14] that uses Hidden Markov Models to do sequence homology search. HMMER has become a very popular suite, and a strong focus of Profile Hidden Markov Models (HMMs) research, with many efforts conducted to improve it and optimize it ([12], [19], [34], [54], [25]).

The original version of HMMER relied on a model architecture similar to the Krogh-Haussler

model, called 'Plan 9'(due to its nine transitions per state triplet). The current version employs the 'Plan 7' model architecture, shown in Figure 3.2. The core of this architecture is similar to Krogh-Haussler/Plan 9, but Plan 7 has no D $\rightarrow$ I or I $\rightarrow$ D transitions, thus reducing the number of transitions to 7. Plus, Insert states have emission probabilities identical to the background distribution, canceling its component in the algorithm.

Some special-states were also added to Plan 7 to allow for arbitrary restarts (thus making it a local alignment) and repeats (multihit alignment). These special states are parametrized to control the desired form of alignment (the HMMER *alignment mode*). HMMER generally supports 6 different combinations of *alignment modes*, formed by combining 2 options: unihit vs multihit alignments, and the basic alignment mode (global, glocal, and local). These are explained bellow:

- Local mode: Aligns regions of the sequence against regions of the model. In unihit local alignment (HMMER's *UNILOCAL*), only one region is matched. In multihit local alignment (HMMER's *LOCAL*), multiple regions in the sequence and model are matched, and the same model region may be aligned multiple times (thus differing from classical Smith-Waterman);
- Global mode: Aligns the whole sequence against the entire model. It corresponds to the Needleman-Wunsch algorithm. The special states self-looping states N, and C are disabled, as well as the $B \rightarrow Mi's$ and $Mi's \rightarrow E$ transitions;
- Glocal mode: The whole model is aligned against a subsequence of the target. This is an interesting particular alignment mode offered by HMMER. To make it work, the $B \rightarrow Mi's$ and $Mi's \rightarrow E$ transitions are disabled (thus forcing the whole alignment of the model) but the N and C self-looping states are enabled, to consume arbitrarily long leading and trailing regions of the target sequence.

The original striped implementation of the Viterbi and Forward algorithms in HMMER used SSE units with 4 channels of (32 bit) floats. Floating-point values are the most suitable to the task, since we are dealing with probabilities with possibly many decimal places.

To increase the overall speedup, the data units were later changed to 8 channels of signed integer words (16 bit), through a discretization process of the floating-point probabilities. The discretization is done by a simple scaling operation, plus to an offset to gain a slightly larger representation space.

The SSE arithmetic uses signed saturation, which automatically limits overflowed values, that can then be checked for high-scoring hits. Underflows would be a problem however, since the algorithms use $-\infty$ scores as nullifying limit values, which should never touch the normal valid scores (i.e. scores without $-\infty$ terms). As such, HMMER uses a discretization range that ensures the normal scores will never underflow for model and sequence lengths below 10^{16} ([16]).

The latest HMMER versions are HMMER 3.0, released in March 2010 [16], and HMMER 3.1b1 released recently in May 2013. Since HMMER 3.0, only the Local mode is supported in full; and,

in particular, in the optimized implementations of the SSE-vectorized search filters. According to the author ([15]), accurate statistics are only available for the Local mode, and the other modes are poorly understood in terms of probabilistic significance. Furthermore, only the Local mode does not underflow the limited-precision presentation used in the SSE implementations.

HMMER Pipeline

HMMER 3.0 introduced a processing pipeline which uses a combination of incremental filters, each more accurate, restrictive and expensive than the previous one. Figure 3.3 gives a representation of the filtering steps of the pipeline.

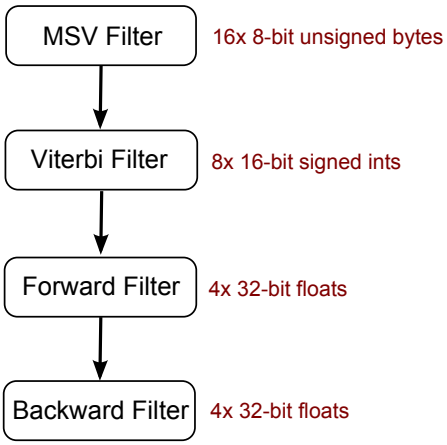


Figure 3.3: Diagram of HMMER's vectorized pipeline.

The MSV (*multiple segment Viterbi*) filter computes an optimal sum of multiple ungapped local alignment fragments. All of these filters have been parallelized with SSE, following Farrar's striped strategy, with increasingly higher precision. The fastest and coarser MSV filter uses 8-bit score values, the Viterbi Filter uses 16-bit scores, and the Forward and Backward filters use the full 32-bit floating point scores. In the case of Viterbi, an 8-bit precision was found to be insufficient and produce unacceptable high errors. The overall pipeline has also been parallelized to run on a multi-threaded, multi-node MPI environment with each thread processing each sequence independently.

4

SIMD Inter-task parallel Viterbi

Contents

4.1 Proposed Solution

The focus of the present work is to develop a parallelization approach based on SIMD (namely SSE on Intel Processors) inter-task vectorization for Profile HMMs algorithms. Rognes in 2011 ([45]) pursued this strategy for alignment algorithms like Smith-Waterman. Building on Rognes' work, the same strategy can be followed for HMMs algorithms. However, as mentioned in the previous chapter, this is a promising new approach, that has not yet been much explored.

This work focuses on one of the most popular HMMs Homology search suites: HMMER [16]. HMMER offers many tools, and one of the its key processing steps used by many of the tools is the Viterbi Decoding algorithm with a Profile HMM. The Viterbi step is currently (as of May 2013) being implemented on SSE with Farrar's intra-task striped pattern. In this work, an alternative solution was developed,

The Rognes-based implementation of Viterbi Decoding created in this work was named *COPS* (Cache-Oblivious Parallel SIMD Viterbi). It is targeted at the HMMER suite and it is compatible with its internal configurations, being mostly interchangeable with the exception of the requirements of Inter-task parallelism (i.e. processing batches of sequences each time instead of just one). Also for this reason, COPS was developed on top of the HMMER suite as a standalone tool instead of being integrated into the HMMER pipeline of search filters (one of which being the Viterbi algorithm). A full integration into the HMMER pipeline was deemed unsuitable, since the pipeline is designed to process only one sequence at a time. Later work may extend this approach to the Forward algorithm as well and as into AVX2, Intel's new vector instruction set (an extension of SSE).

The following sections will present the developed implementation of the Viterbi algorithm with inter-task vectorization, the problems found, and two new methods to improve the strategy used by Rognes. Finally, the last section will describe a coarser-grained multi-threading parallelization of the algorithm itself, running on top the vector inter-task parallelization.

4.2 Rognes-based SSE Inter-task vectorization

The main goal of this work consisted in implementing an inter-task vector parallelization of Viterbi Decoding in Intel's SSE, such as Rognes did for the Smith-Waterman algorithm [45].

Recall that this approach consists of using the N parallel SIMD channels for N different references (i.e. computing the Viterbi equations in parallel for the N sequences). This is known as 'Inter-task parallelism', as described previously in Section 3.1.4.

The initial implementation was largely based on Rognes' vectorization of the Smith-Waterman algorithm, with the changes necessary for the Viterbi problem. Since it is a form of inter-task parallelism, each channel is expected to perform the same operations, and thus the core of the algorithm maintains mostly the logic of the scalar version. The corresponding code for the inner

loop over the model states is presented in Listing A.3.

Initially, 32-bit floating-point values were used, which allowed for four independent channels in the SSE vectors. Four different sequences are fed into these four channels. Mmx, Imx and Dmx are the SSE dynamic programming arrays, respectively for Match, Delete and Insert state values. Mpv, Ipv and Dpv are auxiliary SSE registers to temporarily hold delayed or preempted values. vB and vE are the SSE registers for the E and B flanking states values.

To compute the Match states, there are dependencies on M, D, and I, namely from the previous state triplet and previous sequence symbol (Mpv, Dpv, Ipv). These dependencies must be retrieved in the previous state iteration, *before the writes*, to fetch the values of the previous symbol (indexes (i-1, k-1) in a DP matrix) before they are re-written. Each computed Match value is used to update the E (semi-end) state.

In regards to Insert states, they depend on the M and I values in the same state-triplet, but from the previous sequence token. The M and I values can thus be fetched from the current state iteration, before the updates (saved in Mpv and Ipv). Note that the D $\rightarrow$ I transitions have been removed by design from the model.

Finally, for the Delete states, there are D $\rightarrow$ D and M $\rightarrow$ D dependencies, from the same sequence token but previous state triplet (indexes (i,k-1)). To find these, it is necessary to preemptively compute and save the D and M values in the previous state iteration (Mnext saved and Dcv preemptively computed).

After each inner loop over the state triplets, the special states (E, J, C, N and B) are updated.

The measured performance (number of values computed per second) is mostly dependent on the model length (denoted by M, number of model state-triplets), with hardly any influence from the sequences length (denoted by L). It achieved a 78% performance improvement on the serial version by this initial parallelization with the Rognes-bases strategy. A diagram of the Rognes algorithm applied to HMMER is shown in Figure 4.1.

Initial Optimizations to the Rognes strategy

- Since the Insert emission scores are set to the background distribution in HMMER by default, they yield a null contribution and can be removed from the code.

- It is possible is to pre-compute and save the vB values for each state triplet. This however cannot be done when dealing with a dataset composed of varying-length sequences, since the model must be re-configured with each different sequence length.

- For the main transition scores (i.e. between normal states), a single array was allocated, and the 8 transitions were stored in a interleaved manner, continuous in relation to the state triplets, in the order by which they are fetched in the inner loop. This allows for a better use

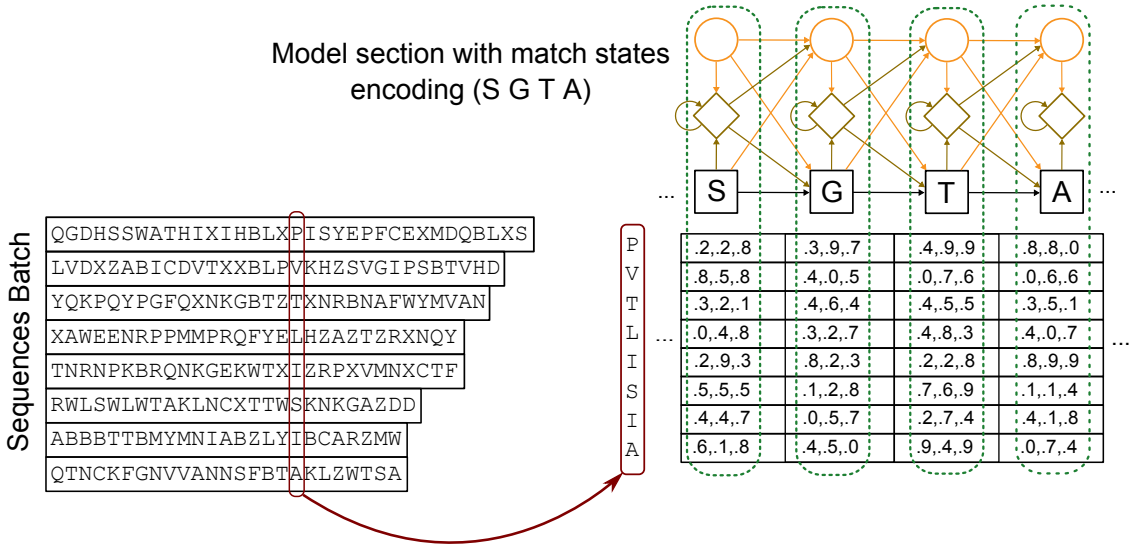


Figure 4.1: Rognes inter-sequence strategy applied to Viterbi Decoding in HMMER.

of memory, avoiding mixed accesses to multiple different memory locations. This arranged pre-allocated layout yielded a performance speedup of 15%.

- Another improvement is unrolling the last iteration of the inner loop. The last iteration is particular since it only needs to compute the match value, hence the other operations can be removed.

4.3 Loading of Emission Scores

Special concern was given to the method of loading and arranging the per-token Match Emission scores, because it was found that this step accounted for a substantial fraction of the overall time spent by the application. First the original method used by Rognes in the Swipe tool was implemented, evaluated, and then a new improved strategy was devised and implemented. The two are analyzed in the following sections.

4.3.1 Rognes method of Loading the Emission Scores

The Match Emission scores present a significant problem: it is impossible to arrange the emission scores in an memory-efficient pattern before the sequence tokens are known. A complete pre-arrangement of all possible $token \times state$ combinations would require $AlphabetSize^{AlphabetSize} \times L \times M$ 16-byte values. For DNA, which has an $AlphabetSize$ of 4, the required memory is manageable. For proteins, which are the usual target of such systems and have an usual $AlphabetSize$ of 21, it exceed the available memory of almost every system.The complete pre-calculation approach is infeasible.

To tackle this problem, initially, the loading of the Match emission scores was done using the method of Rognes' Swipe program [45]. The method consists in pre-loading and arranging each quartet of emission scores ($4xM$, M being the Model length), corresponding to the 4 parallel sequence tokens, necessary for the entire inner loop through the model states. For each new tuple of sequence tokens, an array of M SSE vectors is computed and arranged for efficient access in the inner loop.

The costly arrangement must compute a transposition of the original scores, in successive rounds of 4x4 32-bit values: arrays indexed by symbol are transposed into consecutive tuples of 4 interleaved scores from the 4 different sequences. The transposition is achieved through SSE *unpack* instructions, which interleave the lower or higher-order bits of 2 arguments, starting with a finer-grain unpack (32-bit unpacks when using 32-bit scores) and proceeding to coarser-grained unpacks (ultimately 64-bit unpacks that compute 128-bit values, size of the SSE vectors). This strategy is illustrated in Figure 4.2, with the corresponding pseudo-code shown in Listing A.4.

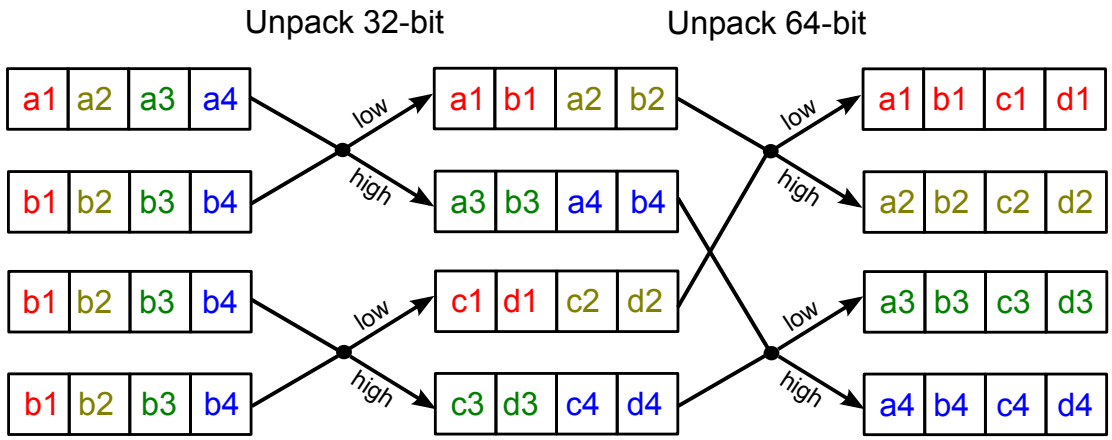


Figure 4.2: Emission scores pre-processing using SSE unpacks, according to Rognes' Swipe tool.

Besides the costly SSE operations in themselves, this method suffers from a poor memory utilization, requiring an additional write (and later read) to store and load the re-arranged scores. The results obtained point to roughly 30% of the program runtime being spent in the arrangement of emission scores, a considerable amount.

4.3.2 Inline method of Loading the Emission scores

The original strategy for the pre-processing of emission scores, as developed by Rognes in Swipe, suffered from a serious performance penalty. In order to relieve this burden, a new method was devised to keep Emission values as closer to the processor as possible, and thus avoid the drawback of the additional re-writing on memory.

The transposition of scores is processed by rounds of 4 (since each SSE vector holds 4 floats,

and the transposition must use a square 'matrix'), so the round of 4 values is the smallest block of data that can be transposed at the same time. The new method computes these 4-floats transpositions inlined between each round of 4 inner loop iterations. Each transposition produces 4 SSE vectors of emission scores, which can then immediately be used to compute 4 state triplets. The transposed scores are merely stored temporarily in close cache memory, and thus avoid the penalty hit of a full round of memory re-writings and improves the cache efficiency. This requires the inner loop be unrolled in 4 iterations.

After this transformation, the new inner loop consists of the code in Listing A.5. The optimization of interleaved scores' loading lead to an execution time roughly 30% to 40% faster than the pre-loading method used by Rognes' tool.

This method requires the number of model states to be a multiple of 4 (for 32-bit floats), to support the 4-state loop step. In order to easily deal with this, the model should be padded with dummy states up to a 4-state barrier. The dummy states carry dummy scores, set to $-infinity$, so that they have a null effect on the final results. These extra dummy states have a negligible effect on the overall performance.

4.4 Discretization to 8x16-bit integer channels

In order to increase the parallelization potential, the original 4 channel x 32-bit float version was converted to an 8channel x 16-bit integer version. This conversion implied a discretization of the floating-point scores, similar to the one employed by HMMER in its Viterbi Filter algorithm.

The discretization used a scale transformation with a factor configurable at compile-time. HMMER's authors estimate an optimal scaling factor of $\frac{500.0}{\ln 2}$, which does not underflow for any $L, M <= 10^{16}$ ([16]).

Although 16-bit scores do not underflow (for any practical application), they may easily overflow when the sequence is slightly similar to the underlying family of the model. The occurrence of such overflows is detected by numerical manipulation of the vE values at the end of the inner loop.

3/2-nat optimization

The 3/2-nat optimization was adopted to simplify the special-state calculations, and it is only valid for local alignment: HMMMER assumes that, for local alignment with reasonable large and non-homologous sequences, only a small section will be matched. The rest of alignment will be merely filled with gaps (NN, CC, and JJ loops), totaling close to L gaps. So, the optimization removes the NN, CC, and JJ loop transitions from the algorithm, and adds the correspondent cumulative sum of their contributions (3.0 for multihit mode and 2.0 for unihit) in the end.

In COPS, it was equally applied the '-2.0 nat approximation' used by HMMER: N $\rightarrow$ N and C $\rightarrow$ C transitions are deleted, and a -2.0 offset bias is added to N in the beginning, and subtracted

after the algorithm finishes. This value approximates the cumulative contribution of N $\rightarrow$ N and C $\rightarrow$ C insertion loops which, for a large L, is given by $\log \frac{L}{L+2}$.

4.5 Model Partitioning to improve the L1 cache utilization

A significant bottleneck was detected in the utilization of the innermost L1 data cache when using large models. This section will analyze the problem, and the solution that was found based on loop-tiling the inner loop.

4.5.1 Problems with First-level Cache Efficiency

At this point, early results showed a considerable performance degradation with an increasing Model length. After some experiments, it was found that the deterioration was caused by an exponential increase in the number of occurring L1 cache misses (the fastest cache level in Intel processors). It was also found that these L1D cache misses were overwhelmingly in the core inner loop of the code ($\sim$97% of total cache misses).

An explanation for the explosion of cache misses seemed likely to be the eviction of the M-length auxiliary and scores' arrays from the cache between each execution of the inner-loop (i.e. between each outer-loop iteration). These arrays are re-used in each new passage through the inner loop, so it would be highly advantageous to keep the data in cache as long as possible. As such, a brief survey of the memory requirements of the most critical section of the code (the inner loop) is warranted. The original Rognes work likely did not suffer from these limitations, since the Smith-Waterman algorithm requires much less memory.

The estimated memory size of the data most heavily accessed by the inner loop is shown in Table 4.1, for both HMMER's ViterbiFilter and this work's COPS.

Table 4.1: Theoretical estimates of memory used by the core inner loop, in bytes

Spec	COPS, 16-bit integers	ViterbiFilter (HMMER), 16-bit integers
Mmx, Dmx, Imx	$3 \times M \times 16$	$3 \times M \times 2$
Transition Scores	$8 \times M \times 16$	$8 \times M \times 2$
Match emission scores	$M \times 16$	$M \times 2$
Auxiliary Emission array	24×16	–
$\sim$20 aux. variables	20×16	$20 \times 16 = 320$
Total	$192 \times M + 700$	$24 \times M + 320$
Total minus E.M. scores	$176 \times M + 700$	$22 \times M + 320$
Max. M to fill a 32KB cache	$\frac{32768-700}{192} \approx 167$	$\frac{32768-320}{22} \approx 1470$

We measured the computation performance and the number of L1D cache misses for the COPS tool of this work and the ViterbiFilter program of HMMER, using models of varying lengths.

The performance was measured in million of state scores computed per second, using the Linux function *ftime*. The cache profiling was conducted using Hardware counters, with the tool OProfile. [32] These tests were conducted on a commercial Intel Core2 architecture, with 32KB of the innermost L1D cache. The Emission match scores are only used once, they are never re-used, so their memory footprint and access pattern is an unavoidable hurdle, whose impact cannot be minimized.

The HMMER striped version uses a fraction of the memory requirements of COPS, mainly due to the shorter inner-loop and related vectors (with only $\sim\frac{M}{16}$ elements instead of M). This reduced memory footprint allows it to avoid the limits of the L1D cache for the inner-loop until a comfortably high model length (roughly 1470 in the estimates). The experimental results bear out this view (Figure 4.3).

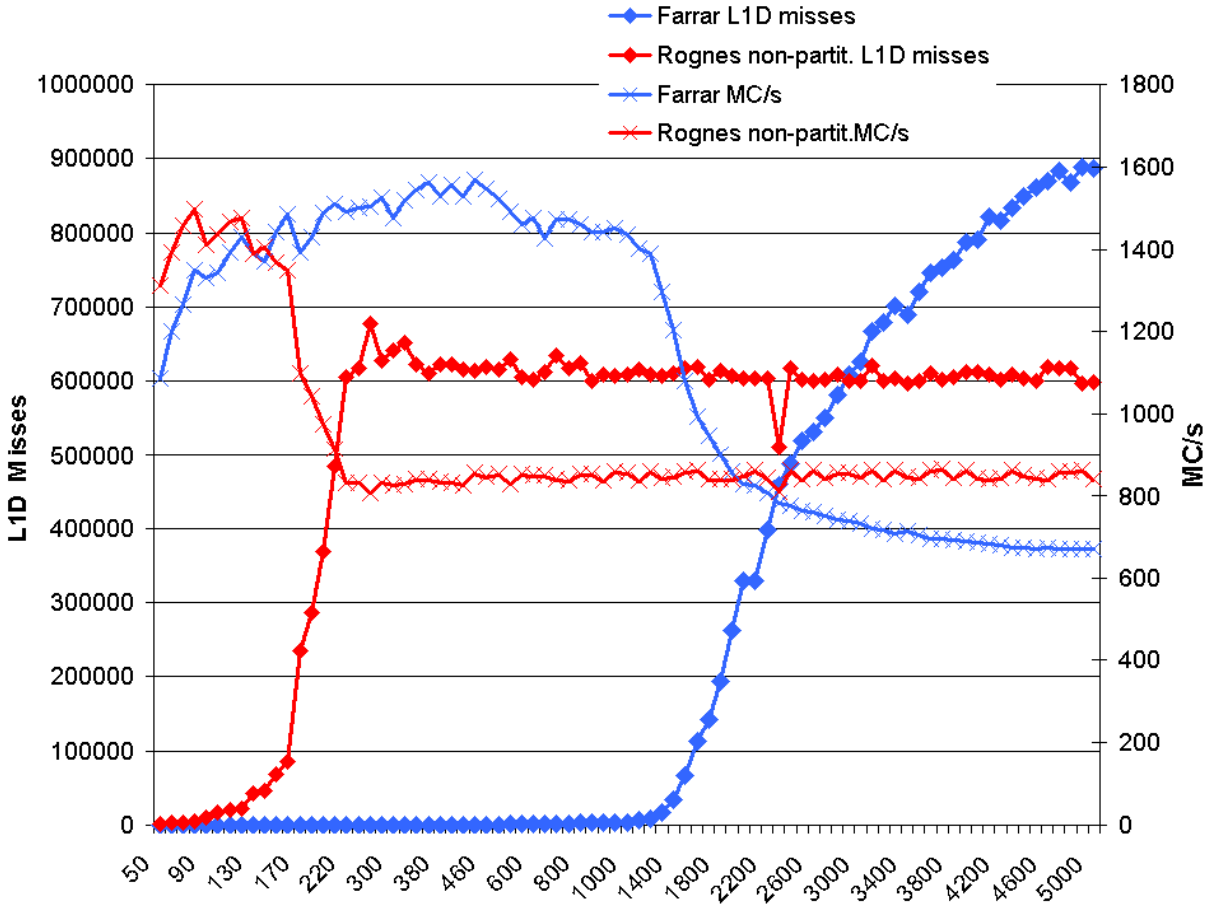


Figure 4.3: Results of cache profiling of HMMER's Farrar-based ViterbiFilter, and the Rognes-based COPS, with a 32KB L1D cache. Measured in millions of cells computed per second (MC/s) and L1 data cache misses.

The theoretical estimates point to *critical point* of full L1D utilization around ∼1470 for ViterbiFilter and ∼167 for COPS, for 32KB L1 data caches. These points coincide rather well with the observed spikes in cache misses and drops in performance, which figure strongly correlated in

the results:

- Rognes-based COPS: from a M of 120 to 220, there is a 26000% increase in cache misses, and a 44% drop in performance. Both the performance and the number of misses stabilize for M $>$ 220.
- Farrar-based ViterbiFilter: between a M of 1200 and 2300, the number of misses rises 50000% with the performance falling 43%. Both the performance and the cache misses continue to deteriorate for longer models, although at a slower pace.

So, for practical applications, where the model length rarely exceeds 2000, COPS had a considerable memory handicap, which left it sorely uncompetitive vis-a-vis the striped ViterbiFilter. The graphic shows how the striped version was 70% faster than the inter-sequence version for Ms between $\sim$180 and $\sim$1400. For models exceedingly long, starting with Ms $>$ 2400, COPS is again competitive and was able to achieve a small speedup against the HMMER implementation.

4.5.2 Partitioning the Model

In order to solve the cache efficiency problem, it was devised a *loop-tiling* strategy based on a partitioning of the model states, that sought to limit the amount of memory required by the core loop. The normal model states are split in blocks of maximum *M.P.* states, and the blocks are iterated over in a new outermost-loop. This corresponds to a standard *loop-tiling* (a.k.a., *strip-mining*) strategy, illustrated by the pseudo-code in Table 4.2.

Table 4.2: Comparison of the inner loop code, before and after partitioning the Model

Original code, non-partitioned	Strip-mined code, partitioned

▷ Loop through the sequence symbols
for $i \leftarrow 1$ to $SequenceLength\ (L)$ **do**
...
▷ Loop through the model state-triplets
for $i \leftarrow 0$ to $M-1$ **do**
▷ Core Viterbi code
...
end for
▷ Update the special states
...
end for
	▷ Loop through the partitions
	for $i \leftarrow 1$ to $Npartitions$ **do**
	▷ Loop through the sequence symbols
	for $i \leftarrow 1$ to $SequenceLength\ (L)$ **do**
	Load_Data_From_Last_Partition(i)
	...
	▷ Loop through the state-triplets
	▷ of the current partition
	for $i \leftarrow 0$ to MP **do**
	▷ Core Viterbi code
	...
	end for
	▷ Update the special states
	...
	Store_Data_For_Next_Partition(i)
	end for
	end for

The outer loop (new middle loop) over the sequences mostly re-uses the same memory locations (except for emission scores), which are accessed in the inner core loop, so these locations

should be kept in close cache. By limiting the model states loop to a maximum of *MP* state-triplets, we can effectively guarantee that the whole sequence loop (the middle loop in the new layout) does not access more than roughly $\sim (176 \times M + 320)$ bytes (disregarding the emission scores). The memory required by the inner loop is then cached in close memory, and repeatedly accessed over all the sequence loop while in cache, thereby reducing drastically the occurrence of cache misses. The maximum partition length, *MP*, is adjusted to achieve an optimal cache occupation, one that fills the available capacity of the closest data cache up to its limit.

There are two memory blocks which cannot be *strip-mined* and thus degrade the performance of this optimization:

- Emission scores, which must be refreshed (re-computed) for each new round of sequence tokens. These values are only accessed once, so it is counter-productive to consider their cacheability.

  An attempt was made to prevent their loading into cache, by using the Intel *software prefetch* instructions for non-temporal access, which tell the processor that the data is *non-temporal* (i.e.; will never be reused) and thus should not evict other data from the cache. However, the non-temporal prefetches could not improve on the hardware prefetching already done by the processor, since the data access patterns are very regular (continuous actually). The number of LLC caches misses (i.e. for the larger L2 and L3 caches) is practically null.

- Dependencies that must be exchanged between partitions. The last Match, Insert, and Delete contributions from each partition have to be carried on to the next partition, and so they have to be saved at the end of each partition. Each partition receives as input one line of previous states, with one state-triplet for each 8-fold round of sequences, and produces as output another line of values to the next partition.

  These dependencies can be minimized to 3 values per sequence round (vE, Mnext, and Dcv) after re-factoring the core code and moving the computation of Mnext with the 3 state dependencies to the end (Table 4.3).

The processing order of the partitions is shown in Figure 4.4.

4.5.3 Problems of Model Partitioning

There are some restrictions to the applicability of model partitioning however. The computation of each symbol-tuple through the model must be independent between each other. In particular, it must not depend on the final result of the previous symbol-tuple, which is the case for multihit alignments (alignment of the model multiple times against a sequence target). Therefore, the model can only be partitioned for unihit alignments (HMMER's 'uni-local mode'). In unihit alignments, the J special state, which encodes the token-to-token dependencies, is disabled.

Table 4.3: Comparison of the inner loop, before and after refactoring the Match computation to minimize the number of dependencies

Original code	Refactored code

▷ Loop through the model state-triplets
for $i \leftarrow 0$ to $M-1$ **do**
$Mnext \leftarrow Max \begin{cases} vB + t_{BM}(k) \\ Mpv + t_{MM}(k) \\ Ipv + t_{IM}(k) \\ Dpv + t_{DM}(k) \end{cases}$
$Mnext \leftarrow Max(Mnext, e_{match}(k))$
$vE \leftarrow Max(vE, Mnext)$
$Dpv \leftarrow Dmx(k)$
$Ipv \leftarrow Imx(k)$
$Mpv \leftarrow Mmx(k)$
$Mmx(k) \leftarrow Mnext$
$Dmx(k) \leftarrow Dcv$
$Imx(k) \leftarrow Max \begin{cases} Mpv + t_{MI}(k+1) \\ Ipv + t_{II}(k+1) \end{cases}$
$Dcv \leftarrow Max \begin{cases} Mnext + t_{MD}(k+1) \\ Dcv + t_{DD}(k+1) \end{cases}$
end for
	▷ Loop through the model state-triplets
	for $i \leftarrow 0$ to $M-1$ **do**
	▷ Use partial value of Mnext
	$Mnext \leftarrow Max \begin{cases} Mnext \\ vB + t_{BM}(k) \end{cases}$
	$Mnext \leftarrow Max(Mnext, e_{match}(k))$
	$vE \leftarrow Max(vE, Mnext)$
	$Dpv \leftarrow Dmx(k)$
	$Ipv \leftarrow Imx(k)$
	$Mpv \leftarrow Mmx(k)$
	$Mmx(k) \leftarrow Mnext$
	$Dmx(k) \leftarrow Dcv$
	$Imx(k) \leftarrow Max \begin{cases} Mpv + t_{MI}(k+1) \\ Ipv + t_{II}(k+1) \end{cases}$
	$Dcv \leftarrow Max \begin{cases} Mnext + t_{MD}(k+1) \\ Dcv + t_{DD}(k+1) \end{cases}$
	▷ Partial computation of Mnext
	$Mnext \leftarrow Max \begin{cases} Mpv + t_{MM}(k) \\ Ipv + t_{IM}(k) \\ Dpv + t_{DM}(k) \end{cases}$
	end for

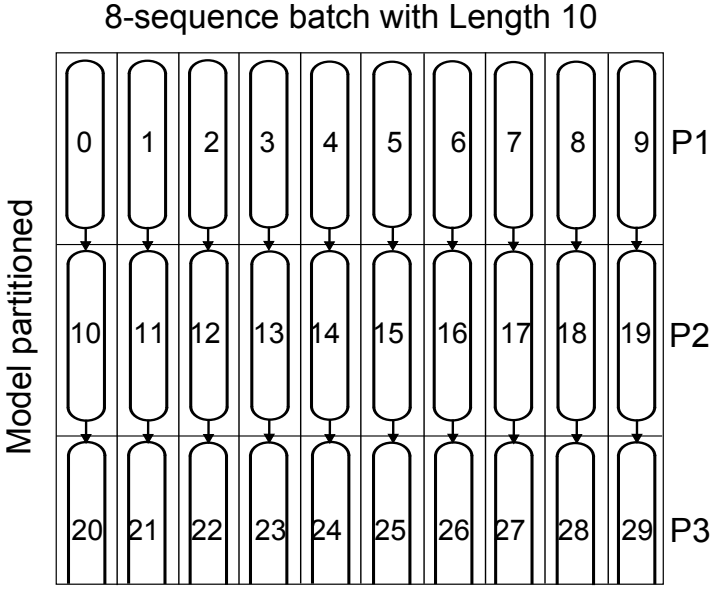


Figure 4.4: Processing pattern of the adopted partitioned model,with an 8-sequence batch of length 10. The numbers represent the processing order of each partition. The arrows show the inter-partition dependencies.

The partitions used must necessarily have a length multiple of 8, to accommodate the loop step of 8 unrolled states. As a result, the model must be expanding up to a length also multiple of 8, by padding with dummy states and scores. The dummy scores are all set to -infinity so that they do not affect the result. The extra states add a slight performance penalty of extra computation.

4.5.4 Determining the Optimal Empirical Partition Length

Overall, the partitioned COPS implementation has an expected memory footprint of around $240 * M + 900$ bytes (corresponding to the original memory requirements of the non-partitioned COPS, plus the additional arrays that are required to store the inter-partition dependencies). It can thereby be estimated the value for the maximum partition length (MP) as the maximum model length that limits the memory footprint to the size of the L1D. Hence the value of MP can be determined by following the formula: $MP = \frac{L1Dsize - 900}{240}$.

Apart from a slight skew towards smaller lengths, justified by the sharing of the L1D cache with other variables not correlated with this processing loop, these estimated MP values coincide very consistently with the best partition lengths that were experimentically observed:

- 112 to 120 states, for 32KB L1D CPUs, (e.g. Intel Core, Core2, Nehalem, Sandy Bridge, Ivy Bridge and Haswell);
- around 48 states, for 16KB L1D CPUs, (e.g. AMD's Opteron Bulldozer and Piledriver);
- 216 to 224 states, for 64KB L1D CPUs, (e.g. AMD's Opteron K8, K10, Phenom I and II).

4.5.5 Evaluation after Partitioning

After partitioning, the overall performance behaved remarkably as expected, maintaining the same level of caches misses and computation speed for any model length:

As a result, COPS managed to be slightly faster than HMMER's ViterbiFilter for models up to $\sim$1200, after which the COPS program quickly gains a close to 2-fold speedup over ViterbiFilter, due to the latter's cache degradation.

Compared to the non-partitioned COPS code, the partitioned version was about 42% faster for long models ($\sim$1000) *before refactoring the inner loop code*. After refactoring the auxiliary storage arrays (Mpv, Ipv, Dpv), the improvement was of 49%. The complete pseudo-code of this final version is presented in Listing A.6 and Listing A.7.

4.6 Batches of Sequences with varying lengths

Since the program runs with 8 parallel sequences in lock-step, there is a problem when the sequences have different lengths. The program must then either ignore the shorter sequences and continue until all sequences are completed (the static approach), or feed in new sequences to replace the finished ones, while the algorithm is running (the dynamic approach used by Rognes).

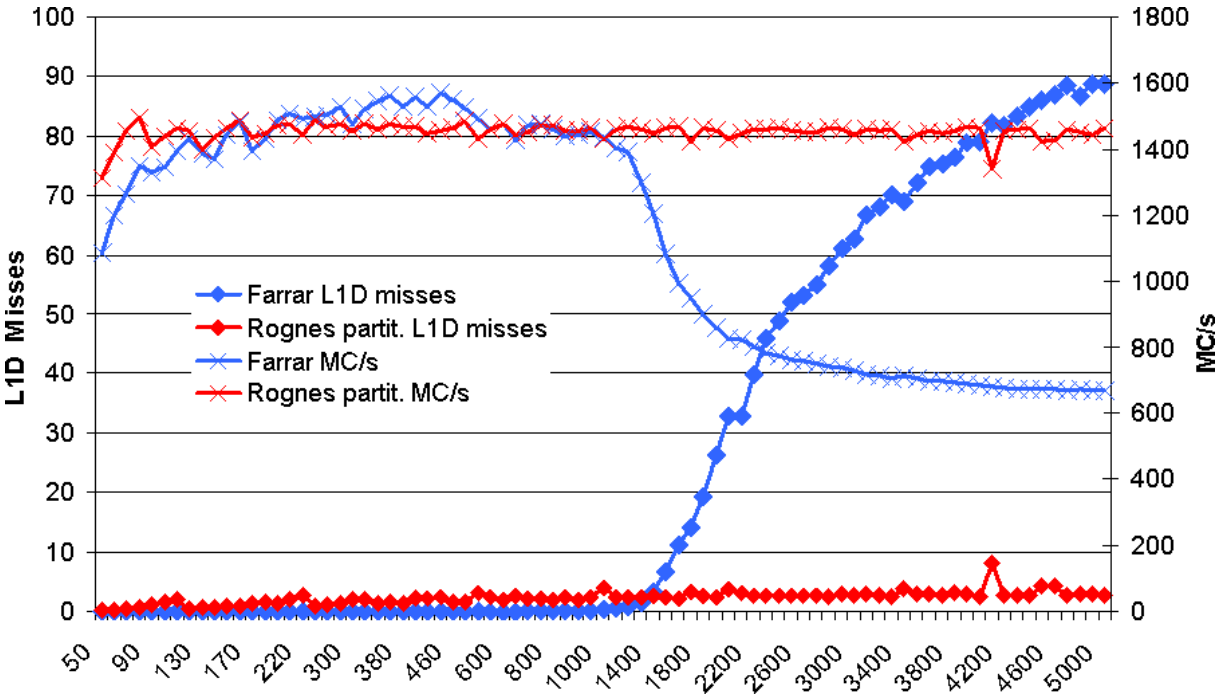


Figure 4.5: Results of cache profiling of HMMER's Farrar-based ViterbiFilter, and the new partitioned COPS. Measured in millions of cells computed per second (MC/s) and L1 data cache misses.

The two possible approaches were implemented in this work, and their performance compared:

- **Static approach: restarting the execution every time to load new sequences**

  The sequences are padded with dummy valid symbols (e.g.; any residue) up to the length of the longest sequences, for each new group of sequences processed. The dummy residue maps to a score array of $-\infty$ scores. These emission scores cancel the updates of vE by the Match states M when they are added to the computation of M.

  In the 32-bit floating point implementation, additional concerns are required: the C $\rightarrow$ C transition cannot be eliminated with dummy scores. It was necessary to mark where each sequence terminates (i.e. its length), compare the limit length in each iteration, and then, by numeric manipulation, nullify the C $\rightarrow$ C transition for iterations beyond the limit length.

  The wasted computation incurred by the dummy symbols was evaluated with a practical-use database (NRDB90) and found to be minimal: it accounted for less than 0.01% of the total computing effort.

- **Dynamic approach: runtime swapping of sequences**

  The Dynamic swapping method was chosen by Rognes for his Swipe tool. In this approach, the sequences are checked in each iteration of the outer loop (loop over sequence symbols) to determine if any has reached its end. Those that have, are exchanged with the next

database sequence, and the appropriate SSE vector elements in the auxiliary data arrays are reset to $-\infty$.

The performance of both methods was evaluated with both a randomized dataset of multiple-length sequences, and the NRDB90, whose sequences have more disparate lengths. On both tests, the Static approach proved to be about $\sim$60% faster than the dynamic approach.

Furthermore, the Rognes dynamic method has a few limitations, which inhibits its use in the present work:

- The sequence loop must be the outer-most loop, and by consequence it would not be possible to employ the Model partitioning optimization.
- The parametrization of the model targeted to each particular sequence Length would equally prove impossible to conduct, since the entire database must be evaluated in a single execution run, without the possibility of re-starts and re-parameterizations.

Even if the swapping method were more efficient, these two limitations are enough to seal the decision of instead using the re-starting method in COPS.

However, with the static re-starting method, there is also an additional source of non-negligible scoring errors in the mis-parametrization of the sequences in the group. Before each execution of the algorithm, the model is reconfigured with the average sequence Length value among the 8 sequences. In particular, the re-configuration computes the transition scores of the special states. Since 8 sequences are processed in lock-step, if their lengths differ, the shorter sequences will yield slightly biased scores. This error can be minimized by first sorting the database sequences, which for practical large databases results in an uniform length for most 8-sequence batches. The re-configuration of the model is a very short step, and does not cause any measurable performance penalty.

The sorting step was done by the standard C implementation of the Quicksort algorithm. Its runtime cost was measured, and determined to be negligble in the context of the application. It took less than 0.01% of the overall runtime, even when sorting large databases with either many small sequences, or fewer but larger sequences.

4.7 Multi-threading the partitions

After having a partitioned model, each partition can be seen as a 'chunk' of data to process. This data layout seems particularly suitable for an additional level of paralellization: multi-threading using a wave-front model of partitioned chunks. From the start, the parallel speedup over the number of threads would expectedly never reach linear growth (mainly due to the unavoidable synchronization and communication between threads). Still, this second parallelization level is an interesting improvement, which could be applied to other areas (e.g.; single tasks that cannot be linearly decomposed into independent threads).

With a partitioned model, it is possible to add Symmetric multiprocessing (SMP) multi-threading to parallelize the partitioned chunks. Some number, N, of partitions are simultaneously processed by N threads, following a wave-front pattern: each thread starts its assigned partition of the i-th sequence residue when the previous thread has finished the previous partition of the same i-th sequence residue (Figure 4.6). The computing pattern is thus 1) left-to-right (ascending by sequence index), and 2) top-to-down (ascending by model state index).

The main-master thread has some special tasks that distinguish it from the worker-slave threads: it is the only thread responsible for loading the sequences, resetting the control and synchronization flags, and re-configuring the model.

There are two synchronization concerns involved in the multi-threaded wave-front strategy: synchronization of all threads at the start of the algorithm (since it is run multiple times using the same N threads), and synchronization of data between threads processing inter-dependent chunks. These two concerns will be explained in detail in the following subsections.

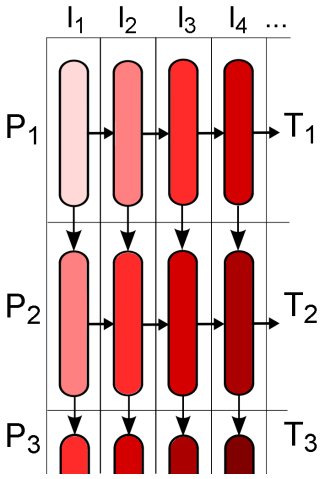


Figure 4.6: Multi-threaded Wave-front pattern. Blocks are processed concurrently in a diagonal pattern (same color for concurrent partitions).

4.7.1 Execution call synchronization

The first level of synchronization concerns the sequentiality of calls to the algorithm, each call running with the same threadpool, requiring synchronization of all threads before starting each new execution of the algorithm. The synchronization here is essential to allow time for the main thread to do its preparation work between runs. On the other hand, it is desirable that the synchronization be as efficient and lightweight as possible.

A few methods were tried, which either did not ensure the proper synchronization, or were not optimally efficient:

- Synchronization barrier from Pthreads library

  This approach consists of the primitive pthread_barrier from the POSIX Pthread library, which acts on a 'barrier' variable initialized with the desired number of threads. When the method 'Wait' is called by a thread, it is blocked by in the barrier until all threads have called the same method. This primitive achieves the synchronization requirement of this project, but it is slower than other methods.

- Per-thread semaphores

  The semaphores are initialized with a value o 0. To start each run, the main thread signals each worker thread that it can start, using the *Post* method. Each worker thread calls the

Listing 4.1 Pseudo-code of synchronization method with Semaphores

```
▷ Declaration:
Semaphore synchSems[NTHREADS]
▷ Initialization:
for i ← 0 to Nthreads do
    SemaphoreInit(synchSems[i], 0)
end for
▷ Worker:
SemaphoreWait(synchSemaphores[myThreadId])
▷ Main:
for i ← 0 to Nthreads do
    SemaphorePost(synchSemaphores[i])
end for
```

Wait method on its corresponding semaphore. This method will either block and wait if the main thread hadn't signaled it yet, or consume one value and proceed if otherwise. The workers that are waiting on the semaphores are woken up by the main thread signals.

The semaphores technique works but it is again slower than other methods.

- Per-thread start flags

  This method uses an array of simple synchronization flags (integers), one per thread, wherein each thread may wait for a command from the main thread.

Listing 4.2 Pseudo-code of synchronization method with Per-thread start flags

```
▷ Worker:
while syncFlags[myThreadId] is False do
    YieldCPU()
end while
▷ after leaving the loop, and seeing a True flag, reset it to False:
syncFlags[myThreadId] = False
▷ Main:
for i ← 0 to Nthreads do
    syncFlags[i] = True
end for
```

Again, this is a functional and lightweight method, but it was deferred in exchange for a more slightly better approach.

In the end, the chosen method was a variation of the Per-thread start flags, that uses a single global execution counter and per-thread local counters:

Listing 4.3 Pseudo-code of synchronization method with Per-thread local counters

```
▷ Main:
Inc globalExecCounter
▷ Worker:
myExecCounter = 0
while True do
    Inc myExecCounter
    while globalExecCounter ≠ myExecCounter do
        YieldCPU()
    end while
    Call DoWork()
end while
```

The worker waits (yielding the CPU) until its local replica counter matches the value of the global counter, whose value is only set by the main thread. There are no data consistency concerns here, simply because outdated values of the global counter will not cause undesired behavior, such as workers' early starts. The only problematic effect of out-of-date values is possibly additional wait cycles for the workers.

Two simple benchmarks were used to evaluate the relative performance of each synchronization method: a first benchmark (Figure 4.7) consisting of a small model and small generated random sequences (∼100 residues each); and a second benchmark (Figure 4.8) that stressed the synchronization primitives alone, with no computation whatsoever.

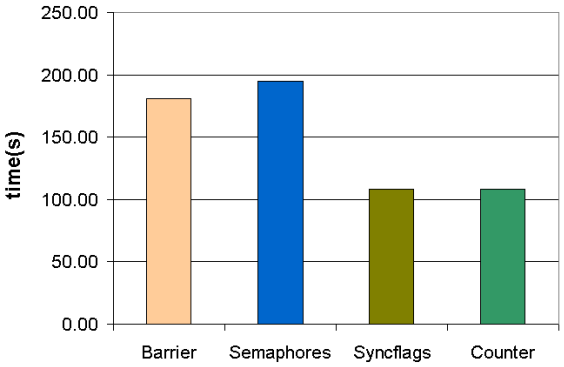


Figure 4.7: First benchmark: COPS code with a 100-length model, 4 threads, and 800K 100-length sequences.

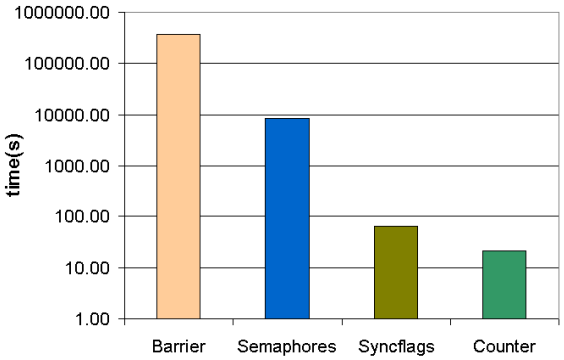


Figure 4.8: Second benchmark: only synchronization primitives, in $80K \times 80K$ rounds. Logarithmic scale.

These results show that the Counter method is indeed the most efficient solution. Note the logarithmic scale on the second benchmark, clearly illustrating the disproportionate efficiency of the no-locks methods vis-a-vis the locking approaches.

Finally, it was evaluated the overall percentage of idle thread time incurred by the chosen method (global Counter + local counters). A battery of tests was run, starting from sequences

with a large amount of wasted computation lines due to partitioning, and down to sequences with no wasted computation. The determined result was that the total idle thread, cumulative of the 4 threads used, ranged from ¡ 1% to a maximum of 5% in the worst case, with an average between 0% and 2%. These values are reasonable and quite positive for the average case.

4.7.2 Partition synchronization

The second level of synchronization concerns the exchange of data between threads processing adjacent partitions. Namely, each thread must send data to the thread processing the succeeding partition, independently for each sequence index. Moreover, each thread must block itself while it is waiting for data from the preceding partition, and this procedure repeats for each and every sequence index. The synchronization overhead is reduced by the decomposition by sequences indices, resulting in relatively small work-chunks of size $PartitionLength \times 1$.

To implement this synchronization requirement, it was chosen a simple and efficient solution:

- $Nthreads$ arrays of flags, one for every pair (partition, sequence symbol), wherein threads can wait and signal one another. The wait is implemented by yielding the CPU and the signaling is done by setting the flag.
- Unique arrays (with length equal to the sequence length) for the data that needs to be exchanged. Only one array is needed for each type of data, because only one transfer of data can occur at any given time for each sequence symbol (i.e.; the data is propagated only one thread at a time, every time the active thread finishes and passes its data to the next partition thread). Only one thread is active processing a sequence index at a given time, any other thread is either in a previous index or waiting on the preceding partition thread.

The chosen synchronization technique has the precious benefit of not using any locks or blocking primitives. No memory barriers are needed, because stale data is not a problem for the program. Data that is not updated will cause no consistency problems, rather it will merely lead to more waiting (yielding) cycles. The flags are only reset by the master thread, before starting the algorithm.

4.7.3 Computing the Practical Threaded Partition Length

A relevant issue to consider is what partition length should be used in each case? This practical partition length differs from the theoretical optimal length by taking into account the computation effort that is wasted by idle threads when the partitions are not evenly divisible by the threads.

The optimal length depends on a few variables: model size, number of threads, and optimal partition length (which is used as a maximum threshold for the practical length), with the stated goal of finding a partition length that maximizes the effective thread use.

1. It is computed the number of minimum (i.e. assuming the threshold length) total chunks required to cover the complete model, and that are a multiple of the number of threads: $MinNchunks = roundtop(\frac{M}{MaxPartLength}, Nthreads)$
2. This minimum number of chunks is used to compute the practical partition length (which has to be a multiple of the SSE vector size, 8): $PartitionLength = roundtop(\frac{M}{MinNchunks}, 8)$
3. The actual number of partitions is finally calculated based on the partition length chosen: $Npartitions = ceiling(\frac{M}{PartitionLength})$

The partitions are then issued to the available threads, using static decomposition and a fixed pinning scheme that ensures the last partition belongs to the master thread (which has the last ThreadId). This is important because it is the master that deals with the algorithm terminations, and extracts and computes the final result.

The following formula was used to determine the start partition of each thread: $StartPartition = (Npartitions + myThreadId) \mod Nthreads$

5

Evaluation

Contents

In this chapter, it will be evaluated the performance gain from each of the main contributions of this work, culminating with the complete multi-threaded version. The results obtained from each benchmark against the relevant comparable software, HMMER's Farrar-based ViterbiFilter vectorization, will be presented and discussed.

5.1 Evaluation Methodology

This section will describe the evaluation methodology and configuration used for the various tests, as well as the features being evaluated.

5.1.1 Benchmark implementations

As previously described in Section 3.2.4, HMMER uses a multi-level processing pipeline to conduct most sequence homology search tasks. This pipeline is composed by a few filtering levels, each more expensive and fine-grained than the previous one. The current pipeline (HMMER version 3.1b1) has the structure presented earlier in Figure 3.3.

The COPS implementation developed in this thesis targets the second level, as an improvement on the Viterbi Decoding filter of the pipeline. However, the whole pipeline processes the sequences sequentially, one by one, and not in parallel. The multi-threaded parallelization of in HMMER is done by running one different pipeline in each thread. The pipeline itself supports only one thread processing a single sequence. This pipeline architecture is thus incompatible with both the Rognes method (inter-task vectorization of N concurrent sequences) and the multi-threading developed in COPS (threading of partitions in a wave-front decomposition).

Given this incompatibility, it was impossible to integrate the developed COPS code into the HMMER pipeline. Instead, the code was packaged into a small stand-alone tool, built on top the HMMER suite. It uses the HMMER libraries and models, and serves as a proof-of-concept for the developed techniques and architecture.

In order to evaluate the overall performance of COPS vis-a-vis the HMMER Viterbi Decoding implementation (i.e., the ViterbiFilter program), a small multi-threaded test tool was created for ViterbiFilter. The benchmark runs with multiple-threads to be fully comparable to the multi-threaded COPS. The used threadpool was implemented efficiently with mutexes and a single synchronized sequence counter.

Besides HMMER's ViterbiFilter, the benchmarks were also run against a serial implementation of Viterbi, optimized for the alignment mode used (Unihit Local alignment).

5.1.2 Evaluation Dataset

To evaluate the developed application, an extensive and thorough range of benchmarks was carried out. The evaluation dataset consists of:

- HMMs sampled from the Dfam database of Homo Sapiens DNA ([4]), with model lengths from 60 to 3000, increasing by a step of roughly 100 model states each time.

  Dfam is a widely used database of protein families, and HMMss constructed to model them. The latest release of Dfam, as of March 2013, uses HMMER3.1b1 to create the Hidden Markov models. The complete list of chosen HMMs from Dfam is presented below (their length is prefixed to the model name):

M0063-U7	M0700-MER77B	M1409-MLT1H-int	M2204-CR1_Mam
M0101-HY3	M0804-LTR1E	M1509-LTR104_Mam	M2334-L1M2c_5end
M0200-MER107	M0900-MER4D1	M1597-Tigger6b	M2434-L1MCa_5end
M0301-Eulor9A	M1000-L1MEg2_5end	M1727-L1P3_5end	M2532-L1MC3_3end
M0401-MER121	M1106-L1MD2_3end	M1817-REP522	M2629-L1MC4a_3end
M0500-LTR72B	M1204-Charlie17b	M1961-Charlie4	M2731-Tigger4
M0600-MER4A1	M1302-HSMAR2	M2101-L1MEg_5end	M2858-Charlie12
			M2991-HAL1M8

- DNA databases: sequenced genomes of *Homo Sapiens* (Human) and *Macaca Fascicularis* (Crab-eating Macaque), retrieved from the NCBI archive.

A wide range of testing configurations were sampled and tried out, to find the best parameters. These tests were mainly described throughout the previous chapter, for instance, the empirical tests to determine the optimal partition length for each target architecture. This extensive testing experience was crucial to tune the parameters for each task requirements, and the overall optimization of the used techniques and implementations. All timings were measured in total walltime, using the Linux *ftime* function.

Table 5.1: Hardware details of the machines used for the evaluation benchmarks. The No. of cores refers to a single NUMA node

Spec . Arch.	AMD Bulldozer	Xeon Nehalem	i7 Sandy Bridge
CPU model	Opteron 6276	Xeon E7-4830	i7-3930K
Frequency	3 GHz	2.13 GHz	3.20 GHz
No. cores	8	8	6
L1D size	16K	32K	32K
L1I size	64K	32K	32K
L2 size	2048K	256K	256K
L3 size	6144K	24576K	12288K

5.1.3 Evaluation architectures

The benchmarks were run on three different architectures: AMD Opteron Bulldozer, Intel Xeon Nehalem, and Intel core i7 Sandy Bridge. Table 5.1 presents the detailed characteristics of the hardware used.

5.2 Results

In this section, the benchmark results obtained in the various hardware setups are presented. Each following subsection will focus one of the versions of the developed tool, after each of the main improvements:

- Inter-task vectorization of the Viterbi algorithm
- Improved method of loading the Emission scores
- Partitioning to Model
- Wave-front multi-threading of the model partitions

The benchmarks were all run against the comparable HMMER's vectorized ViterbiFilter. The computing speeds, measured in millions of cells updates per second (a 'cell' corresponds to a state-triplet in HMMs), and the measured speedup against HMMER, will be presented. The serial implementation consistently maintained a constant performance regardless of the model length, so the speedup of COPS against it varied only with the performance of COPS.

5.2.1 Results for the Initial Rognes-based Inter-task vectorization

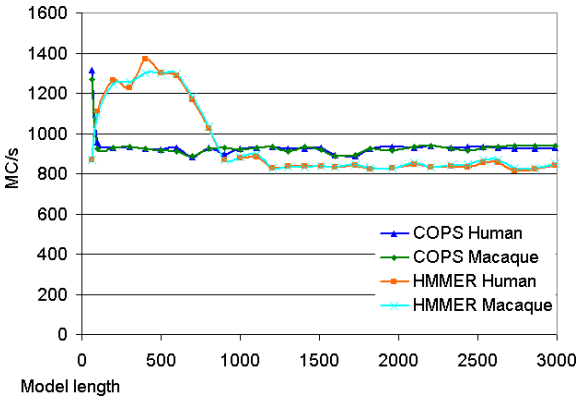


Figure 5.1: Speeds of the initial COPS and HMMER, with the Human and Macaque genomes, on an AMD Opteron Bulldozer.

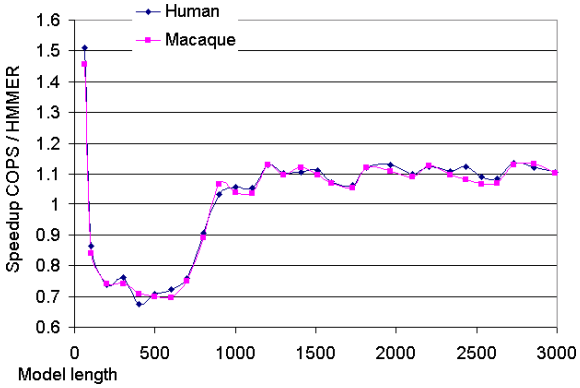


Figure 5.2: Speedups of the initial COPS vs HMMER, with the Human and Macaque genomes, on an AMD Opteron Bulldozer.

In this section, it is presented the results for the initial approach, based on Rognes' work, whose implementation was described in Section 4.2.

These results show that the original Rognes strategy is not able to surpass the performance of HMMER's ViterbiFilter, except in the case of the smaller models (lengths 60 and 100). From a

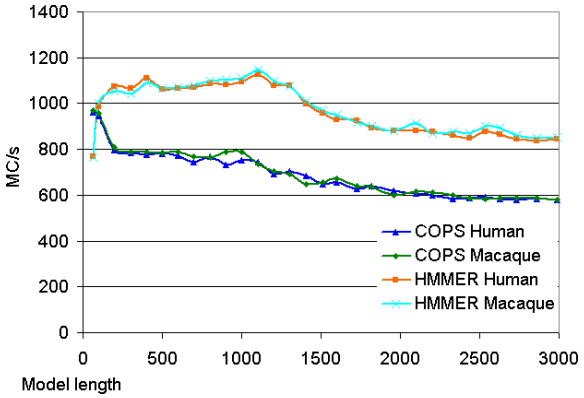


Figure 5.3: Speeds of the initial COPS and HMMER, with the Human and Macaque genomes, on an Intel Xeon Nehalem.

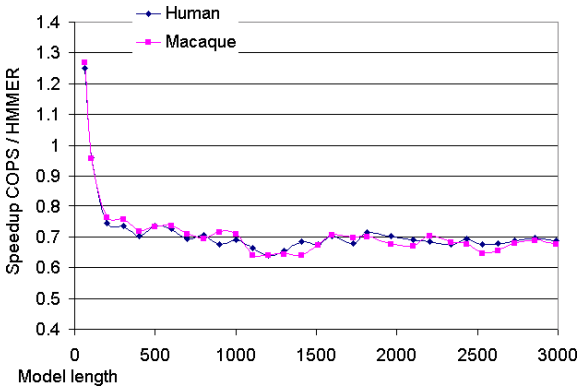


Figure 5.4: Speedups of the initial COPS vs HMMER, with the Human and Macaque genomes, on an Intel Xeon Nehalem.

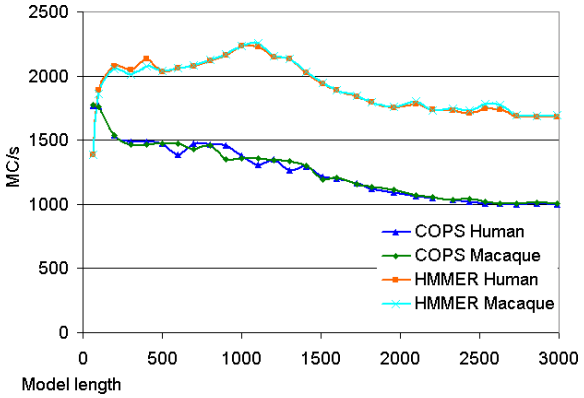


Figure 5.5: Speeds of the initial COPS and HMMER, with the Human and Macaque genomes, on an Intel i7 Sandy Bridge.

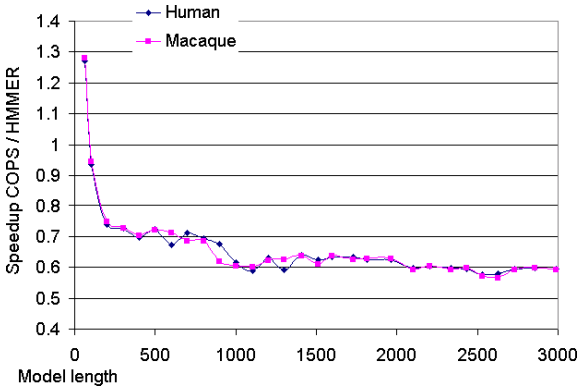


Figure 5.6: Speedups of the initial COPS vs HMMER, with the Human and Macaque genomes, on an Intel i7 Sandy Bridge.

length-100 model to a lengh-200, the performance of COPS drops drastically, as was discussed in Section 4.5.1.

On the Intel Sandy Bridge a substantial performance drop is also noticeable for larger models ($>$ length-1000). This second drop can also be explained by cache limitations, as it will disappear after normalizing the inner loop maximum memory use (see the later results of Section 5.2.3).

In very small models, length $<$ 80 bps, the performance of HMMER's striped version is particularly poor. This poor performance was equally observed in the Smith-Waterman algorithm. It is caused by the substantial overhead of the Farrar's Lazy-F loop, which is run more frequently (relative to the size of the core computation load) when using very small models. The best results of Rognes' Smith-Waterman program against Farrar's were also found in these smaller lengths (for sequence queries).

5.2.2 Results for the Inline loading of the Emission scores

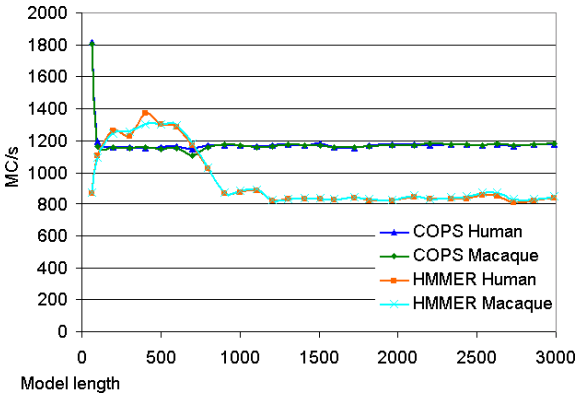


Figure 5.7: Speeds of COPS with Inlined Loading and HMMER, Human and Macaque genomes, on an AMD Opteron Bulldozer.

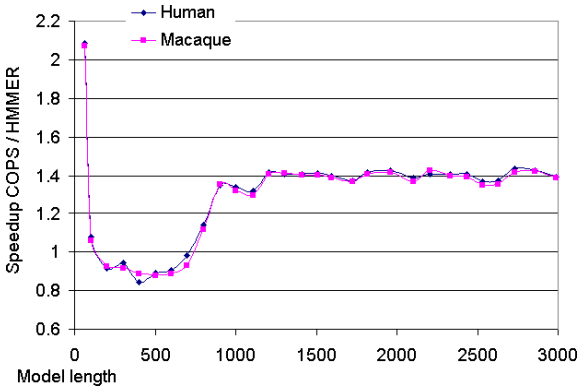


Figure 5.8: Speedups of COPS with Inlined Loading vs HMMER, Human and Macaque genomes, on an AMD Opteron Bulldozer.

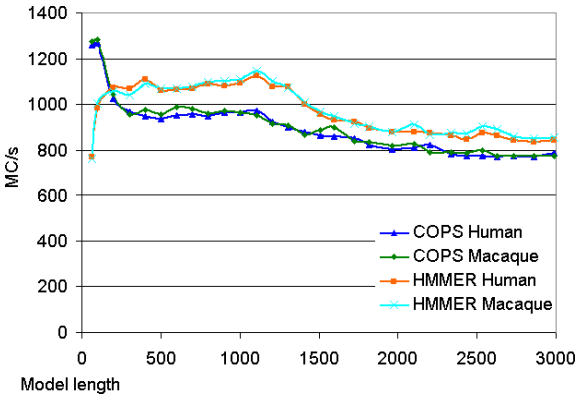


Figure 5.9: Speeds of COPS with Inlined Loading and HMMER, Human and Macaque genomes, on an Intel Xeon Nehalem.

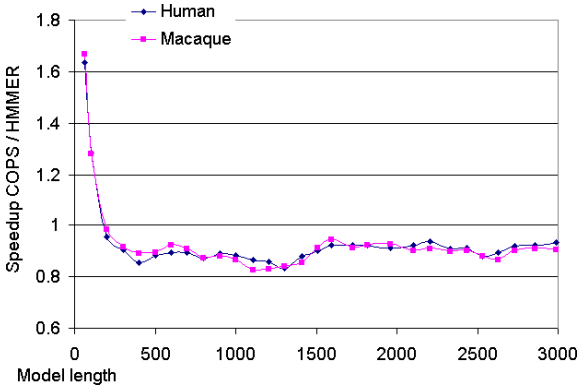


Figure 5.10: Speedups of COPS with Inlined Loading vs HMMER, Human and Macaque genomes, on an Intel Xeon Nehalem.

In this section, Figures 5.7 to 5.12 present the results for the COPS version with the improvement of the loading and arrangement of Emission scores, by inlining the unpack operations in the inner loop. The development of this alternative Inline method was described in Section 4.3.

With this optimization, there is a general substantial performance boost, between 30% and 40% depending on the test machine (see Figures 5.13, 5.14 and 5.15). After this improvement, our COPS tool was able to beat HMMER on some machines (i.e. the AMD Opteron). Still, the performance dependency on model length remained mostly the same. The poor performance of HMMER in the smaller models, and the consequent higher speedup of COPS vs HMMER, remains also unchanged.

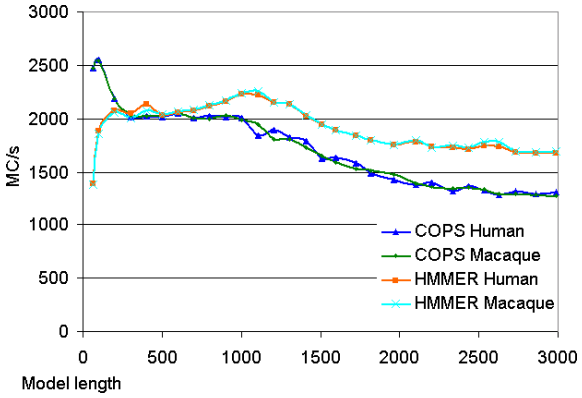


Figure 5.11: Speeds of COPS with Inlined Loading and HMMER, Human and Macaque genomes, on an Intel i7 Sandy Bridge.

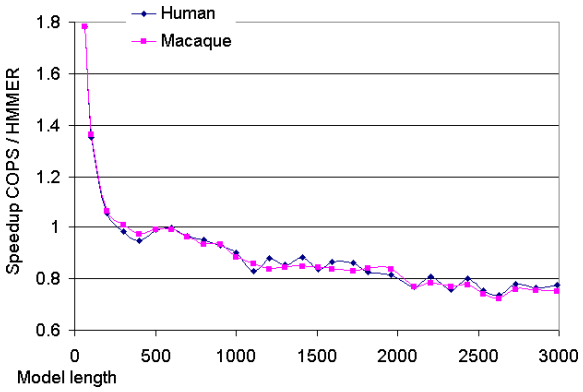


Figure 5.12: Speedups of COPS with Inlined Loading vs HMMER, Human and Macaque genomes, on an Intel i7 Sandy Bridge.

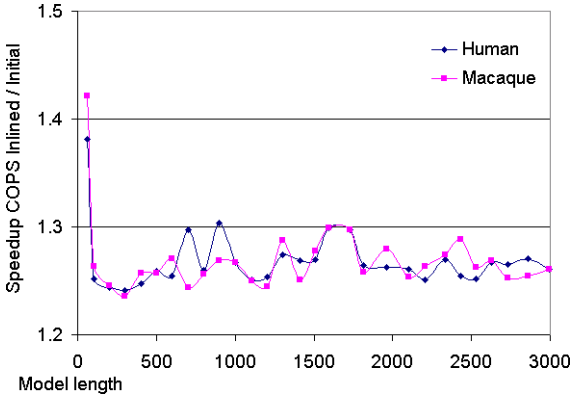


Figure 5.13: Speedups of COPS with Inlined Loading vs Initial COPS, Human and Macaque genomes, on an Intel i7 Sandy Bridge.

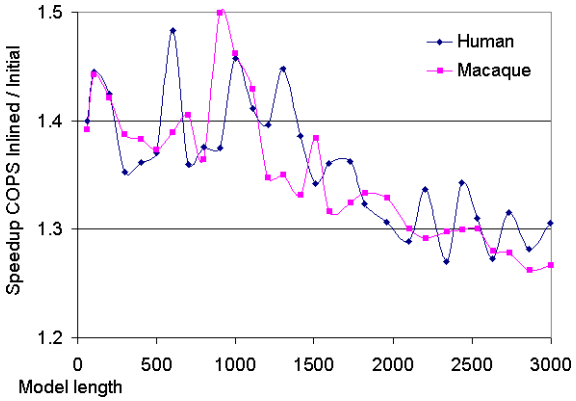


Figure 5.14: Speeds of COPS with Inlined Loading vs Initial COPS, Human and Macaque genomes, on an Intel i7 Sandy Bridge.

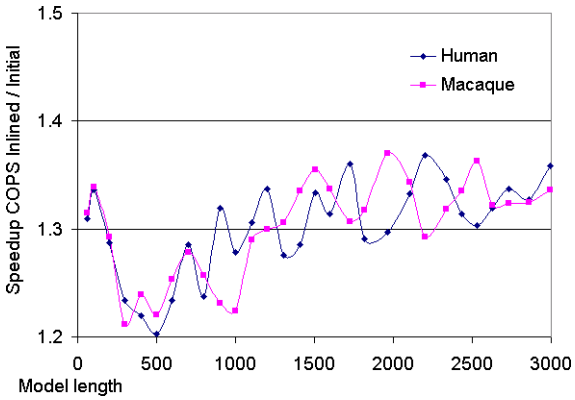


Figure 5.15: Speeds of COPS with Inlined Loading vs Initial COPS, Human and Macaque genomes, on an Intel i7 Sandy Bridge.

5.2.3 Results for the Model Partitioning

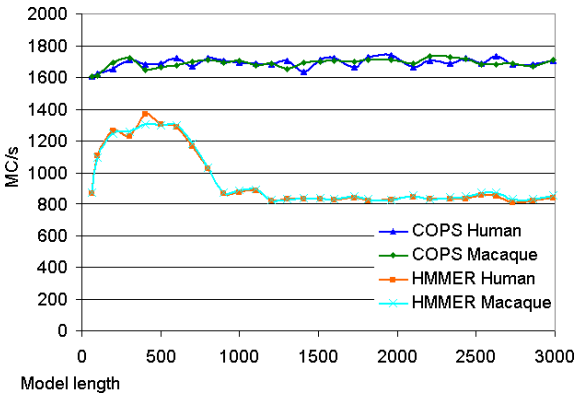


Figure 5.16: Speeds of COPS after Partitioning and HMMER, with the Human and Macaque genomes, on an AMD Opteron Bulldozer.

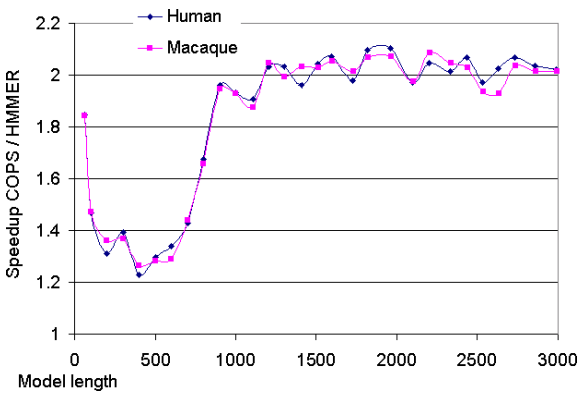


Figure 5.17: Speedups of COPS after Partitioning vs HMMER, with the Human and Macaque genomes, on an AMD Opteron Bulldozer.

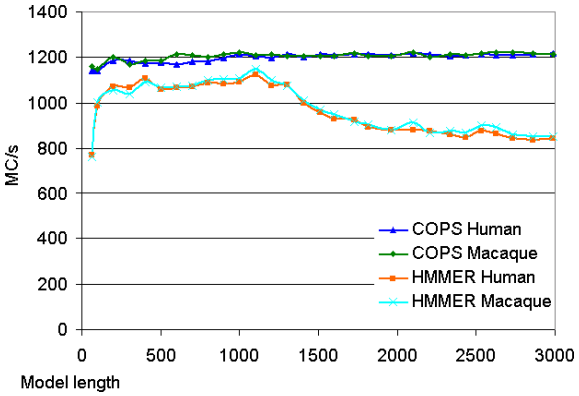


Figure 5.18: Speeds of COPS after Partitioning and HMMER, with the Human and Macaque genomes, on an Intel Xeon Nehalem.

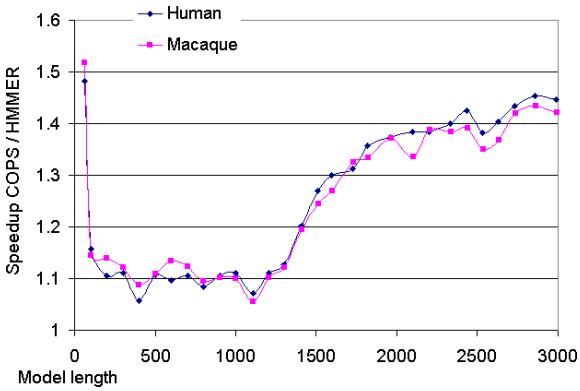


Figure 5.19: Speedups of COPS after Partitioning vs HMMER, with the Human and Macaque genomes, on an Intel Xeon Nehalem.

These are the results for the COPS version after partitioning the HMM model, which has been described in Section 4.5. This evaluated version includes the inline loading of Emission scores.

With model partitioning, the L1 cache size limitation is overcome, and therefore the performance drop from length-100 models to 200-length models, that was evident in the earlier results, disappears. The performance of this version of COPS remains constant with increasing model lengths, in all test machines.

In particular, this improvement is observed equally in Intel's 32KB L1D cache machines, and in AMD's Opterons with a 16KB L1D cache (see Figure 5.16). The optimal partition length was specifically tuned for each test machine (namely, a length of 112-120 for Intel's 32KB caches, and 48 for AMD's 16KB caches). For larger models (length $>$ 1000), COPS is able to beat HMMER on all machines, achieving a speedup ranging from 1.5x on the Intel machines, to 2x on the AMD.

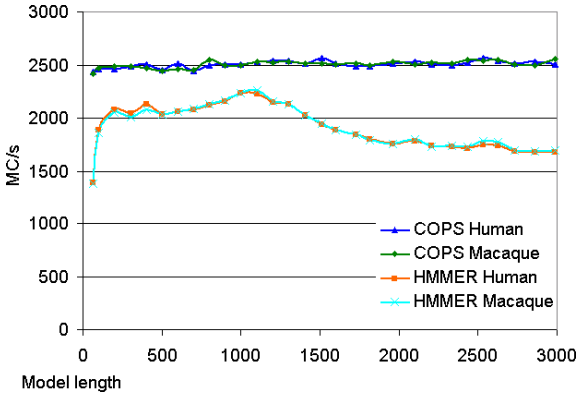


Figure 5.20: Speeds of COPS after Partitioning and HMMER, with the Human and Macaque genomes, on an Intel i7 Sandy Bridge.

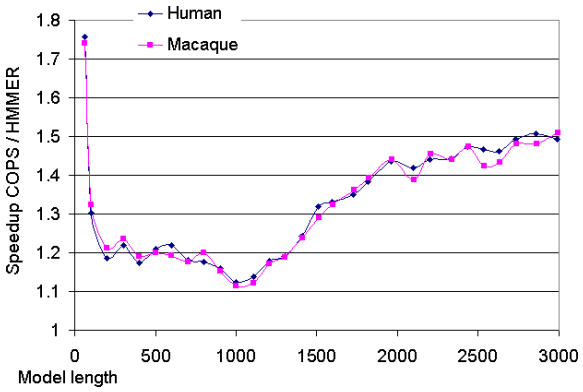


Figure 5.21: Speedups of COPS after Partitioning vs HMMER, with the Human and Macaque genomes, on an Intel i7 Sandy Bridge.

It can be seen in the results that the speedup of COPS vs. HMMER still drops considerably from the smaller M-60 model to the next length-100 model. As referenced before, this drop is explained by the poor showing of HMMER's Viterbifilter in the smaller model. HMMER's striped version executes the *Lazy-F* loop more frequently when the sequence stride is shorter, having a shorter available span to nullify the F dependencies (D for HMMs), and this happens when the model is smaller. Therefore the speedup drop vs. HMMER from the length-60 to the length-100 model is not caused by an efficiency decrease of our version, but by an increase in HMMER. The performance of COPS remains mostly constant in these smaller models, as can be seen in the processing speed graphic.

Therefore, for short models ($<$ 100 bps), COPS achieves a considerable 1.7-fold speedup vs HMMER. For medium-length models (between 200 and 500 bps on 16KB-L1D machines, and up to $\sim$1000 on 32KB-L1D machines), HMMER's implementation is competitive against COPS, reducing our speedup to about 1.2-fold. These are the model lengths wherein the striped version does not exceed the size of the innermost data cache.

Still, the speedups for short models have not been as high as the ones reported by Rognes for Smith-Waterman. Many reasons may justify this, such as Rognes' aggressive assembly optimizations, and the unavoidable differences between the Viterbi and Smith-Waterman algorithms. One significant cause in our view is the overall higher memory requirements of Viterbi. A inter-task approach is bound to required significantly more memory that an intra-task vectorization, 8-fold for some of the data.

For longer models, from 500 bps or 1000 bps depending on the L1D size, it can be observed that the performance of HMMER quickly deteriorates as the model's length increases, and the memory requirements of the standard Farrar approach reach the maximum that the innermost caches can provide. In contrast, COPS is able to consistently maintain the same performance with increasingly long models, thus achieving a 2-fold speedup on AMD's, and 1.5-fold on Intel's,

against HMMER's striped implementation.

The optimized serial benchmark managed a meager result of 228 MC/s on the Core i7, 201 MC/s on the Intel Xeon, and 153MC/s on the AMD, for every model. So, the overall speedup over the non-parallelized code is 11x in the Core i7, 8x on the Intel Xeon, and 8.5x on the AMD.

5.2.4 Results for the Wave-Front Multi-threading

The graphics with the results for the multi-threaded COPS are presented in Appendix A. This version uses a multi-threaded wave-front pattern in the partitions (described in Section 4.7). The evaluation was conducted with an increasing number of threads, 1, 2, 4, 6, and 8 for the 8-core machines.

HMMER's ViterbiFilter is run with independent tasks per thread, in a multi-threaded work pool. Therefore, it is strongly expected that it achieves a quasi-linear speedup over the number of threads, since each thread proceeds independent of the others. The only synchronization point is the access to the work queue, to fetch each new sequence. This synchronization cost is very light - it was measured by comparing against a static work decomposition with same-length sequences (which does not require synchronization primitives), and found to be practically unnoticeable.

The experimental results show that HMMER's Intra-task implementation is able to maintain a linear speedup over the number of threads/scores. This is was the expected outcome, since it is processing each alignment task independently, and there are no inter-thread dependencies (except for the shared work pool).

In the case of COPS, the results show a clear sub-linear speedup, with a considerable drop in the added speed per core, as the number of cores increase. This was also an expected outcome, since there is substantial communication between the threads processing the partitions. Still, other causes may account for some of the lost processing speed. These will be discussed in Section 5.2.5.1. But first, in the next section, the program will be analyzed through Karp-Flatt metric, estimating the serial fraction of the developed program, i.e., the non-parallelized fraction.

5.2.5 Evaluation of the experimental serial fraction by Karp-Flatt's Metric

Karp and Flatt [27] proposed a metric to measure the portion of a program that is inherently sequential and cannot (or has not) been parallelized - the Experimentally-determined serial fraction (e). The metric estimates the value of e by the following formula:

$$e = \frac{\frac{1}{Speedup} - \frac{1}{p}}{1 - \frac{1}{p}}$$

The values derived with this metric from the tests conducted are shown in the graphics below.

In general, the serial fraction consistently decreases with the number of threads, in all models, which substantiate the assumption that most of the performance loss is due to the parallel over-

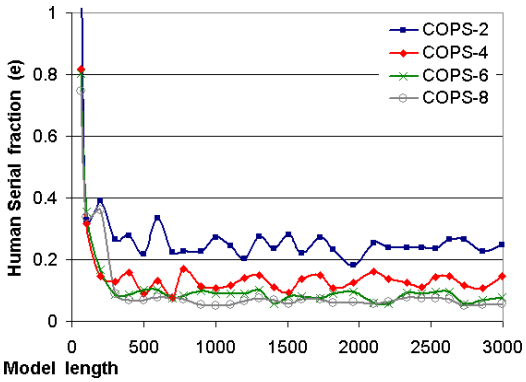


Figure 5.22: Experimental serial fraction using Karp-Flatt's metric, for the AMD Opteron Bulldozer tests with Human DNA.

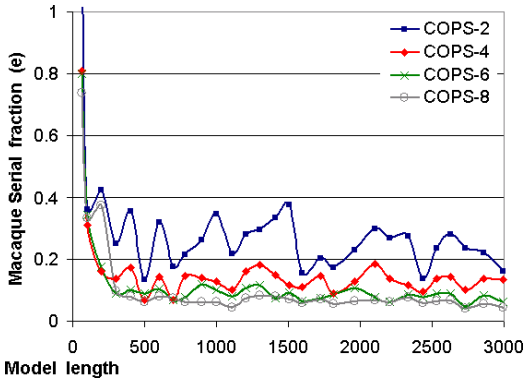


Figure 5.23: Experimental serial fraction using Karp-Flatt's metric, for the AMD Opteron Bulldozer tests with Macaque DNA.

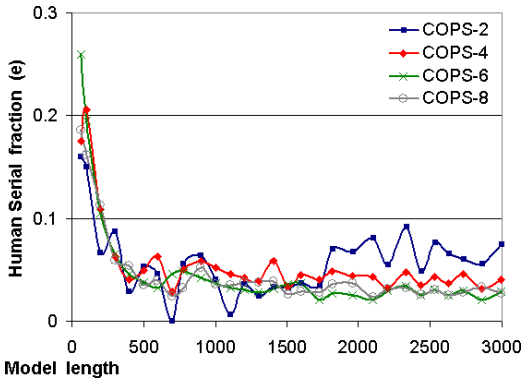


Figure 5.24: Experimental serial fraction using Karp-Flatt's metric, for the Intel Xeon Nehalem tests with Human DNA.

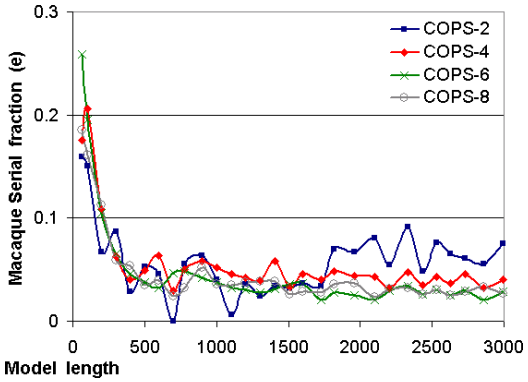


Figure 5.25: Experimental serial fraction using Karp-Flatt's metric, for the Intel Xeon Nehalem tests with Macaque DNA.

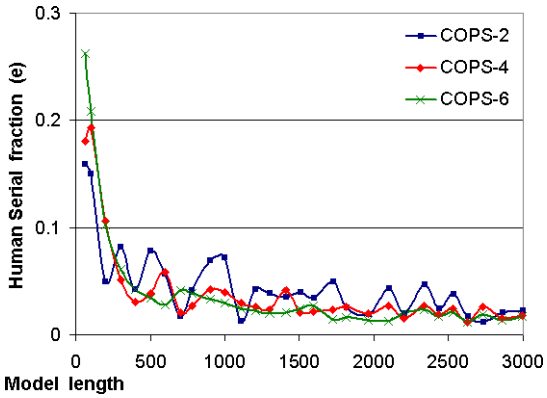


Figure 5.26: Experimental serial fraction using Karp-Flatt's metric, for the Intel i7 Sandy Bridge tests with Human DNA.

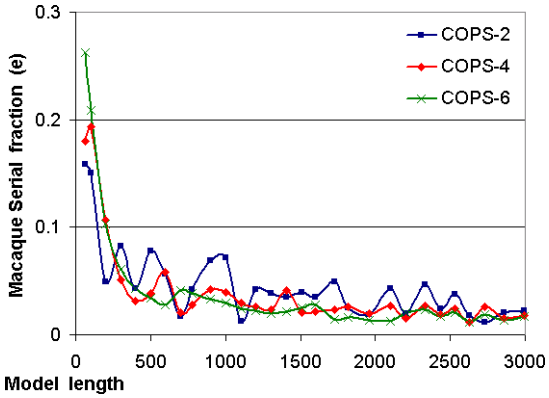


Figure 5.27: Experimental serial fraction using Karp-Flatt's metric, for the Intel i7 Sandy Bridge tests with Macaque DNA.

head, instead of non-parallelized code. However, there is one exception: in the smaller models (length $<= 200$), the serial fraction is both considerable higher (indicating a worse parallelization), and it changes little with the number of threads. This result suggests that there is a substantial

non-parallelized computation load in these models. That can be explained by the difficulty in finding an optimal work division between partitions and threads, as will be further discussed in the next section.

Another interesting point of note are the wide variations in the serial fraction metric along model lengths, reflecting an efficiency of multi-threading that depends on the specific model length. Some model lengths allow for a perfect work decomposition between partitions and threads, while others do not, and leave some unbalanced workloads. This will also be further discussed in the next section.

The parallelization had a lower speedup per added thread on the AMD machine than it did on the Intel machines. The different internal architectures of the two different processor families is a likely culprit for this anomaly, since it was found that even a set of independent, non-communicating processes became considerably slower when running concurrently. Additionally, the AMD machine was deployed in a NUMA topology, with a distributed shared memory. The configuration of the DSM seriously hindered any concurrent contention for resources, even among independent processes running on the same NUMA node.

5.2.5.1 Limitations of Wave-Front Multi-threading

A multi-threading parallelization of the Model partitions, using a wave-front pattern, suffers from some problems that affect its efficiency, and lead to a sub-linear and decreasing parallel speedup:

- **Communication overhead**

  There are two communication instances between the threads:

  - a synchronization point on the flags' arrays, each one shared by only two threads;
  - a data transfer point on partition barriers, which is only accessed before (for writes) and after (for reads) the synchronization point.

  There is therefore no real contention on the data transfer point, since it is guaranteed to be accessed by only one thread at any given time. The synchronization point suffers some contention from its two owner threads, and the wait cycles when one thread is blocked waiting for the other lead to a non-negligible fraction of thread idle time.

- **Start and end delays of the wave-front pattern**

  The wave-front pattern entails some unavoidable wait cycles in the start and end of the alignment. Each thread must wait that the previous thread finished the previous partition column, and as a result there is a incremental delay in the beginning, wherein the necessary delay of each thread increases by one work chunk (i.e. one partition column) for each added thread. In the ending of the alignment, the threads that started earlier have to wait for the ones lagging behind, with wait periods in the reserve order.

- **Unbalanced work division**

  The perfect work division is an even division of the partitions by the threads, and of the model length by the partition length. However it is not always possible to find such a division, especially when the number of threads is higher. The thread work load will become unbalanced in two cases:

  - When the partitions cannot be evenly distributed by the threads, some thread(s) will get less partitions to process, and will have to wait on the others to finish;
  - When the model length is not evenly divisible by the partition length, the last thread will have less work than the others, and thus will also have to wait. This last unbalanced partition represents a smaller waste than the unbalanced number of partitions, which force thread(s) to wait for the computation period of a whole partition.

- **Very short partition lengths**

  When using partitions that are too small, the various overheads (initialization, termination, synchronization, wave-front delays, unbalancing idle times, etc) are a large fraction of the overall runtime, due to the corresponding core computation load that has been reduced by the short partition length. These partition lengths should be avoided, but for very small models it may not be possible.

The quite noticeable lower speedup per added thread in the smaller models is explained by the uneven load balancing between the threads (some threads have less partitions to compute) and by the heavier overheads when using shorter partition lengths. As more cores are added, the speedup of the wave-front COPS vs HMMER decreases considerably in the smaller models, because the inter-task trivial multi-threading implemented for HMMER's ViterbiFilter does not have these load balancing problems in the test databases (i.e. databases with many sequences).

Conversely, the speedup per added core is higher in the longer models, since they have a better load balancing and a smaller overhead from the longer partitions. Therefore, to effectively exploit the multi-threaded wave-front pattern for a high number of threads (i.e. 8 threads), the application's models should be sufficiently large, in order to reduce the incurred overheads with a close-to-maximum partition length, and achieve an even distribution of partitions among threads.

6

Conclusions

Contents

Sequence analysis is an increasingly important field in bioinformatics. By comparing biological sequences, proteins and nucleic acids, it is possible to unveil previously unknown relations between different organisms and organelles, such as evolutionary links. This comparison process, known as Homology Search, is a burgeoning field, fueled by the exponential growth of biological data to search. Recent sequencing technology, like Illumina, Pyrosequencing, SOLiD, and others, have greatly expended the quantity of available data that must be analyzed. The biological databases most used for homology search, such as UniProt, TrEMBL, Genbank, etc, have also grown considerably.

Nowadays, two main methods are used to conduct homology search: aligning sequences against each other with tailored alignment algorithms, and evaluating the probability of natural occurrence of a sequence through the use of a probabilistic model, such as a Hidden Markov Model. Both methods have been extensively developed and are in widespread use, offering an invaluable service to biologists. Due to the heavy computational cost of processing the large biological databases that exist today, it has been essential to exploit the enormous potential of hardware parallel computation to speedup the lengthy searches.

Chapter 2 of this thesis expanded on the most relevant algorithms for alignment and Markov Models that are in use today for homology search, along with some important improvements and optimizations of the standard algorithms. Special attention was given to the optimal Smith-Waterman algorithm, which is the basis of many optimization efforts; and to the Viterbi and Forward algorithms for HMMs. Algorithms for single alignment and HMMs were compared and contrasted, focusing particularly on their similarities and common points, as Dynamic Programming algorithms.

Chapter 3 surveyed the major parallelization approaches to Dynamic Programming algorithms (like the Smith-Waterman and Viterbi). The general hardware architectures were covered - SIMD in general purpose CPUs and in GPUs, and MIMD in clusters and multi-core processors. An important distinction was made between intra-task parallelism where a single task is parallelized, and inter-task parallelism which combines multiples tasks to facilitate the parallel work division.

Three possible decomposition patterns for intra-task parallelism were studied: diagonal, vertical and striped. The striped pattern, first proposed by Farrar, achieved the best speedup and became very popular in many implementations. It is in particular the approach chosen by HMMER, a widely used HMM search tool. In the case of Inter-task parallelism, it was studied the promising work of Rognes in his 2011 Swipe tool. By implementing inter-sequence parallelism on Intel's SSE registers, he managed to surpass Farrar's speedups, for the Smith-Waterman algorithm.

In this work, the objective was to adapt Rognes inter-task strategy to the Viterbi algorithm for HMMs. In particular, to implement it on HMMER and create an alternative Viterbi version that is competitive and surpasses the existing one, which is based on Farrar's pattern. The goal was to

reach in HMMER close to the same speedups of around 2.5-fold to 1.5-fold that Rognes achieved against Farrar's version. The development of the work was detailed in Chapter 4.

The inter-sequence vectorization of Viterbi on SSE was successfully implemented, initially following the methods used by Rognes for Smith-Waterman. The performance was clearly lacking however, failing to reach the same processing speed as HMMER's version. It was then discovered two main issues hindering the program's efficiency:

1. The loading and arrangement of the per-sequence Emission Scores;
2. An exhaustion of the available capacity in the innermost L1D cache, in the inner loop code, that happened for medium/large models.

The first issue was tackled by moving the loading of scores into the inner loop, and doing it inlined only 8 vector values at a time. With the inline loading, the values can be kept in close memory and avoid the memory re-writings of Rognes. This improvement led to a speed increase between 30% and 40%, shown in the experimental results of subsection 5.2.2.

The second issue, the L1D cache exhaustion, was solved by dividing the HMM into chunks, which were named partitions. The partitions are then processed in a new outer loop. The inner loop is *strip-mined* so that it processes only one partition in each iteration, as a way to avoid filling the L1D cache capacity. The result was a constant processing speed for any model size, thus making the program *Cache-Oblivious*. For medium and large models, it is faster than HMMER's version, which suffers from the cache encumbrance.

The developed tool, COPS, achieved roughly the same performance as HMMER's ViterbiFilter version on medium models up to $\sim$1000 model states for Intel and $\sim$500 for AMD, and after this threshold, it yielded an increasingly higher speedup against ViterbiFilter. With the larger models, in all benchmarks, COPS obtained speedups between 1.5 and 2.0 against HMMER. The speedups against a serial optimized version ranged from 8-fold to 11-fold.

This advantage in larger models derives mainly from the improved utilization of the innermost cache, that resulted from partitioning the model. By tweaking the maximum partition length (from $\sim$120 for 32KB L1D caches, to $\sim$50 for 16KB caches, and $\sim$220 for 64KB caches), it was possible to maintain an efficient optimal use of the available memory in the L1D cache to store the arrays that are frequently accessed in the inner loop. As a result, it was possible to keep a high performance level when running large models, with which HMMER's ViterbiFilter suffers considerably. We can observe that the performance of HMMER's implementation quickly deteriorates as the model's length increases, and the memory requirements of the standard striped approach reach the maximum that the innermost caches can provide.

For very small models ($< \sim$100 bps), Farrar's approach also suffers considerably due to more passes through the Lazy-F loop. This weakness of Farrar was the main reason behind Rognes' original comparative speedup in his Smith-Waterman program. The same comparative advantage

and higher speedup (up to ∼1.9-fold) in small models is evident in COPS.

With this work, we have thus shown that an Inter-task vectorization can effectively improve on Farrar's intra-task striped pattern, and achieve a substantial higher speedup that is independent of the cache size. As such, the developed COPS implementation is a significant improvement on HMMER's implementation.

The comparative speedup obtained against Farrar's approach is higher for very small models, and medium or large models, depending on the machine. Therefore, COPs can be most efficient vis-a-vis a striped pattern in applications that use small models (for instance, most proteins); or large models (such as DNA and RNA analysis, speech and audio recognition, etc).

After developing the single-threaded COPS with the workload divided in partitions, it was explored the avenue of multi-threading parallelization using a wave-front pattern. This consisted in an Intra-task parallelization of each COPS execution, as opposed to an Inter-task trivial parallelization with independent executions. The results were fairly good, as expected, although the parallelization is clearly not scalable for a large number of threads, due to the communication overhead. For a small number of threads however, up to about 8, the obtained speedup through multi-threading is strong, not much lower than a linear speedup (i.e. 6.5-fold speedup for 8 threads). Hence, it has been shown to be an interesting parallelization avenue to employ in some areas. Namely, in the parallelization of very large models that are searched only a few times, and for whom, therefore, the trivial inter-task multi-threading is not very useful due to a lack of independent tasks to divide among threads.

6.1 Main Contributions

The main contribution of this thesis is the Inter-task vectorization of HMM algorithms. This was successfully accomplished, although the resulting performance did not meet expectations. Two other novel strategies were later developed to improve the original Rognes-based solution. Overall, the main contributions are listed below:

- Inter-task vectorization of HMMs algorithms, in commodity CPUs (x86's SSE)
- Improved method of loading the Emission scores (which correspond to the Match scores of the Smith-Waterman algorithm)
- Partitioning to Model to fit the inner loops storage arrays in the innermost cache, thus making the program Cache-oblivious
- Multi-threading the model partitions using a wave-front pattern

6.2 Future work

In the author's view, the most promising areas for future work following this thesis are:

- Adapting the Cache-oblivious model partitioning to Farrar's striped pattern. Although it would surely lead to a performance gain on the larger models, it is expected that it would be lower than the speedup obtained by the Inter-task method. It is my view that the introduction of partitions would make the Lazy-F loop be more frequently executed, which would carry a considerable performance cost. Still, as the partitions would be reasonable large (e.g., $\sim$1000 bps for 32-bit L1D CPUs), the Lazy-F overhead would very likely not nullify the benefits of the reduced L1D cache misses.

- Adopting and studying the application of the COPS approach in other areas and tools which require Hidden Markov Models, specially those that use larger models, such as speech recognition, DNA analysis, etc. These areas could benefit greatly from an optimized Viterbi implementation.

- Extending the SSE vectorization to the newest Intel SIMD instruction set, the 256-bit AVX2, which supports operations on 16-bit integers (which AVX1 did not support). Both the Inter-task and the Intra-task striped methods could be ported to AVX2. It is the view of the author that the striped method will obtain a lower speedup than the COPS approach. This expectation is due to the increased frequency of the Lazy-F loop, which happens when the stride is shorter (AVX entails a 2-fold shorter stride for any model length).

Bibliography

[1] Alpern, B., Carter, L., and Su Gatlin, K. (1995). Microparallelism and high-performance protein matching. In 1995 ACM/IEEE conference on Supercomputing, page 24. ACM.

[2] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). Basic local alignment search tool. Journal of molecular biology, 215(3):403–410.

[3] Alves, C., Cáceres, E., and Dehne, F. (2002). Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In Proceedings of the 14th annual ACM symposium on Parallel algorithms and architectures, pages 275–281. ACM.

[4] Bateman, A., Coin, L., Durbin, R., Finn, R. D., Hollich, V., Griffiths-Jones, S., Khanna, A., Marshall, M., Moxon, S., Sonnhammer, E. L., et al. (2004). The pfam protein families database. Nucleic acids research, 32(suppl 1):D138–D141.

[5] Benson, D., Karsch-Mizrachi, I., Lipman, D., Ostell, J., Rapp, B., and Wheeler, D. (2000). GenBank. Nucleic acids research, 28(1):15–18.

[6] Boukerche, A., Batista, R., and de Melo, A. (2009). Exact pairwise alignment using a novel z-align parallel strategy. In IEEE Symposium on Parallel & Distributed Processing, 2009, pages 1–8. IEEE.

[7] Boukerche, A., de Melo, et al. (2007). Parallel strategies for the local biological sequence alignment in a cluster of workstations. Journal of Parallel and Distributed Computing, 67(2):170–185.

[8] Brocchieri, L. and Karlin, S. (2005). Protein length in eukaryotic and prokaryotic proteomes. Nucleic acids research, 33(10):3390.

[9] Dayhoff, M., Schwartz, R., and Orcutt, B. (1972). 22 A Model of Evolutionary Change in Proteins. Atlas of protein sequence and structure, 5:345–352.

[10] de Almeida, T. and Roma, N. (2010). A parallel programming framework for multi-core DNA sequence alignment. In Complex, Intelligent and Software Intensive Systems (CISIS), pages 907–912. IEEE.

[11] Delcher, A., Kasif, S., Fleischmann, R., Peterson, J., White, O., and Salzberg, S. (1999). Alignment of whole genomes. Nucleic Acids Research, 27(11):2369–2376.

[12] Derrien, S. and Quinton, P. (2010). Hardware acceleration of hmmer on fpgas. Journal of Signal Processing Systems, 58(1):53–67.

[13] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. (1998). Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge university press.

[14] Eddy, S. R. (1998). Profile hidden markov models. Bioinformatics, 14(9):755–763.

[15] Eddy, S. R. (2010). Hmmer user's guide.

[16] Eddy, S. R. (2011). Accelerated profile hmm searches. PLoS Computational Biology, 7(10):e1002195.

[17] Farrar, M. (2007). Striped Smith–Waterman speeds database searches six times over other SIMD implementations. Bioinformatics, 23(2):156–161.

[18] Farrar, M. (2008). Optimizing smith-waterman for the cell broadband engine. Sequence Analysis.

[19] Ganesan, N., Chamberlain, R. D., Buhler, J., and Taufer, M. (2010). Accelerating hmmer on gpus by implementing hybrid data and task parallelism. In Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology, pages 418–421. ACM.

[20] Gotoh, O. (1982). An improved algorithm for matching biological sequences. Journal of molecular biology, 162(3):705–708.

[21] Green, P. (1993). SWAT: Smith Waterman Alignment Tool. http://www.genome.washington.edu/uwgc/analysistools/swat.htm.

[22] Gribskov, M., McLachlan, A., and Eisenberg, D. (1987). Profile analysis: detection of distantly related proteins. Proceedings of the National Academy of Sciences, 84(13):4355.

[23] Henikoff, S. and Henikoff, J. (1992). Amino acid substitution matrices from protein blocks. Proceedings of the National Academy of Sciences, 89(22):10915.

[24] Henikoff, S. and Henikoff, J. (1994). Position-based sequence weights. Journal of Molecular Biology, 243(4):574–578.

[25] Horn, D. R., Houston, M., and Hanrahan, P. (2005). Clawhmmer: A streaming hmmer-search implementatio. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing, page 11. IEEE Computer Society.

[26] Hosny, A., Shedeed, H., et al. (2011). An Efficient Solution for Aligning Huge DNA Sequences. International Journal of Computer Applications (0975 ? 8887), 32.

[27] Karp, A. H. and Flatt, H. P. (1990). Measuring parallel processor performance. Communications of the ACM, 33(5):539–543.

[28] Karplus, K., Barrett, C., and Hughey, R. (1998). Hidden markov models for detecting remote protein homologies. Bioinformatics, 14(10):846–856.

[29] Kent, W. (2002). BLAT: the BLAST-like alignment tool. Genome research, 12(4):656–664.

[30] Krogh, A., Brown, M., Mian, I. S., Sjolander, K., and Haussler, D. (1994). Hidden markov models in computational biology: Applications to protein modeling. Journal of molecular biology, 235(5):1501–1531.

[31] Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, pages 707–710.

[32] Levon, J. and Elie, P. (1990). Oprofile - a system profiler for linux. `http://oprofile.sourceforge.net`.

[33] Liao, H., Yin, M., and Cheng, Y. (2004). A parallel implementation of the Smith-Waterman algorithm for massive sequences searching. In Engineering in Medicine and Biology Society. 26th Annual Conference, volume 2, pages 2817–2820. IEEE.

[34] Lindahl, E. (2005). http://lindahl.sbc.su.se/software/altivec/altivec-hmmer,-version-2.html.

[35] Liu, W., Schmidt, B., and Muller-Wittig, W. (2011). CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 8(6):1678–1684.

[36] Liu, Y., Maskell, D., and Schmidt, B. (2009). CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled GPUs. BMC Research Notes, 2(1):73.

[37] Meng, X. and Chaudhary, V. (2005). Exploiting multi-level parallelism for homology search using general purpose processors. In 11th Conference on Parallel and Distributed Systems, volume 2, pages 331–335. IEEE.

[38] Needleman, S. and Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. of Mol. Biol., 48(3):443–453.

[39] Oliver, T., Schmidt, B., and Maskell, D. (2005). Reconfigurable architectures for bio-sequence database scanning on FPGAs. Transactions on Circuits and Systems II: Express Briefs, 52(12):851–855.

[40] Pearson, W. (1991). Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. Genomics, 11(3):635–650.

[41] Pearson, W. and Lipman, D. (1988). Improved tools for biological sequence comparison. Proceedings of the National Academy of Sciences, 85(8):2444.

[42] Rabiner, L. and Juang, B. (1986). An introduction to hidden markov models. ASSP Magazine, IEEE, 3(1):4–16.

[43] Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE, 77(2):257–286.

[44] Rognes, T. (2001). ParAlign: a parallel sequence alignment algorithm for rapid and sensitive database searches. Nucleic Acids Research, 29(7):1647.

[45] Rognes, T. (2011). Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. BMC bioinformatics, 12(1):221.

[46] Rognes, T. and Seeberg, E. (2000). Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics, 16(8):699.

[47] Rudnicki, W., Jankowski, A., et al. (2009). The new SIMD implementation of the Smith-Waterman algorithm on Cell microprocessor. Fundamenta Informaticae, 96(1):181–194.

[48] Rudnicki, W., Maszkowski, R., Bolikowski, L., Cytowski, M., Dobrzynski, M., and Fronczak, M. (2006). Parallel Position-Specific Iterated Smith-Waterman Algorithm Implementation. Proceedings the CUG 2005 conference in Albuquerqe.

[49] Smith, T., Waterman, M., et al. (1981a). Identification of common molecular subsequences. J. mol. Biol, 147(1):195–197.

[50] Smith, T., Waterman, M., and Fitch, W. (1981b). Comparative biosequence metrics. Journal of Molecular Evolution, 18(1):38–46.

[51] Szalkowski, A., Ledergerber, C., et al. (2008). SWPS3: fast multi-threaded vectorized Smith-Waterman for IBM Cell/BE and x86/SSE2. BMC Research Notes, 1(1):107.

[52] Viterbi, A. (April). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. Information Theory, IEEE Transactions on, 13(2):260–269.

[53] Wagner, R. and Fischer, M. (1974). The string-to-string correction problem. Journal of the ACM (JACM), 21(1):168–173.

[54] Walters, J. P., Qudah, B., and Chaudhary, V. (2006). Accelerating the hmmer sequence analysis suite using conventional processors. In Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on, volume 1, pages 6–pp. IEEE.

[55] Waterman, M., Smith, T., and Beyer, W. (1976). Some biological sequence metrics. Advances in Mathematics, 20(3):367–387.

[56] Wozniak, A. (1997). Using video-oriented instructions to speed up sequence comparison. Computer applications in the biosciences: CABIOS, 13(2):145–150.

[57] Yu, C., Kwong, K., et al. (2005). A Smith-Waterman systolic cell. New Algorithms, Architectures and Applications for Reconfigurable Computing, pages 291–300.

A

Code Listings

Listing A.1 Pseudo-code of the Viterbi algorithm for multihit local alignment, with the superfluous transitions already removed.

// Initialization:

$V^B(0) \leftarrow log\ t_{NB}$

for $i \leftarrow 1$ to $SequenceLength\ (L)$ **do**

$V_0^M(i) \leftarrow V_0^D(i) \leftarrow V_0^I(i) \leftarrow V^E(i) \leftarrow -infinity$

for $j \leftarrow 1$ to $ModelSize\ (MS)$ **do**

$$V_j^M(i) \leftarrow log\frac{e_{Mj}(x_i)}{q_{xi}} + Max \begin{cases} V^B\ (i-1) + log\ t_{B_{j-1}M_j} \\ V_{j-1}^M(i-1) + log\ t_{M_{j-1}M_j} \\ V_{j-1}^I(i-1) + log\ t_{I_{j-1}M_j} \\ V_{j-1}^D(i-1) + log\ t_{D_{j-1}M_j} \end{cases}$$

$$V_j^E(i) \leftarrow Max \begin{cases} V_{j-1}^E(i) \\ V_j^M(i) \end{cases}$$

$$V_j^I(i) \leftarrow Max \begin{cases} V_j^M(i-1) + log\ t_{M_jI_j} \\ V_j^I(i-1) + log\ t_{I_jI_j} \end{cases}$$

$$V_j^D(i) \leftarrow Max \begin{cases} V_{j-1}^M(i) + log\ t_{M_{j-1}D_j} \\ V_{j-1}^D(i) + log\ t_{D_{j-1}D_j} \end{cases}$$

end for

// Updates of special states after the symbol's propagation through all the normal states:

$$V_{MS}^E(i) \leftarrow Max \begin{cases} V_{MS}^E(i) \\ V_{MS}^M(i) \end{cases}$$

$$V^J(i) \leftarrow Max \begin{cases} V^J(i) + log\ t_{JJ} \\ V_{MS}^E(i) + log\ t_{EJ} \end{cases}$$

$$V^C(i) \leftarrow Max \begin{cases} V^C(i) + log\ t_{CC} \\ V_{MS}^E(i) + log\ t_{EC} \end{cases}$$

$$V^N(i) \leftarrow V^N(i-1) + log\ t_{NN}$$

$$V^B(i) \leftarrow Max \begin{cases} V^N(i) + log\ t_{NB} \\ V^J(i) + log\ t_{JB} \end{cases}$$

end for

Listing A.2 Pseudo-code of the Forward algorithm for multihit local alignment, using the Interpolation/Lookup approach, and with the superflous transitions removed. The table $TabLog$ is precomputed, as well as the logarithmic transition scores $log\ t_{X_jY_i}$, and emission scores $log\frac{e_{Mj}(x_i)}{q_{xi}}$.

// Initialization:

$F^B(0) \leftarrow log\ t_{NB}$

for $i \leftarrow 1$ to $SequenceLength\ (L)$ **do**

$F_j^M(0) = F_j^D(0) = F_j^I(0) = F^E(i) = -infinity$

for $j \leftarrow 1$ to $ModelSize\ (MS)$ **do**

$$F_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + TabLog\ [\ TabLog\ [\ log\ t_{M_{j-1}M_j} + (F_{j-1}^M(i-1))\ ,\ log\ t_{I_{j-1}M_j} + (F_{j-1}^I(i-1))\ ]\ ,\ TabLog\ [\ log\ t_{D_{j-1}M_j} + (F_{j-1}^D(i-1))\ ,\ log\ t_{B_{j-1}M_j} + (F^B(i-1)\ ]\ ]$$

$$F_j^I(i) \leftarrow TabLog\ [\ log\ t_{M_jI_j} + (F_j^M(i-1)) + \ log\ t_{I_jI_j} + (F_j^I(i-1))\ ]$$

$$F_j^D(i) \leftarrow TabLog\ [\ log\ t_{M_jD_j} + (F_{j-1}^M(i)) + \ log\ t_{M_jD_j} + (F_{j_1}^D(i))\ ]$$

$$F_j^E(i) \leftarrow TabLog\ [F_{j-1}^E(i), TabLog\ [\ F_j^M(i), F_j^D(i)\ ]\ ]$$

end for

Updates of special states after the symbol's propagation through all the normal states:

$F^J(i) \leftarrow TabLog\ [\ t_{JJ} + F^J(i-1)\ , t_{EJ} + F_{MS}^E(i)\ ]$

$F^C(i) \leftarrow TabLog\ [\ t_{CC} + F^C(i-1)\ , t_{EC} + F_{MS}^E(i)\ ]$

$F^N(i) \leftarrow log\ t_{NN} + F^N(i-1)$

$F^B(i) \leftarrow TabLog\ [\ t_{NB} + F^N(i)\ , t_{JB} + F^J(i)\ ]$

end for

Listing A.3 Initial inner loop code of COPS, based on Rognes Smith-Waterman implementation.

$InitializeMmx, Imx, Dmxto - Vinfinity$

▷ Loop through the sequence symbols

for $i \leftarrow 1$ to $SequenceLength\ (L)$ **do**

$LoadEmissionScores(e_{match}, i)$

$xmxE \leftarrow mpv \leftarrow dpv \leftarrow ipv \leftarrow sc \leftarrow dcv \leftarrow -Vinfinity$

▷ Loop through the model state-triplets

for $i \leftarrow 0$ to $M - 1$ **do**

$$Mnext \leftarrow e_{match}(k) + VMax \begin{cases} vB + t_{BM}(k) \\ Mpv + t_{MM}(k) \\ Ipv + t_{IM}(k) \\ Dpv + t_{DM}(k) \end{cases}$$

$vE \leftarrow VMax(vE, Mnext)$

▷ Pre-emptive load of M, D, I

$Dpv \leftarrow Dmx(k)$

$Ipv \leftarrow Imx(k)$

$Mpv \leftarrow Mmx(k)$

▷ Delayed stores of M and D

$Mmx(k) \leftarrow Mnext$

$Dmx(k) \leftarrow Dcv$

▷ Calculate current I(k). e_{insert} always 0 in HMMER's unilocal mode

$$Imx(k) \leftarrow e_{insert}(k+1) + VMax \begin{cases} Mpv + t_{MI}(k+1) \\ Ipv + t_{II}(k+1) \end{cases}$$

▷ Calculate next D, D(k+1)

$$Dcv \leftarrow VMax \begin{cases} Mnext + t_{MD}(k+1) \\ Dcv + t_{DD}(k+1) \end{cases}$$

end for

▷ Compute and update the special flanking states

$vJ \leftarrow VMAX(vJ + t_{JJ}, vE + t_{EJ})$ // always 0 for HMMER's Unilocal mode

$vC \leftarrow VMAX(vC + t_{CC}, vE + t_{EC})$ // t_{EC} is 0

$vN \leftarrow vN + t_{NN}$

$vB \leftarrow VMAX(vN + t_{NM}, vJ + t_{JB})$ // vJ is useless

end for

Listing A.4 SSE code of Rognes method to load and arrange the Emission Scores

```
for i ← 1 to M step 4 do
    ▷ Load original scores
    xmm[0] = _mm_load_ps(EMscoreSeq[0]+k)
    xmm[1] = _mm_load_ps(EMscoreSeq[1]+k)
    xmm[2] = _mm_load_ps(EMscoreSeq[2]+k)
    xmm[3] = _mm_load_ps(EMscoreSeq[3]+k)
    ▷ Interleave 32-bit wide
    xmm[4] = _mm_unpacklo_ps(xmm[0], xmm[1])
    xmm[5] = _mm_unpackhi_ps(xmm[0], xmm[1])
    xmm[6] = _mm_unpacklo_ps(xmm[2], xmm[3])
    xmm[7] = _mm_unpackhi_ps(xmm[2], xmm[3])
    ▷ Interleave 64-bit wide
    xmm[8] = _mm_unpacklo_pd(xmm[4], xmm[6])
    xmm[9] = _mm_unpackhi_pd(xmm[4], xmm[6])
    xmm[10] = _mm_unpacklo_pd(xmm[5], xmm[7])
    xmm[11] = _mm_unpackhi_pd(xmm[5], xmm[7])
    ▷ Store transposed values
    _mm_store_si128(vEmsc+0, xmm[8])
    _mm_store_si128(vEmsc+1, xmm[9])
    _mm_store_si128(vEmsc+2, xmm[10])
    _mm_store_si128(vEmsc+3, xmm[11])
    vEmsc = vEmsc + 4
end for
```

Listing A.5 SSE code of the Inner loop with the Inline method to load and arrange the Emission Scores

for $k \leftarrow 1$ to M step 4 **do**

▷ Load original scores

xmm[0] = _mm_load_ps(EMscoreSeq[0]+k)

xmm[1] = _mm_load_ps(EMscoreSeq[1]+k)

xmm[2] = _mm_load_ps(EMscoreSeq[2]+k)

xmm[3] = _mm_load_ps(EMscoreSeq[3]+k)

▷ Interleave 32-bit wide

xmm[4] = _mm_unpacklo_ps(xmm[0], xmm[1])

xmm[5] = _mm_unpackhi_ps(xmm[0], xmm[1])

xmm[6] = _mm_unpacklo_ps(xmm[2], xmm[3])

xmm[7] = _mm_unpackhi_ps(xmm[2], xmm[3])

▷ Interleave 64-bit wide

xmm[8] = _mm_unpacklo_pd(xmm[4], xmm[6])

xmm[9] = _mm_unpackhi_pd(xmm[4], xmm[6])

xmm[10] = _mm_unpacklo_pd(xmm[5], xmm[7])

xmm[11] = _mm_unpackhi_pd(xmm[5], xmm[7])

▷ Macro with the inner loop code. Arguments: (model state index, xmm array index)

COMPUTE_TRIPLET_STATE(k+0,8)

COMPUTE_TRIPLET_STATE(k+1,9)

COMPUTE_TRIPLET_STATE(k+2,10)

COMPUTE_TRIPLET_STATE(k+3,11)

end for

Listing A.6 Pseudo-code of the proposed COPS approach using SSE2 with 8x16-bit integers and saturated arithmetic. This is the final single-threaded version.

```
1: ▷ Loop through the partitions
2: for p ← 1 to Npartitions do
3:     Initialize Mmx, Imx, Dmx to −∞
4:     ▷ Loop through the sequence symbols
5:     for i ← 1 to SequenceLength (L) do
6:         if p = 0 then
7:             ▷ First partition, initialize all to −∞
8:             xmxE ← Mnext ← Dcv ← −∞
9:         else
10:            ▷ Load data from previous partitions
11:            xmxE ← PxmxE(i)
12:            Dcv  ← PDcv(i)
13:            Mnext ← PMnext(i)
14:        end if
15:        ▷ Loop through the model state-triplets of the current partition
16:        for k ← 0 until k = PartLength or k + p × PartLength > ModelLength step 8 do
17:            ▷ Load original scores, xmm 0 to 8
18:            xmm[0] ← LOAD16(EMscoreSeq[0] + k)
19:            xmm[1] ← LOAD32(EMscoreSeq[1] + k)
20:            ...
21:            ▷ Interleave 16-bit wide, xmm 8 to 15
22:            xmm[8] ← UNPACK_LOW16(xmm[0], xmm[1])
23:            xmm[9] ← UNPACK_HIGH16(xmm[0], xmm[1])
24:            ...
25:            ▷ Interleave 32-bit wide, xmm 16 to 23
26:            xmm[16] ← UNPACK_LOW32(xmm[8], xmm[10])
27:            xmm[17] ← UNPACK_HIGH32(xmm[8], xmm[10])
28:            ...
29:            ▷ Interleave 64-bit wide, xmm 24 to 31
30:            xmm[24] ← UNPACK_LOW64(xmm[16], xmm[18)
31:            xmm[25] ← UNPACK_HIGH64(xmm[16], xmm[18])
32:            ...
33:            ▷ Macro with the inner loop code, xmm 24 to 31
34:            COMPUTE_STATE_TRIPLET(k + 0, 24)
35:            COMPUTE_STATE_TRIPLET(k + 1, 25)
36:            ...
37:        end for
38:        ▷ Compute and update the special flanking states
39:        if k + p × PartLength < ModelLength then
40:            ▷ Not the last partiton, store data for next partition
41:            PxmxE(i) ← xmxE
42:            PDcv(i) ← Dcv
43:            PMnext(i) ← Mnext
44:        else
45:            ▷ Final partition, update the definitive pseudo-states
46:            xmxC ← VMAX16(xmxC, xmxE)
47:        end if
48:    end for
49: end for
50: return Undiscretize(VMAX16(xmxC, t_CT))
```

Listing A.7 Core inner loop code, used in the macro COMPUTE_STATE_TRIPLET(Model State index, Emission Score index). Variable k is the State index

▷ Use M value partially computed in last iteration

$$Mnext \leftarrow VMAX16 \begin{cases} Mnext \\ xmxB + t_{BM}(k) \\ xmx[EMindex]) \end{cases}$$

$$xmxE \leftarrow VMAX16(vE, Mnext)$$

▷ Load scores from last column

$$Dpv \leftarrow Dmx(k)$$

$$Ipv \leftarrow Imx(k)$$

$$Mpv \leftarrow Mmx(k)$$

▷ Compute and store scores of this column

$$Mmx(k) \leftarrow Mnext$$

$$Dmx(k) \leftarrow Dcv$$

$$Imx(k) \leftarrow VMAX16 \begin{cases} Mpv + t_{MI}(k+1) \\ Ipv + t_{II}(k+1) \end{cases}$$

▷ Preempetive computation of next-column D score

$$Dcv \leftarrow VMAX16 \begin{cases} Mnext + t_{MD}(k+1) \\ Dcv + t_{DD}(k+1) \end{cases}$$

▷ Partially compute M score for next column

$$Mnext \leftarrow VMAX16 \begin{cases} Mpv + t_{MM}(k) \\ Ipv + t_{IM}(k) \\ Dpv + t_{DM}(k) \end{cases}$$

B

Results of the Wave-Front Multi-threading

Wave-Front Multi-threading Results on AMD Opteron Bulldozer

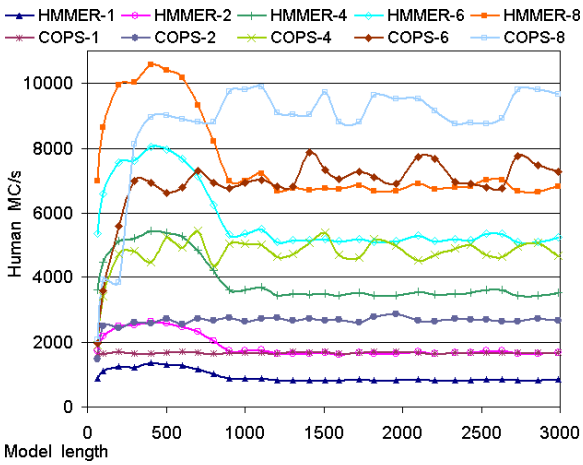


Figure B.1: Benchmark speeds of multi-threaded COPS and HMMER, on an AMD Opteron Bulldozer, for the Human genome

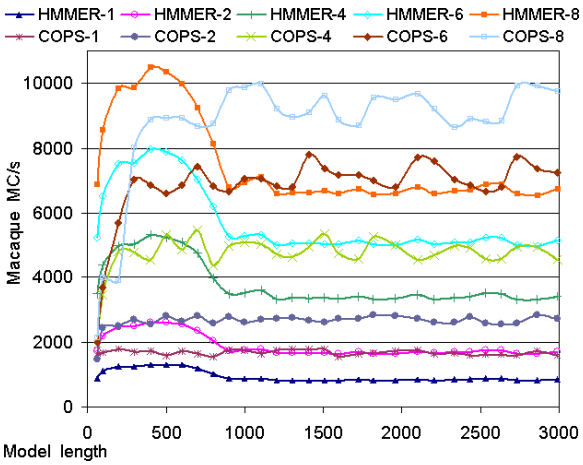


Figure B.2: Benchmark speeds of multi-threaded COPS and HMMER, on an AMD Opteron Bulldozer, for the Macaque genome.

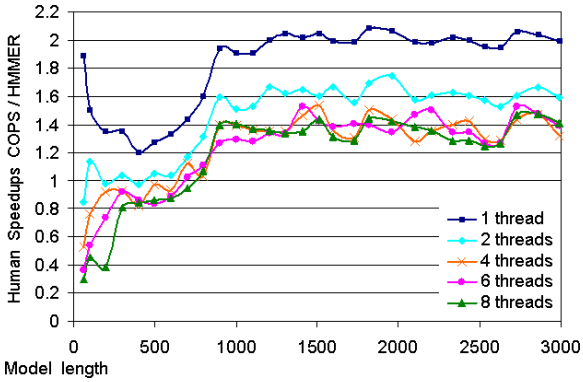


Figure B.3: Benchmark speedups of multi-threaded COPS vs HMMER, on an AMD Opteron Bulldozer, for the Human genome

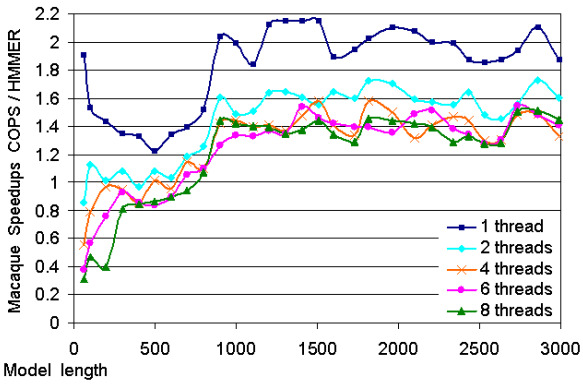


Figure B.4: Benchmark speedups of multi-threaded COPS vs HMMER, on an AMD Opteron Bulldozer, for the Macaque genome.

Wave-Front Multi-threading Results on Intel Xeon Nehalem

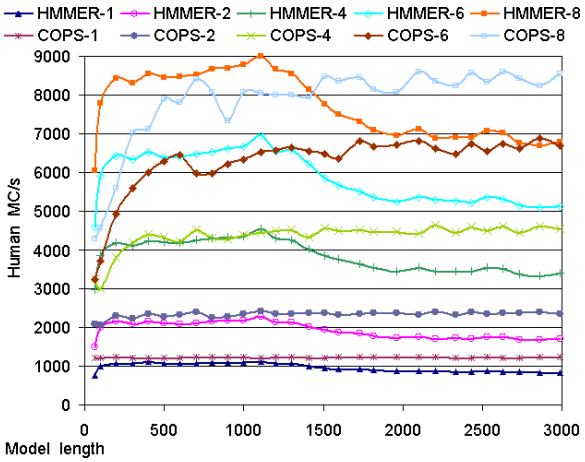


Figure B.5: Benchmark speeds of multi-threaded COPS and HMMER, on an Intel Xeon Nehalem, for the Human genome

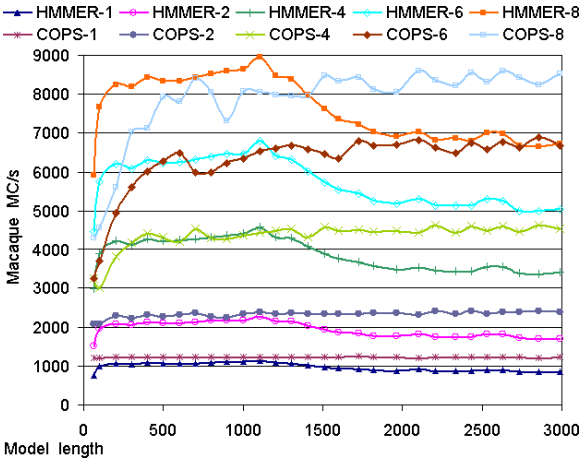


Figure B.6: Benchmark speeds of multi-threaded COPS and HMMER, on an Intel Xeon Nehalem, for the Macaque genome.

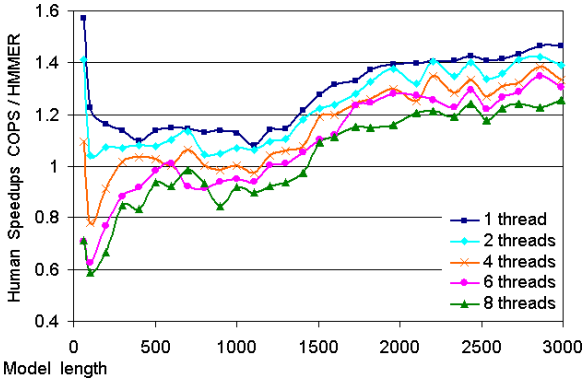


Figure B.7: Benchmark speedups of multi-threaded COPS vs HMMER, on an Intel Xeon Nehalem, for the Human genome

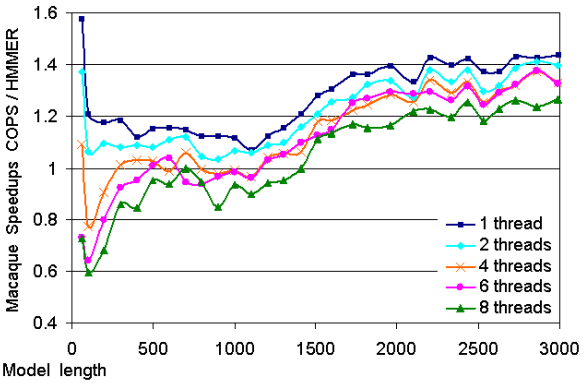


Figure B.8: Benchmark speedups of multi-threaded COPS vs HMMER, on an Intel Xeon Nehalem, for the Macaque genome.

Wave-Front Multi-threading Results on Intel Core i7 Sandy Bridge

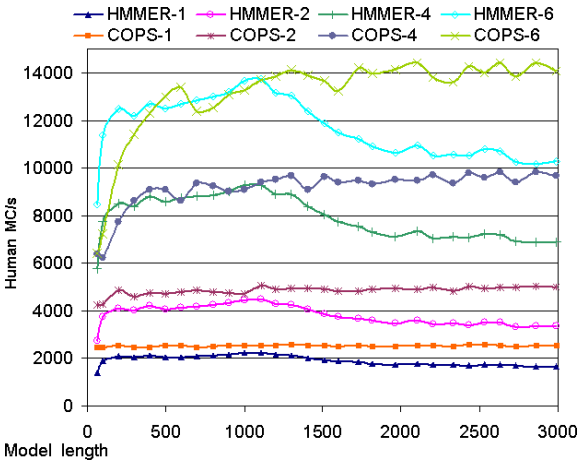


Figure B.9: Benchmark speeds of multi-threaded COPS and HMMER, on an Intel Core i7 Sandy Bridge, for the Human genome

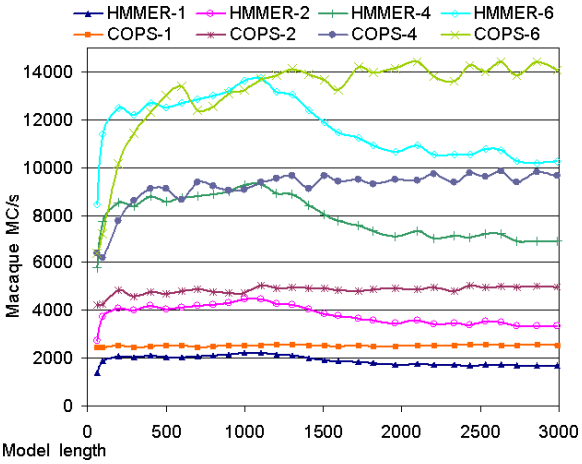


Figure B.10: Benchmark speeds of multi-threaded COPS and HMMER, on an Intel Core i7 Sandy Bridge, for the Macaque genome.

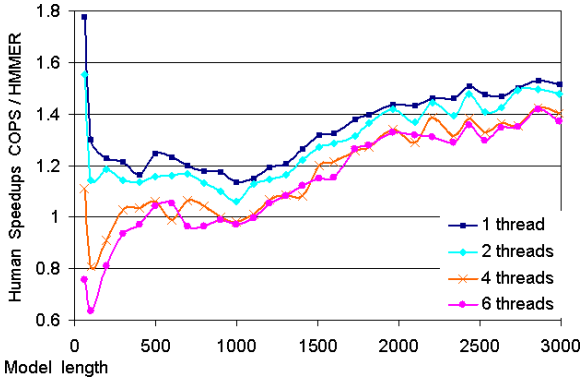


Figure B.11: Benchmark speedups of multi-threaded COPS vs HMMER, on an Intel Core i7 Sandy Bridge, for the Human genome

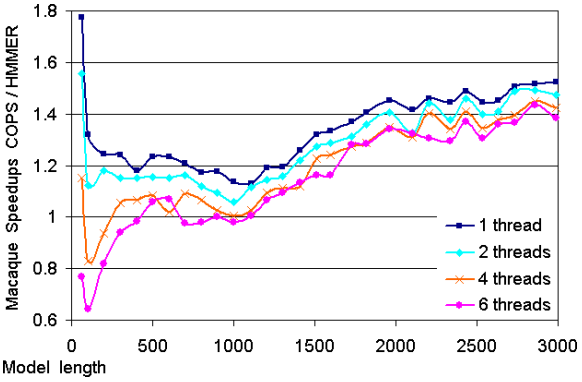


Figure B.12: Benchmark speedups of multi-threaded COPS vs HMMER, on an Intel Core i7 Sandy Bridge, for the Macaque genome.