HARDWARE ACCELERATION FOR MULTIPLE SEQUENCE ALIGNMENT OF PROTEINS

by

İbrahim Ateşpare

B.S., Electronics and Communications Engineering, Yildiz Technical University, 2015

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering
Boğaziçi University

2019

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my supervisor Assist. Prof. Faik Başkaya for his guidance and support throughout my thesis studies. I would like to thank Prof. Günhan Dündar and Prof. Fatih Uğurdağ for their participation in my thesis jury. I also thank all the department members, who have been continuously sharing their knowledge with us in the lectures. Besides, I would like to thank Emre Avara for his contributions to this work. I am sincerely thankful to my friend Yusufcan Demirkan for his help during the writing of this thesis.

I would like to send my special thanks to Pam Beasant, again, who probably made me decide on electronics at an early age.

Lastly, my most important and precious gratitude goes to my parents who have supported me my entire life without expecting any response.

ABSTRACT

HARDWARE ACCELERATION FOR MULTIPLE SEQUENCE ALIGNMENT OF PROTEINS

Multiple Sequence Alignment (MSA), which has been an active research area for a long time, is a computational task that is used in the field of bioinformatics. MSA for proteins involves the process of aligning three or more amino acid sequences. The outputs of this task provide further information for other areas in bioinformatics such as homology searches, phylogenetic analysis and protein structure prediction. The algorithmic solutions for MSA can mainly be split into three groups: Direct approach, progressive approaches and iterative approaches. H4MSA is one of the iterative approaches in the literature, that uses multi-objective optimization and a type of genetic algorithm (GA) to solve the MSA problem.

In this work, a hardware acceleration is proposed for the MSA tasks and H4MSA is selected as the test algorithm. The acceleration is accomplished by designing a hardware block for the evaluation of weighted sum of pairs score (WSPS), which is a commonly used scoring function in MSA applications and also used as an objective function in the H4MSA algorithm. The entire application is built in a CYCLONE V SOC device, where the hardware is implemented in the FPGA fabric and the test algorithm is run on the CPU core of the device. For testing the design, 12 different sequence sets from a well-known database are used, and the results show that the accelerated version of the algorithm achieves speedups of 1.7x to 20x over the CPU version.

ÖZET

PROTEİNLERİN ÇOKLU DİZİ HİZALAMASI İÇİN DONANIM HIZLANDIRMASI

Uzun süredir aktif bir araştırma alanı olan Çoklu Dizi Hizalaması (MSA), biyoinformatik alanında kullanılan sayısal hesaplamaya dayalı bir yöntemdir. Proteinler için MSA, üç veya daha fazla amino asit dizisinin hizalanması işlemi anlamına gelmektedir. Bu işlemin çıktıları; homoloji aramaları, filogenetik analiz, protein yapısı tahmini gibi biyoinformatiğin başka alanları için girdi bilgilerini oluşturur. MSA için literatürdeki çözümler genel hatlarıyla üç gruba ayrılır: Doğrudan yaklaşım, gelişimsel yaklaşımlar ve yinelemeli yaklaşımlar. H4MSA, çok amaçlı eniyileme ve bir tür genetik algoritma içeren, mevcut yinelemeli yaklaşımlardan biridir.

Bu çalışmada, MSA için bir donanım hızlandırması önerilmiş ve test algoritması olarak H4MSA kullanılmıştır. Hızlandırma, MSA uygulamalarında yaygın bir puanlama çeşidi olan ve ayrıca H4MSA algoritmasında bir amaç fonksiyonu olarak da kullanılan, ağırlıklı çiftler toplamı skoru (WSPS) hesabı için donanımsal bir yapı tasarlanarak geçekleştirilmiştir. Tüm uygulama, bir SOC (tek yonga üzerinde sistem) cihazı içine kurulmuş olup; cihazın FPGA bölümünde donanım bloğu inşa edilmiş ve test algoritması cihazın merkezi işlem biriminde (CPU) çalıştırılmıştır. Tasarımın testi için iyi bilinen bir veri tabanından 12 farklı dizi seti kullanılmış olup, sonuçlar donanımsal hızlandırılmış versiyonun, CPU versiyonuna oranla 1.7x ile 20x arasında hız artırımlarına ulaştığını göstermiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

AGP	Affine gap penalty score
g_e	Gap extension coefficient
g_o	Gap opening coefficient
k	Number of sequences of alignment
L	Length of sequence
LD	Levenshtein distance
m	Number of memeplexes
n	Number of solutions in each memeplex
N	Number of evolutionary steps
nge_i	Number of gap extensions in sequence
ngo_i	Number of gap openings in sequence
OPS	Overall pairwise score
P	Population size
$PAGP$	Pairwise affine gap penalty score
PN	Pair number of input alignment
$pnge$	Total number of gap extensions
$pngo$	Total number of gap openings
PS	Pairwise score
s_i	Unaligned sequence
s'_i	Aligned sequence
S	Set of unaligned sequences
SC_{HVA}	Score of HVA
SC_{SVA}	Score of SVA
sn	Sequence number of input alignment
sps_i	Symbol pair score
sr	Speedup ratio
sym_i	Symbol data
t_{HVA}	Processing time of HVA

t_{SVA}	Processing time of SVA
TCS	Totally conserved column score
$W_{i,j}$	Weight of sequence pair
WSP	Sum of weighted pairwise scores
$WSPS$	Weighted sum of pairs score
x_i	Solution of current population
x_{gb}	Global best solution
x_{lb}	Local best solution
x_w	Local worst solution
x_{new}	New solution

LIST OF ACRONYMS/ABBREVIATIONS

AGP	Affine Gap Penalties
BLOSUM	Block Substitution Matrix
CPU	Central Processing Unit
DNA	Deoxyribonucleic Acid
DP	Dynamic Programming
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
HDL	Hardware Description Language
HPS	Hard Processor System
HVA	Hardware Accelerated Version of the Algorithm
MSA	Multiple Sequence Alignment
OPS	Overall Pairwise Score
PAM	Point Accepted Mutation
PF	Pareto Front
PS	Pair Sum
PSA	Pairwise Sequence Alignment
PSGCR	Pair Sum and Gap Count Registers
RNA	Ribonucleic Acid
SOC	System On a Chip
SRAM	Static Random Access Memory
ST	Score Time
SVA	Software Version of the Algorithm
TCS	Totally Conserved Column Score
TT	Total Time
WSPS	Weighted Sum of Pairs Score

1. INTRODUCTION

The information contained in protein sequences are vital for many fields; however, they are not present in a form ready to interpret. Thus, acquiring the structural, functional, and evolutionary relationships between protein sequences becomes an important task, especially in the field of bioinformatics. A first step towards reaching this objective is the computational process called "sequence alignment", which is defined as the arrangement of multiple sequences such that the similarity between them is maximized.

The sequence alignment task that involves three or more sequences has a special name, "Multiple sequence alignment" (MSA). The results of MSA are used to detect regions of variation and conservations within a protein family and also provide further information for areas such as homology searches, phylogenetic analysis and protein structure prediction. There are various MSA algorithms in the literature and most of them consume large computational times to achieve the best possible results. Moreover, processing times mostly grow quadratically with increasing input sequence count. These aspects make the speed performance of MSA algorithms an important metric as biological databases get larger each day. The purpose of this thesis is to decrease these processing times and provide an acceleration for MSA algorithms. For this purpose, a hardware design was proposed, which is tested on an existing multi-objective MSA algorithm called H4MSA [1]. An SOC device is used as the testing platform, where the proposed hardware is constructed on the FPGA fabric and the test algorithm runs on the CPU part of the device.

The general structure of the thesis is described as follows: In this chapter, biological information about proteins and the process of sequence alignment tasks are presented. The next two chapters provide a literature survey of existing MSA algorithms and a detailed review of H4MSA, respectively. In Chapter 4, the proposed hardware design is explained step by step and then the experimental results are shown in Chapter 5. Finally, in the last chapter, the study is concluded with several remarks

and open questions for future research.

1.1. Biological Background

It is necessary to introduce the concepts of proteins and amino acids and their role in life before delving into multiple sequence alignment methods. An amino acid is an organic compound consisting of an amino group, an acidic carboxyl group, and an organic group which determines the uniqueness of each amino acid [2]. The chemical structure of amino acids allows them to bind with each other. This link is called a "peptide bond" and the new sequence structure with multiple amino acids are called "polypeptides". When amino acids join in a polypeptide sequence, they are referred to as residues.

A protein is a special type of polypeptide-sequence, which is a complete biological molecule in a stable configuration that carries some crucial roles in the cell. Essentially, what differentiates these two concepts from each other is the number of residues they contain. While the minimum size of a protein is about 30 residues, most of the proteins that are found in living organisms contain approximately 100 to 1000 residues. The maximum length of proteins can even go up to 27000 residues [3]. It is also important to mention that, while more than 700 amino acids exist in nature [4], proteins in all species, from bacteria to humans, only contain 20 different types among them.

On the other hand, the functional roles of proteins vary greatly from one organism to another. Still, the most important role they play is also the most common one: They are the building blocks of the cells; that is, the majority of the structural elements contained in cells are made of proteins. In larger scales, this leads to the structures that help move the body; i.e., they found the building blocks for the motor skills organisms can perform. Furthermore, they are catalyzers of almost all chemical reactions happening in the cell [5]. Some proteins provide the means for the transportation of important materials between cells. As antibodies, they protect the organism by binding to specific foreign particles, such as viruses and bacteria. Most of the hormones are made of proteins. Besides, proteins control the major activity of genes.

1.2. Definition and Purpose of Sequence Alignment

Sequence alignment, a widely used method in areas such as bioinformatics, natural language processing and finance, refers to the alignment of two or more sequences so that a similarity metric between these can be produced as an output. These sequences usually refer to protein, DNA or RNA sequences when we are in the domain of biology. Throughout this work, we further restrict our attention to protein sequences.

Biologists use the similarity information (according to the types and order of the residues), which is purely a product of the computational alignment operation, and then they draw several conclusions about the relations between the compared sequences and hence use this information in the applications, such as homology searches (i.e., search for common genetic ancestor), phylogenetics (the study of evolutionary relationships among biological entities) and protein structure prediction. Sequence alignment also plays an important role in identifying the human genome. The identification helps to unlock the genetic components for many diseases, and this provides major improvements in applications of medical and pharmaceutical industries, such as personalized medicine, preventive medicine, and gene therapy. Moreover, obtaining functional information on viruses and bacteria by observing their protein structures helps improve the quality of the medicines and the efficiency of treatments. Another application field for sequence alignment is agriculture. By sequencing the genomes and proteins of the plants, stronger crops can be produced, and the quality of the livestock can be improved.

Aligning the sequences gives us some important biological data about the structural, functional, and evolutionary relationships between the sequences [6]. We know that local or global similarities of the compared sequences can be directly related to the structural relationship of the proteins. We can furthermore link this to functional relationship, though the precise relation between the structure and the function of proteins is still an open question. Yet, the literature identified a positive correlation between functional and structural properties. This means that knowing the structure of a protein might provide us information about its function and thus we can hypothesize that

structurally similar proteins may have similar functions.

Other important information we can obtain from the alignment of sequences is the evolutionary relationship between compared sequences. Evolutionary relationship is interpreted through sequence alignment as follows: The theory of evolution tells us that all species share a common ancestor and this common ancestor has been branching into new species throughout millions of years. The formation of new species occurs by genetic mutations, which is defined as structural changes of DNA, RNA, or proteins by insertion, deletion, or substitution (point mutation) events. This means that evolutionary closer sequences have smaller structural changes while more distant ones have larger changes. Sequence alignment is the first step for the detection of these changes. Once the data are aligned, they are post processed in the applications mentioned above.

There are two types of sequence alignments, which are "pairwise sequence alignment" (PSA) and "multiple sequence alignment" (MSA). While PSA considers two sequences at the same time, MSA does the same for multiple (greater than two) sequences [7]. This enables MSA to treat a protein family as a whole group. In real use cases, the size of these groups can vary between 20 to 10000 sequences. There are even some applications, in which the sequence numbers go up to 1.5 million [8]. Compared with PSA, MSA provides us with more biological data because of this property. It is used to detect regions of variation and conservations within a family and plays an important role in the detection of homology in the case, where a new sequence or sequences are added to the already aligned family. Genomic analysis pipelines, secondary structure prediction, and phylogenetic reconstruction also use prerequisite information obtained from MSAs. Furthermore, they may be used to derive profiles or hidden Markov models that can be used to scour databases for distantly related members of the family [9]. Since all these applications need large amounts of data and the complexity increases as the number of the sequence increases, quality and performance of MSA algorithms get more and more important each day.

1.3. Sequence Alignment Process

Basic processes of the two types of alignments are explained in following sections. PSA is explained in more detail, because pairwise scoring of PSA comprises an important part in the most of the MSA algorithms.

The core data of these computational operations is the symbolic representation of proteins in a text file. Each individual protein contains a sequence of capital letters. Each letter represents a unique amino acid residue. Since there are 20 different amino acid residues, 20 different capital letters [10] are assigned to them as shown in Table 1.1.

Table 1.1. 20 amino acids and corresponding symbols.

Name	**Symbol**	**Name**	**Symbol**
alanine	A	leucine	L
arginine	R	lysine	K
asparagine	N	methionine	M
aspartic	D	phenylalanine	F
cysteine	C	proline	P
glutamine	Q	serine	S
glutamic acid	E	threonine	T
glycine	G	tryptophan	W
histidine	H	tyrosine	Y
isoleucine	I	valine	V

1.3.1. Pairwise Sequence Alignment

As mentioned before, the purpose of sequence aligning is to find similarities between proteins. Maximum similarity occurs when two proteins are exactly the same. If two compared sequences are evolutionarily related, that tells us they shared a common ancestor at some point in their history or one of them is itself the ancestor of the other.

Hence, they were actually the same sequence once, and they diverged in their path via various mutations in the course of time, meaning some of their parts started to differentiate. These mutations can be split into two: Indels and point mutations.

Indel refers to the operation of single or multiple residue deletion or insertion during the mutation process. In Figure 1.1, a protein sequence is given with a mutated version underneath of it. The mutation here is the deletion of fifth residue (K) in the base sequence. Unaligned sequences are denoted as s_i, where i indicates the order index of the sequence within the set.

s_1 : KLVRKNILQL
s_2 : KLVRNILQL

Figure 1.1. Unaligned sequence pair with an indel.

The basic operation of sequence alignment is first comparing each residue pair in the same column and then trying to align the most similar ones under each other. The operation does this by putting gap symbols, which are dots or dashes, in between residues. Adding a single gap shifts the rest of the sequence one position to the right, while adding multiple gaps at various positions makes it possible to try every possible combination of alignments, so that the operation can find the best. For example, in the previous figure, scanning through the columns one by one from left to the right, the algorithm will find the exact column-by-column match of the sequences until the fourth column. After fourth, there is no match between residue pairs. The operation then tries to align the rest by putting gaps. Since there are no point mutations in this example, a better alignment means higher number of exact matches. Adding a gap between the fourth residue (R) and the fifth residue (N) of the second sequence will give us the best possible alignment as shown in Figure 1.2 (Note that, aligned sequences are now denoted as s'_i.). This example is a simple one which can be solved easily by the human eye, since it is quite easy to detect with our visual system. With more complex sequence sets consisting of more residues, the efficiency of the algorithm

becomes more important.

s'_1 : KLVRKNILQL
s'_2 : KLVR.NILQL

Figure 1.2. Aligned sequence pair with an indel.

In Figure 1.3, the second sequence is now generated by performing one more deletion and one insertion, in addition to the deletion we have done before. The solution is given in the same figure. Note that insertion creates an extra residue in the second sequence, and this is detected by the added gap in the first sequence.

Unaligned pair:
s_1 : KLVRKNILQL
s_2 : KQLVRNILL

Aligned pair:
s'_1 : K.LVRKNILQL
s'_2 : KQLVR.NIL.L

Figure 1.3. Sequence alignment with multiple indels.

The other type of mutation, the point mutation, refers to the substitution of a residue with another type. Figure 1.4 shows a point mutated version of the same example where the third residue V is changed into T in the second sequence.

s_1 : KLVRKNILQL
s_2 : KLTRKNILQL

Figure 1.4. Unaligned sequence pair with a point mutation.

Point mutation changes the binary situation of match or no match comparison into a scalar score. The structurally closer residue pairs get higher scores while the distant ones get lower scores, and exact matches get the highest score. The scores of the residue pairs are determined by a substitution matrix. Different types of substitution matrices are used depending on the characteristic of the dataset. PAM (Point Accepted Mutation) [11] matrix and BLOSUM (Block Substitution Matrix) [12] are the most commonly used ones. Throughout this work, BLOSUM62 matrix, which is shown in Figure 1.5, is used in all scorings.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Figure 1.5. BLOSUM62 Matrix.

The individual pair scores are then summed up to create the pairwise score of the sequence pair. The mathematical expression of the pairwise score via substitution matrix is given in Equation 1.1. PS refers to pairwise score, sps_i is the symbol (gap or residue) pair score derived from the substitution matrix and L is the length of the sequence pair. The scoring is referred as symbol pair score, because not all the pairs are residues. The symbol pairs that contain a gap are scored as 0 without looking at the matrix.

$$PS = \sum_{i=1}^{L} sps_i \tag{1.1}$$

To summarize PSA as a computational task, the algorithm first gets the unaligned sequences as input data. Then, using the gap adding operator it tries to generate an aligned version of it with the total pairwise score as the objective function. The output of the algorithm is the best possible alignment that the algorithm itself can generate. In Figure 1.6, an example of a pairwise alignment is given. Note that the end of the shorter sequence is filled with gaps to equalize the lengths.

Unaligned pair:
s_1 : DDKDPAHYWDL
s_2 : TADDRHFW

Aligned pair:
s'_1 : ..DDKDPAHYWDL
s'_2 : TADDR...HFW..

Figure 1.6. A PSA example.

The pairwise scoring of the aligned pair in the same example is done using the BLOSUM62 matrix: The first two columns (sps_1, sps_2) are scored as 0 since they contain gaps. Each of the third and fourth symbols pairs (sps_3, sps_4) are scored as 6, since they both contain D pairs. Fifth symbol pair (sps_5) consists of a K and an R, this makes the score 2 for this pair. The looking up operation continues until the end of the alignment. All the symbol pair scores are then summed up, and the pairwise score is obtained as $PS = 36$.

Note that the pairs that contain a gap is scored as 0. Instead of leaving them as 0s, a gap penalty score is assigned to each gap containing a symbol pair in most of the algorithms. The overall gap penalty is later subtracted from the pairwise score and

the final score is achieved. One of the most common gap penalty scoring systems is calculating the penalty with "affine gap penalties" (AGP) [13] method. In this scoring system, the total number of gap openings, which are the first gaps of a gap group, (a gap group is a subsequence that is made of consecutive gaps.) and gap extensions, which are the remaining gaps of a gap group, are calculated separately in both sequences. Then these numbers are multiplied by two coefficients and subtracted from the pairwise sum score, which forms the overall pairwise score (OPS). The mathematical expression of overall pairwise score is presented in Equation 1.2, which is followed by the pairwise AGP score calculation (Equation 1.3), where $pngo$ and $pnge$ are the total number of gap openings and gap extensions in the sequence pair respectively. Gap opening coefficient is denoted as g_o and gap extension coefficient as g_e.

$$OPS = PS - PAGP \tag{1.2}$$

$$PAGP = g_o \times pngo + g_e \times pnge \tag{1.3}$$

The overall pairwise score of the alignment in Figure 1.6, is calculated as follows: The total number of gap openings is 3 since there are 3 gap groups in the alignment pair. The remaining gap count is 4, which makes the total gap extension number 4. The gap opening and gap extension coefficients are selected as 5 ($g_o = 5$) and 0.8 ($g_e = 0.8$) respectively. Hence, the AGP score is calculated as $AGP = 5 \times 3 + 0.8 \times 4 = 18.2$. As a result, the overall pairwise score is calculated as $OPS = 36 - 18.2 = 17.8$.

1.3.2. Multiple Sequence Alignment

The workflow of MSA is similar to PSA. After the algorithm gets the unaligned N number of sequences as input data, using gap adding operator and a scoring method as the objective function, it generates the best possible alignment of the set. Scoring method type in MSA varies in different approaches. Some of them are: Weighted sum of

pairs scoring, totally conserved column scoring, consistency based scoring, probabilistic scoring methods etc. [7]. The first two of them are used in the main test algorithm in this work. Thus, their details are analyzed in this section.

Weighted sum of pairs score (WSPS) is the extended version of the pairwise scoring method we discussed previously. In this method, each sequence pair in the aligned set is scored according to their pairwise scoring initially. Each partial pairwise score is then multiplied by a unique weight that is assigned to each pair. After that, all the weighted pair scores are accumulated, and finally, the total gap penalty score is subtracted from the sum, so that the final score is obtained.

The mathematical expression of WSPS is given by the Equation 1.4. The term WSP is denoted as the sum of weighted pairwise scores in the alignment in this equation. Equation 1.5 gives the mathematical expression of WSP, where i and j indicate the order indices, k is the number of sequences and $W_{i,j}$ is the weight assigned to sequence pair of s'_i and s'_j. $PS_{i,j}$ is the pairwise score corresponding to sequence pair of s'_i and s'_j, which was previously shown in Equation 1.1. Affine gap penalty score (AGP) is presented in Equation 1.6, where ngo_i and nge_i are the numbers of gap openings and gap extensions in the corresponding sequence respectively, g_o and g_e are the gap penalty coefficients and k is the number of sequences in the alignment.

$$WSPS = WSP - AGP \tag{1.4}$$

$$WSP = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} W_{i,j} \times PS_{i,j} \tag{1.5}$$

$$AGP = \sum_{i=1}^{k} g_o \times ngo_i + g_e \times nge_i \tag{1.6}$$

In Figure 1.7, a sample multiple sequence alignment is shown. The WSPS of the aligned set is found with the selected weights, $W_{1,2} = 0.52$, $W_{1,3} = 0.53$, $W_{1,4} = 0.54$,

$W_{2,3} = 0.63$, $W_{2,4} = 0.64$, $W_{3,4} = 0.73$, as follows: The pairwise scores are separately calculated and obtained as $PS_{1,2} = 20$, $PS_{1,3} = 15$, $PS_{1,4} = -2$, $PS_{2,3} = 27$, $PS_{2,4} = 12$, $PS_{3,4} = -2$. Multiplying each score with the corresponding weight and adding the results together gives us the weighted sum of pairs as $WSP = 40.48$. Next, total gap opening and extension numbers are found as 5 and 6 respectively. These numbers are then used to calculate the AGP ($g_o = 5$, $g_e = 0.8$) and the result is obtained as $AGP = 5 \times 5 + 0.8 \times 6 = 29.8$. Finally, the subtraction is executed and the final score is found as $WSPS = 40.48 - 29.8 = 10.68$.

Unaligned set:

s_1 : PDASVRFW
s_2 : AENPRMRNW
s_3 : SLHPEMSNW
s_4 : AESTLME

Aligned set:

s'_1 : ...PDASVRFW
s'_2 : AENP..RMRNW
s'_3 : SLHP..EMSNW
s'_4 : AEST..LME..

Figure 1.7. An MSA example.

Aligned set:

s'_1 : VARGGTWDQ.TL
s'_2 : LVRGG..MLGTK
s'_3 : DFRGSQ...ST.
s'_4 : EVRGC....ETL

Figure 1.8. An aligned set with marked totally conserved columns.

Totally conserved column score (TCS) is the second scoring type, which counts the number of all columns that are completely aligned with the same residue. Maximizing this score establishes more conserved or special regions within the alignment [14].

Figure 1.8 shows TCS of a sample aligned sequence. The columns that contain the exact same residues are marked in the figure, where counting these columns gives us the score as $TCS = 3$.

2. EXISTING ALGORITHMS

There are three main approaches in the literature for solving the MSA problem, which are dynamic programming (DP), progressive, and iterative approaches. DP is a direct approach that gives the optimal result and is commonly used in PSA. On the other hand, the usage of DP is practically unsuitable in MSA, since N-dimensional dynamic programming requires excessive processing time. Clearly, there is a time-optimality tradeoff in this kind of problems. Hence, the fact that the optimal solution is quite hard to compute practically, the problem is solved by transforming it into two new sub-problems, each of which focus only on the one dimension of the tradeoff: 1) Trying to reach a solution in relatively short time. 2) Trying to find the solution that approximates the optimum. The former one is called the progressive approach. Such algorithms try to find high scores in a very short time, usually in one run. The drawback of these algorithms is that the probability of finding a solution far away from optimum is quite high. On the other hand, the iterative approach which tries to approximate the solution is based on the number of iterations you perform. As the number of iterations increases, you get closer to the optimum value. The drawback of this approach is obviously the processing time. This type of algorithms usually require much more time, but since the score improves step by step, they have a better chance to get closer to the optimum.

2.1. Dynamic Programming

Dynamic programming (DP) is the direct method of finding the optimal solution for a sequence alignment problem. This method mentioned here refers to global alignment via the Needleman-Wunsch algorithm [15]. In the case of PSA, the algorithm works as follows: First, each sequence is placed in column and row indexes of an N by M table. N stands for the length of first sequence and M for the second one. Each space in the table is filled according to the updating rules that uses the information obtained from one step left, upper-left and upper values. These rules also involve the scores from a substitution matrix and a gap penalty scheme. After all spaces are filled

from top left to bottom right, the path that has led the way to the end is traced back. The gaps are then added to each sequence according to this path.

MSA version of this method uses N-dimensional dynamic programming, where N stands for the number of sequences. In this version, the same procedure explained above is applied on an N-dimensional table. The drawback of N-dimensional dynamic programming is the fact that the search space increases exponentially as the number of sequences increase. It is shown that finding the optimum result in MSA is an NP-complete problem [16], which in turn makes the dynamic programming approach for MSAs practically inapplicable.

2.2. Progressive Approaches

Progressive MSAs are heuristic methods which take the initial similarities between the pairs into account and build the process upon them. This approach was first invented by Hogeweg [17], then became the most widely used one among the others. It relies on a generated guide tree that places the sequences to the branches of the tree, and then aligning the sequences following the order of tree. The total time is relatively very short, since each sequence is visited only once during the alignment process. Some examples of progressive alignment algorithms are: CLUSTAL [18], CLUSTALW [19], DIALIGN [20], KAlign [21], T-Coffee [9] etc.

CLUSTALW is the most widely used one among the progressive approaches. This algorithm first aligns each pair using dynamic programming and creates initial weights assigned to them. The guide tree then is generated using these pairwise weights with the Neighbour-Joining [22] method. Finally, the sequences are aligned from the tip of the tree towards the root gradually in one run. While this algorithm provides fast and effective results, it has two major drawbacks: The first one is its greedy nature of not changing the positions of previously placed gaps, which might result in getting stuck at a local minimum. The other drawback is the high dependency of the algorithm to initial PSAs. If an error occurs in the first stages, it propagates through all the stages and in turn affects the result [7].

T-Coffee is another commonly used algorithm. This method differs from the previous one in initial weighting and scoring. For determining the weights, the algorithm uses a mixture of global and local pairwise alignment and produces the pre-weights. The process, which is called library extension, then recalculates these weights with the data from each remaining sequence. As a result of this, the initial pairwise weights reflect some of the information from the whole library [9]. This process also generates a scoring scheme where a unique score is attached to each residue pair. These mapped scores then replace a substitution matrix in the final alignment stage. Hence, as a result of these changes, some drawbacks of CLUSTALW are offset, making T-Coffee a better tool for scoring. These changes, however, increase the total runtime of the algorithm (as put forward by the tradeoff), since the library extension operation adds extra processing time to the algorithm.

2.3. Iterative Approaches

The basic idea of iterative approaches is to enhance the result by adding extra iteration steps. The iteration number and the stopping criterion vary in different algorithms. These enhancement steps are not restricted a priori; they can be anything that can potentially improve the result. The guide trees of progressive alignment or the parameters of dynamic programming can be post-processed or full stochastic algorithms like genetic algorithm or simulated annealing can be used as an iterative method.

MUSCLE [23] and MAFFT [24] are some examples of iterative algorithms where guide tree is re-estimated. In MUSCLE, after the progressive alignment is finished, the refinement process starts. The guide tree is cut into two and the profiles remaining in the sub-trees are aligned to each other. If the resulting MSA shows a better score, it is kept, otherwise it is discarded. This continues until a stopping criterion is met. MAFFT also consists of a progressive and an iterative partition. The progressive stage uses a method involving fast Fourier transform (FFT) to identify the homologous parts and reducing the runtime of DP. In the iterative stage, refinements are done by splitting the alignment into two groups and realigning them.

Stochastic MSA algorithms are specific types of iterative approaches which do not rely on any heuristic predetermined method. The initialization of these algorithms is usually fully random. These algorithms try to enhance the objective function in each iteration. The enhancements are applied by modifying the sub-solutions using some rules, which are called 'movements' in these approaches. This randomness provides a level of robustness, but in turn the run-time increases drastically because of the gradual nature. However, since the computational speeds have been increasing in the past decade, we started to encounter more and more stochastic applications in the literature.

A commonly used stochastic approach in MSA is genetic algorithms (GA). The main idea of GA is to imitate the natural selection idea of the evolution theory to find the optimal solution to a problem. The process starts with generating an initial population of random solutions. Then, through some movement operators called "mutations" and "crossovers", new solutions are generated. The new solutions build the next generation and this process continues until the population converges so that no major improvements can be obtained. In each generation, the newborn solutions are called children, whereas the previous ones are called parents. Crossover operation refers to taking some properties from each parent of two and producing the new child by mixing them in some way. Mutation involves only one parent, where some property of the parent is adjusted, usually randomly, to produce the new child.

One of the earliest approaches that uses GA in MSA is SAGA, which uses WSPS as its objective function [25]. SAGA is a popular algorithm that is widely used as a reference to other algorithms that involve GA. SAGA accommodates 22 movement operators in total, of which 2 are crossover and 20 are complex mutations. The algorithm uses a technique in which half of the population that have the worst fitness values is replaced in the next generation. The test results that were derived at that time showed that they were compatible with the other popular MSAs. MSA-GA [26] is another GA approach that uses WSPS as the objective function. MSA-GA uses fewer operators compared to SAGA, and the initial population is not produced fully randomly but seeded from PSA results of the input sequences.

VDGA [27] is another GA approach, in which the authors use vertical decomposition procedure. This procedure is basically dividing the sequences vertically into two or more, solving them individually and then finally recombining them. The same authors also suggested an algorithm called GAPAM [28] in which two new mechanisms to generate the initial population were proposed. The first one is to generate guide trees with randomly selected sequences and the second is shuffling the sequences inside trees. These two algorithms also use WSPS as the objective function. Some other examples in the literature combine GA with other optimization techniques. One of them is RBT-GA [29] where GA is associated with the novel rubber band technique. This algorithm also uses dynamic programming for improving the quality of the results. Another algorithm named GA-ACO [30] combines Ant Colony Optimization with GA. This additional method of optimization is used after the GA is implemented to avoid getting stuck at local maxima.

All algorithms introduced in this section consist of only one objective function. In recent years, MSA started to be treated as a multi-objective optimization problem. This means that multiple criteria have to be optimized separately and neither of them should be sacrificed while the others are improving, resembling the Pareto criterion. MO-SAStrE [31] is an example of multi-objective GA that uses totally conserved columns, non-gaps percentage and STRIKE [32] score as its three objective functions. Moreover; in this algorithm, the initial population is not generated randomly but seeded from some well-known progressive algorithms. Another algorithm from the literature, which is called MSAGMOGA [33], uses three different objective functions: Minimization of gap penalties, maximization of a similarity measure, which is mentioned in the algorithm, and maximization of the support measure, which defines the inclusion of number of sequences in an alignment that increase the quality. HMOABC [34] and H4MSA [1] by Rubio-Largo *et al.* are the last examples that use multi-objective GA. They both share two objective functions that are respectively WSPS and TCS. The next chapter will be devoted to intensively explain H4MSA.

3. A MULTI-OBJECTIVE MSA ALGORITHM: H4MSA

H4MSA is a multi-objective MSA algorithm, formulated by Rubio-Largo *et al.*, which uses a type of genetic algorithm [1]. H4MSA is used as the reference test algorithm for the hardware accelerated WSPS function proposed in this work. The two main reasons that H4MSA is chosen as the test algorithm are:

- H4MSA is an iterative algorithm that uses WSPS (importance of WSPS in terms of hardware design is explained in next chapter).
- H4MSA is a relatively recent algorithm and claims to show a good performance compared to the existing works in the literature.

In the following sections, the algorithm is explained step by step in detail. Note that, the results of the algorithm will be further discussed in Chapter 5.

3.1. Objective Functions

The two objective functions that are improved through the whole algorithm are WSPS and TCS, both explained above in the previous sections. Equation 3.1 shows how the sequence pair weights are calculated for the WSPS. LD refers to Levenshtein distance between two unaligned sequences. Levenshtein distance between two sequences is the minimum number of symbol edits needed to change one sequence into the other [1]. By protein sequences these edits are insertions, deletions and point mutations.

$$W_{i,j} = 1 - \frac{LD(s_i, s_j)}{max(|s_i|, |s_j|)} \tag{3.1}$$

Each solution in the algorithm, which is basically an alignment, is defined as x_i where i is the index of the solution within the population. Each objective function that corresponds to each solution is defined as $f_1(x_i)$ and $f_2(x_i)$ which can be shown in a vector $F(f_1(x_i), f_2(x_i))$ and yet, the multi-objective problem can now be defined as maximizing this vector F. Since each alignment in the population gets two scores, it is not a straightforward task to compare the result of one alignment to another.

Because the alignments must be sorted from best to worst in each generation, a metric for comparison is required. In H4MSA, a comparison method called "Pareto-domination" is used as this metric. Given a vector $v_1(a_1, b_1)$ and another vector $v_2(a_2, b_2)$, by definition, the vector v_1 is said to dominate v_2 if and only if $((a_1 > a_2)$ and $(b_1 \geq b_2))$ or $((a_1 \geq a_2)$ and $(b_1 > b_2))$. Therefore, for any two solutions x_i and x_j, x_i is said to dominate x_j ($x_i \preceq x_j$), if $F(x_i)$ dominates $F(x_j)$ [1].

It is easily seen that there are possible cases where neither of the solutions dominates the other. Even more, there may be more than two solutions that cannot be dominated by each other. All these different groups of non-dominated solutions within the population join separate sets called non-dominated fronts. Hence, the sorting operation becomes front by front sorting instead of sorting of individuals, which is a technique called non-dominated sorting derived from NGSA-2 [35]. Furthermore, the same work uses a metric called "crowding distance" assigned to each member measuring the diversity parameter within the front it is a member of with respect to its own position in there, to develop an additional comparison criterion between the members of the same non-dominated front.

After the population is sorted as fronts, the best front among them gets a special name called "Pareto-front" (PF). The Pareto-front basically contains all the solutions that are not dominated by any other solution in the population. With these definitions, the main objective of H4MSA can now be updated as maximizing the overall score of the Pareto-front.

3.2. Main Algorithm

In this section, the main flow of the H4MSA algorithm is explained step by step. A corresponding flowchart is found in Figure 3.1. The inputs of the algorithm are as follows:

- Set of k unaligned sequences S.
- Substitution matrix.
- Weight coefficient of gap openings g_o.
- Weight coefficient of gap extensions g_e.
- Number of memeplexes m.
- Number of solutions in each memeplex n.
- Number of evolutionary steps N.
- Stopping criterion.

On the other hand, the main outputs of the algorithm are all the solutions found in the Pareto-front by the time the algorithm ends.

The algorithm starts by calculating the pairwise weights ($computeWeights$) and generating P randomly aligned solutions ($generateRandomAlignments$) of the unaligned input S, where $P = m \times n$. Next, the initial Pareto-front ($updatePF$) is created and the algorithm steps into the main loop.

The main loop is the block where the evaluations and movements are executed until the stopping criterion is met. Loop starts with the non-dominated sort of the population ($nonDominantSorting$). Then the operation called "$divideIntoMemeplexes$" is executed. This operation splits the sorted populations into sub-groups called memeplexes, which is a method that is derived from the algorithm called "shuffled frog-leaping algorithm" [36]. The method is a memetic metaheuristic based on evolution of memes carried by the interactive individuals, and a global exchange of information among themselves. Operation distributes the best m solutions of the sorted population to each of the memeplexes, then it moves on with the next m solutions and continues

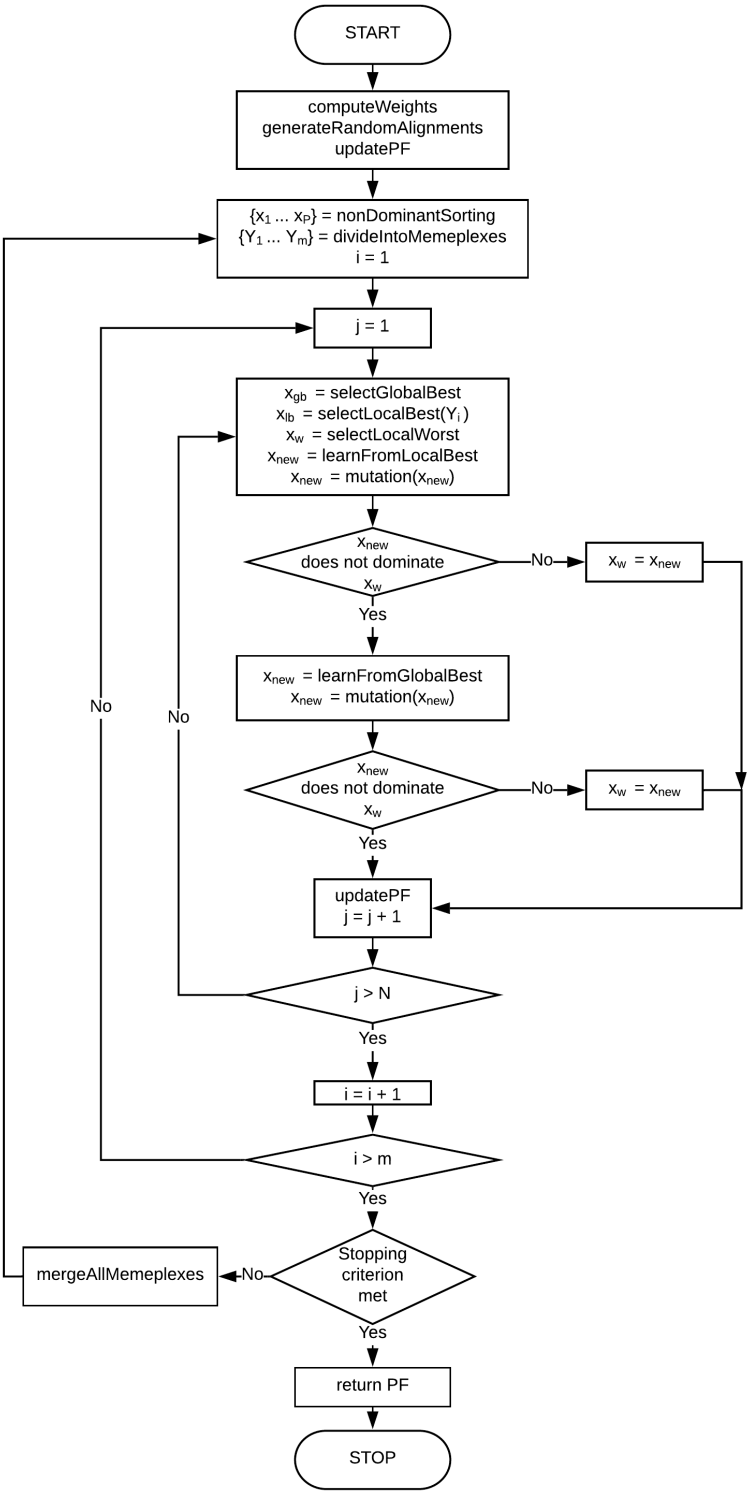

Figure 3.1. Flowchart of H4MSA algorithm.

with this approach until all members are distributed. Each memeplex is represented by Y_i in the flowchart and contains n solutions when the execution of the operation terminates.

In the next step, m separate loops with N iterations are repeated. The outer loop iterates through the memeplexes. Inside the inner loop, the following operations are executed within the selected memeplex by the outer loop. First, the global best solution is selected ($selectGlobalBest$) and labeled as x_{gb}, then the local best ($selectLocalBest$) and the worst solutions ($selectLocalWorst$) are determined within the current memeplex, which are labeled as x_{lb} and x_w respectively. After that, local worst and local best solutions are merged via crossover operation, which is named as "$learnFromLocalBest$" and produces the new child x_{new}. The new solution is then mutated through 4 different operators. Next, the new solution is compared to the local worst. If the new solution dominates the local worst, it is added to the memeplex and the local worst is deleted. However, if x_{new} does not dominate x_w, then the whole crossover and mutation process is repeated between the global best solution and x_w this time. In this case, the crossover is named as "$learnFromGlobalBest$" by the authors. The next section will be explaining all the movements mentioned above in more detail.

This evaluation process is repeated N times by each memeplex. Note that, at the end of every N iterations, the PF is updated, so that the best aligned sets are always ready to be obtained as outputs. After every memeplex is scanned through, they are all merged together so that an unsorted population is recreated ($mergeAllMemeplexes$). Then, the algorithm turns back to the beginning of the main loop and all the operations are repeated. After the stopping criterion is met, the algorithm terminates, and all the aligned sets in the Pareto-front are returned as outputs.

3.3. Movement Operations

Crossover movement is basically cutting two sequences vertically into two and then connecting one side of the first with the other side of the other. The place where the vertical cut occurs is selected randomly on the first parent alignment by the

algorithm. The righthand side (first parent) of the cutting position is selected as the right portion of the new (child) alignment. The order indices of the closest residues to the left of the cutting position are held in memory. Then by the second parent, the portion from the beginning until the residues, with the indices found before, is selected as the left side of the new alignment. These two portions are then merged together as shown in Figure 3.3. Note that if any void space appears between the two portions, it is filled with gaps entirely.

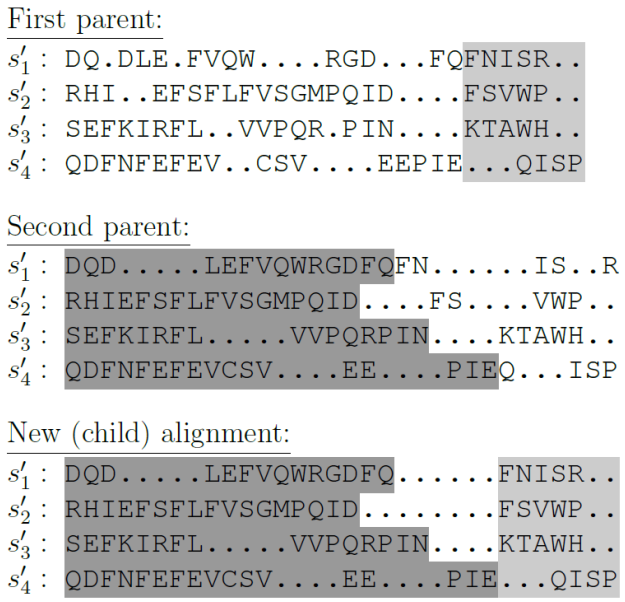

Figure 3.2. Movement: Crossover.

After the crossover, mutations are executed one by one on the new alignment. First mutation movement is called "move a block". In this movement, a block of residues or gaps are selected and moved one step left or right within the alignment. A block of gaps (or residues) is defined as a set of overlapping groups of gaps (or residues) from one or more sequences, each group being delimited by a residue (or a gap) or an end of a sequence [25]. In Figure 3.3, move a block operation of an arbitrary alignment is shown. Note that the direction of the movement and the selection of any block is done randomly.

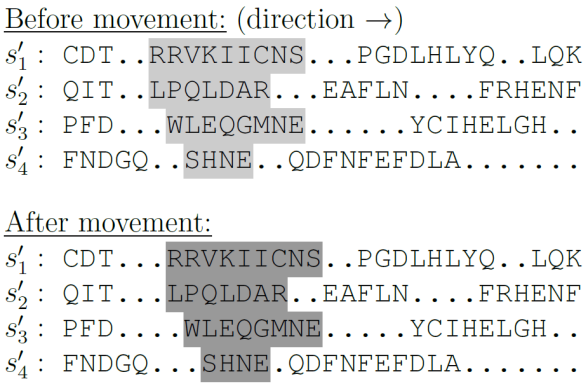

Figure 3.3. Movement: Move a block.

Next mutation movement is called "merge two groups". In this movement, a group of residues or gaps is selected randomly from one random sequence of the alignment. After that, the selected block is merged with the closest neighboring block as depicted in Figure 3.4.

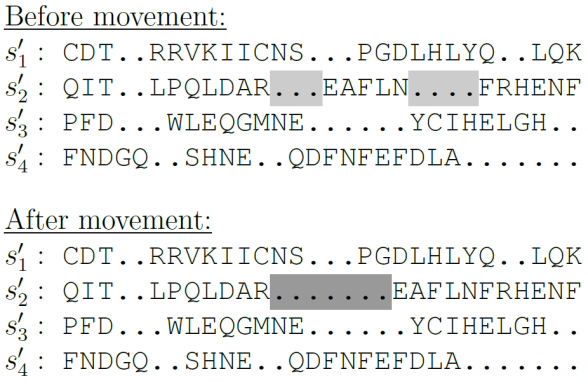

Figure 3.4. Movement: Merge two groups.

The third mutation is named "divide a group". A random group of residues or gaps from a sequence is selected like in the previous movement. Then, it is split into two approximately same sized new groups. Figure 3.5 shows an arbitrary example of this movement.

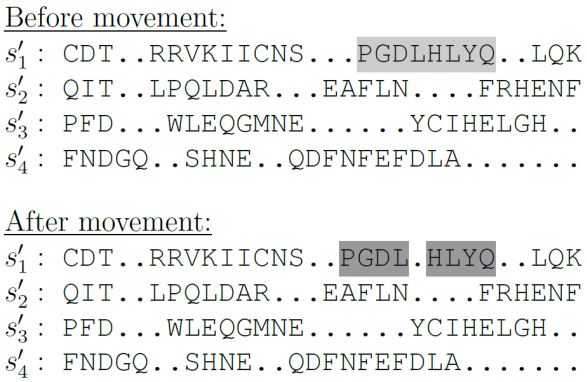

Figure 3.5. Movement: Divide a group.

The last movement is not a mutation, it can be considered as a correction operator. It is called "Compact Alignment" by the authors. The function of the movement is to delete all the columns which contain all gaps, as shown in Figure 3.6. This correction is done in MSAs because the gap columns do not have any information on them; therefore, they should be removed.

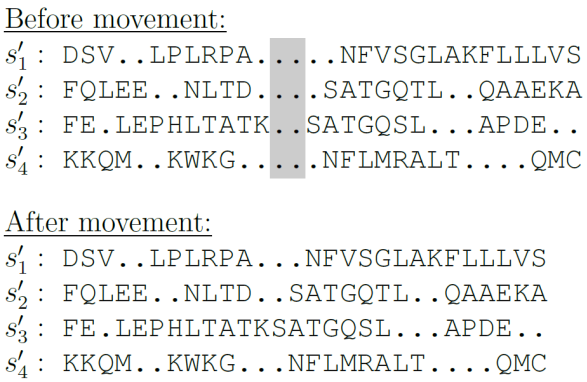

Figure 3.6. Movement: Compact alignment.

4. HARDWARE ACCELERATED WSPS

The hardware acceleration proposed in this work is the evaluation of WSPS function of multiple sequence alignment, by the designed hardware. This choice is made after deep observation of various MSAs in the literature. Hence, the main three reasons for the choosing the acceleration of the WSPS among other functions of MSAs can be listed as follows:

- WSPS is widely used as a scoring function in most of the MSAs.
- The structure of WSPS provides a high parallelization that is very suitable for hardware design.
- WSPS mostly constitutes a high amount of processing time within the most of the MSAs. This makes it a good choice for acceleration, if an increase in overall speed is desired.

After most of the popular MSA algorithms in the literature were reviewed, it was observed that a good amount of recent works involves an iterative stage for post-refinements of the results. This basically means that they should have a scoring function to measure the outcomes of the result, so that they can further improve them. Even though different types of scoring systems have been used in various algorithms recently, WSPS is still a widely preferred scoring function. Therefore, an accelerated WSPS function can provide all these existing algorithms with additional speed upgrades if it is incorporated into them.

The second reason to choose WSPS is its structure. If a computational function requires many independent operations, which means that fewer operations that are dependent to outputs of other operations, it signifies that the function is highly suitable for hardware acceleration, since hardware acceleration is mostly about parallelization. Looking at the structure of WSPS, it is easily seen that most operations in WSPS are independent additions, multiplications, and table lookups. This makes WSPS an even better candidate.

Finally, it is important to mention that the time consumption of WSPS is also very crucial in this context. The previous two reasons are not sufficient by themselves to choose WSPS for acceleration. The percentage of time consumption of WSPS within the main algorithm plays an important role as well. If this percentage is not high enough, the acceleration on the main algorithm will not be high enough to compensate the hardware overheads; thus, there would be no meaning of accelerating it. Considering this aspect, we measured the percentage of WSPS in overall computation time in our test algorithm H4MSA and observed that this ratio is satisfactory. The results of these tests are presented in the next chapter.

In the following section, the possible approaches are discussed and the best among them is explained as the main method of the hardware design. In the second section, logic circuit design of the hardware acceleration block is described in detail. Note that the circuit figures in this chapter do not involve all the actual connections and sub-structures, they are illustrated to give a general idea of their functions.

4.1. Comparison of Design Approaches

The straightforward design for WSPS would be parallelizing the scoring of every single symbol pair, since every pair score is independent from each other. However, the major drawback of this approach is the need for a tremendous amount of hardware. Because every symbol pair needs a lookup block, an adder, a multiplier, and additional logic blocks, the resource demand would be very high. This reason alone prohibits using this approach, but there is also an additional flaw which is the interconnection problem between the input data and main block. Since the platform that is used in this work has a limited data width for the input, the transportation of that amount of data simultaneously would also be inapplicable.

These limitations necessitate a pipelined design, where the operations are divided into timed stages rather than executed all at once. One way to divide alignments into pipelined stages is to split them vertically or horizontally into two or more sub-alignments and treat the sub-alignments sequentially. Since the lookup operation is

done in the vertical direction, horizontal splitting becomes inefficient because it may cause additional waiting during the process. Vertical division on the other hand is the more suitable one.

The decision that needs to be made for vertical splitting is the choice of the number of pieces the alignment should be divided into. The limitation of input interconnection also plays an important role here. In order to obtain the maximum speedup, the full capacity of the interconnection should be used. Therefore, alignments with fewer sequences should be divided into larger pieces, while the alignments with more sequences into smaller ones. On the other hand, after a certain number of sequences, the column size gets larger than the interconnection width and sending one column can only be accomplished in multiple steps. Hence, a general-purpose approach for this problem would be problematic, as these varying parameters increase the complexity of the design. Considering that the MSA applications always aim for alignments with larger sequence numbers, it is decided to use the maximum number of splitting, which is sending the input alignment one column at a time. With this approach, the maximum speedup is obtained when the column size (i.e., sequence number of the alignment) is equal to the interconnection width.

The simple way to design this approach is to create comparison lookup blocks and pass the sequences column by column through these blocks. Although this method seems to be efficient, there is still room for further improvement. In this approach the lookup block number is equal to the pair number in the alignment. Equation 4.1 shows the relationship between pair number PN and sequence number sn within the alignment. The quadratic relationship between PN and sn makes the required number of memory bits for the lookup blocks prohibitively large after 15 - 20 sequences. Since average protein sets that are used in MSA applications involve much larger sequence numbers, this approach becomes unsuitable.

$$PN = \frac{sn(sn-1)}{2} \tag{4.1}$$

In this work, we developed a better technique with the same column by column approach. In contrast to the previous one, the number of lookup blocks in this technique is now equal to the number of sequences. The structure of the comparison block, which defines the core behavior of our novel design, is explained as follows: The block contains k fixed registers ($fReg_i$) and k shift registers ($sReg_i$), where k is the number of sequences and i indicates the order index. The function of these registers is to hold the symbol data (sym_i) and send them to be evaluated in the next stage. Shift registers have an additional function: They are connected in a way that the symbol values inside $sReg_i$ can be shifted to $sReg_{(i+1)}$ at each time step. They are also connected in a circular scheme, so that the value in $sReg_k$ is shifted back to $sReg_1$. At each time step, the symbol in the fixed register $fReg_i$ is compared and scored with the symbol in the shift register $sReg_i$. For example, the symbols in $fReg_2$ and $sReg_2$ or the symbols in $fReg_k$ and $sReg_k$ are compared and scored in parallel.

The process of the block, on the other hand, is described as follows: Initially, when a new column arrives at the block, every symbol sym_i of the column is loaded into the corresponding $fReg_i$. At the same time, the exact copies of the same symbols are also placed in the shift registers, where sym_i is loaded in $sReg_{(i+1)}$ (sym_k is loaded in $sReg_1$). At the next time step, symbols in the shift registers are shifted one step forward so that the symbols in the fixed registers are scored with different ones. This shifting process is executed until all the symbol pairs are compared and scored with each other. After that, the new column is obtained by the block and the process repeats itself until all the columns are finished.

The number of shifts that is needed in an alignment with k sequences is obtained by experimentation. In Table 4.1, a column comparison and scoring case with 5 sequences is represented. The first column of the table shows the symbol data contents of

the fixed registers. The remaining columns illustrate the contents of the shift registers in consecutive time steps. Note that, at each time step, symbol data is shifted one step downwards, which corresponds to shifting from $sReg_i$ to $sReg_{(i+1)}$. It can be observed that with $k = 5$, two shifts are sufficient for all comparison combinations. In Table 4.2, another case with $k = 6$ is given, where three shifts are required. Note that, in this case the bottom half of the last column has some redundant comparisons (they are shown in parentheses). This situation is dealt with the circuit design by ignoring the redundant comparisons. The last case with $k = 8$ is shown in Table 4.3, completing all shifts in 4 time steps.

Table 4.1. Symbol comparison operation with $k = 5$.

Fixed registers	**Time step 1**	**Time step 2**
sym_1	sym_5	sym_4
sym_2	sym_1	sym_5
sym_3	sym_2	sym_1
sym_4	sym_3	sym_2
sym_5	sym_4	sym_3

Table 4.2. Symbol comparison operation with $k = 6$.

Fixed registers	**Time step 1**	**Time step 2**	**Time step 3**
sym_1	sym_6	sym_5	sym_4
sym_2	sym_1	sym_6	sym_5
sym_3	sym_2	sym_1	sym_6
sym_4	sym_3	sym_2	(sym_1)
sym_5	sym_4	sym_3	(sym_2)
sym_6	sym_5	sym_4	(sym_3)

Shift experiments show that the number of time steps required for scoring the whole column is approximately $k/2$. If the scoring is executed in a CPU where all operations are done sequentially, the theoretical total time needed for the whole function

Table 4.3. Symbol comparison operation with $k = 8$.

Fixed registers	**Time step 1**	**Time step 2**	**Time step 3**	**Time step 4**
sym_1	sym_8	sym_7	sym_6	sym_5
sym_2	sym_1	sym_8	sym_7	sym_6
sym_3	sym_2	sym_1	sym_8	sym_7
sym_4	sym_3	sym_2	sym_1	sym_8
sym_5	sym_4	sym_3	sym_2	(sym_1)
sym_6	sym_5	sym_4	sym_3	(sym_2)
sym_7	sym_6	sym_5	sym_4	(sym_3)
sym_8	sym_7	sym_6	sym_5	(sym_4)

is $\frac{kL(k-1)}{2}$, where L is the length of the alignment. The designed method reduces the time to $\frac{kL}{2}$. On the other hand, the computational complexity is reduced from $O(k^2L)$ to $O(kL)$. Note that the calculations are done theoretically, where all the other parameters are fixed. The practical application however, does not show the same behavior, which is analyzed in the next chapter.

4.2. Circuit Design

In this section, the full logic circuit design of the WSPS block is presented. Inner sub-blocks, input output logic and timing controls are explained in detail. The input data for initialization of the function are listed as follows: Set of unaligned sequences, substitution matrix, weight coefficient of gap openings, weight coefficient of gap extensions, pre-calculated pair weights. After initialization, a start signal is sent, which is followed by the column by column transportation of the unaligned set into the hardware. When the transportation terminates, another signal is sent indicating that all input data are successfully transported. On the other hand, the outputs of the hardware are the WSPS value and a data ready signal, which indicates that the entire operation is completed.

The input data accepted by the hardware is the binary representation of the aligned set. This representation is created by replacing the residues with 0s and the gaps with 1s. The reason for the usage of this representation is to reduce the size of alignment. Sending every symbol would need at least 5 times larger data since the symbols should be represented by a minimum of 5 bits. Instead, every new alignment is sent as the so called "binary alignment" and the hardware chooses the next symbol of the alignment, according to the input order of the binary data. A binary alignment example is shown in Figure 4.1. The alignment in the figure has 4 sequences, hence each column input data words contain 4 bits in this case. The first word that enters the hardware is "0010", the second word is "1010" and so on.

An aligned set:

s'_1 : RAEMLRDMVVYNRVDLSKLVTHVYH.....
s'_2 : ....SVPMLV....ADF...MCKKFALDPL
s'_3 : DYPLLLEM....VSAGF...RCNVKQLV..
s'_4 : H..FVKTVSITSRNYQL...LSKLITHRPR

Binary representation:

s'_1 : 00000000000000000000000011111
s'_2 : 11110000001111000111000000000
s'_3 : 00000000111100000111000000011
s'_4 : 01100000000000000111000000000

Figure 4.1. Binary representation of an aligned set.

The block diagram of the WSPS hardware is shown in Figure 4.2. The entire operation can be briefly explained as follows: Input binary data is accepted at "Unaligned sequences block" and "gap count control block". "Unaligned sequences block" then sends sequence symbols to the next block using the information obtained from input data. "Symbol comparison and scoring block" uses the symbol pairs to produce the symbol pair scores and these are then summed up correspondingly inside the pair sum registers at "Pair sum and gap count registers (PSGCR) block". Parallel to this process, "gap count control block" detects the gap openings and extensions in the input sequences, then sends the control signals accordingly to the gap count counters (also registers) in PSGCR registers block. After the calculations are completed, PSGCR block sends the register values to the "output block", where they are multiplied by the

weights and the coefficients obtained from the “weights block”. Lastly, all the weighted partial values are summed up together and form the WSPS.

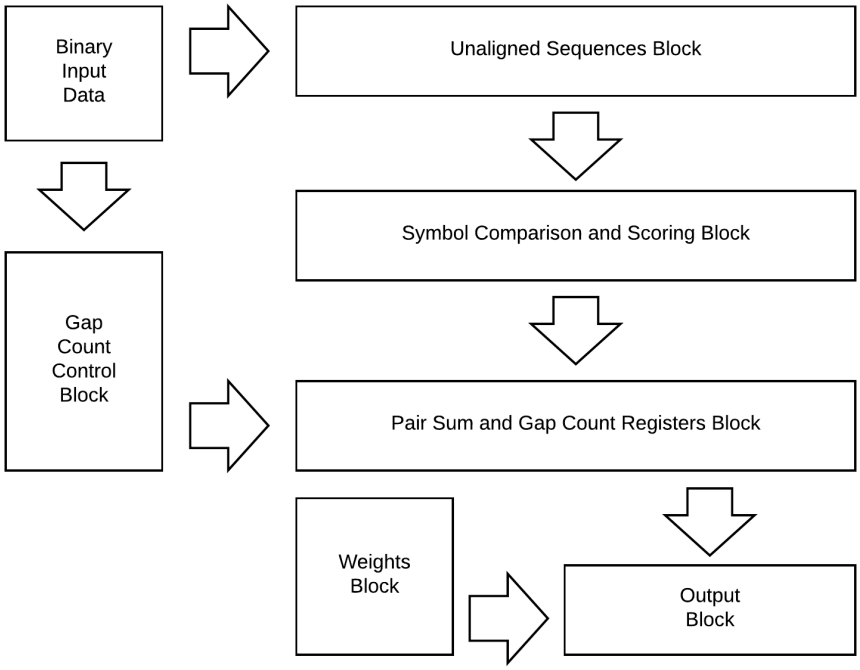

Figure 4.2. Block diagram of WSPS hardware.

The whole operation is also divided into four phases (states), which are listed as follows: “Reset state” is the first phase after the start signal, where all the registers are reset, and then the pair weights and the gap coefficients are loaded into corresponding registers. All other functions are in idle state during this time. Second state is named as “scoring state”, in which pair sums are calculated and gap openings/extensions are counted. After these operations are completed successfully, the hardware gets into the next phase called “multiplication state”. In this state, only the weights, PSGCR and output block are active, in which the multiplication of partial values occurs. The final state is the “output ready state”, where WSPS is created and ready to be read. In this state, all the other operations are terminated and cannot affect the last value of WSPS anymore, hence this state can only be broken with the new start signal.

The structure named “binary input data” in the block diagram represents the binary data source from outside; this structure is not a part of the actual hardware. One of the blocks that gets these data is the unaligned sequence block. The block consists of k memory blocks and counters, where k corresponds to the number of sequences in the set. Counters are connected to the addresses of the memory blocks, which means they control the symbol output that the memory block will send in the next time step. The memory blocks are SRAM and each of them contains one of the unaligned sequences. Additionally, the binary data are propagated through the counter to the memory block, so that the memory block can decide whether to output a residue or a gap. In Figure 4.3, the connections of one of the counters (“symbol counter”) and the corresponding memory block (“sequence data memory”) is shown. If the reached symbol in the current time step is a residue, which means a binary ”0” at the $binIn$ port of the counter, the counter is triggered to increment. This means that one residue has moved forward in the unaligned sequence, and the memory block outputs the next residue. If, however, the current input is a gap, counter does not increment. The binary data are obtained from the $binOut$ port and the memory block sends out a gap symbol, which is basically an all zeros word.

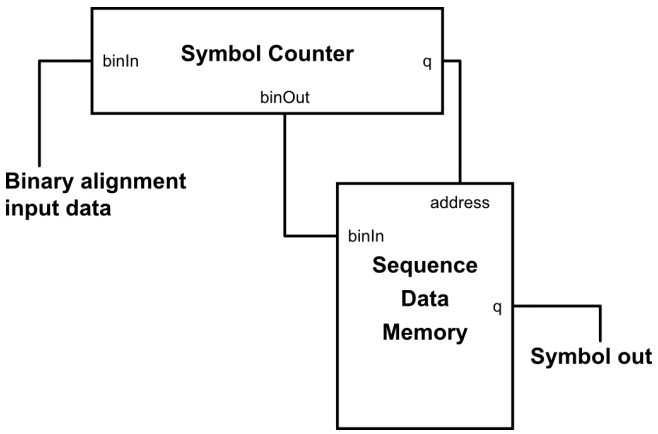

Figure 4.3. Unaligned sequences block logic circuit.

The next block is called “symbol comparison and scoring block”. The block consists of k fixed registers, k shift registers, and k substitution (lookup) matrix memory blocks. This block gets k symbols from the previous block and executes the loading and shifting operations as explained in the previous section. After the initial one cycle of

loading, the block halts the forward movement of the whole pipeline, until the shifting operation is completed. This routine is controlled by a counter which is called "shift counter". It counts up to the total shift number and resets itself after that. At each reset, it sends a global load signal, so that the whole pipeline process proceeds one step forward. In other cases, it enables the shifting operation and the pipeline halts until this operation is completed. At each cycle (regardless of being a load or a shift cycle), the two outputs of each fixed and shift register pairs are concatenated together. The result is sent to the address inputs of the lookup memory blocks. The matrix memory blocks are also SRAMs and contain the whole information of the substitution matrix inside themselves. The memory structure of these blocks is pre-arranged so that the concatenated output of the registers always points to the right pair score value of the substitution matrix. The memory blocks then output the pair score to be sent to the next block.

Figure 4.4 shows a "symbol comparison and scoring block" structure with $k = 3$. Each fixed and shift register is named $fReg$ and $sReg$, respectively. The symbol inputs are the sIn ports and the symbol outputs are the $sOut$ ports. The circular shift connection scheme can be observed by the shift registers, where corresponding shift inputs are called $shiftIn$ and shift outputs are called $shiftOut$. As seen in the figure, "shift control counter" only outputs a control signal named "loadShift" and although it is not shown in the figure, this signal is connected to all registers in the block. Finally, the remaining block type called $subMat$, which is depicted in the figure, is the substitution matrix memory block.

The next structure, which is called "Pair sum and gap count registers block" mainly consists of all the registers where the current pair sum and gap count values are held. There are $k \times sn$ pair sum registers and $2k$ gap count registers in total, where k is the number of sequences and sn is the number of shifts. The function of the gap count registers in the "scoring state" is explained in the next block. On the other hand, the function of pair sum (PS) registers in the same state is represented as follows: In "scoring state" the PS registers are split into groups. Each group is connected to the output of a substitution matrix block. Since one substitution matrix block outputs sn

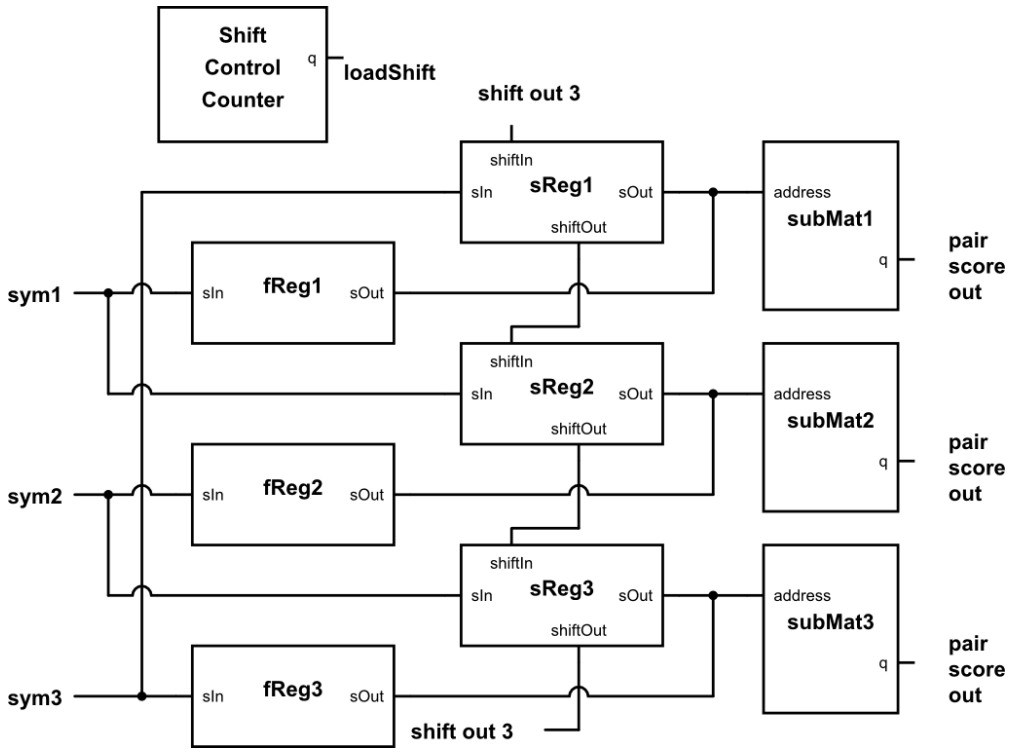

Figure 4.4. Symbol comparison and scoring block logic circuit.

different symbol pair scores in sn consecutive cycles, these scores should be summed up in sn different registers, which makes the number of registers in each block sn. Figure 4.5 shows the $sn = 3$ case. As seen in the figure, registers are connected in a ring formation. At each cycle, the previous pair sum data in $psReg3$ and the current symbol score from the substitution matrix block ($subMat$ score data) are added, where their output is sent to $psReg1$ simultaneously. The other values in the registers are also shifted one step forward in the same direction. The multiplexer (mux) ensures that the ring shift function continues as long as the hardware is in "scoring state" (i.e., the signal $sStage$ is 1). Once the state changes into "multiplication state", the signal $sStage$ becomes 0, hence the mux changes the input direction to the "reg group in" wire. As shown in the figure, this wire is connected to the previous register group and besides, the output of $psReg3$ is connected to the next register group. This means that all PS registers are now connected to each other as a line array, so that in the "multiplication state", the values can be shifted downwards through this array to the output block (the output of the last group is connected to the output block). Note

that, gap count registers in this block also form a line array in "multiplication state". Output of this line array is connected to the input of the first PS register group, so that they follow the PS values from one step behind and propagate to the output block.

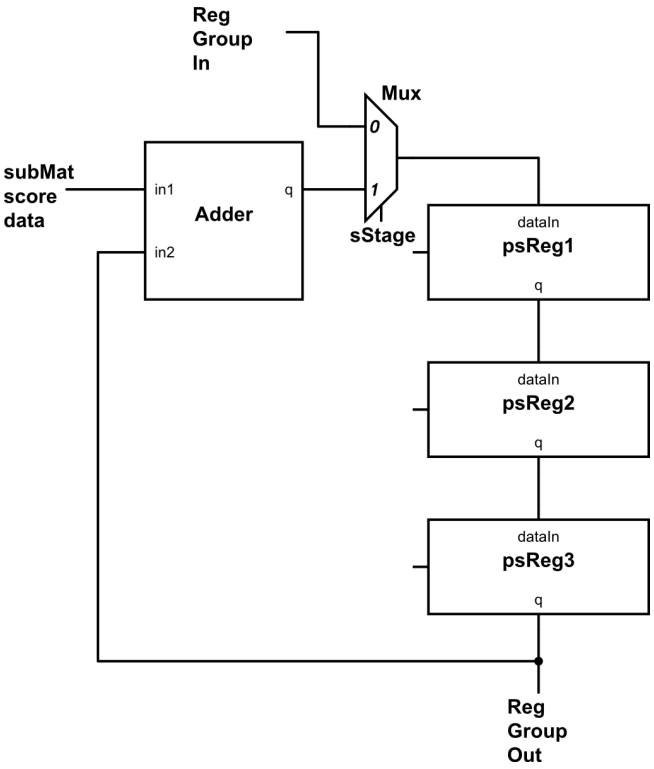

Figure 4.5. PSGCR block logic circuit.

In parallel to scoring operations, the "gap count control block" operates for the gap counting function of the hardware. The main purpose of this block is to get the input binary data and create the gap count control signals for the gap count registers. Figure 4.6 shows a part of this block which is responsible for one sequence. The circuit accepts the binary data at the input of $dff1$, so that at each clock cycle, the next bit of the sequence enters the input of $dff1$ (D flip-flop), while at the same time the previous bit shifts to $dff2$. With the same behavior, the whole sequence propagates through the flip-flops. Therefore, through the whole operation, we can obtain every single consecutive binary pair at the outputs of the flip-flops. The sub-block named "gap count control logic" takes these pairs as inputs and detects the gap openings and extensions in the sequence. If both flip-flops have the value "1" (gap), that means a gap extension is detected. If $dff2$ has the value "0" (residue) and $dff1$ has "1" (gap),

that means a beginning of a gap group is detected, which basically corresponds to a gap opening. When a gap opening is detected, the sub-block sends an enable signal from the *goOut* port to the *cEn* (count enable) port of the *goCounter*, as a result the counter increments its value. When a gap extension is detected, the same procedure occurs over the *geOut* port and *geCounter*. There are total of $2k$ counters in the block, where each counter pair is responsible for a separate sequence. Note that the counters here are part of the PSGCR block, but to show the continuity of the operation, they are depicted in the same figure.

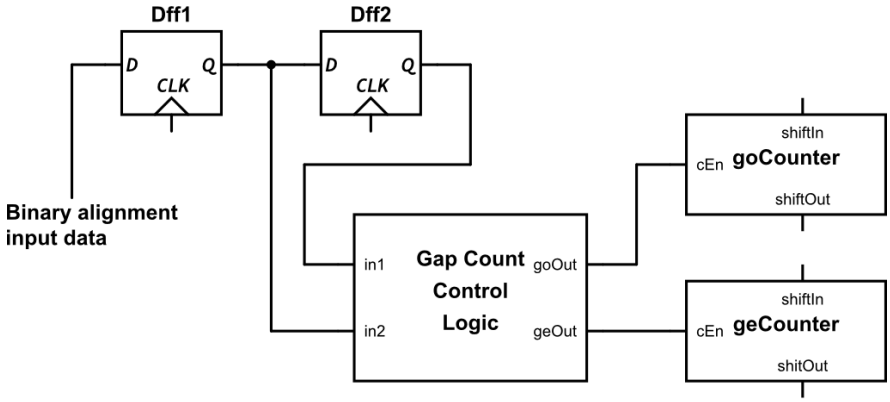

Figure 4.6. Gap count control block logic circuit.

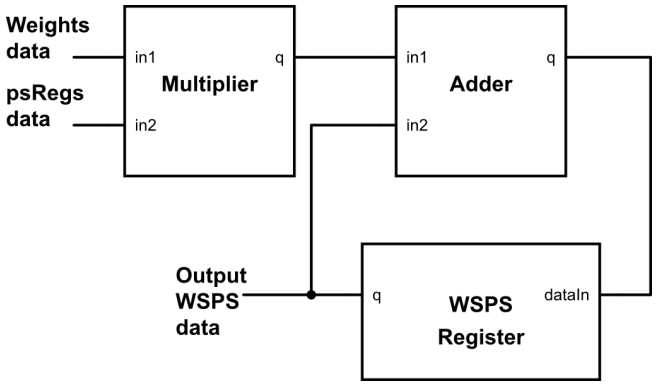

Figure 4.7. Output block logic circuit.

The last block is the "output block", which consists of a multiplier, an adder and a register that holds the WSPS value. This block is idle during the scoring state and begins its operation in the multiplication state. Figure 4.7 shows the connections of the block. At each clock cycle, the block gets a partial value (partial sum or gap count

value) from the PSGCR block and its corresponding weights from the “weights block”. Note that, if the value comes from a gap count register, the corresponding weight is the gap opening or extension coefficient. The two values are multiplied by each other and the output is summed with the value in the WSPS register. This operation continues until the multiplication state ends. When the output is ready, the hardware goes into the last state and no more changes can be made on WSPS register. The final result then can be read from the wire “output WSPS data”.

5. EXPERIMENTS AND RESULTS

In this chapter, results of the test algorithm H4MSA, and the outcomes of the hardware acceleration are examined in separate sections. Since it was not possible to access the H4MSA algorithm itself, it is reproduced with all the details explained in H4MSA article [1]. To produce the hardware accelerated version of the algorithm, the software WSPS function is replaced with our hardware design. Throughout this chapter, the software version of H4MSA that is reproduced in this work will be referred as "SVA" (software version of the algorithm), while the hardware accelerated version will be called "HVA" (hardware accelerated version of the algorithm). Note that, the original algorithm will retain the abbreviation H4MSA.

SVA and HVA are both tested with a widely used sequence set database called BALiBASE 3 [37]. The database provides an unaligned version and a reference aligned version of each sequence set it contains. The sets of sequences in the database are further organized in six subsets (groups) according to their families and similarities [1], which are named as RV11, RV12, RV20, RV30, RV40, RV50.

The platform that is used in this work is DE1-SoC Board [38]. The board is equipped with a CYCLONE V [39] SOC device, which consists of an FPGA fabric and a dual-core ARM Cortex-A9 CPU, which is named as "hard processor system" (HPS). SVA and the software partition of HVA are written in C language that runs on a Linux-based operating system in HPS. The hardware, on the other hand, is designed with Verilog HDL and constructed on the FPGA fabric of the SOC device. Note that, in all the tests, clock speeds of HPS and FPGA are 900 MHz and 50 MHz respectively.

5.1. Test Algorithm Results

The original H4MSA algorithm uses the parameter configurations as follows: $m = 5$ (5 memeplexes), $n = 20$ (20 solutions per memeplex), and $N = 10$ (10 loops per each memeplex). The stopping criterion is selected as 50 thousand evaluations. This means

that, after 50 thousand evaluations (the crossover and mutation process) are executed, the algorithm terminates and the results are printed out. The substitution matrix is BLOSUM62, the gap opening and extension coefficients are 6 and 0.85 respectively.

H4MSA uses the benchmark program called "bali score", which is provided by BALiBASE, for assessing the performance of the algorithm. This program gets the user aligned set as input, compares it with the corresponding reference alignment and produces two quality metrics called Q and TC. Q (quality) indicates the number of correctly aligned residue pairs divided by the number of residue pairs in the reference alignment. On the other hand, TC (total column) is the number of correctly aligned columns divided by the number of columns in the reference alignment [1].

The results of Q and TC scores, which are obtained by the original H4MSA article, are shown in Table 5.1. Note that, the name "H4MSA" that is presented in table is the version of the algorithm that involves an extra step called "local search". The one with name "H4MSA*" does not contain this step. The experiments are done by running the algorithm 30 times and obtaining the average of these results for each member of the dataset. The dataset is also tested with the known algorithms in the literature. The article shows that, results of H4MSA receives the best scores among them in most of the cases [1].

Table 5.1. Q and TC scores of H4MSA.

Name	**H4MSA Q**	**H4MSA TC**	**H4MSA* Q**	**H4MSA* TC**
RV11	0.7618	0.6007	0.5720	0.3532
RV12	0.9524	0.8905	0.8913	0.7302
RV20	0.9410	0.5571	0.8670	0.3347
RV30	0.8803	0.6349	0.7393	0.3963
RV40	0.9354	0.6682	0.8317	0.4535
RV50	0.8938	0.6364	0.7770	0.4660

The algorithm that is developed in this work (SVA) is the reproduced version of H4MSA without the local search (H4MSA*). For the test of the algorithm, same parameter configurations with the original H4MSA is used. When the sub-steps of the algorithm are examined in runtime, it is observed that the outputs of every movement and evaluations are correct. However, the obtained TC and Q values are lower than the ones that the original H4MSA claims. These obtained results are given in Table 5.2, where the first column indicates the names of the group, second and third columns show the results that we acquired. To obtain the results, 10 different sets from each group is run 30 times and the average result of every run is calculated.

Table 5.2. Q and TC scores of SVA.

Name	**SVA Q**	**SVA TC**
RV11	0.1201	0.0654
RV12	0.0832	0.0332
RV20	0.0973	0.0204
RV30	0.0674	0.0258
RV40	0.0934	0.0343
RV50	0.1105	0.0747

Possible reasons of the score differences between our SVA and the original H4MSA can be listed as follows:

- Some crucial details of the movements are not mentioned clearly (definition of gap block is problematic, whether merging of blocks during the movements are allowed or not is unclear, etc.).
- All the decisions by the movements (choosing the sequence, choosing gap or residue group, choosing the direction of the movement, etc.) are also said to be made randomly in H4MSA. However, the order of these decision and the randomness measures are not explained in detail.
- The step after obtaining a better solution is not explained clearly. It is mentioned that the new solution is replaced with the local worst in the memeplex, but

whether the new solution is re-sorted within the memeplex or whether it is valid to be used in the next evaluation or not is unclear.

The listed aspects above may have caused the score differences between the original H4MSA and the reproduced SVA. After trying different variants of the above-mentioned decisions, the results could not be improved.

5.2. Hardware Acceleration Results

The results of various experiments that are performed in the hardware accelerated version of H4MSA (HVA) are reported in this section. The tests mainly focus on the speed and timing comparison between the two versions and the accuracy of the results.

HVA is the variation of SVA, in which the WSPS hardware is replaced with the software version of the function. The remaining parts are the exact copy of SVA. The communication between the hardware and HPS is provided by the SOC device itself via a 32-bit wide interconnection bus. This basic flow of the communication is explained as follows: When the WSPS function starts, the start signal is sent from HPS to the hardware. Then, the software splits the binary alignment into 32-bit words and the alignment is sent one word at a time. This data transfer is made through a FIFO block; hence without the need of any extra communication protocol, hardware reads the continuous flow of the 32-bit words. When the transfer is over, an all 1s word is sent from the HPS, which terminates the further reads by the hardware. After that, the HPS waits for the output ready signal, and reads the final WSPS result.

The current version of HVA is highly parameterized for different number of sequences. This parameterization allows using alignment sets with even sequence numbers. To efficiently use the 32-bit interconnection, only the sets with sequence numbers that are factors of 32 are selected as inputs. Therefore, the test sets from the database are chosen with four different sequence numbers, which are $k = 4$, $k = 8$, $k = 16$ and $k = 32$. For each separate k, three different alignments with different lengths are chosen, so the total number of the test alignments is 12. Properties of the chosen sets

from the database are shown in Table 5.3. The first column states the name of the set, the second column indicates the sequence number, the third and fourth columns indicate the maximum and average residue lengths of the set respectively. For further comparison of the lengths, a graph is created in Figure 5.1, where the average lengths and the maximum lengths can be observed side by side. Note that, the database cannot provide enough number of sets for each sequence number. To solve this problem, sequence numbers of two sets are modified by adding extra sequences. This is a valid action, because the comparison of the scores with the reference alignments are irrelevant in the hardware experiments. To distinguish the modified sets, the letters "MD" are added at the beginning of their names.

Table 5.3. Properties of the selected sets.

Name	**Sequence Number**	**Max Length**	**Avg Length**
BB11001	4	91	86
BB12025	4	215	202
BB11004	4	456	421
BB11002	8	193	98
BB11032	8	403	295
BB12013	8	772	710
MD30017	16	370	315
BB20020	16	527	288
BB20001	16	697	295
MD20023	32	302	220
BB40030	32	614	349
BB20026	32	1016	604

The first step of speed comparison is to measure the processing times of both versions. In Table 5.4, processing time (in seconds) results of each set, which are run in both SVA and HVA, are shown. Two metrics called "score time" (ST) and "total time" (TT) are used to reflect the results. Total time is the total runtime that the whole algorithm consumes. Score time is the duration which the algorithm spends only inside

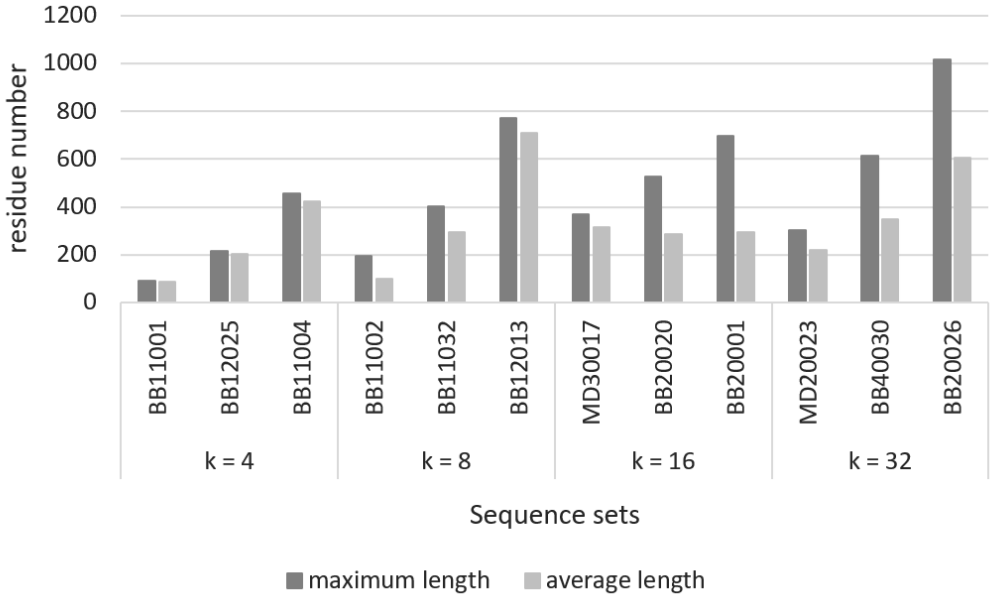

Figure 5.1. Comparison of sequence set lengths.

the WSPS function. In SVA, this represents the duration of software WSPS, while in HVA, this is the total time consumed by the WSPS hardware and the communication overhead together. Each set is run 10 times and the average results of the 10 runs are placed in the table. For accuracy, all the measurements are made with a hardware performance clock, which is implemented in the FPGA fabric.

The ratio of the score time within the total time is an important measure, because it shows the ratio of maximum processing time that can be theoretically reduced after the WSPS is accelerated. This means that, accelerating the WSPS is only meaningful for the overall speedup, if this measure is high enough. In Figure 5.2, the corresponding percentage values for the SVA are depicted. It is observed that, after a sequence number of 4, WSPS consumes almost all the processing time within the algorithm. This clearly shows that accelerating the WSPS function within H4MSA is an effective method for speeding up the whole algorithm. The same percentages are given in Figure 5.3, this time for the HVA, where the decrease of the values by all sets are clearly seen as expected. An additional observation is that, after sequence number of 8, the percentages are still over 90% despite the value drops. This provides further

Table 5.4. Processing time results of the sets.

Name	**ST SVA(s)**	**TT SVA(s)**	**ST HVA(s)**	**TT HVA(s)**
BB11001	7.814	10.19	3.423	5.934
BB12025	19.225	22.314	8.292	11.273
BB11004	43.878	46.855	19.851	23.364
BB11002	79.565	83.401	16.594	20.283
BB11032	206.54	210.91	43.158	48.952
BB12013	477.47	484.02	100.70	107.01
MD30017	1172.7	1182.3	120.81	129.23
BB20020	1191.1	1199.9	127.74	135.75
BB20001	1263.5	1271.9	129.11	136.25
MD20023	4513.2	4528.5	217.23	229.21
BB40030	8062.9	8082.3	425.26	439.46
BB20026	12721	12750	623.87	643.58

opportunities for extra accelerations for the sets with large sequence numbers.

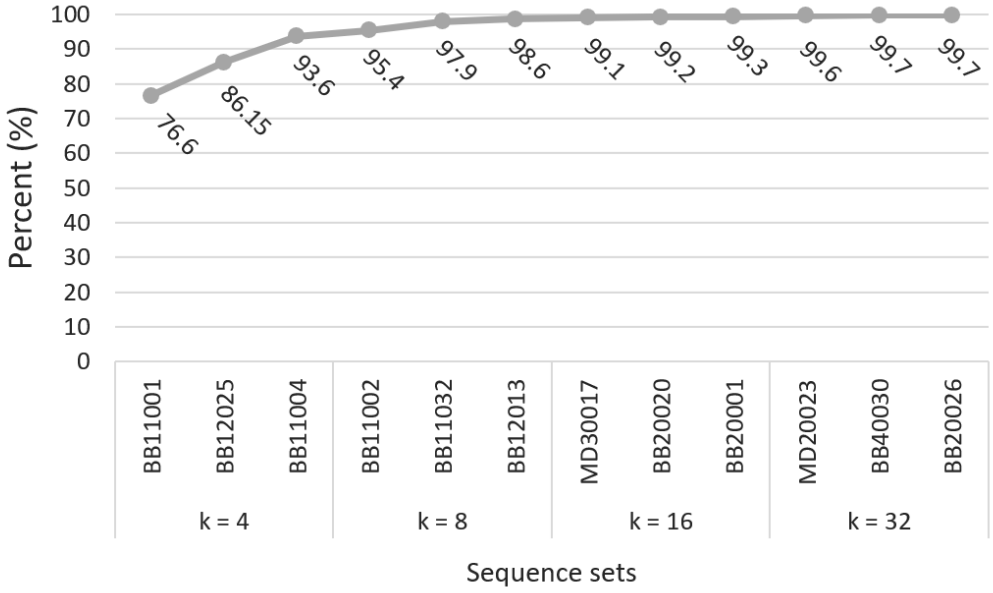

Figure 5.2. Percentages of score times within the total times for SVA.

Since the remaining part of the algorithm mainly consists of the movements, a separate time analysis for the movement durations are also carried out. The results of the movement times are represented side by side in Figure 5.4. Since the ratio of the durations within the whole process is excessively low, the values are given in seconds instead of percentages. The increase rate of the durations approximately follows the average residue number of the sets. However, since the movement functions involve high randomness, the increase rates also show variations. Note that, movements and scorings are not the only parts of the algorithm. The remaining part involves the functions like, initial population creation, file readings, memory allocations, etc. But since they consume even smaller percentage of processing time, they are neglected as test results.

The reason for the short durations of movement functions is the method that H4MSA uses. H4MSA uses a notation, in which the information of the sequences is held as gap groups, instead of keeping individual symbols separately. Since the number of gap groups are much smaller than the number of symbols, this notation provides

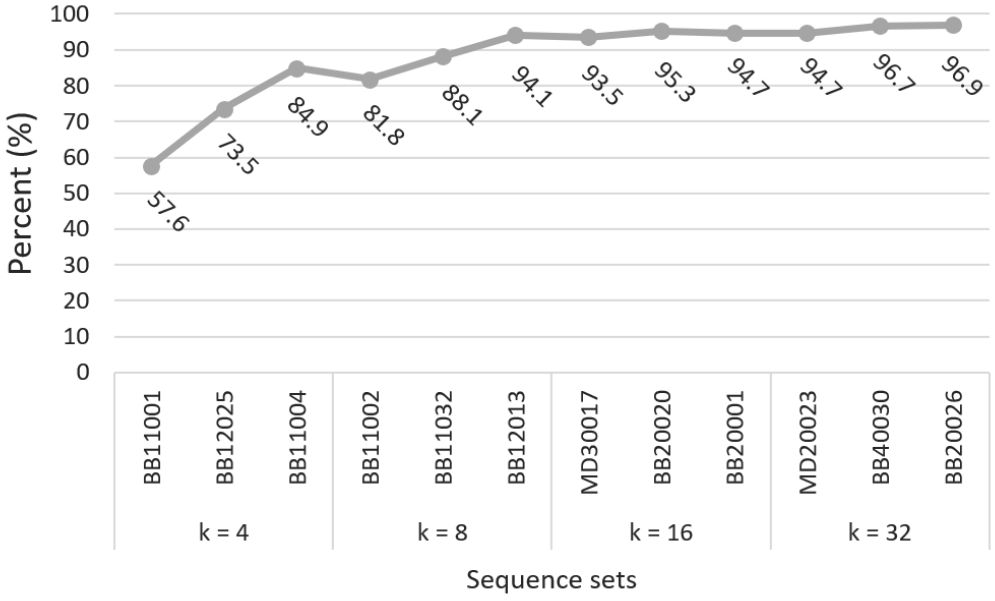

Figure 5.3. Percentages of score times within the total times for HVA.

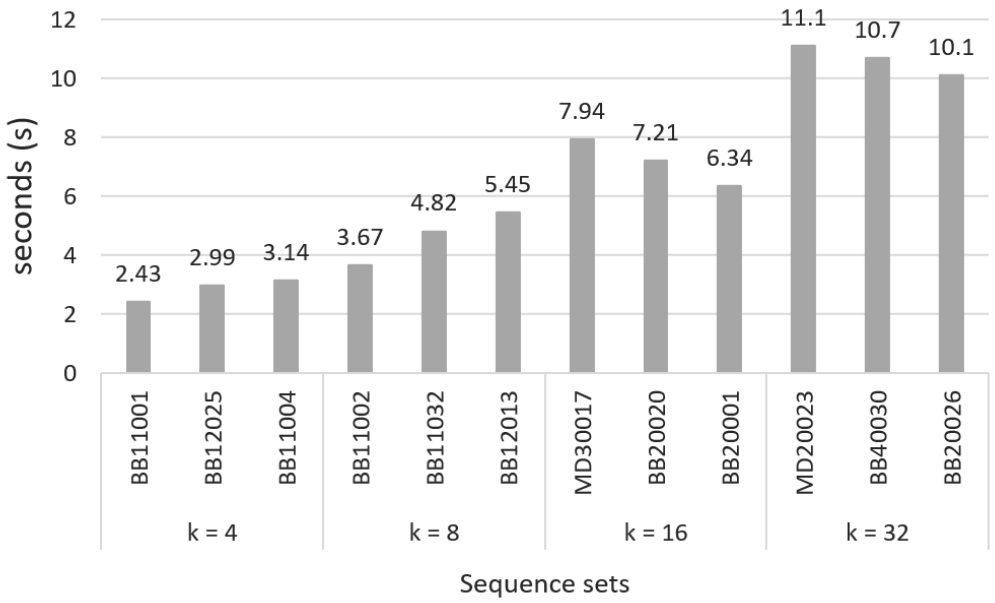

Figure 5.4. Movement durations for each sequence set.

much shorter loops within the movement functions. This notation however cannot be used by the WSPS, since each residue pair should be evaluated in this function. This situation creates the large difference between the processing times of the two partitions of H4MSA.

The next experiment is the accuracy test, in which the scores of HVA are compared with corresponding SVA scores. The test is carried out as follows: A temporal version of H4MSA is created, in which both hardware and software WSPS functions are placed. They evaluate the same alignments separately and print out the scores. The algorithm is run with each test set, at each run, 10 random score pairs are selected, then the percentage error is calculated using the pairs. In Equation 5.1, calculation of the percentage error is presented, where SC_{HVA} and SC_{SVA} are the hardware and software scores respectively.

$$error = \frac{|SC_{HVA} - SC_{SVA}|}{SC_{HVA}} \times 100 \tag{5.1}$$

In Table 5.5, sequence sets and corresponding average error values (average of the 10 errors of each set) are given. The overall average error is calculated as 0.011%, which is an acceptable value. The main reason of this error is the usage of fixed-point numbers by the hardware, while the software uses floating-point. In hardware, all the numbers are initially multiplied by 1024 (which corresponds to 10 bit shift for integers). After all the calculations are done, the fixed-point WSPS result is changed back to floating point in the software. The error can be decreased to zero by using floating-point arithmetic in hardware, but since the error is small enough, fixed-point approach is also suitable.

Speed comparison results are the most important measures, since the main purpose of this work is accelerating the overall speed of MSA. The comparisons basically involve the calculation of the ratio between processing times (taking score time or total

Table 5.5. Percentage errors of the scores.

Name	**Error**	**Name**	**Error**
BB11001	0.011%	MD30017	0.006%
BB12025	0.013%	BB20020	0.011%
BB11004	0.011%	BB20001	0.016%
BB11002	0.019%	MD20023	0.008%
BB11032	0.021%	BB40030	0.007%
BB12013	0.006%	BB20026	0.009%

time as reference) of SVA and HVA. The speedup ratio (sr) calculation is formulated in Equation 5.2, where t_{SVA} and t_{HVA} are the corresponding processing times of SVA and HVA respectively.

$$sr = \frac{t_{SVA}}{t_{HVA}} \tag{5.2}$$

In Figures 5.5 and 5.6 , the speedup values for score time and total time of each individual sets are illustrated. As shown in the figures, speedup ratios vary between 2.2x and 20.7x for the score time, 1.7x and 20.1x for the total time. Graphs clearly show that speedup ratios of the sets that share the same sequence numbers are close to each other. This was an expected behavior since the acceleration rate of designed hardware scales with sequence number.

To give a better picture of the overall scaling relationship, the average speedups with respect to sequence numbers are shown as a bar graph in Figure 5.7 and as a line graph in Figure 5.8, in which score and total time ratios are shown with different lines. In both ratios, a nearly linear relationship can be observed, where doubling the sequence number also approximately doubles the speedup value. It is also observed

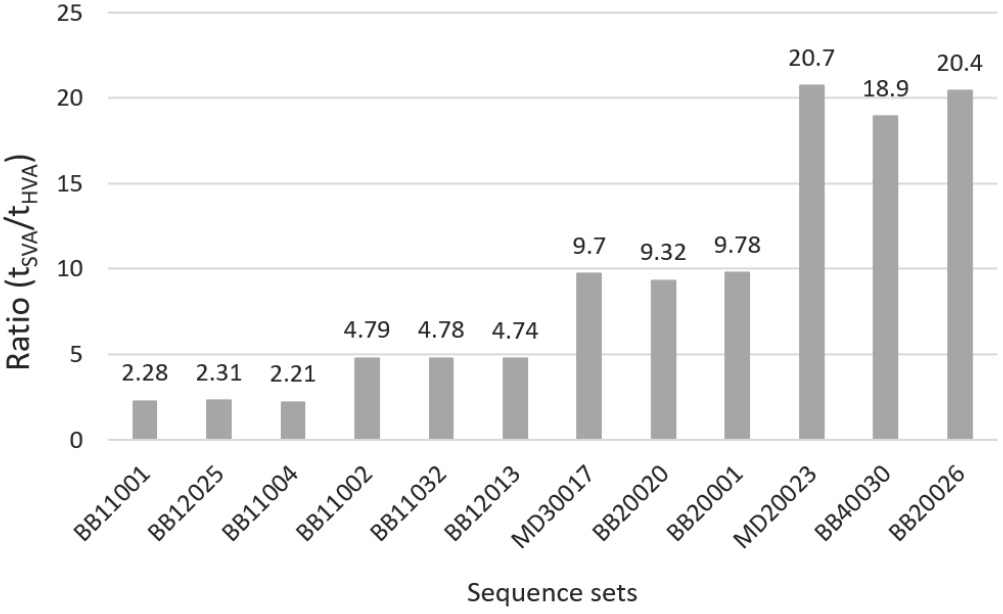

Figure 5.5. Speedup ratios for score time.

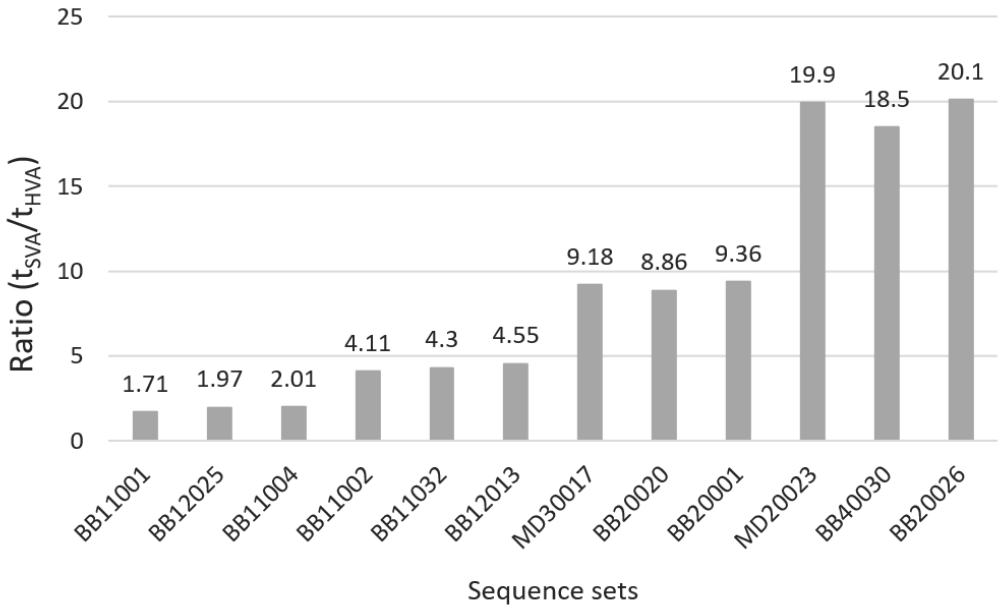

Figure 5.6. Speedup ratios for total time.

that, the difference between the two durations decreases as the sequence number k rises; which is an expected result, since the ratio of the score time within the algorithm highly increases in sets with larger k.

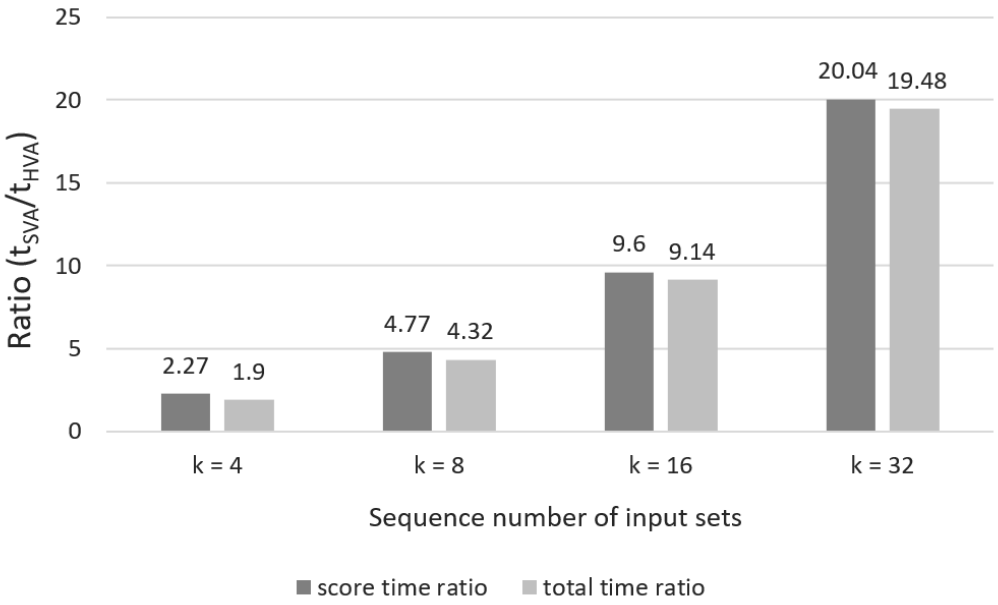

Figure 5.7. Overall speedup comparison.

In previous chapters, the theoretical scoring times are calculated as $\frac{kL(k-1)}{2}$ and $\frac{kL}{2}$ for software and hardware respectively. Hence, the theoretical speedup ratio term is obtained by dividing the software time by the hardware time and found as $(k-1)$. On the other hand, the results of the tests show that, the experimental speedup ratio is approximately $k/2$ (k additionally contains an exponent which is about 1.05, but since it is relatively small, it is neglected). It is seen that the main difference between the two terms is the constant coefficient of 1/2. The possible reasons that create this difference can be listed as follows:

- The main factor that decreases the coefficient value is the clock speed difference. Clock speed of the HPS is 900 MHz while the hardware uses a 50 MHz clock.
- In theoretical calculation of the processing time of SVA, each symbol scoring is assumed to be done in one cycle. However, since it is a pure CPU application, each scoring operation obviously consists of multiple instructions. This adds

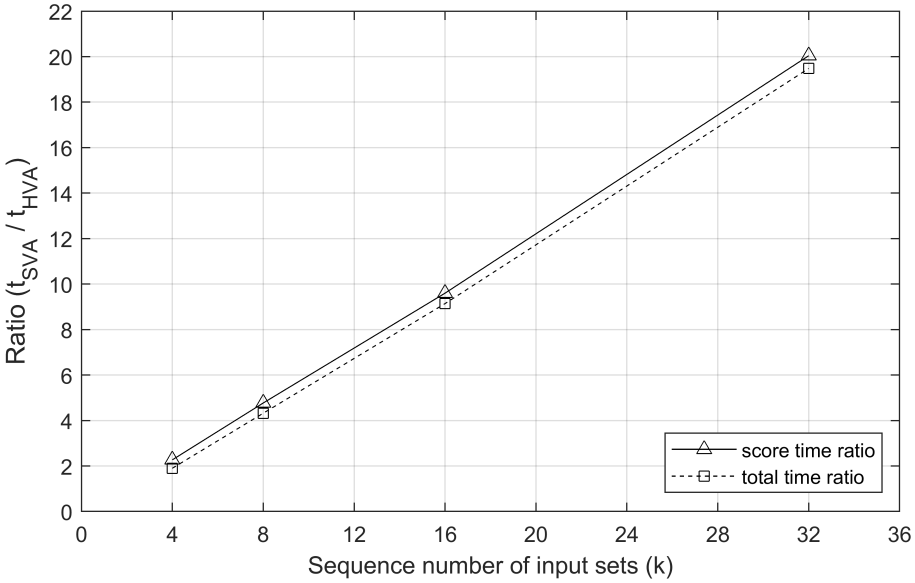

Figure 5.8. Speedup ratios vs. sequence numbers.

extra time to the overall duration of the algorithm, which is the main factor of the coefficient value increase.

- Since the SVA runs on an operating system, it is possible that various background processes are adding redundant time to the process.
- The memory access time of HPS and the communication overhead of the hardware are also crucial factors that affects the scoring time.

When all the factors that are mentioned above are considered together, they change the speedup ratio term from $(k-1)$ to $k/2$. Although the coefficient variation creates a difference in speedups of individual runs, to explain the overall scaling behavior of the speedup ratio, computational complexities should be examined. The theoretical complexity of the design is already calculated as $O(kL)$ in Chapter 4. Since k is the dominant term of the expression $k/2$, the computational complexity of the practical design is also achieved as $O(kL)$. In conclusion, we can claim that experimental results are compatible with the theoretical calculations and the obtained scaling behavior is proven to be true by the measurements.

6. CONCLUSION AND FUTURE WORK

MSA algorithms require a large amount of time to be effectively computed. This study proposes a hardware design that improves the speed performance of MSA algorithms with a special focus on a particular member of this class, H4MSA, a multi-objective algorithm. The improvement relies on the following observations: First, a detailed analysis of the time, consumed for the computation of the algorithm, demonstrates that WSPS is the leading factor by a significant margin. Second, the structure of the WSPS function itself hints that it is highly suitable for a hardware design. The hardware block of WSPS evaluation is designed with a novel method explained in Chapter 4 and the acceleration is accomplished by replacing the WSPS function of H4MSA with the designed hardware version. For testing the achieved speedups, scoring times and total runtimes of SVA and HVA are measured separately. Average speedups corresponding to the sets with 4, 8, 16, 32 sequences are 2.27x, 4.77x, 9.6x, 20.04x for score time and 1.9x, 4.32x, 9.14x, 19.48x for total time respectively. Thus, the results confirm that the study satisfies its main objective of accelerating MSA.

Still, there are open questions for potential future work. Recall that the speedup ratio of the current version scales with the sequence number of the input alignment, which means more speedup is achieved only if the column size of the alignment gets larger. Developing an approach, where longer alignments with fewer sequences show greater speedup ratios, can be quite useful. Furthermore, fixing the speedup ratio to a constant value regardless of the sequence number would be an even more significant contribution. Another aspect that can potentially be improved is the reproduced version of H4MSA (SVA) that we used as the benchmark test algorithm for this study. As the test results in Chapter 5 clearly shows, the scores of the SVA are not compatible with the ones that the original algorithm claims. Moreover, the physical platform we used imposes major limitations for this work. An upgrade to a setup with a high end CPU and FPGA together with a powerful communication interface would layout the actual practical limits of the design.

REFERENCES

1. Rubio-Largo, Á., M. A. Vega-Rodríguez and D. L. González-Álvarez, "A hybrid multiobjective memetic metaheuristic for multiple sequence alignment", *IEEE Transactions on Evolutionary Computation*, Vol. 20, No. 4, pp. 499–514, 2015.

2. Hughes, A. B., *Amino acids, peptides and proteins in organic chemistry, analysis and function of amino acids and peptides*, Vol. 5, John Wiley & Sons, 2013.

3. Fulton, A. B. and W. B. Isaacs, "Titin, a huge, elastic sarcomeric protein with a probable role in morphogenesis", *BioEssays*, Vol. 13, No. 4, pp. 157–161, 1991.

4. Barrett, G. C. and D. T. Elmore, *Amino acids and peptides*, Cambridge University Press, 1998.

5. Sung, W.-K., *Algorithms in bioinformatics: A practical introduction*, CRC Press, 2009.

6. Russell, D. J., *Multiple sequence alignment methods*, Springer, 2014.

7. Chowdhury, B. and G. Garai, "A review on multiple sequence alignment from the perspective of genetic algorithm", *Genomics*, Vol. 109, No. 5-6, pp. 419–431, 2017.

8. Chatzou, M., C. Magis, J.-M. Chang, C. Kemena, G. Bussotti, I. Erb and C. Notredame, "Multiple sequence alignment modeling: methods and applications", *Briefings in bioinformatics*, Vol. 17, No. 6, pp. 1009–1023, 2015.

9. Notredame, C., D. G. Higgins and J. Heringa, "T-Coffee: A novel method for fast and accurate multiple sequence alignment", *Journal of molecular biology*, Vol. 302, No. 1, pp. 205–217, 2000.

10. Ryadnov, M. and F. Hudecz, *Amino Acids, Peptides and Proteins: Volume 42*,

Vol. 42, Royal Society of Chemistry, 2017.

11. Altschul, S. F., "Amino acid substitution matrices from an information theoretic perspective", *Journal of molecular biology*, Vol. 219, No. 3, pp. 555–565, 1991.

12. Henikoff, S. and J. G. Henikoff, "Amino acid substitution matrices from protein blocks", *Proceedings of the National Academy of Sciences*, Vol. 89, No. 22, pp. 10915–10919, 1992.

13. Altschul, S. F. and B. W. Erickson, "Optimal sequence alignment using affine gap costs", *Bulletin of mathematical biology*, Vol. 48, No. 5-6, pp. 603–616, 1986.

14. Rojas, I. and F. Ortuño, *Bioinformatics and Biomedical Engineering: 5th International Work-Conference, IWBBIO 2017, Granada, Spain, April 26–28, 2017, Proceedings*, Vol. 10209, Springer, 2017.

15. Needleman, S. B. and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *Journal of molecular biology*, Vol. 48, No. 3, pp. 443–453, 1970.

16. Wang, L. and T. Jiang, "On the complexity of multiple sequence alignment", *Journal of computational biology*, Vol. 1, No. 4, pp. 337–348, 1994.

17. Hogeweg, P. and B. Hesper, "The alignment of sets of sequences and the construction of phyletic trees: an integrated method", *Journal of molecular evolution*, Vol. 20, No. 2, pp. 175–186, 1984.

18. Higgins, D. G. and P. M. Sharp, "CLUSTAL: a package for performing multiple sequence alignment on a microcomputer", *Gene*, Vol. 73, No. 1, pp. 237–244, 1988.

19. Thompson, J. D., D. G. Higgins and T. J. Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice", *Nucleic acids research*,

Vol. 22, No. 22, pp. 4673–4680, 1994.

20. Morgenstern, B., K. Frech, A. Dress and T. Werner, "DIALIGN: finding local similarities by multiple sequence alignment.", *Bioinformatics (Oxford, England)*, Vol. 14, No. 3, pp. 290–294, 1998.

21. Lassmann, T. and E. L. Sonnhammer, "Kalign–an accurate and fast multiple sequence alignment algorithm", *BMC bioinformatics*, Vol. 6, No. 1, p. 298, 2005.

22. Saitou, N. and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees.", *Molecular biology and evolution*, Vol. 4, No. 4, pp. 406–425, 1987.

23. Edgar, R. C., "MUSCLE: multiple sequence alignment with high accuracy and high throughput", *Nucleic acids research*, Vol. 32, No. 5, pp. 1792–1797, 2004.

24. Katoh, K., K. Misawa, K.-i. Kuma and T. Miyata, "MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform", *Nucleic acids research*, Vol. 30, No. 14, pp. 3059–3066, 2002.

25. Notredame, C. and D. G. Higgins, "SAGA: sequence alignment by genetic algorithm", *Nucleic acids research*, Vol. 24, No. 8, pp. 1515–1524, 1996.

26. Gondro, C. and B. P. Kinghorn, "A simple genetic algorithm for multiple sequence alignment", *Genetics and Molecular Research*, Vol. 6, No. 4, pp. 964–982, 2007.

27. Naznin, F., R. Sarker and D. Essam, "Vertical decomposition with genetic algorithm for multiple sequence alignment", *BMC bioinformatics*, Vol. 12, No. 1, p. 353, 2011.

28. Naznin, F., R. Sarker and D. Essam, "Progressive alignment method using genetic algorithm for multiple sequence alignment", *IEEE Transactions on Evolutionary Computation*, Vol. 16, No. 5, pp. 615–631, 2012.

29. Taheri, J. and A. Y. Zomaya, "RBT-GA: a novel metaheuristic for solving the multiple sequence alignment problem", *Bmc Genomics*, Vol. 10, No. 1, p. S10, 2009.

30. Lee, Z.-J., S.-F. Su, C.-C. Chuang and K.-H. Liu, "Genetic algorithm with ant colony optimization (GA-ACO) for multiple sequence alignment", *Applied Soft Computing*, Vol. 8, No. 1, pp. 55–78, 2008.

31. Ortuno, F. M., O. Valenzuela, F. Rojas, H. Pomares, J. P. Florido, J. M. Urquiza and I. Rojas, "Optimizing multiple sequence alignments using a genetic algorithm based on three objectives: structural information, non-gaps percentage and totally conserved columns", *Bioinformatics*, Vol. 29, No. 17, pp. 2112–2121, 2013.

32. Kemena, C., J.-F. Taly, J. Kleinjung and C. Notredame, "STRIKE: evaluation of protein MSAs using a single 3D structure", *Bioinformatics*, Vol. 27, No. 24, pp. 3385–3391, 2011.

33. Kaya, M., A. Sarhan and R. Alhajj, "Multiple sequence alignment with affine gap by using multi-objective genetic algorithm", *Computer methods and programs in biomedicine*, Vol. 114, No. 1, pp. 38–49, 2014.

34. Rubio-Largo, Á., M. A. Vega-Rodríguez and D. L. González-Álvarez, "Hybrid multiobjective artificial bee colony for multiple sequence alignment", *Applied Soft Computing*, Vol. 41, pp. 157–168, 2016.

35. Deb, K., A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II", *IEEE transactions on evolutionary computation*, Vol. 6, No. 2, pp. 182–197, 2002.

36. Eusuff, M., K. Lansey and F. Pasha, "Shuffled frog-leaping algorithm: a memetic meta-heuristic for discrete optimization", *Engineering optimization*, Vol. 38, No. 2, pp. 129–154, 2006.

37. Thompson, J. D., P. Koehl, R. Ripp and O. Poch, "BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark", *Proteins: Structure, Function, and Bioinformatics*, Vol. 61, No. 1, pp. 127–136, 2005.

38. Terasic, *DE1-SoC User Manual*, 2015, `https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=eac30a7aaacf5187a4ace0d613cd4676`, accessed in June 2019.

39. Intel, *Cyclone V Device Overview*, 2018, `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf`, accessed in June 2019.