2011-05-20

Accelerated Large-Scale Multiple Sequence Alignment with Reconfigurable Computing

G Scott Lloyd
Brigham Young University - Provo

Accelerated Large-Scale Multiple Sequence Alignment with Reconfigurable Computing

G. Scott Lloyd

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Quinn O. Snell, Chair
Mark J. Clement
Michael J. Wirthlin
Sean C. Warnick
Parris K. Egbert

Department of Computer Science

Brigham Young University

June 2011

ABSTRACT

Accelerated Large-Scale Multiple Sequence Alignment
with Reconfigurable Computing

G. Scott Lloyd
Department of Computer Science, BYU
Doctor of Philosophy

Multiple Sequence Alignment (MSA) is a fundamental analysis method used in bioinformatics and many comparative genomic applications. The time to compute an optimal MSA grows exponentially with respect to the number of sequences. Consequently, producing timely results on large problems requires more efficient algorithms and the use of parallel computing resources. Reconfigurable computing hardware provides one approach to the acceleration of biological sequence alignment. Other acceleration methods typically encounter scaling problems that arise from the overhead of inter-process communication and from the lack of parallelism. Reconfigurable computing allows a greater scale of parallelism with many custom processing elements that have a low-overhead interconnect. The proposed parallel algorithms and architecture accelerate the most computationally demanding portions of MSA. An overall speedup of up to 150 has been demonstrated on a large data set when compared to a single processor. The reduced runtime for MSA allows researchers to solve the larger problems that confront biologists today.

Keywords: communication network, communication protocol, computer architecture, interface, modules, dynamic programming, field-programmable gate array, FPGA, sequence alignment, reconfigurable hardware, traceback

ACKNOWLEDGMENTS

Returning to graduate school was a decision we made as a family. I would like to thank my wife Susan and our children Sarah, Anna, Maxwell, Natalie, and Sadie for their faith, encouragement and support through the years to realize our goal. Even with the challenges, they remained true and committed. I appreciate their tolerance while living in the limited space of student housing. Susan says she conducted her own research about thriving in a small space. I am also thankful for the maintenance staff in student housing and their timely response to support calls.

I am grateful for the assistance and support of my adviser Dr. Quinn Snell. He has always made sure that I have everything necessary to conduct research, including equipment and software. Dr. Snell's experience with parallel processing and high-performance computing has been fundamental to this research. I appreciate the trust that was shown by giving me the freedom to specify the needed hardware within our budget constraints.

Dr. Mark Clement and Dr. Mike Wirthlin, in conjunction with Dr. Snell, have given appreciated guidance and direction. They have provided valuable feedback on my thesis topic and dissertation writing. Additionally, they have guided the scope of the research to allow completion in a reasonable time. Work within the research area has required expertise from other disciplines. Dr. Snell and Dr. Clement have provided mentorship in molecular biology and Dr. Wirthlin has brought invaluable experience with FPGA hardware and reconfigurable computing. Dr. Sean Warnick and Dr. Parris Egbert have also been exemplary committee members in sharing their time and offering constructive comments.

Special thanks go to my parents Bud and JoAnn. Their continual prayers in my behalf have been felt. They have championed me through the process, and even though my father passed away before I could finish, his hope for me continues to offer strength.

I am indebted to my Father in Heaven for His unconditional support. When searching for answers to research problems, key insights were revealed. When wrestling with confidence, assurance was given. Often, the divine response was delivered through the great individuals mentioned above.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Searching and comparing biological sequences in genomic databases are essential processes in molecular biology. Fueled by new sequencing technology, the collection of genetic sequence data is increasing exponentially each year and consists mostly of nucleotide (DNA/RNA) and amino acid (protein) symbols. The comparison of hundreds or thousands of genomic sequences presents a significant computational challenge. The large data sets under analysis require high-performance computing methods to produce timely results. Some of the larger problems encountered by biologists today can not be adequately addressed by current methods.

Biologists and other researchers use sequence alignment as a fundamental analysis method to find similarities between sequences, predict protein structure, identify important genetic regions, and facilitate drug design. For example, sequence alignment is used to derive flu vaccines [56] and is used by the nation's BioWatch [29] program in identifying DNA signatures of pathogens. Alignment is also used in comparative genomic applications such as whole genome alignment [11] and whole genome phylogeny [14]. Sequence alignment consists of matching characters between sequences and positioning them together in a column (see Figure 1.1). Gaps may be introduced in columns where matches do not occur to reflect an insertion or deletion evolutionary event. Pairwise alignment involves two sequences and multiple alignment considers three or more sequences.

The computational cost for an optimal sequence alignment increases exponentially with the number of sequences. Finding an optimal multiple sequence alignment (MSA) is NP-

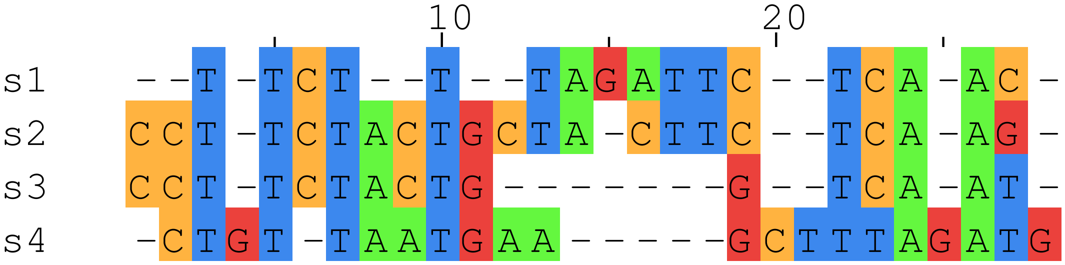

10
20
s1 --T-TCT--T--TAGATTC--TCA-AC-
s2 CCT-TCTACTGCTA-CTTC--TCA-AG-
s3 CCT-TCTACTG-------G--TCA-AT-
s4 -CTGT-TAATGAA-----GCTTTAGATG

Figure 1.1: Example multiple alignment

hard in complexity [22, 96], which means that the exact solution to the problem is intractable for data sets with more than a few sequences. This complexity poses a challenge for sequence alignment programs to return results within a reasonable time period as biologists compare greater numbers of sequences. Consequently, many MSA approaches use heuristics to arrive at a sub-optimal solution in a reasonable time. Even with current heuristic methods, an alignment program may run for days or even weeks depending on the number of sequences and their length.

1.1 Potential Impact and Importance

In some cases, getting a result from a MSA is time critical. For instance, molecular diagnostic designers commonly align all the available large bacterial genomes and would like a result within a couple of hours. There are now dozens of genomes available for many bacterial species, and this will soon be hundreds or thousands. Currently, the best methods available require days for the larger alignment tasks. The contributions of this work will let researchers supporting biodefense or public health agencies react quickly to pathogen outbreaks or attacks by significantly reducing the time to solve this large-scale MSA problem. Rapid identification of a biothreat with accelerated MSA will allow subsequent diagnostic or countermeasure design efforts to begin promptly.

New parallel algorithms and a new architecture are presented to accelerate the most computationally demanding portions of large-scale MSA with reconfigurable computing. This work advances the state of the art in MSA by overcoming the scaling problems typically encountered in current methods and by reducing the time required to compute an alignment by up to two orders of magnitude compared to a desktop computer on data sets consisting of hundreds or thousands of sequences. One example viral data set from the family Herpesviridae has 142 sequences with an average length of 167 kb. Current MSA programs take about one week to align this data set on a desktop computer, while the contributions of this work allow estimated execution to complete within 12 minutes.

1.2 Pairwise Sequence Alignment

Global and local alignment are the most common pairwise alignment problems. Global alignment [65] considers sequences from end to end and finds the best overall alignment. A variation known as semiglobal alignment finds the best overlap between sequences and allows unmatched ends to extend without penalty (free end gaps). Local alignment [82] identifies the sections with greatest similarity and only aligns the subsequences.

A more formal definition of global alignment follows. Given a pair of *sequences* $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ of *length* $|A| = m$ and $|B| = n$ from the finite alphabet Σ, a *pairwise sequence alignment* is obtained by inserting *gap characters* “-” into A and B. The aligned sequences A' and B' from the extended alphabet $\Sigma' = \Sigma \cup \{\text{“-”}\}$ are of equal length such that $|A'| = |B'|$.

An alignment score provides a metric to assess the quality of an alignment and represents a measure of similarity between sequences. For pairwise alignment, it is the sum of similarity values for each pair of aligned characters. Characters that match have a positive value while those that mismatch have a lower or negative value. Any character aligned with a gap also contributes a negative value to the alignment score. In practice, the similarity value comes from a substitution matrix that reflects the probability of substituting one character for another. Let the function $s : \Sigma \times \Sigma \rightarrow \mathbb{Z}$ determine the similarity of two characters and let l denote the length of an alignment. The *pairwise scoring function* is then given by

$$F_{PW}(A', B') = \sum_{1 \leq i \leq l} s(a'_i, b'_i).$$

The goal is to find an optimal pairwise alignment of A and B such that for all possible alignments, the score is maximal. Pairwise alignment is typically solved with dynamic programming (DP), which fills a two-dimensional matrix with score values. Let H denote the DP matrix and the element $H[i, j]$ the similarity score of sequences $a_1 a_2 ... a_i$ and $b_1 b_2 ... b_j$. Let α represent the cost of inserting or deleting a gap. An optimal alignment is obtained by maximizing the score in each element of H. The values of H are determined by the following recurrence relations for $1 \leq i \leq m$ and $1 \leq j \leq n$:

$$\begin{aligned}
H[0,0] &= 0, \\
H[i,0] &= H[i-1,0] + \alpha, \\
H[0,j] &= H[0,j-1] + \alpha, \\
H[i,j] &= \max \begin{cases} H[i-1,j-1] + s(a_i, b_j), \\ H[i-1,j] + \alpha, \\ H[i,j-1] + \alpha. \end{cases}
\end{aligned}$$

The matrix fill occurs in a forward scan from upper left to lower right because of dependencies from neighboring elements. This dependency limits the amount of parallelism that is achievable in computing the matrix to the elements along the scan wavefront. Following a forward scan, traceback starts from a designated lower right position and follows a path to upper left, thereby determining the best alignment.

1.3 Multiple Sequence Alignment

The definition of a multiple sequence alignment is a generalization of pairwise alignment. Given an ordered set of sequences $S = \langle s_1, s_2, ... s_n \rangle$, a *multiple sequence alignment* (MSA)

$A = \langle a_1, a_2, ...a_n \rangle$ is obtained by inserting *gap characters* "-" into s_i such that the *aligned sequences* $a_i \in A$ are of equal length with $|a_i| = k$.

To determine the quality of a MSA, a more complex scoring function than the one for pairwise alignment is needed. Various assumptions about the relationship between multiple sequences lead to several possible scoring methods. The weighted sum-of-pairs (WSP) method is popular among MSA programs. It assumes that sequences are related by an evolutionary tree and that sequence weights are derived from this tree. The WSP method calculates a total score from the weighted pairwise score of all sequences. Let $F_{WSP} : A \to \mathbb{Z}$ be a *WSP scoring function* for an MSA A such that

$$F_{WSP}(A) = \sum_{1 \le i < j \le n} w_{i,j} \sum_{1 \le l \le k} s(a_i[l], a_j[l])$$

where n is the number of sequences, k is the length of aligned sequences, $w_{i,j}$ is the weight given to a pair of sequences, and the function $s : \Sigma \times \Sigma \to \mathbb{Z}$ determines the similarity of symbol $a_i[l]$ with $a_j[l]$.

The choice of scoring method inherently affects the nature of the alignment algorithm. After choosing a scoring function, a suitable algorithm is determined to maximize the score and thereby produce an optimal alignment. More specifically, the *MSA problem* is to find an alignment A given a set of sequences S such that for all possible alignments of S, the score $F_{WSP}(A)$ is maximal. Several scoring methods and MSA algorithms have been proposed and are described in a thorough review by Gotoh [33].

The DP solution to pairwise alignment may be extended to multiple alignment with an N-dimensional scoring matrix where N is the number of sequences. However, because of exponential time and space scaling problems, optimal alignment algorithms like DP are limited to a small number of sequences. Even with restricted search space strategies [8], exact algorithms are limited in the number of sequences.

1.3.1 Progressive Alignment

The most common heuristic algorithm used to solve the MSA problem is progressive alignment [27, 68, 88]. Other heuristic algorithms have been studied, but they generally provide poorer quality or suffer from greater computational cost with limited improvement in alignment quality [67]. Progressive algorithms are also robust in that they converge to a solution even with divergent input sequences. Since progressive alignment is a greedy strategy, mistakes in placing gaps at early stages will remain throughout the process. To compensate for early mistakes, iterative refinement algorithms have been developed that repeat certain stages of the process a fixed number of times or until there is no improvement in the alignment quality [19, 32, 36, 40].

Progressive algorithms successively perform pairwise alignment on the most similar sequences and groups of sequences, until all sequences are aligned. A progressive alignment is accomplished in three main stages.

Stage 1: All sequences are compared pairwise with each other and the score is stored in a similarity matrix.

Stage 2: A guide tree is constructed from the similarity matrix, with the leaves of the tree representing the sequences.

Stage 3: Following the branches of the guide tree from the leaves to the root, sequences and groups are pairwise aligned.

The guide tree indicates the order of combining groups with each node specifying a pairwise alignment of the left and right groups (see Figure 1.2).

The execution time for the first stage of progressive alignment typically dominates the overall computation and increases exponentially as more sequences are aligned. Using an optimal pairwise sequence alignment algorithm, the first stage complexity is $O(N^2L^2)$, where N is the number of sequences and L is the length. Hence, more recent large-scale MSA algorithms use less-costly sequence comparison methods in the first stage to reduce the

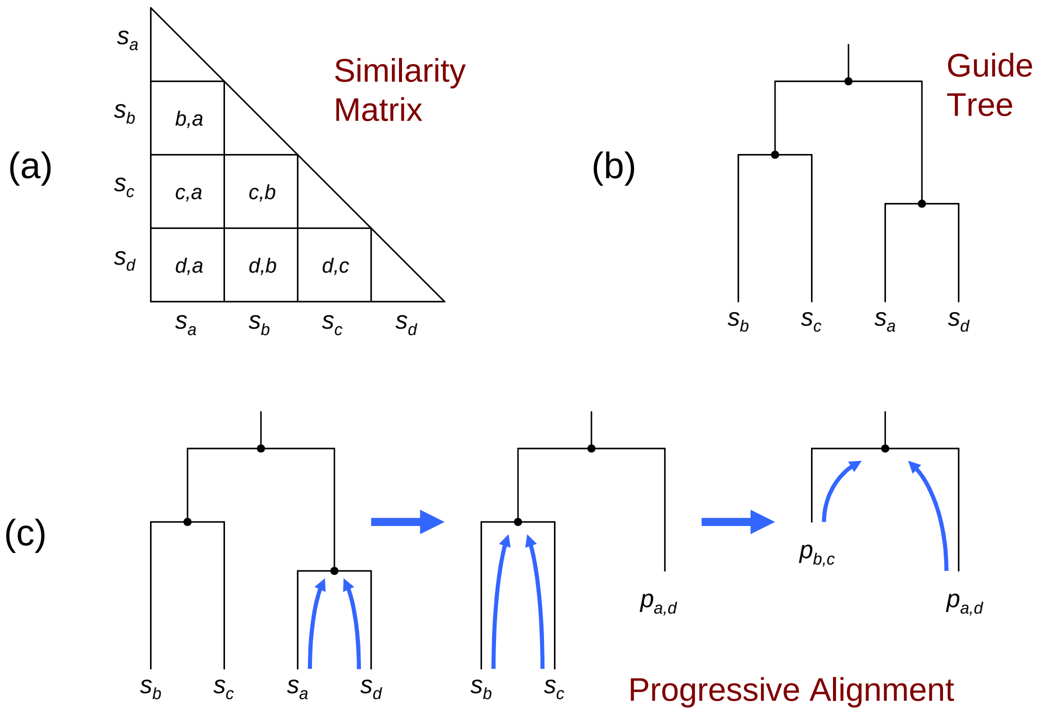


Figure 1.2: The three stages of progressive alignment: (a) compare all sequences to form a similarity matrix; (b) use the similarity scores and a clustering method to build a guide tree; and (c) progressively align sequences s_i and groups of sequences $p_{i,j,\ldots}$ in an order guided by the tree.

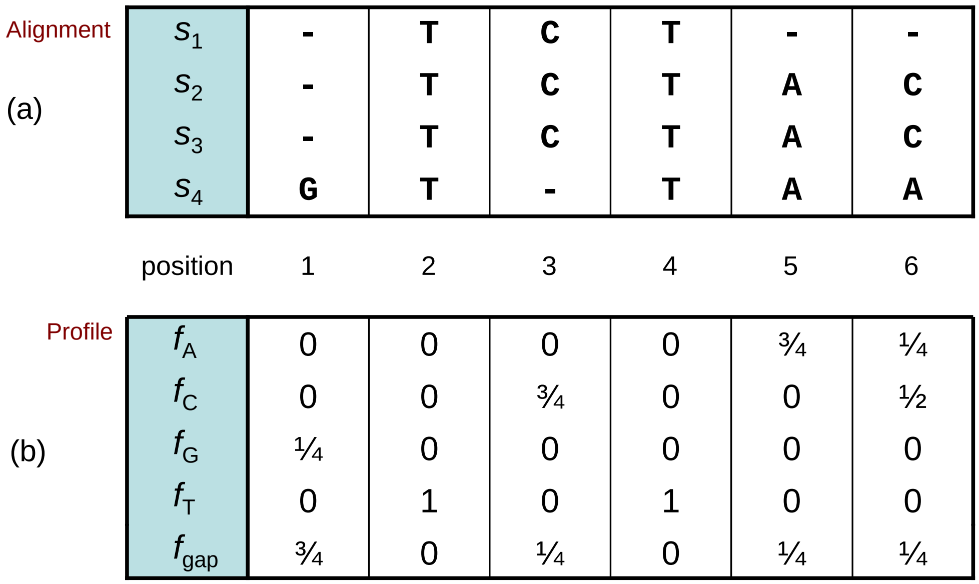
overall computation time. For example, Kalign2 [43] uses approximate string searching [63], MAFFT [40] and MUSCLE [19] use an alignment-free comparison method based on word counts [94], and MAVID [6] avoids the first stage entirely and uses a random guide tree for the first iteration of progressive alignment.

The second stage groups the most similar sequences together on terminal branches of the guide tree. Common clustering methods for tree construction include UPGMA [83] and neighbor-joining (NJ) [79]. In addition to tree topology, these methods provide branch lengths that reflect a measure of sequence divergence at each level of the tree. Individual sequence weights may be derived from branch lengths that lie in a path to the root.

In the third stage, instead of aligning two groups of sequences directly, profiles are often created first. A profile is derived from the aligned sequences below a branch of the guide tree. Figure 1.3 shows an example profile derived from a group of aligned sequences. Individual sequence weights and position specific weights may be applied in the profile calculation based upon tree branch lengths and local patterns in the sequences. Once two profiles are created, they are aligned with a DP algorithm in a similar way to sequences. Gaps inserted into the profiles are inserted into the corresponding columns of the groups. When a single sequence is aligned with a group of sequences, it is treated like a simple profile.

1.4 Acceleration

One method of addressing the computational demand of MSA is with supercomputers that use parallel processors to speed up the computations. While supercomputers are somewhat effective at accelerating MSA, they may only be available to larger institutions because of their significant cost. Even if a supercomputer is available, waiting in a queue for a shared resource limits accessibility. Furthermore, supercomputer approaches for progressive MSA begin to show scaling problems beyond 32 processors [86] because of inter-process communication overhead and a lack of parallelism. The best published speedup on a cluster system is 40 with 80 processors [104]. The desire for a solution with greater performance



Alignment

(a)

s_1	-	T	C	T	-	-
s_2	-	T	C	T	A	C
s_3	-	T	C	T	A	C
s_4	G	T	-	T	A	A
position	1	2	3	4	5	6

Profile

(b)

f_A	0	0	0	0	¾	¼
f_C	0	0	¾	0	0	½
f_G	¼	0	0	0	0	0
f_T	0	1	0	1	0	0
f_{gap}	¾	0	¼	0	¼	¼

Figure 1.3: Each position in a profile consists of a vector with character frequencies f_N for the corresponding column in a group of aligned sequences. (a) Multiple alignment of sequences s_i. (b) Profile derived from the alignment.

and accessibility has motivated research into hardware acceleration methods using Graphics Processing Units (GPUs), the Cell Broadband Engine (Cell BE), and reconfigurable computing with Field-Programmable Gate Arrays (FPGAs). For MSA applications, acceleration methods have demonstrated a desktop solution with up to 42 times the performance of a single workstation.

Some challenges must be overcome to realize the benefits of acceleration methods. The increased performance comes at a price that is usually in the form of greater design complexity and longer development times. Acceleration methods require a deep understanding of the system's memory and communication architecture to obtain the best performance. Parallel portions of applications must be identified and matched with appropriate hardware resources. Consequently, this exposure to architecture detail leads to greater difficulty in developing application programs.

Acceleration of sequence alignment with reconfigurable computing has demonstrated a performance advantage over other methods. The most significant reason is that the FPGA's configurable logic operates with fine-grained parallelism, which allows hundreds or thousands of operations to occur in parallel within the chip. The FPGA acts as a coprocessor to accelerate repetitive or parallel portions of an application. These portions are programed into the FPGA's logic and execute at hardware speed on data supplied from the host. For many applications, reconfigurable computing has the potential to deliver supercomputing performance to a desktop computer for about the same cost as an additional computer.

When applied to MSA, reconfigurable computing with FPGAs has the potential to overcome the scaling problems that result from a lack of parallelism and from high communication overhead. Processing elements that wait idle on communication reduce the parallelism and performance of an application. Reconfigurable computing supports many custom processing elements and a local interconnect that provides communication with very low overhead when compared with commodity processors. Accurately aligning thousands of genomic sequences becomes feasible in a reasonable time period with sufficient parallelism provided by reconfigurable hardware. With the current exponential increase in available sequence data, an accelerated system capable of reducing the time for large-scale MSA by two or more orders of magnitude compared to a desktop computer would greatly benefit the bioinformatics community.

Chapter 2

Related Work

Most efforts to accelerate bio-sequence applications with hardware have focused solely on database searches and have employed a pairwise local comparison algorithm. Ramdas and Egan [77] discuss several FPGA based architectures in their survey. Other pairwise comparison accelerators have also been described in [25, 50, 84]. Given a query sequence, an entire genetic database is scanned to find other sequences that are similar. Searching a genetic database for matches with a bio-sequence is similar in nature to an Internet search that returns hits sorted by relevance. Most of the search acceleration occurs on the forward scan of a DP-based algorithm when the query is compared with the database sequences. Only the highest-scoring matches are retained for later alignment, which requires an additional traceback procedure. Since the matches are relatively short and few in number, traceback acceleration provides little benefit to database searches; therefore, it is absent from most implementations.

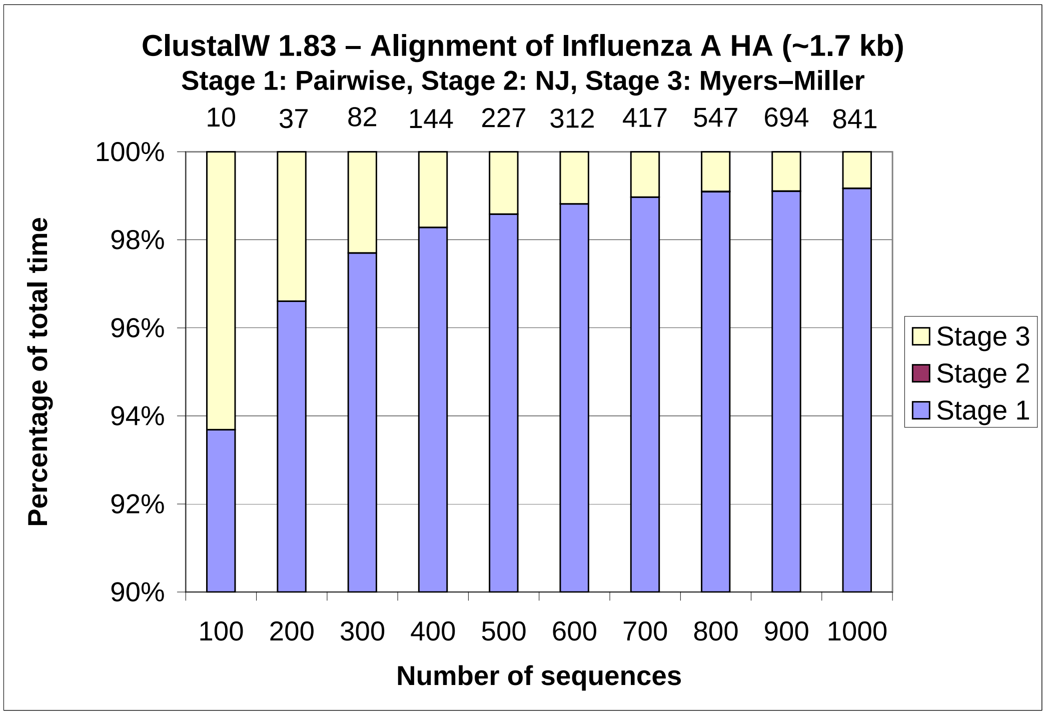
A few methods to accelerate MSA with hardware have been demonstrated. Existing acceleration methods fail to use all the available parallel resources in every stage of MSA; consequently, performance is reduced in some stages with idle processors. Greater performance may be achieved, however, with more parallelism and by accelerating additional stages of the algorithm.

2.1 Acceleration Challenges

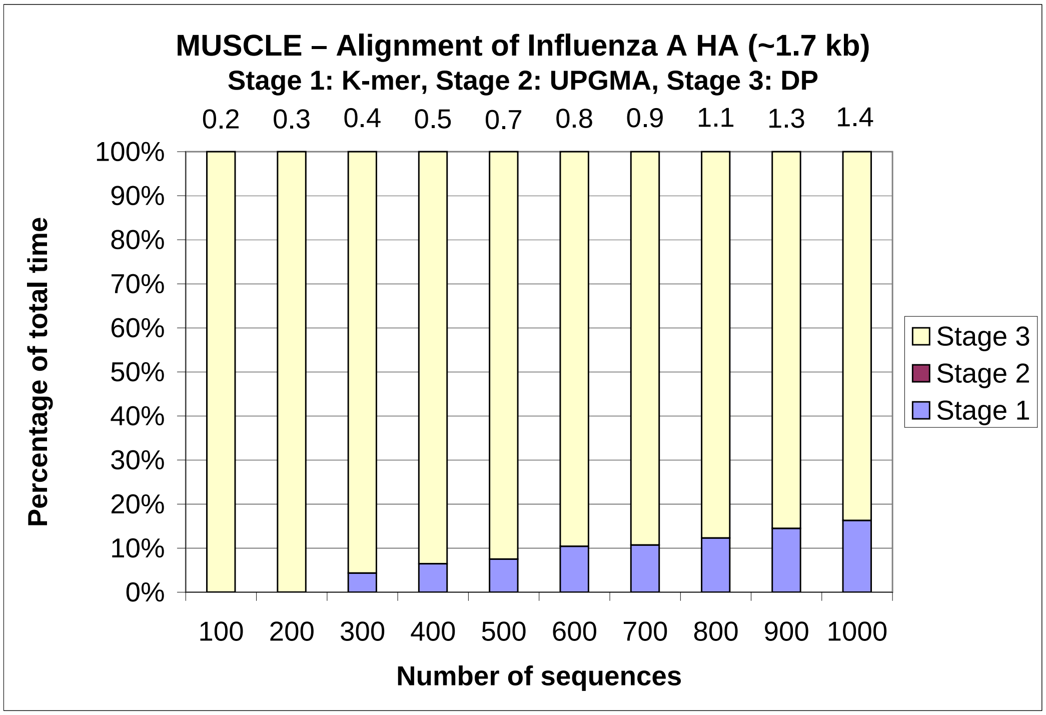
Among the accelerated MSA approaches, some or all of the stages of progressive alignment are parallelized with varying amounts of success. Figure 2.1a shows the proportion of time spent in each stage of the well-known ClustalW [88] program for several problem sizes. The first stage receives the most attention since it usually dominates the computation time and it is easily parallelized by independently comparing all of the sequences. The second stage receives little attention since it requires the least computation time of all the stages for a small number of sequences. However, the third stage still takes a significant amount of computation time and warrants acceleration.

Without accelerating the third stage of progressive MSA, Amdahl's law [1] limits the overall speedup. For example, if the third stage takes 5% of the computation time, the overall speedup is limited to about 20 even if the other stages are infinitely fast. If the time in Stage 1 is reduced with faster comparison techniques, then the acceleration of Stage 3 becomes more critical. Newer programs like MUSCLE and MAFFT use a faster alignment free comparison method; therefore, the third stage dominates the computation time (see Figure 2.1b). Even though these newer methods show greater performance, most of the related work has still focused on accelerating ClustalW where the first stage dominates the run time.

Many of the acceleration methods struggle to find enough parallelism in the third stage of progressive alignment where the DP data dependencies restrict the parallel calculations to a wavefront. Each process associated with a cell or block of cells along the wavefront must communicate with three neighbors. Because of frequent and small communication between these processes, fine-grained parallelism with low-latency communication is required to efficiently compute the wavefront. Without access to fine-grained resources, many methods will take a coarse-grained approach and traverse branches of the guide tree in parallel or use a recursively parallel version of the Myers-Miller [64] pairwise alignment algorithm. Nevertheless, parallelism in the guide tree may be poor if the tree is not balanced. Further-



(a)



(b)

Figure 2.1: The proportion of time spent in the stages of ClustalW and MUSCLE is depicted for the Influenza data set. Only one iteration of MUSCLE was run for the comparison. The total time for an alignment in minutes is shown at the top of each column. This data set is comparable in size to those used in related work.

more, both of these strategies have limited parallelism at the root of the tree and in the first steps of recursion.

Reconfigurable computing with FPGAs has demonstrated a performance advantage in accelerating the DP solution to pairwise alignment with a systolic array of custom processors in programmable logic. However, this advantage has not previously been demonstrated for the third stage of progressive alignment because of the extra complexity associated with aligning profiles. This work is unique in that it addresses the acceleration of profile alignment in Stage 3 with reconfigurable computing and demonstrates an overall speedup of up to 150 on large data sets. The next section describes some related examples of parallel MSA in more detail.

2.2 Parallel MSA

Parallel MSA algorithms can be classified into several different categories depending on the underlying solution strategy. Some common classifications include optimal, progressive, iterative, stochastic, and hybrid. Only progressive and progressive-iterative methods will be discussed because of their relevance to this work. Within each section, the parallel examples are organized by the system type and include a description of the parallel approach. The examples discussed show a contribution in either overall performance or in the application of a parallel algorithm.

Parallel algorithms are often categorized by their granularity, which refers mainly to the frequency of communication between parallel tasks. A coarse-grained algorithm typically has fewer, larger tasks that communicate every second or less, while a fine-grained algorithm may have many, smaller tasks that communicate more frequently. The classification is not rigid, and some examples may not properly fit one category or the other.

2.2.1 Progressive Methods

Almost all of the parallel MSA examples referenced in this section are based on the popular ClustalW program [88]. Several reasons for this popularity exist. First of all, the method has been trusted by biologists for almost two decades with quality that is still comparable to more recent algorithms. The trust is gained with alignment results that are similar to biologists expectations. Even though newer methods may have better performance or quality, ClustalW has become a recognized benchmark standard. The algorithm provides fairly good alignments across a diverse range of sequence types. Also, the alignment algorithm is relatively fast, simple, and understandable. The source code is freely available and well supported, which provides broad access to biologists and allows researchers to experiment with algorithmic variations without starting from scratch.

Vector. The first published attempt to parallelize MSA was described by Tajima [85]. Only the DP calculations were vectorized on a FACOM VP-200 vector supercomputer. However, accelerating DP calculation with vector parallelism is problematic because vector machines typically do not include a table lookup vector operation that is needed to return the similarity score between two characters. To avoid a table lookup, the implementation by Tajima returns zero if symbols are equal and one otherwise. Parallelism was introduced with a FORTRAN 77/VP compiler that vectorizes DO loops when possible for a speedup of 4 on sequence-to-sequence alignment and a speedup of 2–3 on sequence-to-group alignment. A more recent attempt to use vector parallelism was reported by Chaichoompu and Kittitornkun [10] with only a speedup of 1.23 on ClustalW. The Intel C++ compiler was used with the /QxP option to generate Streaming SIMD Extensions (SSE) instructions for a Pentium processor.

Multiprocessor. Mikhailov et al. [60] parallelized all three stages of ClustalW on a shared-memory SGI Origin machine to demonstrate a speedup of 10 with 16 processors. The first stage is easily parallelized since each of the pairwise comparisons can be executed independently and the results stored in a similarity matrix without any conflict. Mikhailov introduced coarse-grained, task-level parallelism with OpenMP [72] directives. Since the first stage typically dominates the run time of ClustalW, most of the speedup comes from parallelizing this stage. The degree of parallelism with this method is limited to the number of all-to-all pairwise comparisons, which is $N(N-1)/2$.

A notable feature of Mikhailov's effort is the parallelization of the guide tree calculation in the second stage, whereas it is often overlooked in many other implementations because it requires the least computation time of all the stages. The clustering algorithm used in the second stage repeatedly finds a minimum element in the similarity matrix. Mikhailov's method searches each row of the matrix in parallel for the minimum element and then reduces the row-wise results to find the overall minimum. In this case, the parallelism is limited to the number of rows in the similarity matrix.

Since Mikhailov uses loop-level parallelism, only a portion of the available parallelism was realized in the third stage. During group-to-group alignment, ClustalW calls a function to determine the score for aligning two profile positions. A profile is derived from a group of sequences and consists of the character frequencies for each column in a group. A temporary matrix with dimensions equal to the length of each profile can be used to store the scores. Mikhailov's method precalculates each element of the scoring matrix in parallel.

Using message passing on a distributed-memory system, Li [45] also parallelized all the stages in ClustalW-MPI for an overall speedup of 14.6 on a 16 processor cluster. The test data consisted of 500 protein sequences with a length of 1100. Li used a fixed-size bundling strategy in the first stage to schedule 80 pairwise alignments to processors in a batch, thus reducing the frequency and overhead of communication. Li was the first to publish more sophisticated parallel methods for the third stage of progressive alignment. One method computes alignments at terminal nodes of the guide tree in parallel (see Figure 1.2c). The problem with this method is that an unbalanced tree can severely limit the number of

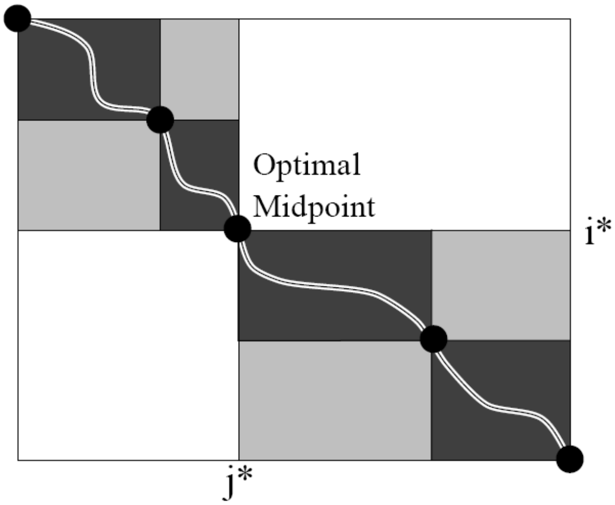


Figure 2.2: Myers and Miller subdivision algorithm. Eugene W. Myers and Webb Miller, Optimal alignments in linear space, *Computer Applications in the Biosciences : CABIOS*, 1988, Vol. 4, No. 1, p. 14, by permission of Oxford University Press.

parallel tasks. Furthermore, even a balanced tree will have limited parallelism near the root. In practice, the guide tree is usually unbalanced.

Another parallel method used by Li is based on the recursive Myers-Miller DP algorithm [64]. The Myers-Miller algorithm solves the pairwise alignment problem by dividing the DP matrix in half and then scanning from opposite corners towards the middle (see Figure 2.2). Where the two scans meet, an optimal midpoint in the traceback path is determined. This point becomes the corner of two subblocks which are in turn divided and scanned for midpoints. The recursion continues until a trivial alignment is encountered. The forward and backward scans can occur briefly in parallel, but must join before determining the midpoint. Each time a midpoint is found, two new subtasks can be spawned for the subblocks. Similar to guide-tree parallelism, recursive parallelism is also limited in the first steps. Using both guide-tree and recursive parallelism, Li only achieved a speedup of 4.3 on 16 processors in the third stage, while the first stage realized a customary linear speedup of 15.8.

The best performance reported for ClustalW using multiprocessors was by Tan et al. [86] with an overall speedup of 35 on an SMP-cluster system with 40 nodes and 80 processors. A speedup of 80 and 9.2 was obtained for the first and third stages respectively. In the third stage, Tan's method also distributes group-to-group alignments to system nodes using a method similar to Li that is based upon guide-tree and recursive parallelism. The main contribution comes from computing the forward and backward DP scans in parallel on processors within a node.

Load balancing strategies can improve the parallel efficiency in the third stage. Luo et al. [55] proposed a dynamic scheduling algorithm that estimates the execution time and communication cost for each task. Since the input for a node in the guide tree is dependent on a prior node, task costs are dynamically estimated after each task completes. The scheduler considers these costs and the current workload of the processors when making scheduling decisions. A peak efficiency of 0.75 is achieved with a speedup of 6 on 8 processors. In a later work, Tan et al. [87] also proposed a load balancing strategy based on tree accumulation. A speedup of 18 was achieved with 32 processors when aligning 3998 protein sequences. The small speedup achieved in the third stage, which is under 10 in most cases, limits the overall speedup of progressive algorithms on multiprocessor systems.

Cell. Sachdeva et al. [78] ported the ClustalW application to the Cell platform for a Stage 1 speedup of 6.51 compared with a Xeon (Woodcrest) processor, but the overall performance was slower by a factor of 1.58. The significance of their effort comes from being the first to experiment with the Cell as a MSA accelerator and illuminating the challenges that must be overcome to achieve a performance improvement. Most of the speedup in the first stage comes from vectorizing the pairwise alignment computations and executing them in parallel on the Cell's eight Synergistic Processing Units (SPUs). However, vector performance is challenged on the SPUs with multiple branches in the DP code and also with table lookups for the similarity score. The second and third stages were executed on the Cell's single, 64-bit Power Processing Unit (PPU). The lower performance of a PPU compared with a Xeon processor explains the overall performance degradation.

Vandierendonck et al. [93] accelerated ClustalW on two Cell BEs by a factor of 8 when compared with a 2.13 GHz Intel Core2 Duo processor running a single thread. Stage 1 is parallelized by vectorizing DP matrix calculations and scheduling the independent pairwise alignment tasks across the 16 available SPUs. Vandierendonck applied loop unrolling and loop skewing optimizations to compute the DP scan with diagonal vector operations. These optimizations are also applied to the group-to-group DP calculations in the third stage. Vandierendonck discovered that a significant portion of the third stage is spent calculating the similarity score between two profile positions. When comparing profile positions, all the character and gap frequencies must be considered. Similar to Mikhailov, these scores are precalculated in parallel, but in Vandierendonck's case, a more sophisticated scheme is proposed to pass precomputed scores from producer tasks through queues to consumer tasks. The PPU executes the sequential portions of ClustalW and load balances worker threads across SPUs with a dynamic scheduler.

Using a Playstation3, Wirawan et al. [100] achieved a peak speedup of 108 for the first stage when compared to a 3.0 GHz Pentium 4. The data set consisted of 1000 protein sequences with an average length of 446. Only the first stage was accelerated, and no overall speedup was reported. Wirawan used a sequence comparison algorithm that differs from ClustalW and has been previously demonstrated on FPGA [69] and GPU [49] accelerators. A count of matching characters is normally determined from an alignment, but in this algorithm the number of identical characters is computed directly by the recurrence relations during the forward scan of DP. By avoiding a full alignment, which requires a traceback procedure, better performance is realized.

GPU. Weiguo Liu et al. [49] were the first to publish MSA acceleration on GPUs and achieved a Stage 1 speedup of 11.7 compared with a 3.0 GHz Pentium 4 processor. Stages 2 and 3 were executed sequentially on the Pentium processor for an overall speedup of 7.2. A single GPU card (GeForce 7800 GTX) was programmed with OpenGL Shading Language

(GLSL). The sequence comparison algorithm uses the same recurrence relations demonstrated on FPGA [69] and Cell [100] accelerators to assist in calculating the number of identical characters.

Yongchao Liu et al. [51] demonstrated an overall peak speedup of 41.53 on 1000 sequences of average length 858 with 1 GPU card (GeForce GTX 280) when compared with a 3.0 GHz Pentium 4. All three stages of ClustalW are accelerated by the GPU, with the parallel portions programmed using CUDA. When pairwise-alignment and guide-tree parallelism is low, cells of DP matrix calculations are computed in parallel. Since CUDA does not support recursion, a stack-based iterative version of the Myers-Miller algorithm was developed. This new version was used for both pairwise-alignments and group-to-group alignments. A separate paper [52] describes the parallel algorithm for the second stage. The neighbor-joining algorithm [79] is accelerated by computing the two innermost loops in parallel. Threads that compute minimum elements for square blocks of the distance matrix are scheduled on the GPU. The best speedup obtained in each of the three stages is 47.13, 11.08, and 5.9 respectively. Again, the small gain in the third stage limits the overall speedup.

FPGA. Reconfigurable computing approaches accelerate the first stage of MSA by computing pairwise alignments with a pipeline of processing elements (PEs). This linear systolic array operates with fine-grained parallelism along a wavefront of cells in the DP matrix. The ClustalW algorithm does not use the score obtained from a pairwise alignment directly. Instead, the number of identical characters in an alignment are used to compute the fractional identity. Oliver et al. [70] accelerates the first stage of ClustalW, but leaves the second and third stages for execution on the host processor. Rather than actually aligning the sequences, a custom algorithm on the accelerator counts the number of identical characters during the forward scan without performing traceback. The best overall speedup was 13.3 compared to ClustalW running on a 3.0 GHz Pentium 4. For Stage 1, a PCI-based accelerator board reached a peak speedup of 50.9 with 92 PEs in a Xilinx XC2V6000.

In another approach, Lin et al. [47] demonstrated an overall speedup of 34.6 using 10 Altera Stratix PEIS30 with a total of 3072 PEs. For the first stage, a speedup of 1697.5 was achieved when compared with a 2.8 GHz Xeon. The number of identical characters is deduced from the comparison score returned from the accelerator and the sequence lengths. Even with the impressive speedup in the first stage, the overall speedup is still limited by the third stage. Greater performance may be achieved, however, by accelerating the third stage of progressive alignment.

2.2.2 Progressive-Iterative Methods

Iterative refinement algorithms have been developed to correct mistakes induced by the greedy strategy of progressive alignment. Most commonly, the iterative algorithms repeat subgroup alignment in the third stage to remove misplaced gaps. A more recent version of ClustalW now includes an iteration option to improve alignment quality, but this quality comes at the expense of more run time. To compensate for the lengthened run time, parallel methods have been introduced to some of the iterative applications. A few programs other than ClustalW have gained enough acceptance to warrant a parallelization effort.

MUSCLE. The iterative approach of MUSCLE starts with two rounds of basic progressive alignment and then repeats tree-guided group-to-group alignments until convergence is reached. As shown in Figure 2.3, a round consists of the three stages that are familiar to progressive alignment. The first two rounds derive pairwise similarity scores during Stage 1 in different ways, wherein the first round uses a faster alignment-free method based on k-mers and the second round uses the multiple alignment from the prior round.

Deng et al. [15] parallelized MUSCLE for a speedup of 15.2 on a 16 processor SMP system using OpenMP. The target data set consists of 50–150 proteins of average length 330. In the first and second rounds, group-to-group alignment following the guide tree is executed in parallel. A queuing module is used to schedule the tasks and make sure children

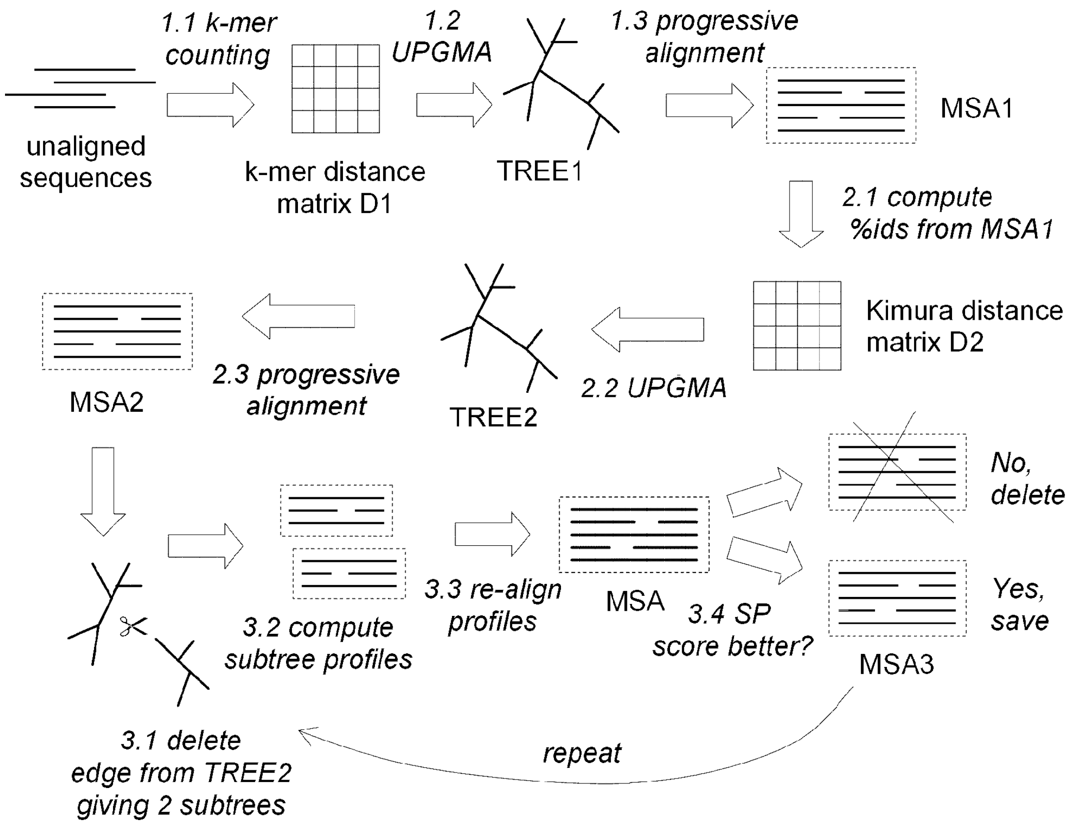

1.1 k-mer counting
1.2 UPGMA
1.3 progressive alignment
unaligned sequences
k-mer distance matrix D1
TREE1
MSA1
2.1 compute %ids from MSA1
Kimura distance matrix D2
2.2 UPGMA
TREE2
2.3 progressive alignment
MSA2
No, delete
Yes, save
MSA3
MSA
3.4 SP score better?
3.3 re-align profiles
3.2 compute subtree profiles
3.1 delete edge from TREE2 giving 2 subtrees
repeat

Figure 2.3: MUSCLE Algorithm. Robert C. Edgar, MUSCLE: multiple sequence alignment with high accuracy and high throughput, *Nucleic Acids Research*, 2004, Vol. 32, No. 5, p. 1793, by permission of Oxford University Press.

nodes are aligned before parent nodes; however, as discussed before, this tree-based method has limited parallelism, and consequently, poor performance is reported for this stage with a speedup between 1 and 2. Most of the speedup comes from parallelizing and executing independently the all-to-all pairwise comparisons in the second round. Deng opted to use a more compute intensive probabilistic sequence comparison algorithm in the second round; therefore, most of the execution time was spent in this stage.

PRALINE. The progressive method of PRALINE has a pairwise sequence alignment stage and a progressive profile alignment stage that correspond to Stages 1 and 3, but the guide tree formation of Stage 2 is avoided. Instead of following a guide tree to align sequences and groups, PRALINE repeatedly chooses the next highest scoring pair to align until all sequences and groups are aligned to produce the final alignment. The highest scoring pair is determined by comparing all sequences with each other at first, and then comparing the aligned pair with the remaining sequences after each iteration.

A parallel implementation of PRALINE by Kleinjung et al. [42] realized a speedup of 10 with 25 processors on a distributed system using a set of 200 random sequences that are 200 residues in length. The pairwise sequence alignment stage is parallelized in the usual way by distributing pairwise alignments tasks to separate processors. In the progressive profile alignment stage, only the comparison of sequences and groups is parallelized. This occurs in a similar way to the first stage by distributing the comparison tasks, but each iteration must collect the results before selecting the highest score.

T-Coffee. While T-Coffee follows a progressive strategy, the first stage consists of a few extra steps that generate a library of pairwise alignments. This library is later used in the third stage to score alignments with a consistency-base objective function. After a round of basic progressive alignment, T-Coffee can iteratively refine the multiple alignment as an option. Each sequence is removed in turn from the multiple alignment and realigned with the remaining sequences.

Zola et al. [104] implemented a parallel version of T-Coffee using a master-worker architecture and message passing to obtain an overall speedup of about 40 on a system with 80 CPUs. Most of the parallelism comes from distributing pairwise alignment tasks with dynamic scheduling for a near linear speedup during library generation. In the progressive alignment stage, a sophisticated dynamic scheduling strategy is used that follows the guide tree, but almost no speedup is seen in this stage with more than 16 CPUs.

Chapter 3

Thesis statement

The strengths of a host microprocessor and an FPGA accelerator are applied to MSA acceleration. New parallel algorithms and a new high-performance architecture are proposed that bring these strengths together to accelerate MSA on a reconfigurable computing system. A main component of the proposed work accelerates the third, progressive-alignment stage of MSA by quantizing the profiles before they are aligned on the FPGA.

Through fine-grained parallelism provided by an FPGA accelerator, a progressive MSA application can produce comparable quality alignments in less time than currently known methods.

The new algorithms are incorporated into an existing MSA program to demonstrate accelerated large-scale MSA. Portions of the MSA application still execute on the host computer since it is efficient at executing serial code with dynamic data structures. The highly-parallel portions of the application are optimized with SSE instructions or accelerated on reconfigurable hardware.

The work consists of the following components with corresponding contributions that are fundamental to accelerating MSA. The next three chapters correspond with these components.

- A reconfigurable computing architecture
- An accelerated pairwise alignment algorithm
- A discrete profile alignment algorithm for the third stage of MSA

As a foundation to the other proposed components, the reconfigurable computing architecture provides a modular framework and an interconnect standard for program development on the FPGA. The modular framework helps developers partition complex FPGA resources (e. g. processors, memory, I/O devices) into manageable units that are connected with a high-performance network. Without an efficient communication architecture, parallel computational resources are limited in performance. This architecture has been published as "A Packet-Switched Network Architecture for Reconfigurable Computing," *ACM Transactions on Embedded Computing Systems*, 9, 1, Article 7 (October 2009), 17 pages.

Accelerated pairwise alignment is used in the third stage as a part of the discrete profile alignment algorithm. A novel aspect of the pairwise alignment algorithm is the ability to handle long sequences of DNA. This algorithm has been published as "Hardware Accelerated Sequence Alignment with Traceback," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 762362, 10 pages, 2009. For pairwise alignment, a speedup of 300 has been demonstrated when compared to a 2.4 GHz Core2 processor.

The contributions of the new discrete profile alignment algorithm in conjunction with the pairwise alignment algorithm advance the capabilities and performance of MSA. This work is the first known to accelerate the third stage of progressive alignment on reconfigurable hardware.

Chapter 4

A Packet-Switched Network Architecture for Reconfigurable Computing

Published in

ACM Transactions on Embedded Computing Systems,

Vol. 9, No. 1, Article 7 (October 2009).

Abstract

A packet-switched network architecture named Qnet and programming interface is presented that simplifies the integration of reconfigurable computing modules within a field-programmable gate array (FPGA). Qnet provides an abstraction layer to the designer of FPGA accelerator modules that hides the complexities of the system, while supporting a high degree of parallelism and performance. The architecture facilitates system design with pluggable, reusable modules. A network protocol is described that supports a three-party communication scheme between an initiator, a sender and a receiver. This protocol allows a master device to manage the state of other devices and the data flow within the system. An example using a high-level language is given. The Qnet architecture opens the computational power of FPGAs to computer scientists and software developers.

4.1 Introduction

The need for greater computational performance pervades many disciplines. In addition, embedded system designers must balance power consumption, cost and other factors to meet application requirements. As microprocessors reach limitations on clock frequencies, parallel solutions are used to meet the performance challenge. For example, multiprocessor systems-on-chips have been successfully employed in many high-performance embedded platforms.

Reconfigurable computing is another approach used to address demanding computational problems [7, 30]. Field-programmable gate arrays (FPGAs) are commonly used in reconfigurable systems as coprocessors to accelerate repetitive or parallel portions of an application. These portions benefit from the FPGA's configurable logic that operates with fine-grained parallelism. While FPGAs provide logic as a configurable resource, research is leading to other devices that contain a mix of higher-level configurable components; for instance, processors, floating-point and integer units, caches, memory interfaces, and on-chip networks.

4.1.1 Benefits and Challenges

For some applications, reconfigurable computing has demonstrated a performance advantage over microprocessors with gains ranging up to several orders of magnitude. This increase in performance may be traded for reduced system size, power, and cost. The potential for these benefits motivates ongoing research; however, several challenges remain for wide scale adoption of reconfigurable computing [21].

Since FPGAs operate at slower clock rates, they must use a high-degree of parallelism to exceed the performance of commodity processors. Overcoming this challenge often requires careful analysis to devise or choose a parallel algorithm suitable for the problem.

Managing the flow of data through an FPGA is another research challenge. For FPGAs to effectively participate in computations, adequate communication paths are required. I/O bottlenecks between the processor and FPGA frequently limit reconfigurable systems from greater performance [91]. Given the need for parallelism, scalability, and throughput, research is converging to on-chip networks as the connection architecture of choice [13, 39, 57, 74]. As FPGA capacities increase, serial on-chip buses do not scale to handle the intermodule communication demands; however, on-chip networks do scale to meet the demands with parallel data paths while still comparing favorably with buses on resource usage.

Programming reconfigurable systems poses yet another challenge. The FPGA is a vast array of configurable logic. Traditionally, hardware description languages, such as VHDL and Verilog, have been used to specify FPGA configurations. Unfortunately, these languages are unsuitable for writing algorithms in a way familiar to software developers. A current trend is to use algorithmic, high-level languages offered by a few commercial vendors.

Although languages and design tools ease the development of an FPGA accelerator module, connecting it to other system resources (processors, memory, I/O devices) remains tedious. Since each resource has a different hardware interface, system integration often requires hardware expertise. One approach to the problem is to provide platform development kits with libraries that can be called and linked directly to the accelerator code, but this ties the resources to the accelerator and restricts other clients from accessing them.

Given the current challenges, a standard communication architecture is needed for reconfigurable computing. The Message Passing Interface (MPI) [62] and CORBA [71] are examples of standards that address similar challenges. These standards allow interaction with resources at remote locations in the computing environment while hiding the complexities of the system. This benefit and others, such as portability and interoperability, may also be realized by reconfigurable computing through a standard communication architecture. Beyond the standards offered by MPI and CORBA, a reconfigurable computing standard will need to offer a common framework for both hardware and software components distributed across chip boundaries.

4.1.2 Requirements

A communication architecture meeting the following requirements addresses or mitigates many of the challenges of reconfigurable computing:

Network: Provide a high-performance, efficient, communication network that connects various modules with a standard hardware interface.

Protocol: Provide data transport and access to another module's resources via the network with user-extensible functions, data types, and access patterns.

Interface: Provide a communication abstraction layer or application programming interface to the network for the module designer (e. g. C language binding, VHDL module).

Methodology: Allow module design and verification in a high-level language of choice with module connection and system integration assisted by a platform-level tool.

This research introduces a network architecture named Qnet along with a flexible protocol and a programming interface. Qnet is a packet-switched network that connects modules within a reconfigurable system. Modules encapsulate sharable devices or resources and may reside within or be external to an FPGA. In conjunction with Qnet, the Distributed Access Protocol (DAP) is presented, which offers a unified solution to data management. Qnet provides a highly parallel framework for computationally intensive problems that meets the prescribed requirements and opens the computational power of FPGAs to computer scientists and software developers.

4.2 Background and Related Work

The Qnet protocols and architecture build on related work in several areas. The communication protocols discussed in this section originated with research involving conventional clustered computing systems. The network interface cards in these systems often provide hardware assistance in packet processing. Qnet generalizes the concept of packing and unpacking data from each end of a communication channel through the *access pattern,* which will be discussed later. Extending work at the hardware level, Qnet provides hardware assistance to access memory in common patterns through a unified network interface.

4.2.1 Communication

Communication methods are usually classified as one-sided or two-sided. A one-sided method [95, 2] accesses the memory of a remote processor without direct involvement of the other processor. The communication activity is transparent to the remote node. Even though only one side is actively involved in the communication, two parties still participate. A direct memory access (DMA) engine or a processor under interrupt must transfer the data on the remote node.

Two-sided communication methods use the Send/Receive paradigm. Both parties are actively involved in the communication and each specifies buffers of data. Examples of this paradigm include the sockets interface and MPI.

Each of these methods has strengths and weaknesses; nevertheless, most are memory centric and not general enough to directly access the functional resources of various FPGA and host modules. A single, unified protocol is needed that allows modules to initiate data transfers and operations within the system. Since all user data access patterns can not be anticipated, the protocol needs to be extensible.

4.2.2 Architecture

With the increase in size of FPGAs, Diniz and Park [16] envision the need for data reorganization engines that offload the burden of data management and manipulation in available memories. Some common memory access patterns are listed (e. g. splitting, padding, merging, transpose, scatter/gather). A programmable switching network is proposed for managing the flow of data between memories, but other module types and nonmemory access patterns are not addressed.

A considerable number of on-FPGA communication architectures have been proposed. Mak et al. [57] presents a taxonomy of on-FPGA communication architectures with representative examples. Pionteck et al. [74] also gives an overview of several architectures specific to dynamic reconfiguration. Most closely related to this work is DIMEtalk [80], a tool that configures a communication network within an FPGA system. Nodes connect user defined hardware to the network through two types of interfaces: RAM and FIFO memory.

Of the known on-chip architectures, none present a single network application programming interface for use by both processor-based modules and hardware accelerator modules. Also, none provide a single application layer protocol for managing operations and data flow in the system. The on-chip network architectures only address the packet routing issues up to the transport and network layers and do not present a programming interface from the module's perspective.

4.3 Qnet

Qnet satisfies the module interconnect requirement in an FPGA system as specified in Section 4.1.2. The module types vary in function and purpose, like those found in typical bus-based FPGA systems (e. g. processors, memory interfaces, peripherals, accelerators). Figure 4.1 shows a system constructed from modules connected through a standard hardware interface. Modules designed as initiators access resources in other modules through the network infrastructure. A layered network protocol allows modules to implement the functionality needed at the endpoint without imposing unnecessary complexity, whether in hardware or software. Qnet can also coexist with bus-based modules that do not require the benefits of a network.

The hardware interface standard for Qnet modules is key to fulfilling the methodology requirement. Modules may be implemented by any method, as long as they conform to

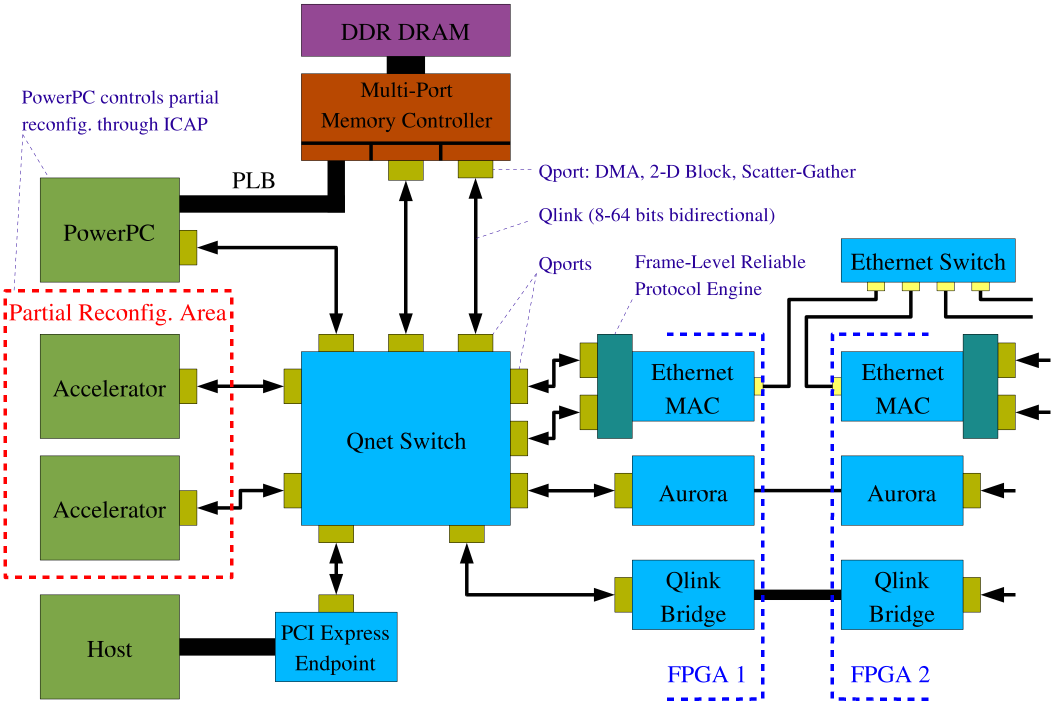


Figure 4.1: Example Qnet system

the Qnet interface standard. Compatible modules may be integrated into a system with the assistance of a platform-level tool, such as XPS in the Xilinx Embedded Development Kit (EDK) [101]. Through a graphical user interface, modules are selected from a library, placed, and connected. The tool generates the requisite hardware description files that bind the modules, and it automates design rule checks and other tests for misconfiguration.

The main contributions of Qnet address the interface and protocol requirements:

- An application programming interface to interact with other modules in the system. The interface is the same for both processor based modules and hardware accelerator modules, and it hides system hardware details opening FPGA resources to software developers.
- A three-party network communication protocol. This protocol allows a master module to manage the operation of other modules and the data flow within the system.
- The generalization of data access patterns that are applied at network endpoints. This provides the ability to use diverse module resources over the network.

- The flexibility to define data access patterns. Users with specific needs can define a new access pattern that operates within the Qnet architecture without defining a new protocol and a new programming interface.

4.3.1 Network Components

The basic network components consist of switches, Qports, and Qlinks. As the central figure in the network, the switch provides a communication path to other modules (see Figure 4.1). Depending on the number of modules in the system, a switch with the appropriate number of ports is typically used. Qports are the interface between modules and the network and are the addressable endpoints of a communication. Qports are connected by Qlinks, which consist of paired, unidirectional, point-to-point communication channels that can be implemented with varying bit widths (e. g. 8, 16, 32, 64-bits). Qlinks implemented with narrow widths minimize resources for lower-speed devices, while wider links support higher-speed devices such as PCI Express [4]. Each Qport has word-based flow control that will apply back-pressure on a link, delaying communication until the port is ready to receive. Hence, packets are not arbitrarily discarded, and the requirement to buffer an entire packet at the input of a module is removed while still maintaining performance.

Qnet allows for different designs and implementations of the switch, depending on the application requirements. For instance, switch ports may have no buffering or various amounts of buffering determined by the application packet size and flow patterns. The address table size, port priority, and forwarding method are not specified by the architecture. Switches may be implemented with various port widths and internal data path widths to meet the bandwidth demands of the application. Latency through the switch can be as low as one clock cycle.

Accelerator modules often need multiple data paths for high performance. Examples of this include matrix and vector functions with two operands and a result. These functions can be implemented as a pipelined accelerator module with streaming data on two Qports.

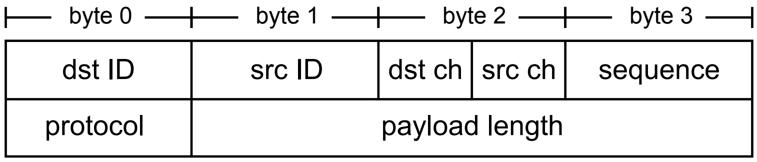


Figure 4.2: Qpacket header

Since Qlinks are bidirectional, one link can handle operand A and a result, while the other link handles operand B. The switched architecture of Qnet supports concurrent data paths to a module when the data comes from separate sources. When both operand streams come from a common memory source, a multiport memory controller [73, 102] is an option that can supply the streams on concurrent links up to the available memory bandwidth.

More sophisticated architectures beyond a single switch connecting on-chip modules can be constructed with Qnet. For instance, modules on multiple FPGAs can be networked with the appropriate bridge at the chip interface, which may range from simple parallel signals to a multigigabit serial interface. From a module's perspective, access to remote network resources is through a simple Qport connection, and the network pathway is invisible to the programmer.

4.3.2 Network Protocol

The lowest abstraction offered by Qnet is the reliable transfer of data packets between endpoints. Packets consist of a small header (see Figure 5.5) and a payload that ranges in size up to 16 MB. Endpoints are specified by a unique port identifier and a channel number. The port identifier is used to route a packet to a specific Qport and the channel number is a means to address subprocesses within a module, thereby providing virtual channels over a single Qlink. Normally, if bandwidth and latency are an issue, multiple Qlinks are used in a module design instead of sharing a Qlink with multiple channels.

Qnet has hardware support for marking the header/payload boundary of a packet. On receipt of a packet, the boundary marking simplifies the separation of the header from the payload, which translates to a simpler protocol stack and better communication performance.

The device driver or application flexibly sets the boundary marker, which is implemented with a sideband signal asserted on a word in a packet.

4.4 Distributed Access Protocol

The Distributed Access Protocol (DAP) enables processors and modules to access data and functionality in other Qnet modules with a standard abstraction. For instance, memories of various types distributed through the system can be accessed via the network. When a processor sends a transfer request message to a memory module using DAP, the module responds with the data. A data transfer engine in the memory module's Qport reads the memory in the specified access pattern and forms a packet on the link. The memory type can be on-chip RAM or off-chip SRAM/DRAM. Accelerator modules and slave devices implementing I/O functions are accessed through standard and user defined messages delivered over Qnet.

The protocol is generally applicable to any network with reliable delivery at the transport layer, which allows DAP to be implemented over TCP/IP on a cluster system. In the context of this article, DAP is implemented in hardware and layered on the basic Qnet packet as described in Section 4.3.2.

4.4.1 Three-Party Communication

The subjects of three-party communication involve: (i) an initiator, (ii) a sender, and (iii) a receiver. With three-party communication, the initiator initiates a data transfer, but is not required to participate in the transfer between the other parties. The transfer is initiated by sending a short transfer request message to the sender, which completes the request by transferring data to the receiver (see Figure 4.3). With existing methods, like Active Messages and RDMA, the initiator is also the sender or the receiver. Within an embedded system, the three-party communication scheme allows a controller to manage devices and data flow within the system without being directly in the data path. Control packets injected

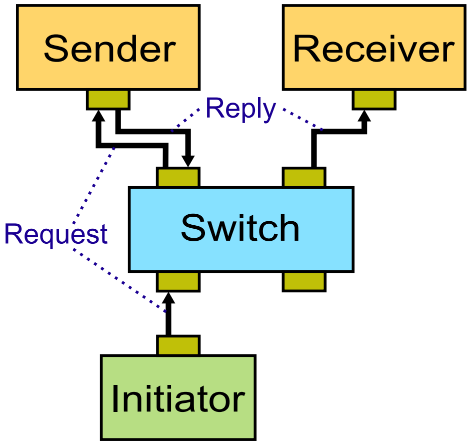


Figure 4.3: Example of three-party communication

into the network initiate the movement of data from one module to another. Multiple sources for control packets are possible in a system.

To visualize the benefit of three-party communication, take, for example, a case where a processor is tasked with sending a large block of data from memory to a device. With DAP, a transfer request from the processor is sent to the memory module specifying the device as the destination. Data flows directly from memory to the device. A two-party scheme requires two transfers, one from the memory to the initiator, and another from the initiator to the device. Another approach, used by distributed DMA, is for the processor to configure and initiate a DMA transaction on the device, which then performs the data transfer. The three-party communication scheme provides functionality similar to distributed DMA, but without the shared-bus limitations.

4.4.2 Access Patterns

An access pattern specifies a method or a template for processing data at each end of a communication channel. As an example, Bove et al. [5] use an access pattern to produce a stream of data to a processor by addressing a multidimensional array in random access memory. DAP generalizes the access pattern concept to include more than memory centric patterns. Access patterns are defined by a type and a variable number of parameters. At

Table 4.1: Sample of Access Patterns

Access Pattern	Description
DEFAULT	receiver determines pattern
SEQUENTIAL	access sequential addresses
OFFSET	access sequential addresses from an offset
FIXED	access a fixed location or port
BLOCK	access a 2-D block of memory
TRANSPOSE	access matrix elements in column major order
MODIFY	modify an element in place by an increment
TREQUEST2	initiate a transfer with reply back to initiator
TREQUEST3	initiate a transfer to a 3rd party
MESSAGE	initiate user defined functions or send status/control
CODE	execute packet payload as code

the sending end, a packet payload is assembled according to an access pattern, while on the receiving end, the packet payload is disassembled. DAP supports different access patterns at the source and destination of the data packet, thus enabling data reorganization through a transfer. For example, the transpose operation can be performed while moving matrix data from a distributed memory to an accelerator, thereby avoiding the need for a transposed copy of the matrix.

DAP access patterns encompass several standard patterns as well as user-defined patterns. A new access pattern is added by defining the access pattern type and the corresponding parameters. A process to handle the new type is implemented at the endpoint. The DAP protocol need not change to accommodate a new access pattern. Some access patterns have parameters of fixed length, others have variable length parameters. A few standard access patterns are listed in Table 4.1.

Access patterns are flexibly specified at communication endpoints. Packet headers convey access patterns for use at remote endpoints. The DEFAULT access pattern serves as a null pattern that allows a receiver to specify the access pattern, otherwise the access pattern in the header is used. The SEQUENTIAL access pattern is a fundamental access pattern, and is similar to the memory copy function. Given a starting address, data is moved to or from sequentially incrementing addresses up to a maximum length (see Figure 4.4a). More

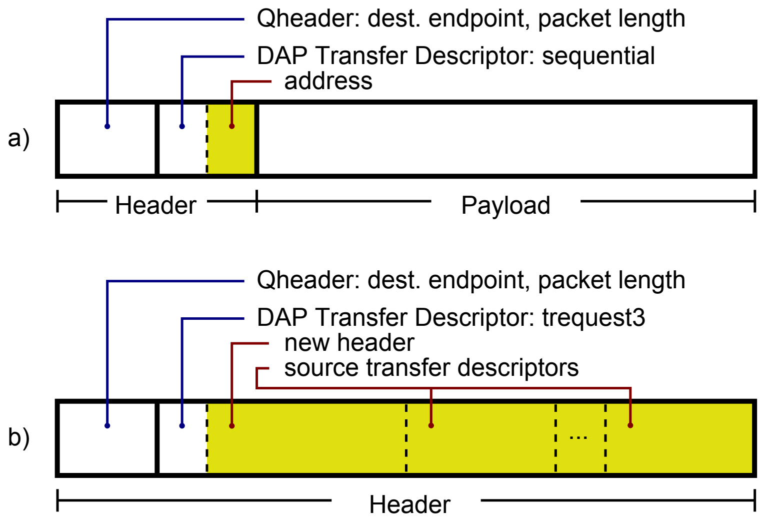

Qheader: dest. endpoint, packet length
DAP Transfer Descriptor: sequential
address
a)
Header
Payload
Qheader: dest. endpoint, packet length
DAP Transfer Descriptor: trequest3
new header
source transfer descriptors
b)
...
Header

Figure 4.4: Packet format: (a) sequential write data packet, (b) transfer request packet. The highlighted portions are the access pattern parameters.

sophisticated processor-in-memory (PIM) functions are feasible with access patterns. For example, custom code fragments sent in a packet with the CODE access pattern are executed near the memory at full bandwidth.

The TREQUEST2 and TREQUEST3 access patterns initiate a transfer at another endpoint. The two-party version assumes the destination of the reply is the initiator, while the three-party version specifies another destination. In the three-party case, the destination endpoint is used in building a new header for the reply packet. The request packet also has one or more transfer descriptors used to build the reply packet payload (see Figure 4.4b). Each transfer descriptor contains an access pattern that is applied during payload construction.

Scatter/gather operations are accomplished by chaining transfer descriptors together within a transfer request access pattern or through sending separate packets. Under the control of an initiator, fragments of memory are moved about the system via the network in a consistent method. These operations benefit packet processing applications by avoiding the need to copy data. Other high-performance applications benefit by efficiently dispersing data to parallel processing elements and collecting the results.

The generalized access pattern provides the ability to use diverse module resources over the network, all with a single protocol. Access patterns are applied at each end of a communication instance to produce and consume packet payloads. When a module is attached to the network with defined access patterns, its resources may be controlled and accessed via network communication.

4.4.3 Programming Interface

The DAP application programming interface provides a consistent method to access modules and their resources over the network through a small set of routines, which are used to build higher-level routines with specialized functionality. The programming interface defines data structures specific to DAP. A `QPORT` is system dependent and provides a reference to a module's Qnet port. Modules may have more than one Qport. An `ENDPOINT` consists of an identifier and a channel number. A `TDESC` represents a transfer descriptor, which contains an access pattern type and length along with flags and a tag. The flags specify options such as completion notification, synchronization control, and packet marker settings, while the tag is used to match reply packets with a waiting receiver.

A packet is formed by a call to `DAP_SendHdr` to send the header, followed by `DAP_SendPay` to send the packet payload. Likewise, `DAP_RecvHdr` and `DAP_RecvPay` are called to receive a packet. The payload routines are not necessary when a payload does not exist. As for the header routines, the endpoint parameter specifies the destination when sending, and returns the source when receiving. The length parameter specifies the maximum packet length on send and returns the packet length on receive. The transfer descriptor and parameters operate in a similar fashion. The flags specify packet marker settings and other options for the local transfers.

For streaming applications, a packet payload may be transferred in segments with multiple calls to the payload routines `DAP_SendPay` and `DAP_RecvPay`, after transferring the header. Since Qlink flow control is handled on a word basis, packet data may be transferred one word at a time. This alleviates the need to buffer a whole packet when the payload contains a sequence of elements. To avoid buffering, elements must be aligned and transferred in multiples of the link width. Data is available for immediate consumption at the receiving endpoint without waiting for a complete packet transfer.

This programming interface is available to accelerator modules written in a high-level language, such as Handel-C [58], and also to programs running on a host machine written in C/C++. In the latter case, an application may communicate with other Qnet modules through a host interface, such as PCI Express. From a programmer's perspective, accessing network resources is accomplished through a Qport with a consistent programming interface.

4.5 High-Level Example

To illustrate the usage of DAP and its programming interface, Handel-C is used to implement an accelerator for multiplying two matrices. In this tutorial example, the accelerator is connected to the network through two Qports and receives operand data from two separate memory modules, which contain the square matrices A and B. Data transfers between memory and the accelerator are initiated by a third party. The accelerator is designed to calculate the dot product of two vectors. To implement a matrix multiply, the row vectors of A and the column vectors of B are sent systematically to the accelerator. The resulting scalar values are transferred from the accelerator to the final locations in C.

The most significant portion of the accelerator code is shown in Figure 4.5, which depicts the three phases of the dot product routine. The first phase demonstrates calls to `DAP_RecvHdr` and `DAP_RecvPay` to acquire the operand vectors A and B. The access patterns are discarded in this case, since the streams are assumed to supply consecutive vector elements. The vectors A and B are received in parallel on the two input channels by using the Handel-C `par` construct. The second phase performs the dot product producing a scalar result. Phase three demonstrates sending the result from the accelerator to a destination memory module with a call to `DAP_SendHdr` and `DAP_SendPay`. The destination endpoint

```
void DotProduct(chan QPORT *inA,
                chan QPORT *inB,
                chan QPORT *outB,
                ENDPOINT *dst_ep,
                TDESC *dst_tdesc,
                unsigned 32 *dst_param,
                unsigned n) // elements
{
    unsigned 32 A[MAX_ELEM], B[MAX_ELEM];
    unsigned 32 result;
    unsigned i;

    // 1) Receive Operands in Parallel
    par {
        ENDPOINT    epA,     epB;
        unsigned 8  lengthA, lengthB;
        TDESC       tdescA,  tdescB;
        unsigned 32 paramA,  paramB;
        DAP_RecvHdr(inA, &epA, &lengthA,
                    &tdescA, &paramA, 0);
        DAP_RecvHdr(inB, &epB, &lengthB,
                    &tdescB, &paramB, 0);
    }
    par {
        DAP_RecvPay(inA, A, n*ELEM_SIZE, 0);
        DAP_RecvPay(inB, B, n*ELEM_SIZE, 0);
    }
    // 2) Multiply-Accumulate
    result = 0;
    for (i = 0; i < n; i++) {
        result += A[i] * B[i];
    }
    // 3) Send Result
    DAP_SendHdr(outB, dst_ep, ELEM_SIZE,
                dst_tdesc, dst_param,
                F_BEGP|F_MARK);
    DAP_SendPay(outB, &result, ELEM_SIZE,
                F_ENDP);
}
```

Figure 4.5: Example accelerator code

```
ENDPOINT req_ep;
TDESC req_tdesc;
unsigned 32 req_param[6];

req_ep.id = src_id;
req_ep.ch = src_ch;
req_tdesc.ap_type = APT_TREQUEST3;
req_tdesc.ap_length = 24;
req_tdesc.flags = 0;
req_tdesc.tag = 0;
// destination header info.
req_param[0] = ;// endpoint
req_param[1] = ;// packet length
// destination transfer desc.
req_param[2] = ;// TDESC w/default AP
// source transfer desc.
req_param[3] = ;// TDESC w/sequential AP
req_param[4] = ;// address
req_param[5] = ;// length

DAP_SendHdr(out, &req_ep, length,
            &req_tdesc, req_param,
            F_BEGP|F_MARK|F_ENDP);
```

Figure 4.6: Example transfer request

and access pattern arguments are determined prior to calling the dot product routine by interpreting a transfer request from the initiator. This instructional example can be made more efficient by pipelining the multiply-accumulate operations and double-buffering the operands so that computation overlaps communication. Partitioning computation on the accelerator to perform a submatrix multiply would also improve performance.

The initiator controls the matrix multiply by issuing a series of three-party transfer requests to the memories and accelerator. For each dot product, two requests to memory are needed for the operand vectors and one request to the accelerator for the result. Figure 4.6 shows a transfer request for a row of A. A transfer request for a column of B is similar except a transpose access pattern is used instead of a sequential access pattern. In a transfer request for the result, the destination access pattern contains the address of an element in matrix C. The request is delivered to the source of the transfer by assigning the source ID as the request packet endpoint. The request parameters include three sections: (i) the destination header information, (ii) the destination transfer descriptor, and (iii) the source

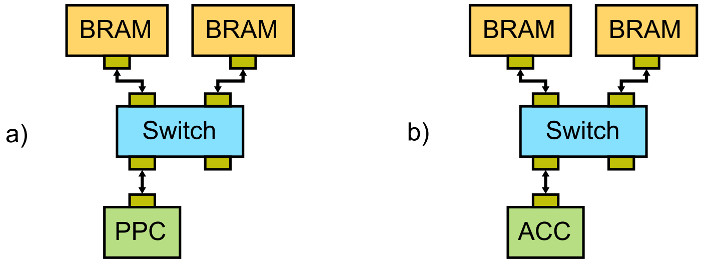


Figure 4.7: Benchmark systems with (a) PowerPC, and (b) accelerator as initiator.

transfer descriptor. Since the request does not involve a payload, only `DAP_SendHdr` is called with all the packet marker flags.

4.6 Experimental Setup

Two basic systems were implemented to evaluate Qnet performance (see Figure 4.7). A PowerPC processor initiates transfers with two block RAM memory modules in one system and a hardware accelerator initiates transfers in the other system. The Xilinx EDK assisted system construction with component placement, connection, and configuration through the graphical user interface. Each component, except for the accelerator module, was written in VHDL. All components were made available in the EDK IP Catalog. The benchmark systems were implemented on a Xilinx Virtex-4 FX100 FPGA, which contains two embedded PowerPC processors and 42,176 slices of configurable logic.

Switch. A 4-port switch connects the three other modules using 32-bit Qlinks that run at 200 MHz. To minimize resource usage and latency, the implementation uses a fixed address table, a fixed port priority resolution scheme, and a cut-through packet forwarding method. Once the switch determines the route, the connection from input port to output port remains the same for all following words of the packet. The full packet is not buffered in the switch.

PowerPC. The Virtex-4 embedded PowerPC is interfaced with Qnet through the On-Chip Memory (OCM) interface. Two 4 KB, dual-port RAMs and a set of control and status registers are memory mapped into the processor address space. One dual-port RAM is used for transmission and the other is used for reception. The PowerPC runs at 400 MHz, while the OCM interface is clocked at 100 MHz. The benchmark application accesses the Qnet modules through a library written in C and compiled with gcc for the PowerPC.

Accelerator. The accelerator module is written in Handel-C and compiled with the DK Design Suite [59] to produce an EDIF file. A stub module, compatible with the EDK, instantiates the Handel-C accelerator through the standard Qport signals. The Handel-C to Qnet programming interface is implemented through a library written in Handel-C, and is callable by the accelerator application.

FPGA Block RAM. The FPGA Block RAM memory module is implemented using 64 KB of on-chip block RAM. The module responds to DAP read and write requests, allowing memory access over the network; and it supports concurrent read and write operations. The block RAMs use the 200 MHz Qnet clock.

4.7 Results

Three benchmarks are run on each of the evaluation systems: (i) A one-sided write from the initiator to memory, (ii) a three-party communication involving a transfer between memory modules, and (iii) a one-sided read from memory to the initiator. The initiator in the systems is either the PowerPC or the Handel-C accelerator. Benchmark timing surrounds a loop that executes the communication instances 2^{20} times in a streaming fashion. When clocked at 200 MHz, the maximum theoretical bandwidth of a 32-bit Qlink is 763 MB/s per direction (1 MB = 2^{20} bytes). The theoretical minimum latency for each of the benchmarks is 30 ns, 50 ns, and 80 ns respectively, which is determined only by packet word length.

Table 4.2 shows the modest resource usage of the various Qnet components used by the benchmark systems. The Qnet components in each system account for about 4% of the total FPGA slice resources.

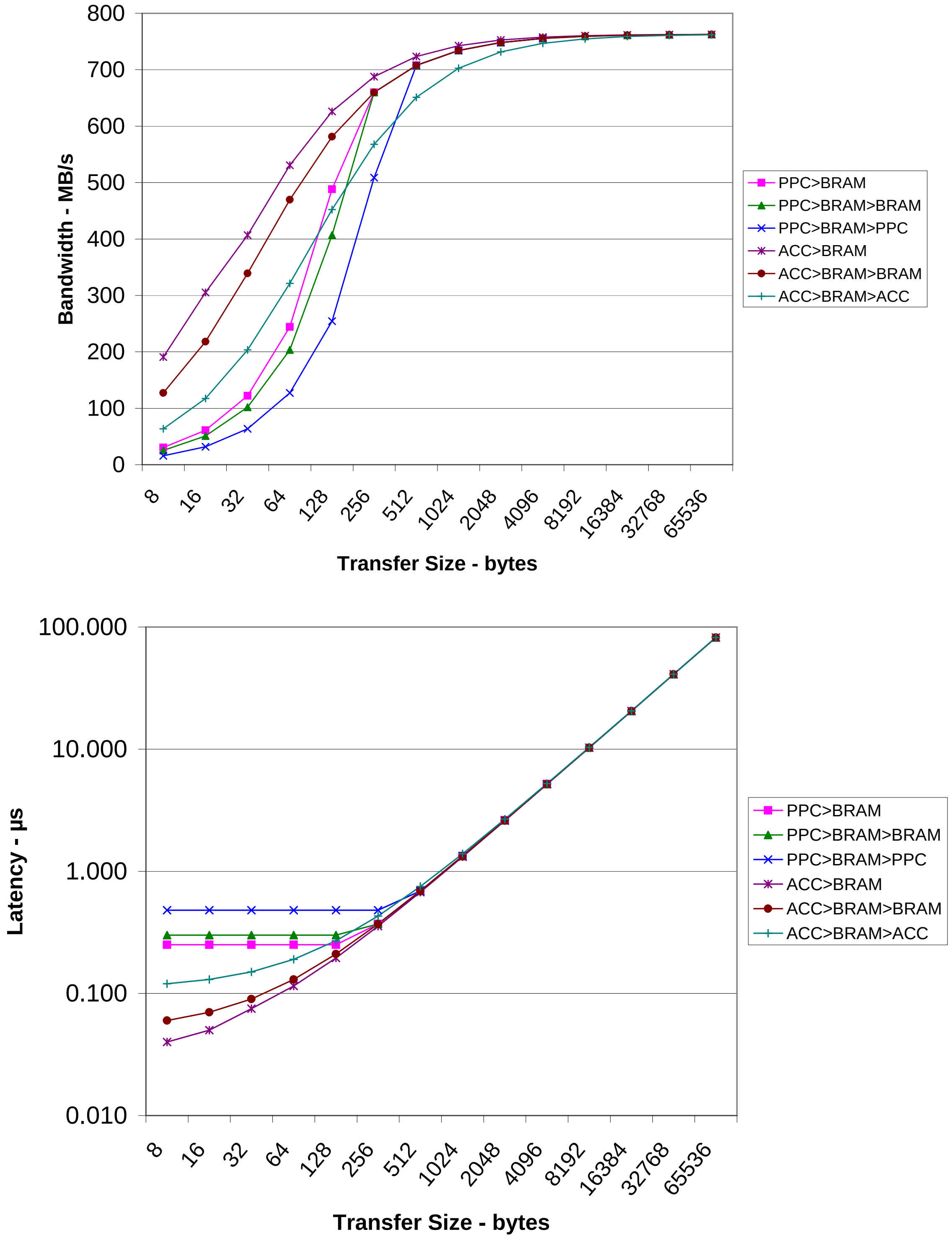


Figure 4.8: Qnet performance. The legend indicates the parties involved in the transfer where PPC = PowerPC, ACC = Accelerator, BRAM = Block RAM. Note that accelerator results are very near theoretical peak values.

Table 4.2: Qnet Resource Usage

Qnet Component	Slices	FPGA Percentage
4-Port Switch	464	1.1%
Block RAM Qport	504	1.2%
PowerPC Qport	189	0.4%

Figure 4.8 shows the Qnet communication performance for the two benchmark systems. When the PowerPC is the initiator, both reads and writes to block RAM (BRAM) reach a bandwidth of 756 MB/s. Transfers between BRAMs with 64 KB packets reach 762 MB/s, which is within 1 MB/s of the theoretical maximum. The latency for a small write packet to BRAM is 250 ns. When reading BRAM, a small packet is returned in 480 ns. The processor interface adds about 200 ns of latency for each packet send or receive.

By avoiding processor overhead, the hardware accelerator module greatly reduces the one-sided write latency to 40 ns. This is only 10 ns away from the theoretical minimum latency. The three-party transfer is reduced to 60 ns, and a read takes 120 ns. For larger packet sizes, the accelerator bandwidth is comparable with the PowerPC.

4.8 Conclusion

Qnet addresses many of the challenges associated with reconfigurable computing. Qnet encourages parallelism, provides high-performance communication, and supports modular design with high-level languages. Thus, the Qnet architecture facilitates computer scientists and software developers in utilizing the computational power of FPGAs.

The key features of Qnet are its modularity and standard interfaces. The hardware interface enables a platform-level tool to configure the system with pluggable components from a library and to integrate user-developed accelerator modules. The programming interface hides the complexities of the system and facilitates code reuse. Through communication packets, the Distributed Access Protocol allows any system module to access another's resources. Standard and user-definable access patterns direct packet assembly and disassembly at endpoints. With a three-party communication scheme, control packets injected into the system may initiate data flow and invoke operations.

The packet-switched architecture of Qnet realizes high performance and efficient resource usage on FPGA technology. For example, a 32-bit Qlink achieves a bandwidth of 762 MB/s for large packets and a latency of 40 ns for small packets, which are very near the theoretical peak values. Also, a 4-port switch requires only about 1% of the FPGA's resources. In support of parallelism, the switched architecture allows concurrent communication paths between modules.

Chapter 5

Hardware Accelerated Sequence Alignment with Traceback

Published in the

International Journal of Reconfigurable Computing,

Vol. 2009, Article ID 762362

Abstract

Biological sequence alignment is an essential tool used in molecular biology and biomedical applications. The growing volume of genetic data and the complexity of sequence alignment present a challenge in obtaining alignment results in a timely manner. Known methods to accelerate alignment on reconfigurable hardware only address sequence comparison, limit the sequence length, or exhibit memory and I/O bottlenecks. A space-efficient, global sequence alignment algorithm and architecture is presented that accelerates the forward scan and traceback in hardware without memory and I/O limitations. With 256 processing elements in FPGA technology, a performance gain over 300 times that of a desktop computer is demonstrated on sequence lengths of 16000. For greater performance, the architecture is scalable to more processing elements.

```
--TTCT--T--TAGATTC
CCTTCTACTGCTA-CTTC
```

Figure 5.1: Example pairwise alignment

5.1 Introduction

Searching and comparing biological sequences in genomic databases are essential processes in molecular biology. The collection of genetic sequence data is increasing exponentially each year and consists mostly of nucleotide (DNA/RNA) and amino acid (protein) symbols. Approximately 3 billion nucleotide pairs comprise the human genome alone. Given the large volume of data, sequence comparison applications require efficient computing methods to produce timely results.

Biologists and other researchers use sequence alignment as a fundamental comparison method to find common patterns between sequences, predict protein structure, identify important genetic regions, and facilitate drug design. For example, sequence alignment is used to derive flu vaccines [56] and by the nation's BioWatch [29] program in identifying DNA signatures of pathogens. Sequence alignment consists of matching characters between two or more sequences and positioning them together in a column. Gaps may be inserted in regions where matches do not occur to reflect an insertion or deletion evolutionary event. A count of the matching characters results in a measure of similarity between the sequences. Pairwise alignment involves two sequences (see Figure 5.1) and multiple alignment considers three or more sequences. Finding the optimal multiple sequence alignment is NP-hard in complexity. As a first step, multiple alignment algorithms [88, 68] often compute a pairwise alignment between all the sequences.

Global and local pairwise alignment are the two most common alignment problems. Global alignment [65] considers both sequences from end to end and finds the best overall alignment. Local alignment [82] identifies the sections with greatest similarity and only aligns the subsequences. Both alignment problems are typically solved with dynamic programming (DP), which fills a two dimensional matrix with score or distance values in a forward scan from upper left to lower right, followed by a traceback procedure. Traceback occurs from a designated lower right position following a path to upper left, thereby determining the best alignment.

The computational cost for an optimal sequence alignment increases exponentially with the length of each sequence and with the number of sequences. This complexity poses a challenge for sequence alignment programs to return results within a reasonable time period as biologists compare greater numbers of sequences. Using current methods, an alignment program may run for days or even weeks depending on the number of sequences and their length.

Unlike most acceleration methods that focus on sequence comparison, this research describes and evaluates a space-efficient, global sequence alignment algorithm and architecture that includes traceback for implementation on reconfigurable hardware. Given a pair of sequences, the accelerator returns a list of edit operations constituting the optimal alignment. A library of accelerator functions is easily incorporated into multiple sequence alignment programs that run on platforms equipped with reconfigurable hardware.

5.2 Related Work

Most efforts to accelerate bio-sequence applications with hardware have focused on database searches. Ramdas and Egan [77] compare several of these architectures in their survey. Given a query sequence, an entire genetic database is scanned looking for other sequences that are similar. Searching a genetic database for matches with a bio-sequence is similar in nature to a search of the web that returns “hits” sorted by relevance. Accelerating a database search is a simpler problem than alignment. Only the score for the comparison is computed by hardware in the forward scan, whereas alignment requires traceback in addition to the forward scan. The sequence comparison problem can be mapped to a linear systolic array of processing elements (PEs) requiring $O(\min(m, n))$ space, where m and n are the lengths of the sequences. However, global alignment necessitates extra storage for traceback pointers and a traceback procedure, which are not addressed by sequence comparison solutions.

Traceback support in hardware has the most benefit when the traceback path spans a significant portion of the DP matrix. Global alignment applications realize the greatest performance gain because the traceback path extends across the entire DP matrix, whereas local alignment applications with a shorter path show less benefit. After a forward scan in hardware, any alignment in software must recompute the DP matrix and traceback pointers for the section of interest before determining an optimal traceback path. For instance, accelerated database search applications may compute an alignment in software only between high-scoring matches and the query sequence after the comparison phase. These search applications usually run in acceptable time with relatively short query sequences; however, comparative genomic applications commonly align long sequences at greater computational cost and stand to benefit from accelerated alignment. Examples include whole genome alignment [11], whole genome phylogeny [14], and computation of pathogen detection signatures [81].

The predominant, non-parallel algorithms for global sequence alignment are described by Gotoh [31] and Myers-Miller [64]. Both algorithms execute in $O(mn)$ time. The algorithm presented by Gotoh requires $O(mn)$ space, while the algorithm of Myers-Miller needs only $O(\log m + n)$ space, but it incurs a factor of 2 time penalty. Most of the space is used to hold values of the DP matrix and the traceback pointers. Saving all traceback pointers in an array requires only one forward scan through the DP matrix followed by one traceback pass. Otherwise, multiple passes through the DP matrix are required if not saving all the traceback pointers. The downside of saving all the traceback pointers is the $O(mn)$ space requirement, which can be significant for longer sequence lengths or prohibitive when limited by FPGA memory.

A few efforts propose hardware methods for accelerating pairwise alignment and traceback. The work presented by Hoang and Lopresti [37] describes an FPGA architecture which consists of a linear systolic array of PEs that output traceback data. However, the type of sequences are limited to only DNA and the sequence length is limited by the number of PEs on the accelerator (a couple of hundred nucleotides). The work by Jacobi et al. [38] and VanCourt-Herbordt [92] suggest accelerated traceback methods, but with few details. The sequence length accommodated by their accelerators is also limited by the number of PEs on the accelerator like the one described by Hoang. Another limitation of the Hoang and VanCourt methods is that traceback cannot be overlapped with another forward scan since the systolic array is used for both scan and traceback.

The methods presented by Yamaguchi et al. [103] and Moritz et al. [61] allow longer sequences by partitioning the sequences through the pipeline of PEs. Nevertheless, the traceback data must be saved to external memory, since the size of the data exceeds the amount of available internal FPGA memory. Hence, the traceback performance of both methods is limited by the FPGA bandwidth to external memory. The design described by Benkrid et al. [3] also partitions sequences, but the size of FPGA memory ultimately limits the length of sequences that are aligned with hardware acceleration. Operating at 100 MHz, a systolic array with 256 PEs requires at least 6.4 GB/s of memory bandwidth to store 2-bit traceback data from each PE. As PE densities and clock frequencies increase, the external memory bandwidth is easily exceeded. Internal FPGA memory has sufficient bandwidth, but even modest sequence lengths of 16 K require 64 MB of traceback store, which far exceeds current FPGA internal memory capacities.

The global alignment algorithm presented in this paper overcomes the memory size and bandwidth limitations of FPGA accelerators and does not limit the sequence length by the number of PEs. Long sequences of DNA and protein are accommodated by the algorithm through a space-efficient traceback procedure that is accelerated in hardware. Traceback may occur in parallel with the next forward scan since it is implemented in a separate process from the systolic array.

5.3 Algorithm

The general algorithm is described first followed by the FPGA architecture in the next section. The algorithm is based on dynamic programming (DP), but partitions the problem into slices for the FPGA hardware. A description of the general sequence alignment problem is also found in [65, 31].

Given a pair of *sequences* $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ of *length* $|A| = m$ and $|B| = n$ from the finite alphabet Σ, a *sequence alignment* is obtained by inserting *gap characters* "-" into A and B. The aligned sequences A' and B' from the extended alphabet $\Sigma' = \Sigma \cup \{\text{"-"}\}$ are of equal length such that $|A'| = |B'|$. Let the function $s : \Sigma \times \Sigma \to \mathbb{Z}$ determine the similarity of symbol a_i with b_j, and the constant α represent the cost of inserting/deleting a gap. Let H denote the DP matrix and the element $H[i, j]$ the similarity score of sequences $a_1a_2...a_i$ and $b_1b_2...b_j$. An optimal alignment is obtained by maximizing the score in each element of H. The values of H are determined by the following recurrence relations for $1 \leq i \leq m$ and $1 \leq j \leq n$:

$$\begin{aligned}
H[0,0] &= 0, \\
H[i,0] &= H[i-1,0] + \alpha, \\
H[0,j] &= H[0,j-1] + \alpha, \\
H[i,j] &= \max \begin{cases} H[i-1,j-1] + s(a_i,b_j), \\ H[i-1,j] + \alpha, \\ H[i,j-1] + \alpha. \end{cases}
\end{aligned} \tag{5.1}$$

The matrix fill occurs in a scan from upper left to lower right because of dependencies from neighboring elements. During the forward scan, a pointer $p \in$ {DIAG, ABOVE, LEFT} indicates the current selection of the MAX function in Equation 5.1. Given a tie, fixed priority resolves the selection. The value of p is saved to the traceback matrix T, thus $T[i, j] = p$. Following the forward scan, traceback proceeds from $T[m, n]$ to $T[0, 0]$,

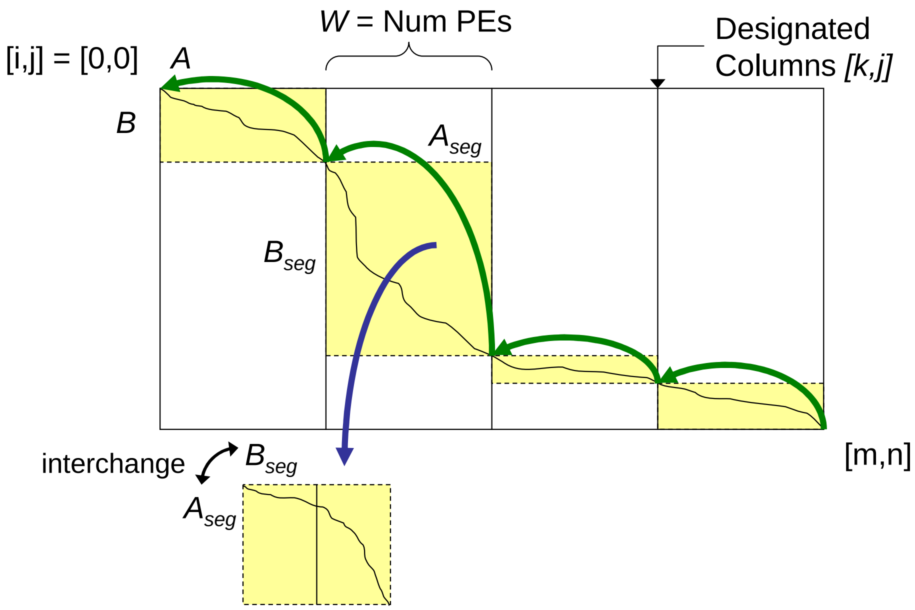


Figure 5.2: Forward scan and traceback

thereby determining the best alignment. The result is a list of edit operations $e \in \{\text{SUBSTITUTE, INSERT, DELETE}\}$.

The scan algorithm presented here builds upon the space-saving concepts described by Edmiston et al. [20], and the divide-and-conquer scheme of Guan and Uberbacher [35]. Since sequence lengths are often longer than the number of PEs available in a systolic array, the problem is often partitioned [48]. The forward scan consists of two fundamental scan procedures ScanPartial and ScanFull. The partial and full descriptors refer to the amount of traceback data saved by the procedures. ScanPartial partitions the DP matrix H into slices of width W. The slices are processed iteratively. The result of processing each slice is a column of traceback pointers $R[k, j]$ that refer to a row in a prior slice (see Figure 5.2). The designated columns k are given by $k \in \{c \,|\, c \bmod W = 0 \vee c = m\}$. The row pointers form a partial traceback path through H that link only the right-most columns of each slice. Given that p indicates the heritage of element $H[i, j]$, the following recurrences for $1 \leq i \leq m$ and $1 \leq j \leq n$ determine R.

if $i \bmod W = 1$ then

$$R[i,j] = \begin{cases} j-1 & \text{if } p = \text{DIAG} \\ j & \text{if } p = \text{LEFT} \\ R[i,j-1] & \text{if } p = \text{ABOVE} \end{cases}$$

else

$$R[i,j] = \begin{cases} R[i-1,j-1] & \text{if } p = \text{DIAG} \\ R[i-1,j] & \text{if } p = \text{LEFT} \\ R[i,j-1] & \text{if } p = \text{ABOVE} \end{cases}$$

Only the designated columns of R are actually stored, which correspond to the right-most columns of a slice. The values for the other columns are retained temporarily with a vector variable that follows the wavefront of the scan. In contrast, the ScanFull procedure does not partition the DP matrix and produces a full matrix T of traceback pointers that refer to adjacent elements of H.

The TracePartial procedure differs from TraceFull in that the partial set of traceback pointers from R are followed instead of the full set from T. The row pointers, from $R[m,n]$ to $R[0,0]$ in designated columns, identify waypoints on the optimal path through the DP matrix. Since the row pointer in $R[k,j]$ refers to a row in a prior slice, a block between the columns is identified, along with corresponding segments of A and B. The segments of A and B are passed to ScanFull and TraceFull to determine the full path from $[k,j]$ back to $[k_{prev},\, R[k,j]]$. The alignment results from each block are concatenated and thereby form a complete path from $[m,n]$ to $[0,0]$.

Since the vertical height of a block (the length of a B segment) is unbounded, the traceback space available to the Full procedures may be exceeded. To avoid this case, a vertical threshold Y is defined such that if exceeded, the Partial procedures are called instead, with the segments of A and B interchanged in the calls. Figure 5.3 shows the algorithm, which is central to bounding the memory required for traceback. TracePartial

```
procedure TracePartial($A, B, m, n, R, E$)
{
    $x_2 \leftarrow m$, $y_2 \leftarrow n$
    while ($x_2 > 1$) do
        $x_1 \leftarrow \lfloor (x_2 - 1)/W \rfloor \cdot W + 1$
        $y_1 \leftarrow (x_1 > 1 \wedge y_2 \geq 1)\,?\,R[x_2, y_2] + 1 \,:\, 1$
        $xlen \leftarrow x_2 - x_1 + 1$, $ylen \leftarrow y_2 - y_1 + 1$
        if ($ylen = 0$) then
            Add $xlen$ DELETE operations to $E'$
        else if ($ylen \leq Y$) then
            ScanFull($A_{x_1}, B_{y_1}, xlen, ylen, T$)
            TraceFull($A_{x_1}, B_{y_1}, xlen, ylen, T, E'$)
        else // interchange $A$ and $B$
            ScanPartial($B_{y_1}, A_{x_1}, ylen, xlen, R'$)
            TracePartial($B_{y_1}, A_{x_1}, ylen, xlen, R', E'$)
            $\forall e \in E'$ : replace DELETE $\Leftrightarrow$ INSERT
        end if
        $E \leftarrow E \cup E'$
        $x_2 \leftarrow x_1 - 1$, $y_2 \leftarrow y_1 - 1$
    end while
}
```

Figure 5.3: Algorithm for TracePartial

is called recursively a maximum of once. Any segments passed to the Full procedures will not exceed W and Y in length because of the partitioning done by ScanPartial. In the worst case, the length of sequence A is bounded by the first call to ScanPartial and the length of B is bounded by the second call.

5.4 Architecture

The global alignment accelerator is implemented using Qnet [53], an open-source packet-switched network architecture similar to DIMEtalk [80]. Qnet components interconnect the host and other FPGA accelerator modules in the system. The architecture facilitates system design with reusable modules that encapsulate sharable devices or resources. Qnet encourages parallelism by offering concurrent, high-performance data paths between modules. Figure 5.4 shows the alignment system constructed with Qnet modules and components. A few

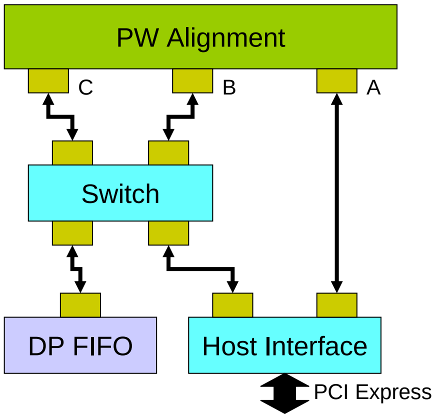


Figure 5.4: System architecture

specifics of Qnet are given before describing the alignment accelerator module and system operation.

5.4.1 Qnet Components

The basic network components consist of a switch, Qports, and Qlinks. As the central figure in the network, the switch provides a path for communicating packets to other modules. Qports are the interface between modules and the network, and are the addressable endpoints of communication. Qports are connected by Qlinks, which consist of paired, unidirectional, point-to-point signaling channels that are each 32-bits wide in this system, but may be implemented with other bit widths. Each Qport has word-based flow control that will apply back-pressure on a link, delaying communication until the port is ready to receive. Hence, packets are not arbitrarily discarded, and the requirement to buffer an entire packet at the input of a module is removed while still maintaining performance. Qnet communication performance has been shown to be very near the theoretical max bandwidth between modules on the FPGA while also maintaining latencies very near theoretical minimums.

Qnet reliably transfers data packets between endpoints through a simple protocol that requires minimal FPGA resources. Packets consist of a small header (see Figure 5.5) and a payload of variable size. The header specifies the source and destination endpoints with

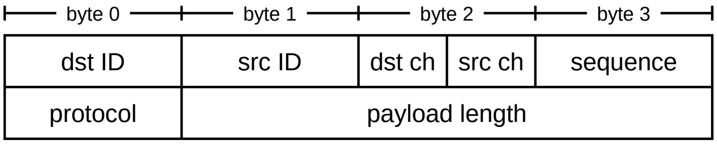


Figure 5.5: Qpacket header

unique port identifiers and also indicates the payload length. When a packet header enters the switch, the output port is determined from the destination endpoint and remains the same for all following words of the packet. With a cut-through packet forwarding method, the full packet is not buffered in the switch. Packets that enter the switch simultaneously with different destinations pass through concurrently. This architecture allows parallel data transfer on all ports of an accelerator module.

5.4.2 System Modules

Host Interface. The host computer communicates with the FPGA accelerator through the PCI Express [4] module, which contains DMA engines and translates PCI packets into Qnet packets. Two ports on this module allow both sequences to be sent in parallel to the accelerator.

DP Matrix FIFO. If the length of sequence A is longer than the number of PEs in the accelerator, the DP matrix H must be processed in slices of width W = (num. PEs) as described in Section 5.3. After processing a slice, the right column of DP matrix values will exit the pipeline of PEs. These H values are sent in a packet to the DP matrix FIFO and retained for processing the next slice through the pipeline. Any packet sent to the DP matrix FIFO will be returned to the originating Qport, as indicated by the packet header, thus cycling the pipeline output to the input. The FIFO may be implemented with any memory technology of sufficient bandwidth and size to handle the stream of data from the PE pipeline. Since only one H value exits the pipeline each clock cycle, the bandwidth requirement is not excessive.

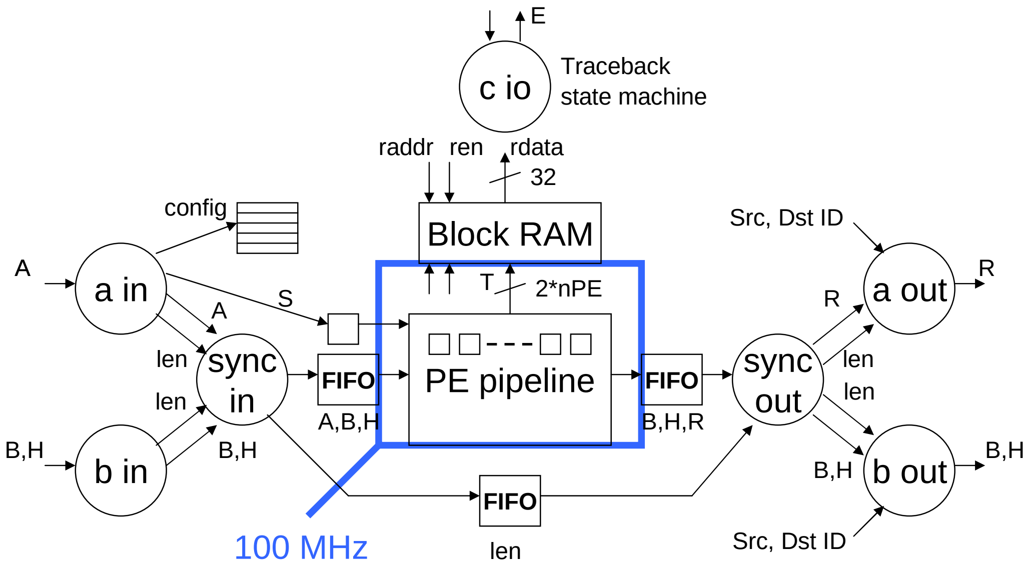


Figure 5.6: Pairwise alignment module

Pairwise Alignment Module. The compute intensive portions of the alignment algorithm are performed by the pairwise alignment module, which contains the pipeline of PEs. This module has three Qports through which the sequences are provided and results are returned (see Figure 5.4). In parallel, Sequence A is input on port A and sequence B is input on port B, while the traceback results are returned on port C.

Figure 5.6 shows the internal architecture of the alignment module. The front-end of the pipeline synchronizes the A and B streams of symbols, and the back-end sends the partial traceback results R out on port A and the H values on port B. The symbols of sequence B that flow through the pipeline are merged with the H values on output, since they will also be needed in processing additional slices. Merged B and H values that exit the pipeline are sent in a packet to the DP matrix FIFO. As sequence A is fed into the pipeline, merged B and H values from the end of the pipeline flow from the alignment module through the DP matrix FIFO and back into the front-end of the pipeline at port B. This cycle occurs for each slice of the scan, except for the last.

Most systems commonly load a segment of A into the pipeline and then shift in B, whereas this system enters A and B in parallel [24]. Sequence B is shifted in as usual, but

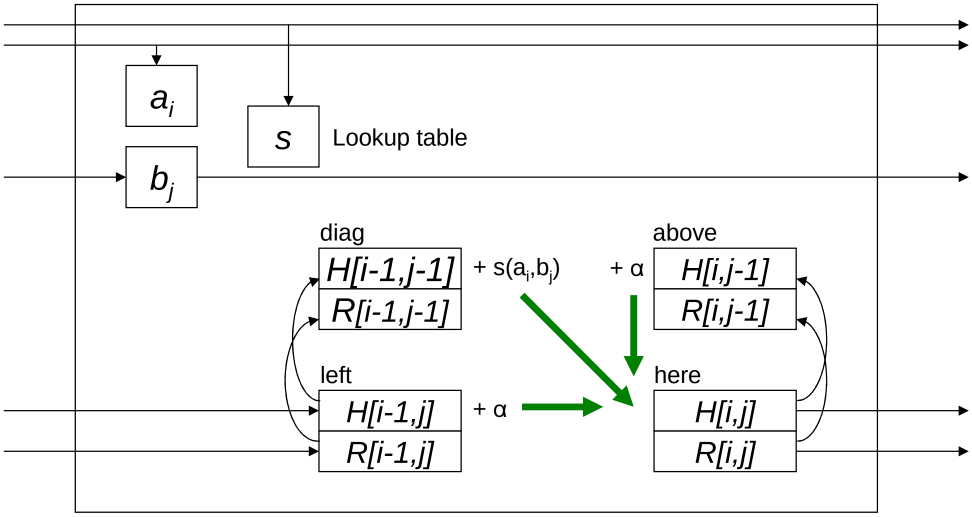


Figure 5.7: Processing element architecture

A is bussed to each PE and latched when the first symbol of B reaches a PE in the pipeline (see Figure 5.7). The recurrence equations described in Section 5.3 are calculated by the PEs each time a pair of symbols enter the pipeline. As a forward scan proceeds from upper left to lower right, the pipeline of PEs operate in parallel along an anti-diagonal wavefront through the DP matrix. Figure 5.8 shows the progression of symbols in the pipeline and shows the mapping of PEs to DP matrix cells over several cycles.

Both of the forward SCAN procedures are implemented by the pipeline of PEs. SCANPARTIAL enables the R (partial row pointer) output, while SCANFULL enables the T (full traceback pointer) output. Configuration bits in the packet header of sequence A determine which pointer type is enabled. For each slice processed by SCANPARTIAL, a column of R is returned to the host in a packet. SCANFULL will only process one slice, while saving the full traceback data in FPGA block RAM, which has the bandwidth to store pointers from every PE in parallel. The vertical threshold Y, as described in Section 5.3, is determined by the depth of FPGA block RAM allocated to full traceback.

A state machine implements the TRACEFULL procedure that follows the pointers saved in block RAM by SCANFULL. To initiate a full traceback, a request packet is sent to Port C of the pairwise alignment module from the host. The results, a list of edit operations

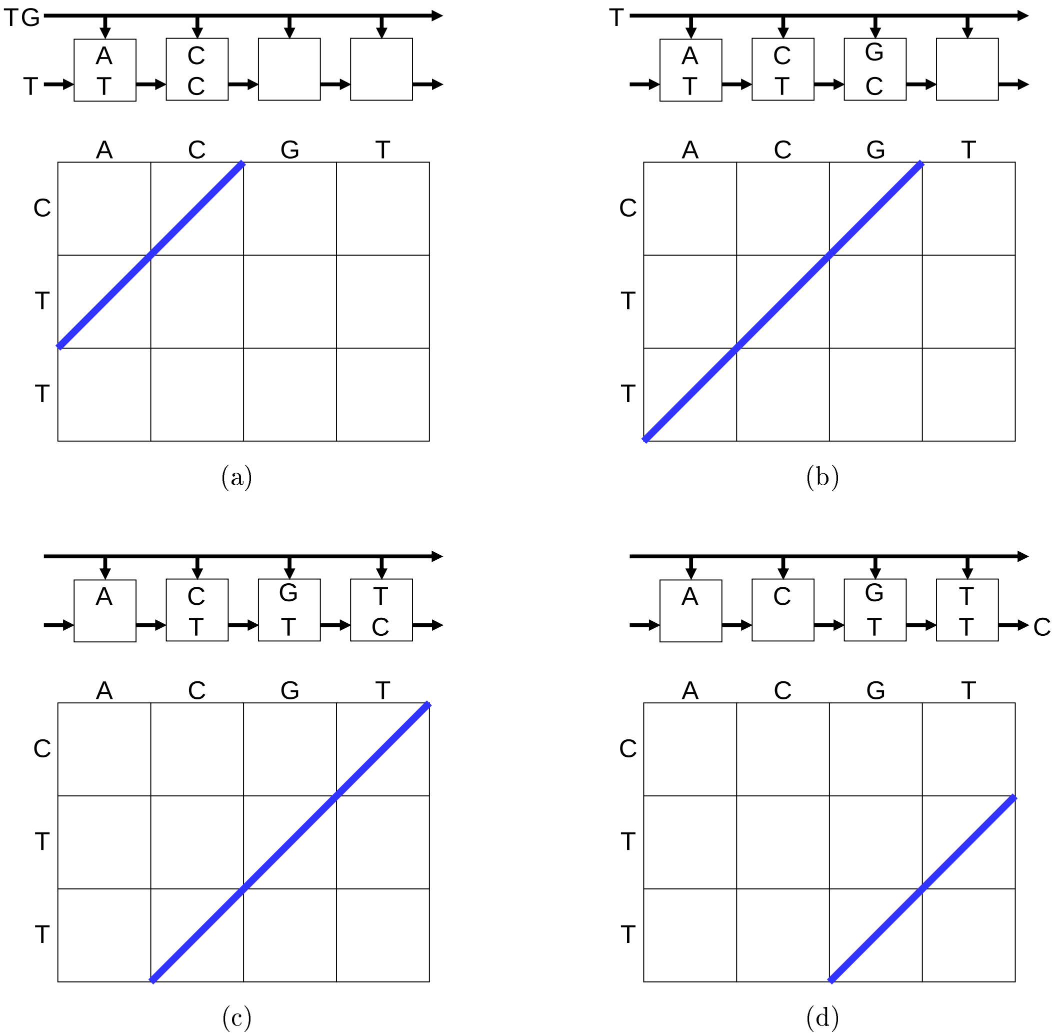


Figure 5.8: Symbol flow and the corresponding DP matrix wavefront for sequential cycles of the PE pipeline.

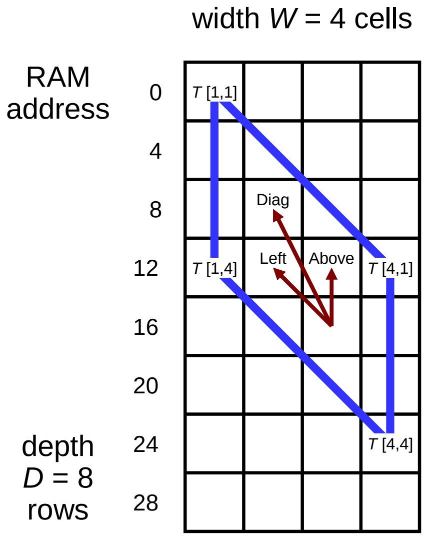


Figure 5.9: The traceback matrix T is skewed in memory. The pointers show how to address neighboring cells during traceback in the skewed matrix.

$e \in E$, are returned to the host from Port C. TracePartial is implemented in software on the host, but calls the Full procedures for most of the work (see Figure 5.3).

Access to traceback pointers $T[i, j]$ in block RAM requires a skewed addressing scheme because of the storage method used in the forward scan. Storing a diagonal wavefront of pointers as a row in block RAM skews the traceback matrix T in memory (see Figure 5.9). A full traceback begins with a request packet that contains the cell address of $T[1, 1]$ and the lengths of sequences A and B. The address of $T[1, 1]$ is saved at the start of a full forward scan and will always be the lowest address in a row (leftmost). From the address of $T[1, 1]$ and the width W of block RAM in cells, the address of $T[m, n]$ is calculated:

$$\begin{aligned} m' &= m - 1, \\ n' &= n - 1, \\ addr_{T[m,n]} &= addr_{T[1,1]} + W(m' + n') + m'. \end{aligned}$$

Traceback proceeds from $T[m, n]$ to $T[0, 0]$ following the pointer in each accessed cell. Given a traceback pointer p from the current cell, the following equation determines the address of the next cell in block RAM.

$$addr = \begin{cases} addr - (2W + 1) & \text{if} \, p = \text{DIAG} \\ addr - (W + 1) & \text{if} \, p = \text{LEFT} \\ addr - W & \text{if} \, p = \text{ABOVE} \end{cases}$$

Since block RAM is dual-ported, traceback reads can occur while the next forward scan concurrently saves pointers in another portion of the traceback memory. Address calculations into block RAM wrap around when the range is exceeded.

5.4.3 System Parameters

Most system parameters are implemented with VHDL generics. For example, symbol width, number of PEs, traceback memory depth, and various register sizes are all specified at a high level in the module hierarchy and passed as generics to lower modules. This allows different configurations of the accelerator with minimal changes to the source. Protein sequences require 5-bit symbols and DNA sequences require at least 2-bit symbols. Mega-length sequences may be handled by the architecture and algorithm by setting system constants and rebuilding a system. The number of PEs is scalable to match the target hardware resources.

Several system parameters affect the maximum sequence length L_{max} that can be processed by the accelerator. As mentioned previously, the DP matrix FIFO must be deep enough to hold the merged B symbols and H values that come from the end of the pipeline. The FIFO length limit is determined by $L_F = N_{FIFO}/N_{BH}$, where N_{BH} denotes the number of bytes for a single B-H pair and N_{FIFO} denotes the DP matrix FIFO size in bytes. Also, the substitution and gap costs combined with the H register size affect the maximum sequence length. Each stage of the pipeline increments an H value by the gap cost α or the result of the similarity function $s(a_i, b_j)$. To avoid H register overflow, the H length limit is

$L_H = (2^{N_H - 1} - 1)/I_{max}$, where N_H denotes the number of bits in H registers, and I_{max} denotes the maximum absolute value of the gap cost α or the similarity function s. In conjunction with the other parameters, the R register size affects the maximum sequence length. A register for R must hold an index into sequence B without overflow. Given N_R, the number of bits in R registers, the R length limit is $L_R = 2^{N_R} - 1$. From the contributing length limits, the maximum sequence length is determined by $L_{max} = \min(L_F, L_H, L_R)$.

5.5 Timing Model

A timing model is presented for the sequence alignment algorithm and architecture described in Sections 5.3 and 5.4. First, constants for the system are defined with the values in parenthesis being specific to the evaluation system:

W = number of PEs (256)

Y = threshold for length of sequence B (768)

C_{pad} = cycles to pad pipeline (8)

t_s = communication startup (1.5 µs)

t_h = host overhead (3 µs)

t_{clk1} = period of clock 1 $({}^{1}\!/_{100\,\mathrm{MHz}})$

t_{clk2} = period of clock 2 $({}^{1}\!/_{150\,\mathrm{MHz}})$

Timing varies as a function of the following variables:

l = $|A'| = |B'|$, aligned length

m = $|A|$, length of sequence A

n = $|B|$, length of sequence B

N_{slice} = $\lceil m/W \rceil$, number of slices

The time for processing a slice is determined by the length of B or the length of the pipeline plus padding, whichever is greater. Flush time depends on how much of sequence B is left in the pipeline after processing a slice and is calculated from the length of B minus padding (zero limited) or the length of the pipeline, whichever is less:

$$\begin{aligned} t_{slice} &= t_{clk1}[\max(n, W + C_{pad}) + 1] \\ t_{flush} &= t_{clk1}\min(W, \max(0, n - C_{pad})) \end{aligned}$$

Based on the previous definitions, execution times for the SCAN and TRACE procedures are:

$$\begin{aligned} t_{scanF} &= t_{slice} + t_{flush} + 4t_s \\ t_{traceF} &= t_{clk2}(2l + 4) + 2t_s \\ t_{scanP} &= N_{slice}(t_{slice} + t_s) + t_{flush} + 4t_s \\ t_{traceP} &= N_{slice}(t_{scanF} + t_{traceF}) \end{aligned}$$

Finally, the time to perform a global sequence alignment is given by:

$$t_{align} = \begin{cases} t_{scanF} + t_{traceF} + t_h & \text{if}\, m \leq W \wedge n \leq Y \\ t_{scanP} + t_{traceP} + t_h & \text{else} \end{cases}$$

This analytical model matches experimental results and predicts the scalability and performance of the architecture under various system configurations.

5.6 Experimental setup

Application. Three global alignment implementations are tested in the evaluation: (1) as a baseline, a software-only version of the algorithm presented in this paper; (2) a version accelerated by the FPGA; and (3) an implementation of the Myers-Miller global alignment algorithm for an additional point of reference. The host computer is used to evaluate the software only versions of the algorithms. Seq-Gen [76] produced varying lengths of test sequences ranging from 128 to 16383 symbols for the evaluation. The applications use a gap cost of −2, a substitution score of 1, and a match score of 2.

Host. The host platform consists of a desktop computer with a 2.4 GHz Intel Core2 Duo processor running Fedora 6 Linux as the operating system. All benchmark applications execute in a single thread and are compiled with gcc using -O3 optimization. For accurate timing, the processor's performance counters are used.

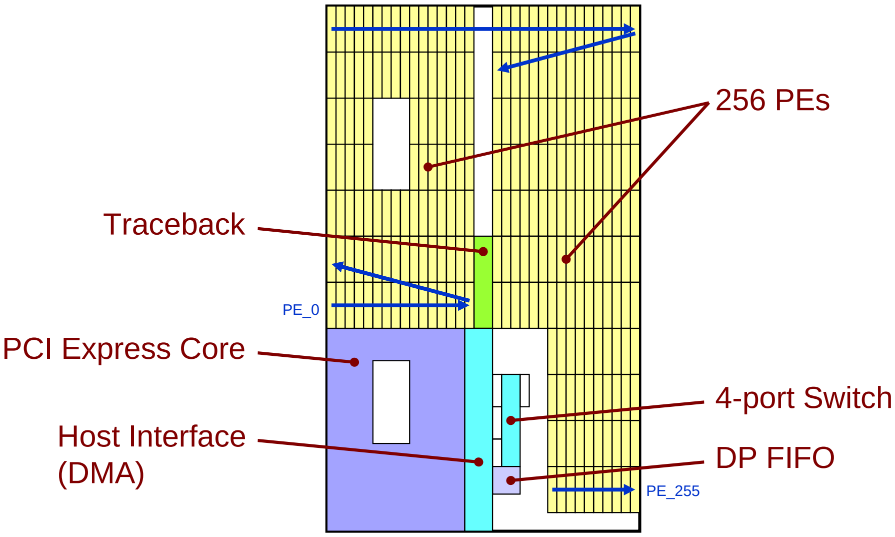


Figure 5.10: FPGA floorplan

Accelerator. An 8-lane PCI Express add-in card with a Xilinx Virtex-4 FX100 FPGA provides the hardware acceleration. To conserve FPGA resources, only 4 of the 8 PCI Express lanes are used in the experimental system. All of the components are implemented in VHDL. As shown in Figure 5.4, a 4-port switch connects the three FPGA modules using 32-bit Qlinks that run at 150 MHz. For simplicity and minimal latency, the switch is implemented with a fixed address table and a fixed port priority resolution scheme. The DP matrix FIFO uses 64 KB of FPGA block RAM, which is enough to hold 16 K entries of B symbols and H values. Driven by a 100 MHz clock, the pipeline consists of 256 PEs placed in a tiled pattern. DNA and protein sequences are accommodated with 5-bit symbol values. An 8-bit look-up table that requires one block RAM per PE implements the similarity function $s(a_i, b_j)$. Each PE outputs a 2-bit traceback pointer p that is stored in traceback memory, which is instantiated in 64 KB of block RAM with a width of 512 bits and a depth of 1024. The traceback memory depth determines the Y threshold. Within the system, DP matrix values H and row pointer values R both require 16-bits.

Through the use of constraints and floor planning, 90% slice utilization is achieved. First, an area shape and size constraint for one PE is determined, in this case, by repeated

Table 5.1: Resource usage

Component	Slices	FPGA Percentage
PCI Express	6175	14.6%
Host Interface	1221	2.9%
4-Port Switch	448	1.1%
Traceback	283	0.7%
DP FIFO	192	0.5%
PE (one)	111	0.3%

Table 5.2: Speedup between implementations

Sequence Length	t_{FPGA} µs	$\frac{t_{Myers}}{t_{FPGA}}$	$\frac{t_{Host}}{t_{FPGA}}$
511	64	131	107
1023	128	171	124
2047	327	264	181
4095	969	357	236
16383	11696	471	304

place and route trials. Then, given this shape and size, a simple (75 line) Perl script tiles the PEs in a programed pattern by generating area constraints for each PE. Keep-out areas are also given to the Perl script. The text output from the Perl script is pasted into the user constraints file for use by the place and route tools along with the other constraints. Only slice resources are constrained for the PEs, since the block RAM needed for each PE may not reside within the area constraint. To meet timing, the first and last PEs of the pipeline are kept closer to the Qport interfaces of the switch and alignment module, which is shown in Figure 5.10 along with the tiling pattern. The traceback block RAMs are constrained to a centrally located area of the FPGA to minimize path lengths from distant PEs. For proximity to the traceback memory, the traceback state machine is also centrally located. Table 5.1 shows the relative resource usage of the various components.

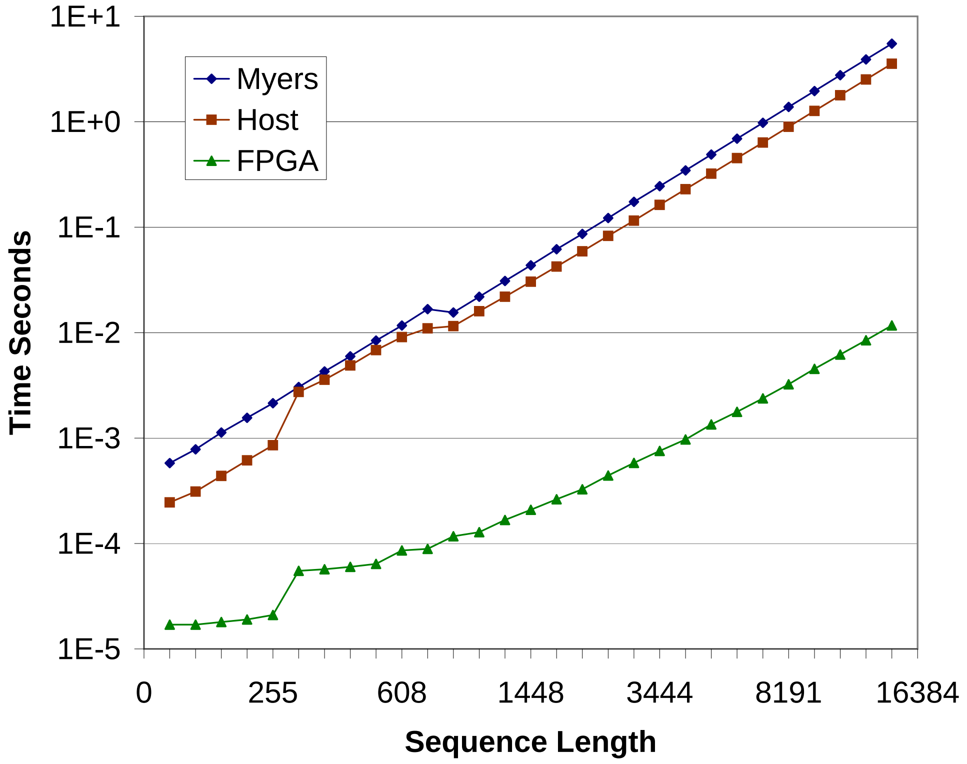
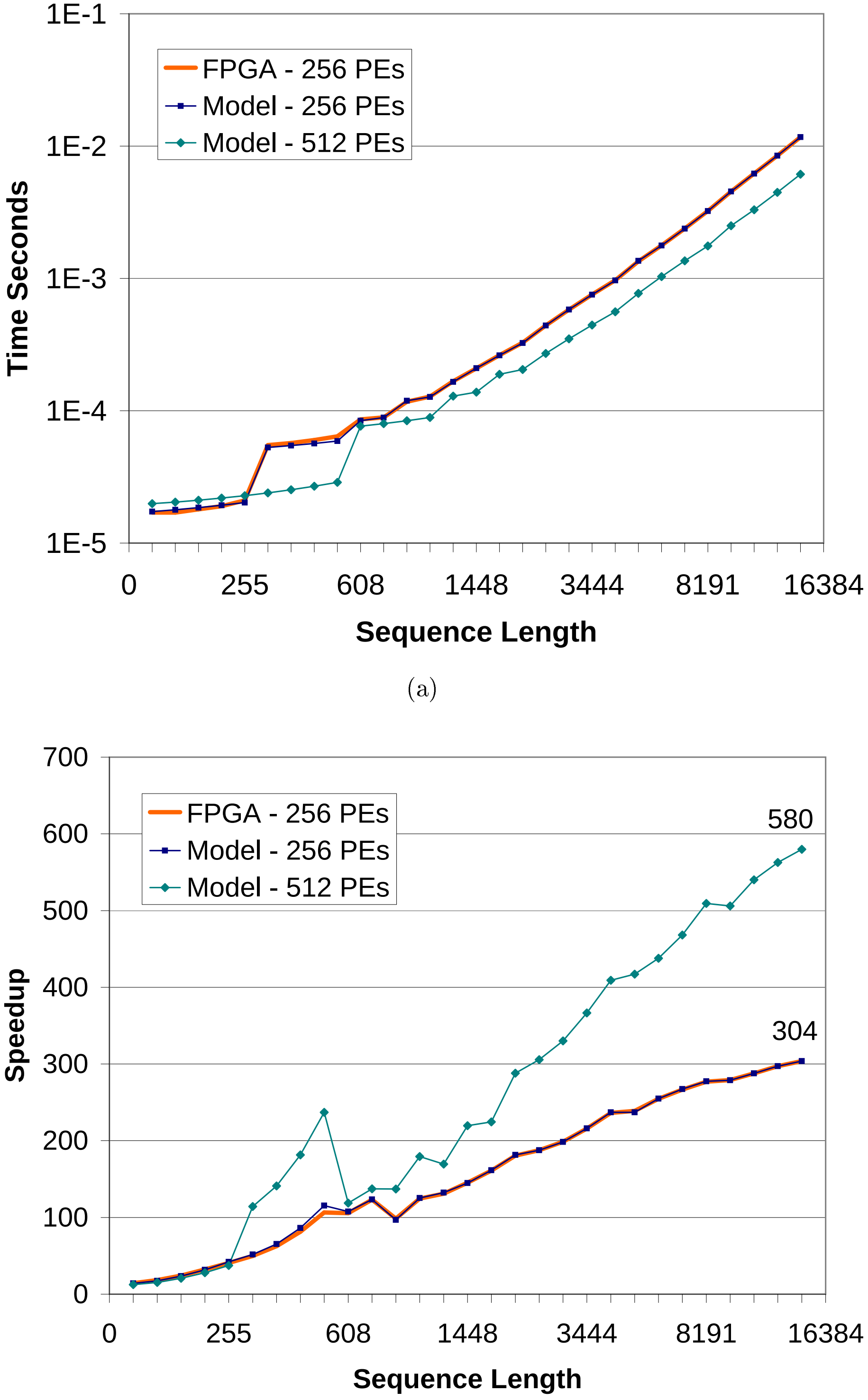


Figure 5.11: Global alignment execution time

5.7 Results

Figure 5.11 shows the performance of the three global sequence alignment implementations with varying lengths of sequences and Table 5.2 compares the speedup between the implementations. The host-only version averages a speedup of 1.6 over the Myers-Miller implementation and the accelerated version achieves a max speedup of 304 over the host version. During the forward scan, the accelerator reaches a peak dynamic programming rate of 25.6×10^9 cell updates/s (CUPS). Traceback occurs at a peak rate of 75×10^6 pointers/s. Figure 5.12 shows the actual performance compared with the timing model from Section 5.5. For longer sequences, the actual performance is near the theoretical peak. The timing model suggests a high degree of scalability for the presented algorithm and architecture. For example, performance predicted by the model gives a speedup of 580 with 512 PEs operating at 100 MHz on a larger FPGA.



(a)

(b)

Figure 5.12: Timing model compared with actual FPGA performance. The model is nearly indistinguishable from the FPGA time. (a) sequence alignment execution time, (b) speedup relative to the host-only version.

Supported by the low communication overhead of Qnet, sequences of length 10 or greater are aligned faster on the accelerator. Sending a single packet from the host to the accelerator takes minimally 1.5 µs. The demonstration system takes a minimum of 14 µs for an alignment with most of the time being attributed to the overhead of several packets, since only 2.65 µs is required for a single pipeline fill and flush once sequences are ready at the front-end of the pipeline.

Sequences shorter than W have a lower bound on alignment time, because unused PEs must be filled with null symbols. Longer sequences realize greater performance on the accelerator because the pipeline does not require a flush between adjacent slices. Adjacent slices need only 1 cycle of spacing in the pipeline. Longer sequences are also more efficient because of proportionately less time spent in the traceback. The average traceback time relative to the forward scan can be visualized in Figure 5.2 as the area of the sub-blocks relative to the area of the whole matrix.

Even though the algorithm presented here requires $O(mn)$ space, the traceback memory is reduced by a significant constant. For example, given sequences with 100 K symbols, saving all the traceback data requires 2.5 GB. By saving the partial traceback pointers in a system with 256 PEs, the traceback data is reduced to 78 MB. Perhaps more importantly, the necessary memory bandwidth to store the partial traceback pointers is reduced to a practical level that is achievable between the host computer and the FPGA accelerator. With the pipeline running at 100 MHz and 16-bit R values, the partial traceback data rate is only 200 MB/s.

Qnet provides communication bandwidth up to 600 MB/s per link in each direction between modules, which exceeds the rate needed by the alignment module to maintain maximum throughput in the pipeline. With excess bandwidth at each end of the pipeline, stalls occur infrequently. Sequences enter the alignment module on ports A and B at a rate of 100 MB/s. Concurrently, partial traceback pointers exit port A at 200 MB/s destined for the host, and merged B-H values exit port B at 400 MB/s destined for the DP FIFO.

Notice that the presented algorithm does not limit the sequence length by the number of PEs or by the amount of full traceback memory. Matching system parameters, such as the number of PEs and the size of traceback memory, to the available FPGA resources maximizes performance. The experimental results and timing model together demonstrate the scalability of the algorithm without memory bandwidth limitations.

5.8 Conclusion

With the presented algorithm and architecture, long sequences are globally aligned with supercomputing performance on reconfigurable hardware. A speedup over 300 is achieved with the example implementation on FPGA technology when compared to a desktop computer. The architecture is scalable to larger capacity FPGAs for a further increase in performance. Beyond sequence comparison, the full alignment of long sequences is accelerated without memory and I/O bottlenecks through a space-efficient algorithm. After executing traceback in hardware, the accelerator returns a list of edit operations to the host, which constitutes an optimal alignment. Other global alignment acceleration methods only address sequence comparison, limit the sequence length, or exhibit memory and I/O bottlenecks.

The key features of the algorithm are the bounded space requirement for full traceback memory and the reduced space for partial traceback memory. These space reductions enable high-performance alignment of long sequences on a reconfigurable accelerator and are a match for FPGA memory capacities and bandwidth. Only 64 KB of FPGA block RAM is used for full traceback in the demonstrated implementation. Partial traceback data sent to the host at a rate of 200 MB/s is supported by commodity FPGA boards.

Future work includes combining coarse-grained parallel methods [75] with the fine-grained parallelism of this method for multiplied performance gain on reconfigurable computing clusters. Also, the advantages of the presented method are applicable to accelerating local alignment. A general-purpose accelerated alignment library that consists of both local and global methods may be applied to multiple sequence alignment codes with minimal effort.

Chapter 6

Accelerated Large-Scale Multiple Sequence Alignment

Submitted to

BMC Bioinformatics

Abstract

Multiple sequence alignment (MSA) is a fundamental analysis method used in bioinformatics and many comparative genomic applications. Prior MSA acceleration attempts with reconfigurable computing have only addressed the first stage of progressive alignment and consequently exhibit performance limitations according to Amdahl's Law. This work is the first known to accelerate the third stage of progressive alignment on reconfigurable hardware. We reduce subgroups of aligned sequences into discrete profiles before they are pairwise aligned on the accelerator. Using an FPGA accelerator, an overall speedup of up to 150 has been demonstrated on a large data set when compared to a 2.4 GHz Core2 processor. Our parallel algorithm and architecture accelerates large-scale MSA with reconfigurable computing and allows researchers to solve the larger problems that confront biologists today.

6.1 Introduction

Biologists and other researchers use multiple sequence alignment (MSA) as a fundamental analysis method to find similarities among nucleotide (DNA/RNA) or amino acid (protein) sequences. The compute time for an optimal MSA grows exponentially with respect to the number of sequences. Consequently, producing timely results on large problems requires more efficient algorithms and the use of parallel computing resources. Reconfigurable computing hardware, such as Field-Programmable Gate Arrays (FPGAs), provides one approach to the acceleration of biological sequence alignment. Other acceleration methods typically encounter scaling problems that arise from the overhead of inter-process communication and from the lack of parallelism. Reconfigurable computing allows a greater scale of parallelism using many fine-grained custom processing elements that have a low-overhead interconnect.

The most common algorithm used to solve the MSA problem is progressive alignment [27, 68, 88]. This algorithm consists of three main stages. The first stage compares all the sequences with each other producing similarity scores only. Since this stage is easily parallelized, it has traditionally been the focus of parallelization efforts; however, speedup is limited without accelerating the following stages. The second stage of MSA groups the most similar sequences together using the similarity scores to form a tree that guides alignment in the next stage. Finally, the third stage successively aligns the most similar sequences and groups of sequences until all the sequences are aligned. Groups of aligned sequences are converted into profiles before alignment with a pairwise dynamic programming algorithm. A profile represents the character frequencies for each column in an alignment. In Stage 3, traceback information from full pairwise alignment is required to align groups of sequences.

In this work, a new method for accelerating the third stage is described that reduces subgroups of aligned sequences into discrete profiles before they are pairwise aligned on the accelerator. Our pairwise alignment algorithm [54] produces the required traceback information and does not limit the sequence length by the number of processing elements (PEs) or by the amount of block RAM on the accelerator. Other hardware acceleration methods are inadequate for use in the third stage because the sequence length is severely limited or only similarity scores are computed.

Alignment quality of the new method is assessed with the BRAliBase benchmark RNA alignment database [99] that consists of 18,990 RNA alignments and with the MDSA data set [9]. Discrete profile alignment is shown to have comparable quality to other popular MSA programs and an accelerated version of the program demonstrates two orders of magnitude speedup.

6.2 Related Work

Most efforts to accelerate bio-sequence applications with hardware have focused solely on database searches and have employed a pairwise local comparison algorithm. Ramdas and Egan [77] discuss several FPGA-based architectures in their survey. Other pairwise comparison accelerators have also been described in [25, 50, 84]. A few methods to accelerate MSA with hardware have been demonstrated, but they fail to use all the available parallel resources in every stage of MSA; consequently, performance is reduced in some stages with idle processors.

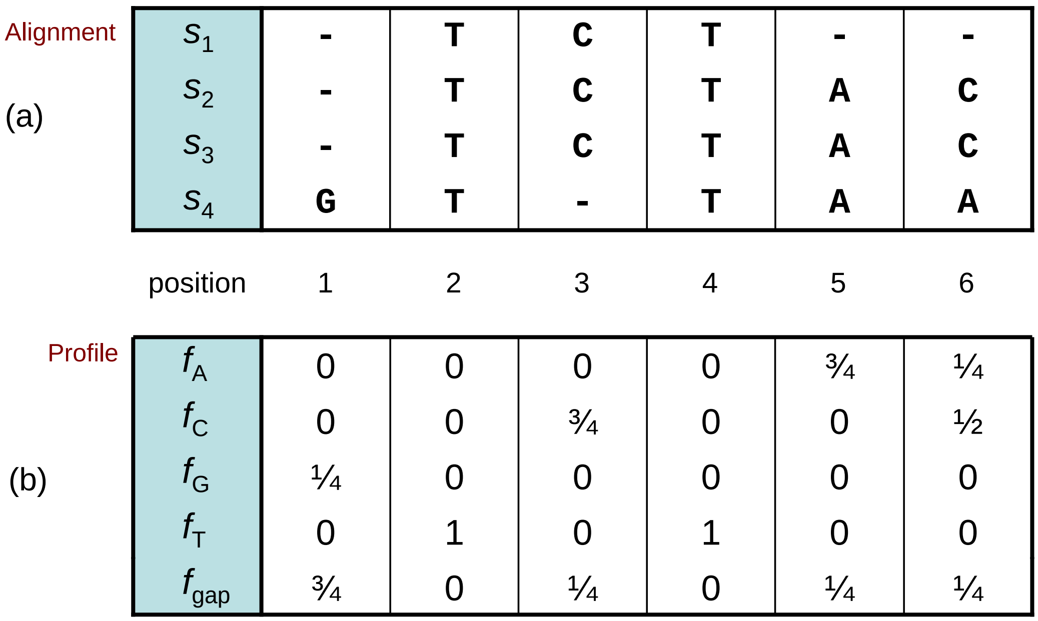
Without accelerating the most time consuming stages of progressive MSA, Amdahl's law [1] limits the overall speedup. For example, if the third stage takes 5% of the computation time, the overall speedup is limited to about 20 even if the other stages are infinitely fast. If the time in Stage 1 is reduced with faster comparison techniques, then the acceleration of Stage 3 becomes more critical. Newer programs like MUSCLE [19] and MAFFT [41] use a faster alignment-free comparison method; therefore, the third stage usually dominates the computation time. Even though these newer methods show greater performance, most of the related work has still focused on accelerating ClustalW where the first stage dominates the run time.

Multiprocessor-Supercomputer. Most attempts to accelerate MSA have been on shared-memory or distributed-memory systems using a coarse-grained parallel approach. Mikhailov et al. [60] shows a 10x speedup with 16 processors by parallelizing all three stages of ClustalW [88] with OpenMP [72] on a shared-memory SGI Origin machine. A notable feature of this effort is the parallelization of the guide tree calculation in the second stage. Deng et al. [15] parallelized several stages of MUSCLE [19] to realize a speedup of 15 on a 16 processor shared-memory machine. Several attempts [12, 17, 45, 46] have been made to parallelize ClustalW on distributed-memory systems using message passing. In these cases, Stages 1 and 3 were parallelized with the best performance reported by Lin et al. [47] having a speedup of 29 on 64 CPUs. Tan et al. [86] achieved a speedup of 35 on a hybrid multiprocessor-cluster system of 40 nodes with 80 CPUs. In the third stage, Tan's method distributes group-to-group alignments to system nodes using a method that is based upon guide-tree and recursive parallelism. The main contribution comes from computing the forward and backward DP scans in parallel on processors within a node. The small speedup achieved in the third stage, which is under 10 in most cases, limits the overall speedup of the progressive algorithms.

Cell BE. Recently, the Cell Broadband Engine has received attention as an accelerator for MSA. Vandierendonck et al. [93] have accelerated ClustalW by a factor of 8 when compared with a 2.13 GHz Intel Core2 Duo processor running a single thread. Stages 1 and 3 were parallelized on two Cell BEs by vectorizing DP matrix calculations and scheduling independent tasks across the 16 available synergistic processing elements. Using a Playstation3, Wirawan et al. [100] achieved a peak speedup of 108 on the first stage when compared with a 3.0 GHz Pentium 4. Overall, a speedup of only 13.7 was observed on 1000 sequences with an average length of 446. However, the announcement from IBM to discontinue Cell production for technical computing [26] may diminish further interest in the Cell.

GPU. Another popular acceleration technology is the general purpose graphics processing unit. Its commodity nature has sparked much interest outside of the graphics community as an acceleration engine. Liu et al. [51] accelerated all three stages of ClustalW on the GPU, with the parallel portions programmed using CUDA [66]. When independent task and guide tree parallelism is low, cells of DP matrix calculations are computed in parallel. An overall peak speedup of 41.53 was demonstrated on 1000 sequences of average length 858 with 1 GPU card (GeForce GTX 280) when compared with a 3.0 GHz Pentium 4. The best speedup obtained in each of the three stages is 47.13, 11.08, and 5.9 respectively. Again, the small gain in the third stage limits the overall speedup.

FPGA. Reconfigurable computing approaches accelerate the first stage of MSA by computing pairwise alignments with a pipeline of processing elements (PEs). This linear systolic array operates with fine-grained parallelism along a wavefront of cells in the DP matrix. The ClustalW algorithm does not use the score obtained from a pairwise alignment directly. Instead, the number of identical characters in an alignment are used to compute the fractional identity. Oliver et al. [70] accelerates the first stage of ClustalW, but leaves the second and third stages for execution on the host processor. Instead of actually aligning the sequences, a custom algorithm on the accelerator counts the number of identical characters during the forward scan without performing traceback. The best overall speedup was 13.3 compared to ClustalW running on a 3.0 GHz Pentium 4. For Stage 1, a PCI-based accelerator board reaches a peak speedup of 50.9 with 92 PEs in a Xilinx XC2V6000. In another approach, Lin et al. [47] demonstrated an overall speedup of 34.6 using 10 Altera Stratix PEIS30 with a total of 3072 PEs. For the first stage, a speedup of 1697.5 was achieved when compared with a 2.8 GHz Xeon. The number of identical characters is deduced from the comparison score returned from the accelerator and the sequence lengths. Even with the impressive speedup in the first stage, the overall speedup is still limited by the third stage. Greater performance may be achieved, however, by accelerating the third stage of progressive alignment.



Alignment

(a)

s_1	-	T	C	T	-	-
s_2	-	T	C	T	A	C
s_3	-	T	C	T	A	C
s_4	G	T	-	T	A	A
position	1	2	3	4	5	6

Profile

(b)

f_A	0	0	0	0	¾	¼
f_C	0	0	¾	0	0	½
f_G	¼	0	0	0	0	0
f_T	0	1	0	1	0	0
f_{gap}	¾	0	¼	0	¼	¼

Figure 6.1: Each position in a profile consists of a vector with character frequencies f_N for the corresponding column in a group of aligned sequences. (a) Multiple alignment of sequences s_i. (b) Profile derived from the alignment.

6.3 Discrete Profile Alignment

The third stage of MSA pairwise aligns profiles in a similar way to sequences, but it must also work with the extra information in profiles. Each position of a profile designates a point in continuous profile space with a vector of character frequencies (see Figure 6.1 and Figure 6.2). Profile-based MSA applications typically use floating-point numbers or scaled integers to represent these character frequencies. The extra size and dimension of profiles, in relation to sequences, adds to the complexity of alignment. Hence, a reduced representation of profiles that retains as much information as possible simplifies alignment. By reducing profiles to discrete profiles—essentially sequences with an extended alphabet—they may be aligned with a simpler pairwise sequence alignment algorithm.

The concept of discrete profile space was introduced by Eskin [23] with application to DNA motif search, which finds relatively short patterns in a subject sequence. For instance, when searching for promoter sequences, query profiles have a length of about 8–12 positions.

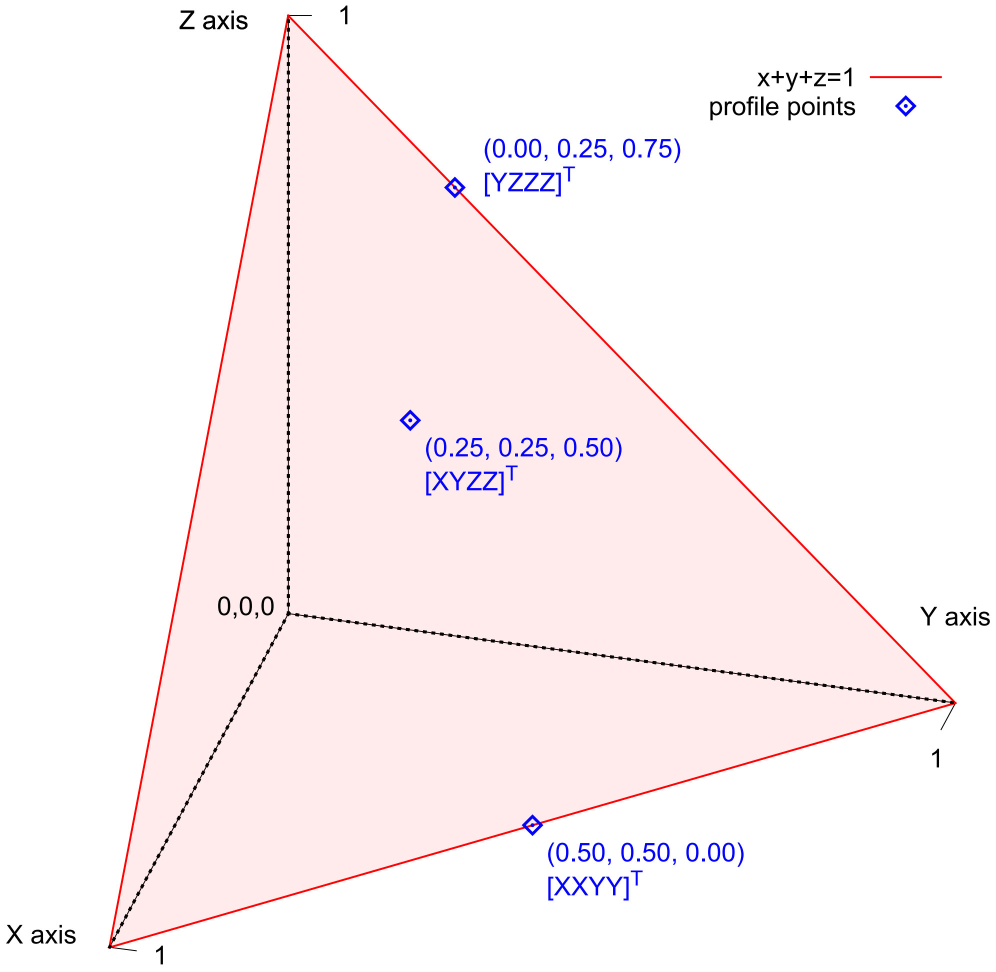


Figure 6.2: In three dimensions, profile space is a triangle on the plane $x+y+z=1$; however, five dimensions are required to represent DNA alignments. Points in profile space are shown with coordinates and an aligned column example (transposed). The corners of profile space represent columns of an alignment that contain all the same character.

In Eskin's method, a motif is represented as a small, discrete profile that contains the probabilities of finding each nucleotide at the respective positions. A similar work by Wang and Stormo [97] partitions a four-dimensional continuous profile space into 15 subspaces based upon a supervised learning algorithm. Each dimension corresponds to a nucleotide frequency f_N with the constraint $\sum f_N = 1$. Any point falling within a partition is then represented by a discrete profile symbol.

For the application of discrete profile space to MSA, a few issues and extensions must be addressed. For example, an additional dimension must be added to profile space to accommodate gaps. Also, sample points from profile space must be selected for representation with discrete symbols, and substitution costs need to be calculated between these sample points. Furthermore, a reduction method from continuous space to discrete symbols must be devised that can operate efficiently on genomic-sized profiles.

6.3.1 Sample Points

Five dimensions in profile space are required to represent profiles that contain nucleotide and gap character frequencies. Each position of a profile can be mapped to a point that falls on the bounded hyperplane $f_A + f_C + f_G + f_T + f_{gap} = 1$ in Euclidean space where $0 \leq f_N \leq 1$. To reduce the number of possible points, a discrete number of sample points are selected from continuous profile space. These sample points and a corresponding discrete symbol represent nearby points in profile space.

A selection algorithm determines sample points by projecting lattice points $\boldsymbol{p}$ in D-space onto the profile hyperplane according to the parametric equation $\boldsymbol{p}' = t\boldsymbol{p}$, where $t = (1 - \sum p_i)/D$. Lattice points (see Figure 6.3) are evenly spaced by a distance of $1/L$ in each dimension; however, only points that lie in a band near the hyperplane are considered. Given the sum of lattice point coordinates $S = \sum p_i$, the considered points fall between $(1 - \frac{D-1}{L}) \leq S \leq (1 - \frac{1}{L})$. Intuitively, these lattice points reside on parallel hyperplanes that are a distance of $\varepsilon = \sqrt{D}/DL$ from each other (see Figure 6.4). Corners of profile space that

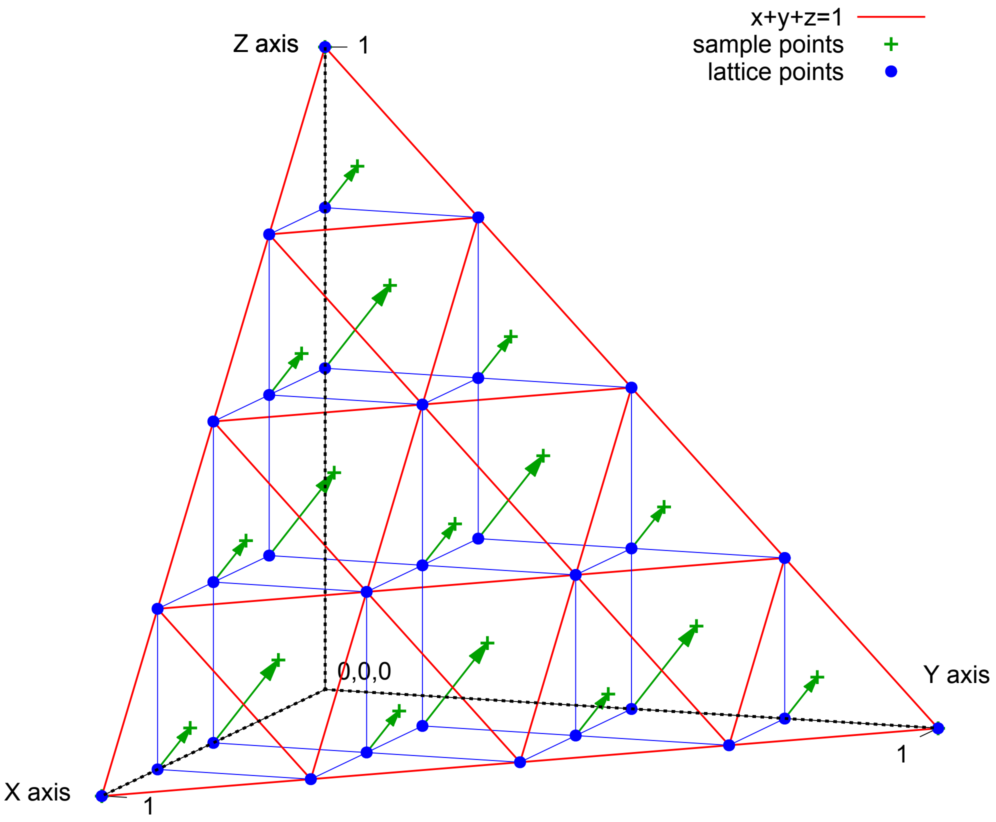


Figure 6.3: Sample points are determined by projecting lattice points onto the profile plane.

consist of all one nucleotide are also included as sample points, but the point indicating a profile of all gaps is excluded.

The number of sample points is reduced further by filtering points that represent less probable nucleotide frequencies. Nucleotides from the same group, either purine or pyrimidine, have a higher probability of being aligned, while those from different groups have a lower probability. Substitution tables reflect this probability in their cost values and influence alignment algorithms accordingly. Therefore, sample points with a high frequency of both purines and pyrimidines are eliminated if they meet the condition

$$(f_i + f_j > 0.75) \wedge (|f_i - f_j| < 0.30)$$

where $i \in \{\text{A,G}\}$ and $j \in \{\text{C,T}\}$.

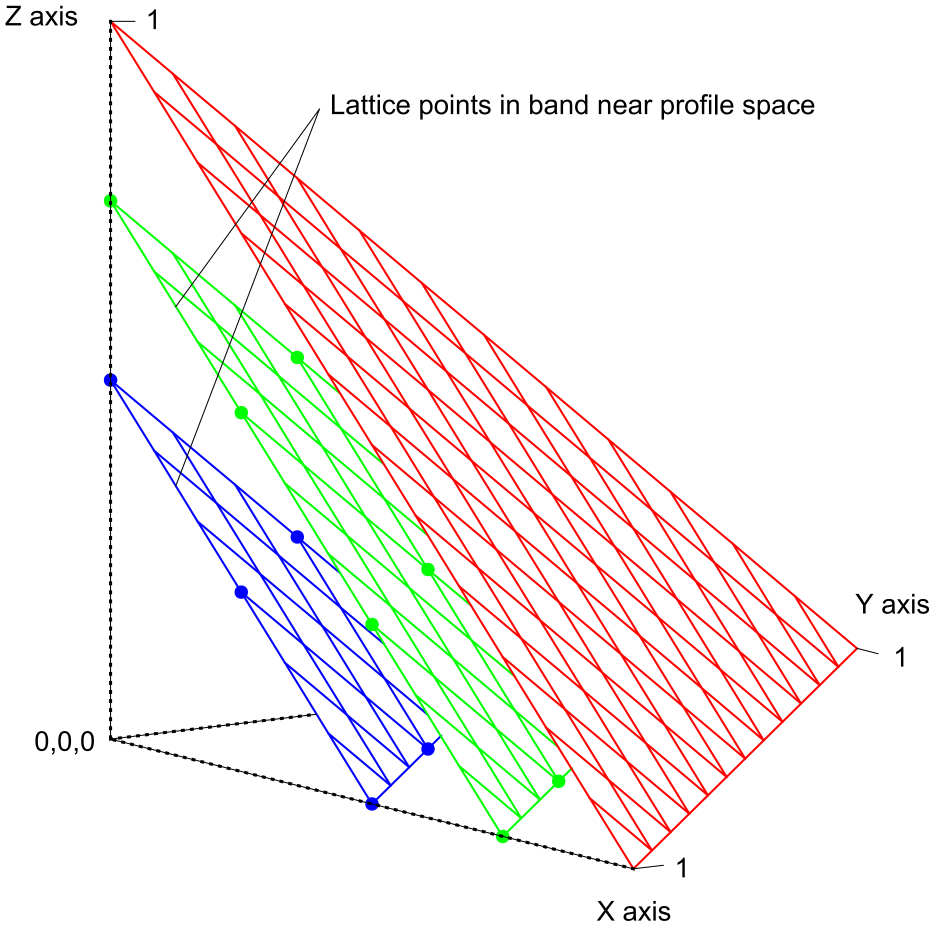


Figure 6.4: Planes parallel to profile space are separated by a distance of $\varepsilon = \sqrt{D}/DL$. For this example, $D = 3$ and $L = 4$.

6.3.2 Substitution Table

After sample points in profile space are selected, the substitution cost between these representative points can be determined. Instead of calculating the cost every time sample points are compared during alignment, the cost can be computed once and stored in a new sample substitution table. The discrete symbols associated with each sample point become the indices into the table and the codebook for a quantization algorithm.

Substitution costs between sample points are computed from the individual nucleotide frequencies and substitution costs. Since a hardware constrained implementation of the sample substitution table may only have 4 or 8-bit entries, a scaling factor adapts the range of computed values to fit within entry size limits. Given the nucleotide substitution table s of size $N \times N$, an array of sample points c, and a scaling factor β, the substitution cost s' between each sample point is determined by

$$s'_{i,j} = \left[\sum_{m=1}^{N} \sum_{n=1}^{N} c_{j,n}\, c_{i,m}\, s_{n,m} \right] \beta.$$

The substitution cost of a gap and a nucleotide is the gap extension cost plus one. This prevents a gap in one sequence from being followed by a gap in the other sequence during pairwise alignment of discrete profiles.

6.3.3 Reduction

For the accelerator to sustain maximum performance, the host system must supply reduced profiles at the accelerator's input data rate (see Figure 6.5). Profiles are reduced to discrete profiles to support a simpler, higher-performing pairwise alignment algorithm on an accelerator that only aligns sequences of symbols. A new quantization technique is used for this reduction on the host to reach the needed performance. For each continuous profile position, the reduction algorithm searches for a nearby sample point and then returns the corresponding discrete symbol. Finding a nearby point in less time is preferred to a

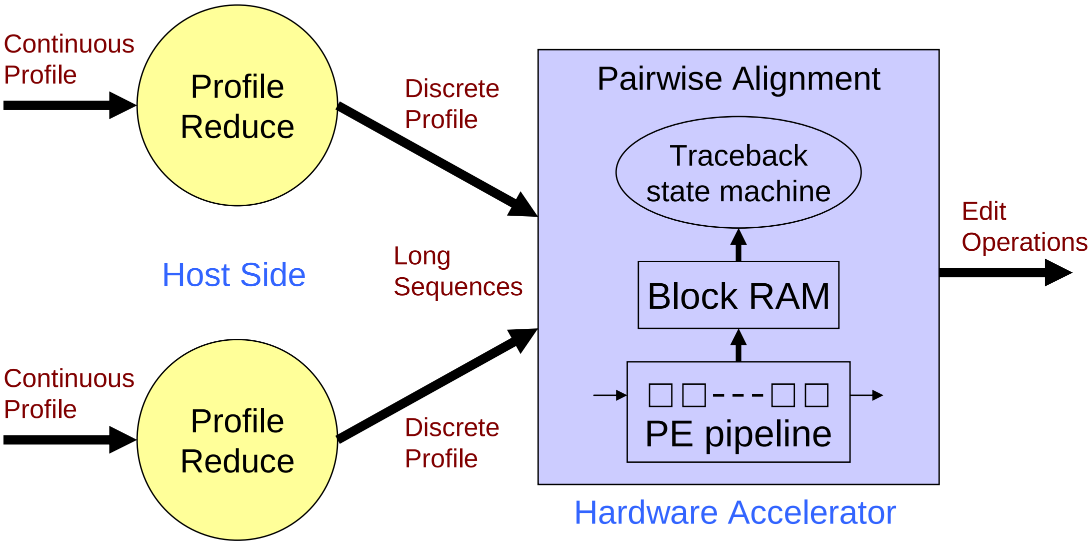


Figure 6.5: Profile reduction before alignment

nearest neighbor search with greater overhead. Also, constraining the search to the profile hyperplane $\sum f_N = 1$ allows for some optimization.

A near neighbor search finds a sample point that is close to the given continuous point, but not necessarily the closest point. This relaxation of proximity allows the search to proceed in deterministic time, and thereby keep up with the accelerated pairwise alignment. Search begins by scaling and truncating each nucleotide frequency to form a partially quantized point. Then these integral coordinates are used as indices into a lookup table R that contains references to nearby sample points. The scale factor determines the number of quantization levels for each coordinate and also the size of the lookup table. As a result of the search, points in continuous profile space are mapped to a small set of symbols that represent sample points. Not every element of the D-dimensional lookup table requires storage since the partially quantized points lie within a scaled distance of $(D-1)\varepsilon$ from the profile hyperplane. A ragged array with only the needed locations is used to implement the lookup table R.

6.3.4 Example

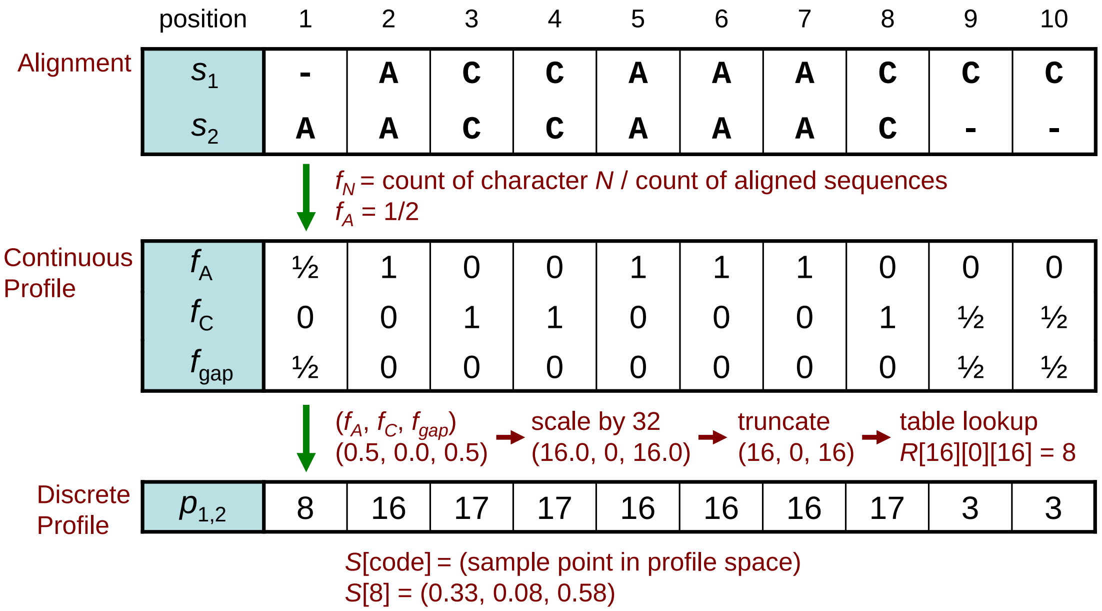
An example of discrete profile alignment is presented starting with two groups of aligned sequences. Profile formation, reduction, and pairwise alignment of the profiles are included in this example. A simplified alphabet $\Sigma' = \{\mathrm{A}, \mathrm{C}, \text{“-”}\}$ is used so that the character frequencies correspond with the X, Y and Z axes of a depictable three-dimensional profile space.

Figures 6.6 and 6.7 show instances of profile calculation and reduction. Each profile position is calculated independently and corresponds with a column of aligned sequences. Given two groups of sequences $\{s_1, s_2\}$ and $\{s_3, s_4\}$, continuous profiles are calculated by counting the occurrence of characters in each column and dividing by the number of sequences to produce a vector of frequencies (f_A, f_C, f_{gap}). Profile reduction proceeds by scaling each vector by 32 and truncating the values to form indices into the three-dimensional reduction table $R_{A,C,gap}$. These table lookup values, which are references to nearby sample points, are used for each position of the discrete profiles $p_{1,2}$ and $p_{3,4}$. Figure 6.9 depicts two points in profile space and the nearby sample points found by lookup in the reduction table R. Figure 6.8 shows the discrete profile alignment process and the final alignment of the original groups. The discrete profiles $p_{1,2}$ and $p_{3,4}$ are aligned with a pairwise algorithm that returns the edit string $E_{1,2,3,4}$ composed of the operations $e_i \in \{\mathrm{(Mis)Match}, \mathrm{Insert}, \mathrm{Delete}\}$. The edit operations also apply to the groups of sequences $\{s_1, s_2\}$ and $\{s_3, s_4\}$ because of the position correspondence between alignments and derived profiles.

6.4 Methods

The following components were incorporated into MUSCLE [19], an open-source MSA program, to demonstrate accelerated large-scale MSA.

- SSE accelerated sequence similarity algorithms for the first stage of MSA
- A discrete profile alignment algorithm for the third stage of MSA
- An FPGA accelerated pairwise alignment algorithm [54]



Alignment

position	1	2	3	4	5	6	7	8	9	10
s_1	-	A	C	C	A	A	A	C	C	C
s_2	A	A	C	C	A	A	A	C	-	-

f_N = count of character N / count of aligned sequences
$f_A = 1/2$

Continuous Profile

f_A	½	1	0	0	1	1	1	0	0	0
f_C	0	0	1	1	0	0	0	1	½	½
f_{gap}	½	0	0	0	0	0	0	0	½	½

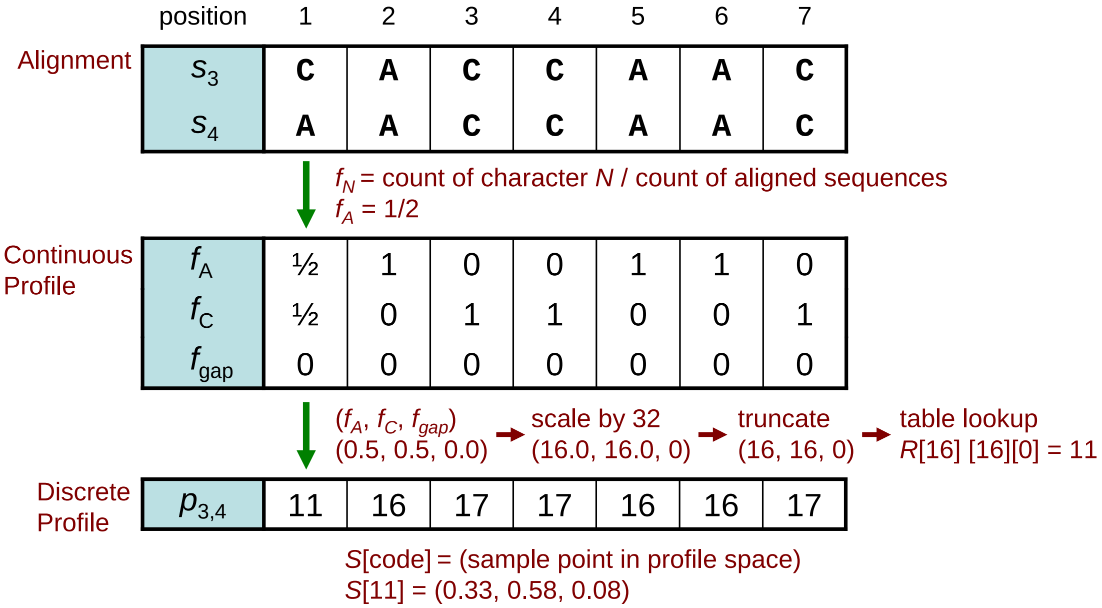
(f_A, f_C, f_{gap}) (0.5, 0.0, 0.5) → scale by 32 (16.0, 0, 16.0) → truncate (16, 0, 16) → table lookup R[16][0][16] = 8

Discrete Profile

$p_{1,2}$	8	16	17	17	16	16	16	17	3	3

S[code] = (sample point in profile space)
S[8] = (0.33, 0.08, 0.58)

Figure 6.6: From the alignment $\{s_1, s_2\}$, a continuous profile is derived and then reduced to form the corresponding discrete profile $p_{1,2}$. S is a table of sample points.



Alignment

position	1	2	3	4	5	6	7
s_3	C	A	C	C	A	A	C
s_4	A	A	C	C	A	A	C

f_N = count of character N / count of aligned sequences
$f_A = 1/2$

Continuous Profile

f_A	½	1	0	0	1	1	0
f_C	½	0	1	1	0	0	1
f_{gap}	0	0	0	0	0	0	0

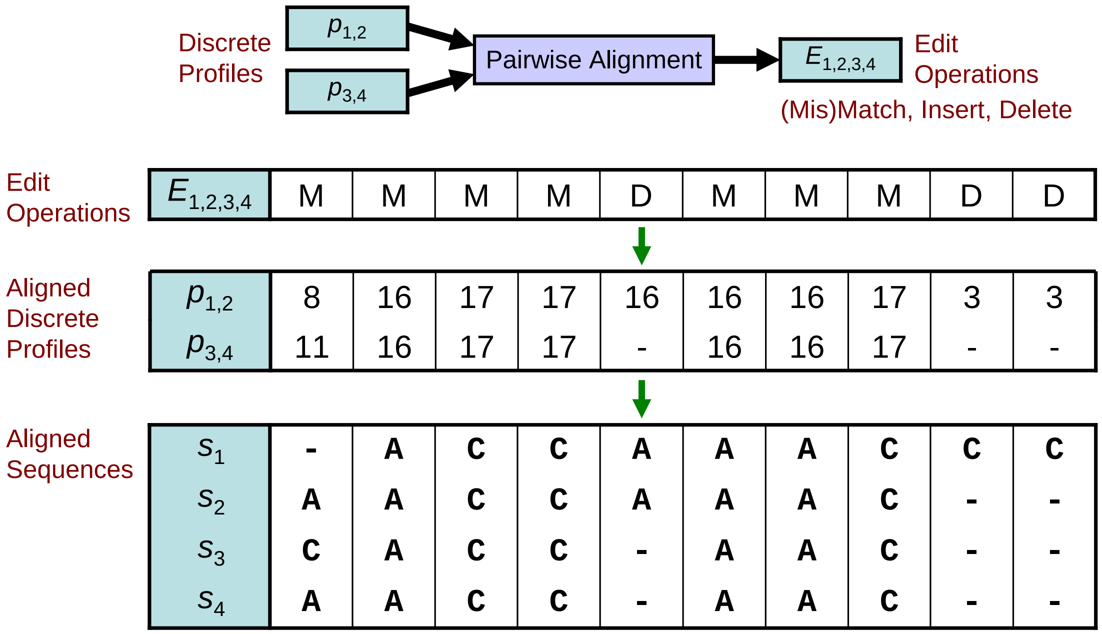
(f_A, f_C, f_{gap}) (0.5, 0.5, 0.0) → scale by 32 (16.0, 16.0, 0) → truncate (16, 16, 0) → table lookup R[16] [16][0] = 11

Discrete Profile

$p_{3,4}$	11	16	17	17	16	16	17

S[code] = (sample point in profile space)
S[11] = (0.33, 0.58, 0.08)

Figure 6.7: From the alignment $\{s_3, s_4\}$, a continuous profile is derived and then reduced to form the corresponding discrete profile $p_{3,4}$. S is a table of sample points.



$E_{1,2,3,4}$	M	M	M	M	D	M	M	M	D	D

$p_{1,2}$	8	16	17	17	16	16	16	17	3	3
$p_{3,4}$	11	16	17	17	-	16	16	17	-	-

s_1	-	A	C	C	A	A	A	C	C	C
s_2	A	A	C	C	A	A	A	C	-	-
s_3	C	A	C	C	-	A	A	C	-	-
s_4	A	A	C	C	-	A	A	C	-	-

Figure 6.8: A pairwise alignment algorithm treats discrete profiles as sequences. The resulting edit operations $E_{1,2,3,4}$ indicate the computed alignment between the discrete profiles $p_{1,2}$ and $p_{3,4}$, and the corresponding groups of sequences $\{s_1, s_2\}$ and $\{s_3, s_4\}$.

Corresponding code in MUSCLE was replaced with our highly-parallel code that uses SSE instructions and the FPGA accelerator. Discrete profile alignment replaced the float-based alignment used in each step of progressive alignment. Also, sequence similarity calculations were optimized with SSE instructions. The vectorized code includes the comparison of k-mer counts and the counting of identical symbols.

Two versions of the modified MUSCLE are used for analysis. One version (MUDISC) implements our pairwise alignment in software on the host, while the other (MUFPGA) accelerates pairwise alignment on the FPGA. MUDISC is compared with other popular MSA programs such as ClustalW [88], Kalign [43], MAFFT [41], MUSCLE [19], and POA [34]. For those programs that support iterations, a maximum of two are used. The non-accelerated MSA programs and MUDISC execute only on the conventional processor and MUFPGA additionally uses the FPGA accelerator. Both alignment quality and program performance are measured.

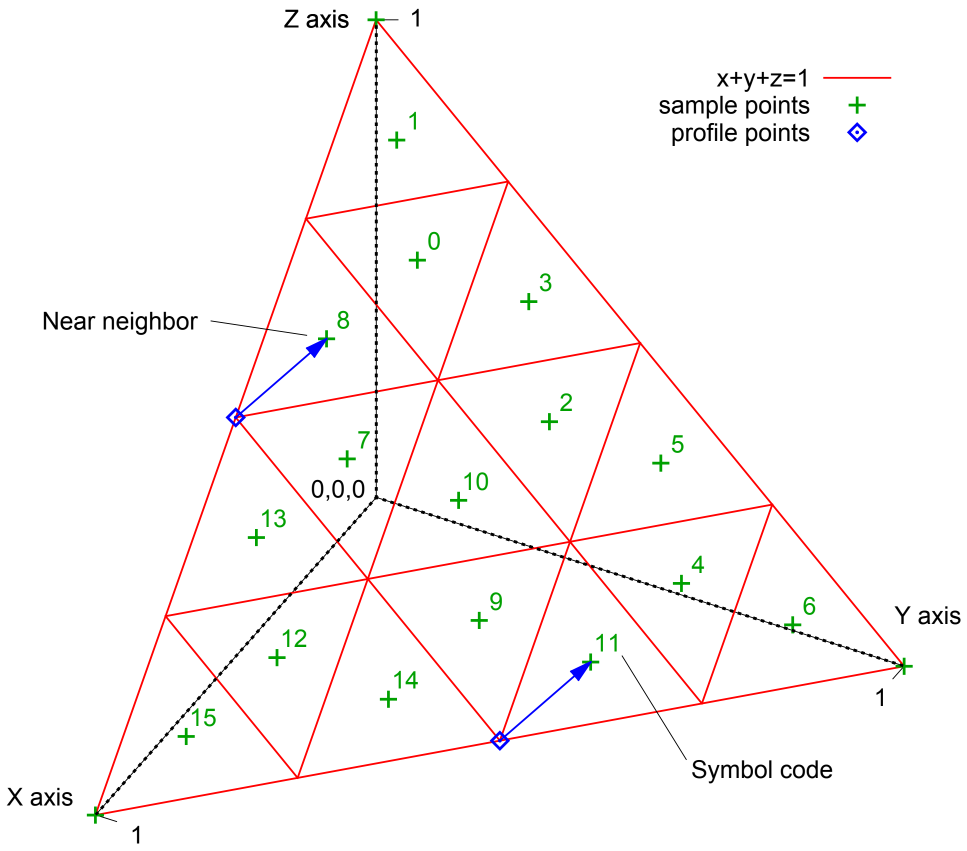


Figure 6.9: Given two profile points, nearby sample points and associated symbol codes are shown.

Nucleotide adaptations [9] of the BAliBASE [89] and SMART [44] reference alignments are used to compare the quality of the MSA programs. BAliBASE alignments have been determined to be correct based upon known three-dimensional structure. Another assessment of alignment quality is obtained with the BRAliBase benchmark RNA alignment database [99] that consists of 18,990 RNA alignments. Unaligned versions of the reference alignments are realigned to produce test alignments. Reference and test alignments are then compared with scoring programs to produce an accuracy metric between 0 and 1.

A performance analysis uses a few large-scale, viral data sets as shown in Table 6.2. Overall program performance for MUSCLE v3.8.31, MUDISC, and MUFPGA is measured by the wall-clock time needed to align a data set and includes all three stages of progressive alignment. For accurate timing, the host processor's performance counters are used.

The host platform consists of a desktop computer with a 2.4 GHz Intel Core2 Duo processor running 64-bit Fedora 13 Linux as the operating system. All benchmark applications execute in a single thread and are compiled with gcc using -O3 optimization. An 8-lane PCI Express [4] add-in card with a Xilinx Virtex-4 FX100 FPGA provides the hardware acceleration for pairwise alignment. Acceleration occurs on a pipeline of 256 PEs driven by a 100 MHz clock. Each PE requires one block RAM to implement the substitution cost $s'_{i,j}$ as a lookup table. The accelerator supports linear gap costs and up to 64 points in profile space with 6-bit symbol values.

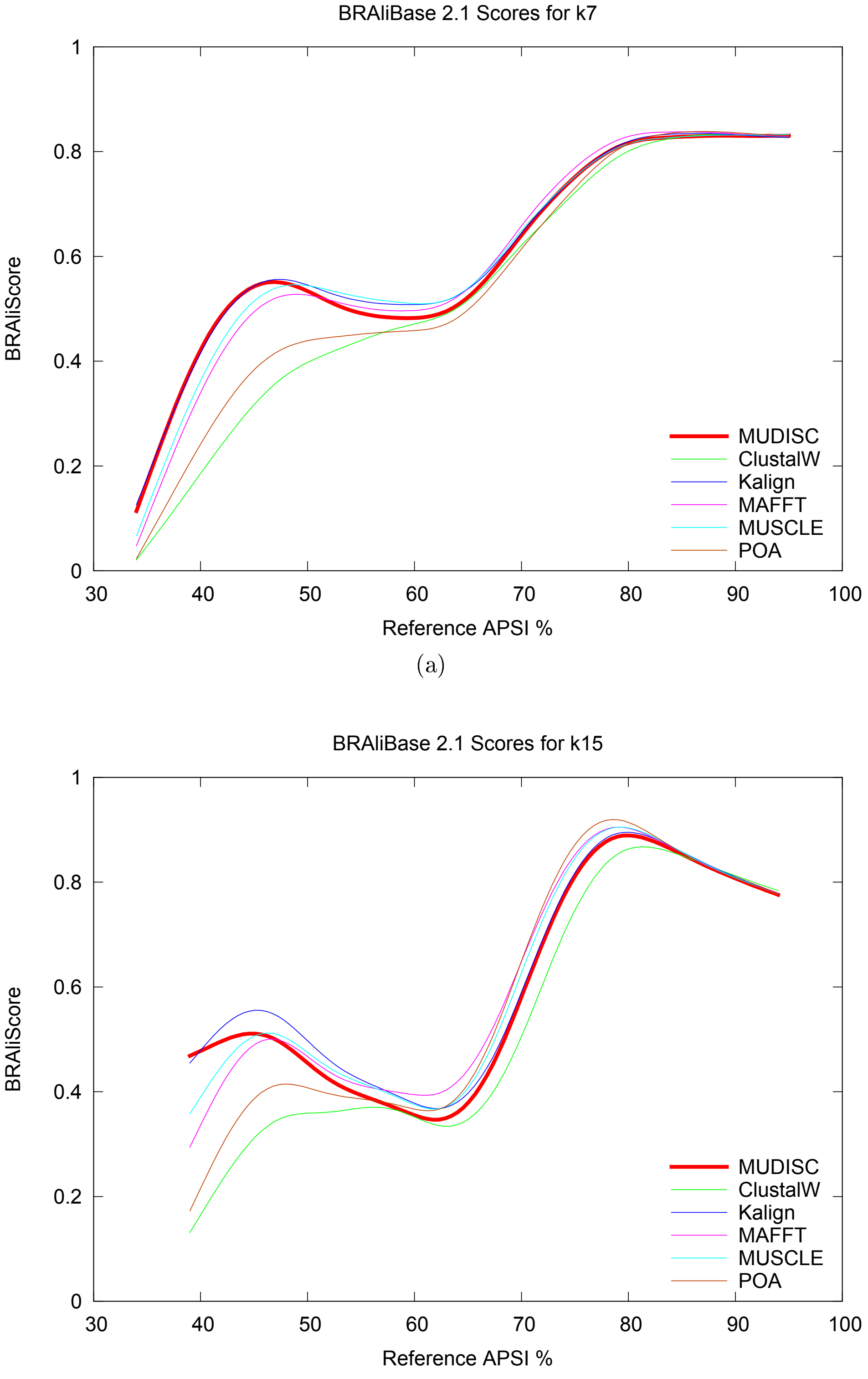
6.5 Results

Alignment quality with BRAliBase 2.1 is depicted in Figure 6.10 for MUDISC and several other MSA programs. The BRAliScore, which reflects the alignment accuracy, is plotted in relation to the average pairwise sequence identity (APSI) of the reference alignment. Identical sequences have an APSI of 100%. BRAliScore is composed of two independent scores and is calculated by multiplying the fractional identity (FI) [18] and the structure conservation index (SCI) [98]. The FI score is based on the fraction of matching characters between the test and reference alignment, whereas the SCI is not based on the reference alignment, but indicates the amount of secondary structure conserved in the multiple alignment. A local smoothing of score values is applied with the acsplines option in gnuplot and a weighting factor of 5e-3. Above 60% APSI, there is little difference in the alignment quality between the programs; however, MUDISC is one of the top performers on data sets below 60% APSI.

A comparison of alignment quality with the MDSA reference sets is reported in Figure 6.11. The Q score, which is equivalent to the sum-of-pairs score (SPS) score [90], is shown in relation to the APSI. Unlike the FI, the Q score only considers residue pairs correctly aligned in the test alignment compared with the reference and does not count residue-gap pairs. The acsplines smoothing option is again used, but with a weighting factor of 1e-2. MUDISC is on par with other MSA programs down to about 40% APSI and is still comparable in accuracy below 40% APSI.

The average alignment quality of MUSCLE and MUDISC is shown in Table 6.1. A variant of MUDISC that uses the nearest neighbor search method is also shown. According to the Friedman rank test [28] with an adjustment for ties, the difference in quality between the near and nearest neighbor search method is not significant. Even though the average scores are very similar, the difference between MUSCLE and MUDISC is significant with MUSCLE ranking higher on BRAliBase and MUDISC ranking higher on the MDSA data set.

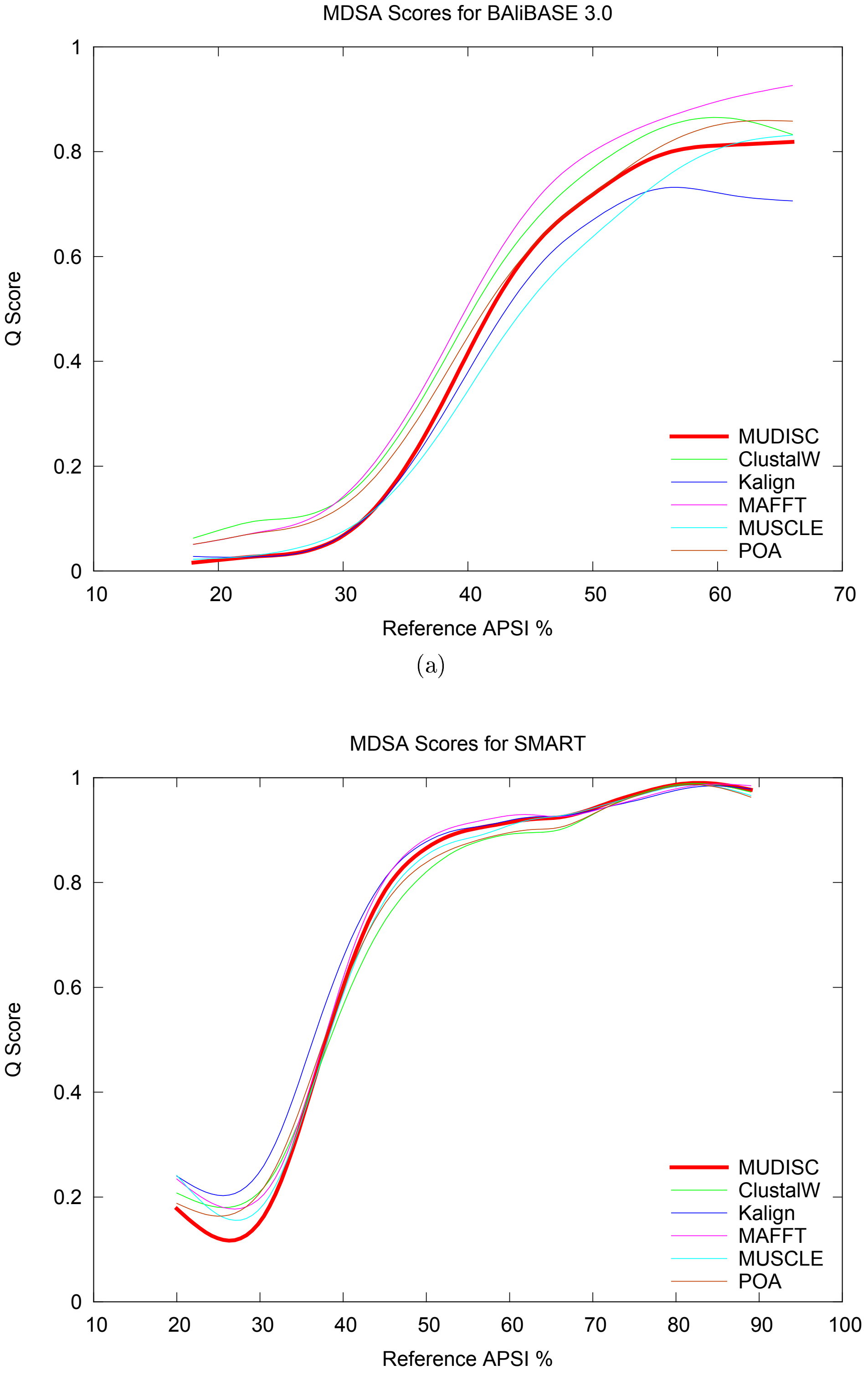
Program run times for MUSCLE, MUDISC, and MUFPGA are reported in Table 6.2. MUFPGA obtains an overall speedup of 33 relative to MUSCLE on the Influenza data set and a speedup of 154 on the HIV data set. Run times on the Corona and Herpes data sets are estimated since the accelerator currently only supports sequence lengths up to 16 K. To calculate these values, the pairwise alignment time in MUDISC is reduced by a factor of 290, which is extrapolated from timings on the Influenza and HIV data sets. Pairwise alignment in the third stage is accelerated by a factor of 176 on the Influenza data set and a factor of 283 on the HIV data set.



(a)

(b)

Figure 6.10: MUDISC (the new method) is compared with several alignment programs on a seven (a) and fifteen (b) sequence RNA reference set from BRAliBase 2.1. A higher score indicates better quality and is shown in relation to the average pairwise sequence identity (APSI). MUDISC uses discrete profile alignment.



(a)

(b)

Figure 6.11: MUDISC (the new method) is compared with several alignment programs on the MDSA data set which contains nucleotide adaptations of the BAliBASE (a) and SMART (b) reference alignments. BAliBASE includes reference sets 1–7. MUDISC uses discrete profile alignment.

Table 6.1: Comparison of MUSCLE and MUDISC alignment quality

Reference Set	MUSCLE Avg. Score	P-value, Rank	MUDISC (near) Avg. Score	P-value, Rank	MUDISC (nearest) Avg. Score
BRAliBase k7	0.6846	1.42e-4, >	0.6851	0.875, >	0.6839
BRAliBase k15	0.6914	1.18e-3, >	0.6819	0.285, <	0.6814
MDSA BAliBASE	0.3125	9.58e-9, <	0.3623	0.806, <	0.3629
MDSA SMART	0.6195	7.46e-3, <	0.6295	0.830, >	0.6316

The average quality scores for MUSCLE and MUDISC are shown on four reference sets. BRAliScore is reported for BRAliBase, and Q score is reported for MDSA. A variant of MUDISC that uses the nearest neighbor search method is also shown. The P-values from a Friedman rank test indicate the difference between two adjacent programs, and the relational symbols indicate which program ranked higher.

Table 6.2: Overall MSA Speedup: Program run times are in HH:MM:SS.

Data Set	Sequences	Avg. Length	MUSCLE	MUDISC	MUFPGA	Speedup
Influenza	12,104	1,717	01:25:40	00:20:03	00:02:35	33
HIV	2,144	9,019	02:09:29	01:28:42	00:00:50	154
Corona	400	29,531	03:17:47	02:22:43	00:00:41*	284*
Herpes	142	167,043	136:38:20	52:31:05	00:11:26*	716*

* Estimated

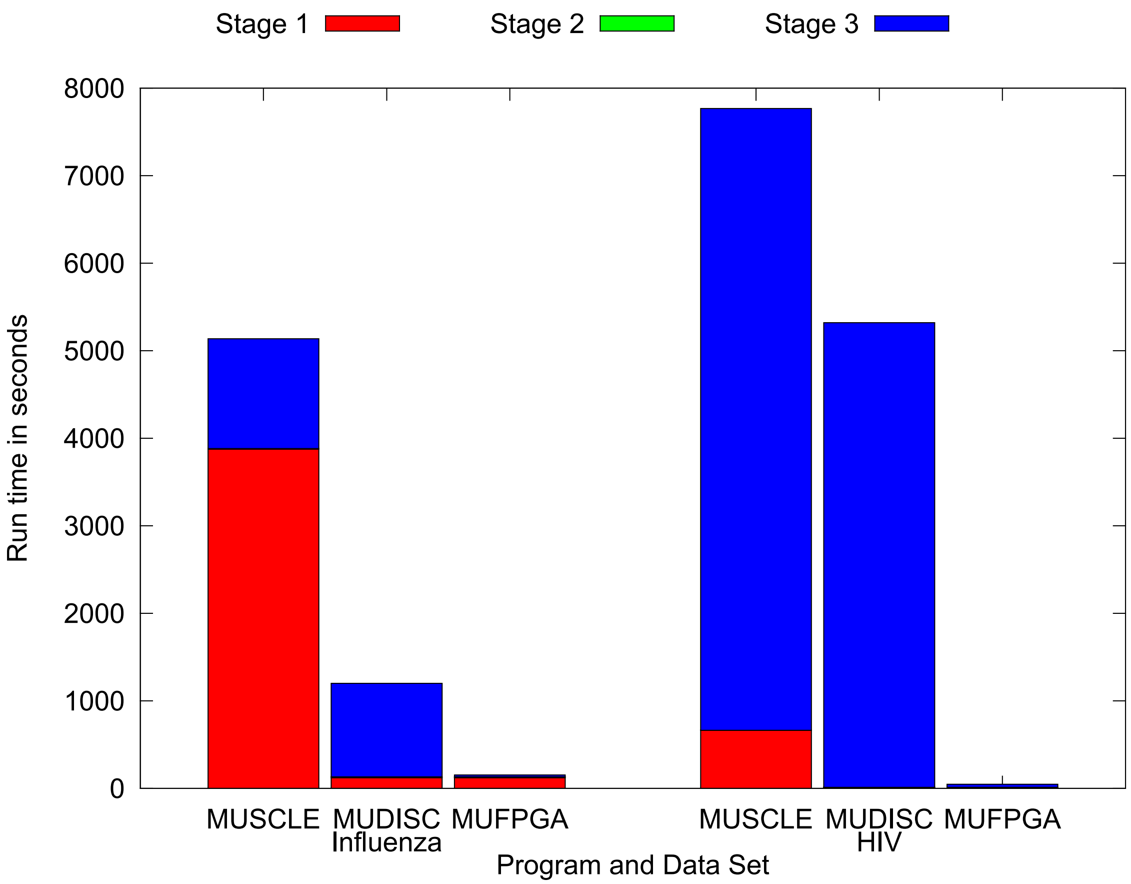


Figure 6.12: Overall program runtimes are shown on the Influenza and HIV data sets with a breakdown of time spent in each stage.

Figure 6.12 shows the proportion of time spent in the three stages of alignment on the Influenza and HIV data sets. The time for each stage includes both iterations. SSE acceleration improves the first stage run time with a speedup of 31 on the Influenza data set and a speedup of 79 on the Corona data set. Notice that the proportion of time spent in similarity calculations on the Influenza data set is greater with more sequences and limits the overall speedup. For pairwise alignment in the third stage, a speedup of 300 is obtainable on the FPGA accelerator when compared to a 2.4 GHz Core2 processor [54]. The profile reduction rate ranges from 44.1 to 98.7 MB/s and the reduction time ranges from 5.4 to 11.7% of the pairwise alignment time on the accelerator.

6.6 Conclusion

The discrete profile alignment algorithm presented here produces alignments with quality comparable to other leading MSA programs and enables the acceleration of progressive alignment. A speedup over 150 is demonstrated when discrete profile alignment is combined with an FPGA accelerator that uses a fine-grained parallel approach for the DP calculations of pairwise alignment. Previous coarse-grained approaches are limited by insufficient parallelism, particularly in the third stage of MSA. The discrete profile alignment algorithm in conjunction with a fast pairwise alignment algorithm advance the capabilities and performance of large-scale MSA. A key component of our method is a fast profile reduction algorithm on the host that can supply sequences at a rate comparable to the accelerator's input data rate. The reduction algorithm uses a near neighbor search in hyper-dimensional profile space to quantize profile positions at a rate up to 100 Mpos/s on a single core. Since this rate is sufficient to support the high-end performance of reconfigurable computing, other acceleration methods based on GPUs or SSE instructions may also be a viable option.

Minimizing the time for sequence similarity calculations in Stage 1 is also important to achieve significant speedup, especially for data sets with large numbers of sequences. Using SSE instructions reduces the time for sequence similarity calculations by a factor of 30–80. Thousands of sequences can be aligned in a few minutes when Stage 1 is accelerated with SSE instructions and Stage 3 is accelerated with reconfigurable computing.

Future work includes the demonstration of discrete profile alignment with GPU and SSE versions of our pairwise alignment algorithm. Another area of investigation is to apply the coarse-grained parallelism of cluster supercomputers and the fine-grained parallelism of reconfigurable computing to multiple sequence alignment. Since this work only uses a single core and one accelerator, a cluster with reconfigurable computing could provide an estimated 20–30x speedup beyond this work.

Chapter 7

Summary

Multiple sequence alignment is accelerated with new parallel algorithms and a novel high-performance architecture on a reconfigurable computing system. The new method produces alignments with quality comparable to other leading MSA programs and uses the combined strengths of a host microprocessor and accelerator to reduce the overall run time. Through fine-grained parallelism provided by an FPGA accelerator, the time required to compute a multiple alignment is reduced by a factor of up to 150 compared to a desktop computer on large data sets. Biologists and other researchers with large-scale alignments can see results within minutes instead of days through the contributions of this work. The following sections summarize several contributions that are needed to reach the demonstrated alignment quality and performance.

7.1 Contributions

The contributions of this work are as follows:

- Reconfigurable Computing Architecture
  - Access module functionality through the Distributed Access Protocol
  - Support user defined data access patterns
  - Promote developer productivity by partitioning complex FPGA resources
- Accelerated Pairwise Alignment
  - Provide traceback capability for alignment of long sequences
  - Develop dynamic programming algorithm with bounded space on an accelerator
- Discrete Profile Alignment
  - First progressive, profile alignment accelerated on FPGA
  - Define profile space with gaps
  - Develop sample point selection algorithm
  - Develop near neighbor search algorithm
  - Develop efficient host data structures and quantization technique
- Sequence Similarity Calculations
  - Vectorize the comparison of k-mer counts with SSE instructions
  - Vectorize the counting of identical symbols with SSE instructions

7.1.1 Reconfigurable Computing Architecture

While reconfigurable computing has demonstrated a performance advantage in some applications, several challenges must be overcome. Some examples include avoiding I/O bottlenecks and managing the flow of data to parallel processing elements. Another challenge is integrating user-developed modules with FPGA system resources like memory and I/O devices. Nevertheless, these challenges can be mitigated with a standard communication architecture for reconfigurable computing.

This research introduces a network architecture named Qnet along with a flexible protocol to address some of the challenges associated with reconfigurable computing. Qnet is a packet-switched network that connects modules within a reconfigurable system through parallel high-performance communication channels. Modules with a standard interface simplify system integration and encapsulate sharable devices or resources. In conjunction with Qnet, the Distributed Access Protocol (DAP) is presented, which offers a unified solution to data management.

The basic network components consist of a switch, Qports, and Qlinks. As the central figure in the network, the switch provides a path for communicating packets to other modules. Qports are the interface between modules and the network, and are the addressable endpoints of communication. Qports are connected by Qlinks, which consist of paired, unidirectional, point-to-point signaling channels. Each Qport has word-based flow control that will apply back-pressure on a link, delaying communication until the port is ready to receive. Hence, packets are not arbitrarily discarded, and the requirement to buffer an entire packet at the input of a module is removed while still maintaining performance.

Qnet reliably transfers data packets between endpoints through a simple protocol that requires minimal FPGA resources. Packets consist of a small header and a payload of variable size. The header specifies the source and destination endpoints with unique port identifiers and also indicates the payload length. When a packet header enters the switch, the output port is determined from the destination endpoint and remains the same for all following words of the packet. With a cut-through packet forwarding method, the full packet is not buffered in the switch. Packets that enter the switch simultaneously with different destinations pass through concurrently. This architecture allows parallel data transfer on all ports of an accelerator module.

The key features of Qnet are its modularity and standard interfaces. The hardware interface enables a developer to integrate pluggable components from a library with user-developed accelerator modules. Flexible protocols hide the complexities of the system and facilitate code reuse. Through communication packets, the Distributed Access Protocol allows any system module to access another's resources. Standard and user-definable access patterns direct packet assembly and disassembly at endpoints. Qnet also provides a three-party communication scheme that allows control packets injected into the system to initiate data flow and invoke operations.

7.1.2 Accelerated Pairwise Alignment

Unlike most FPGA acceleration methods that focus only on sequence comparison, this work presents a pairwise, global sequence alignment algorithm and architecture that includes traceback for implementation on reconfigurable hardware. Given a pair of sequences, the accelerator returns a list of edit operations constituting the optimal alignment. An accelerated library of alignment functions is easily incorporated into sequence analysis programs.

The presented global alignment algorithm overcomes the memory size and bandwidth limitations of FPGA accelerators and does not limit the sequence length by the number of PEs. Long sequences of DNA and protein are accommodated by the algorithm through a space-efficient traceback procedure that is accelerated in hardware. Traceback may occur in parallel with the next forward scan since it is implemented in a separate process from the systolic array.

The contributions of the algorithm are the bounded space requirement for full traceback memory on the FPGA and the reduced space for partial traceback memory on the host. These space reductions enable high-performance alignment of long sequences on a reconfigurable accelerator and are a match for FPGA memory capacities and bandwidth. The performance scales almost linearly with more PEs up to the sequence length and is maximized by increasing the number of PEs and size of traceback memory to match the available FPGA resources.

7.1.3 Discrete Profile Alignment

A new method to accelerate the third stage of progressive alignment reduces continuous space profiles into discrete profiles before they are pairwise aligned on reconfigurable hardware. By reducing each position of a profile to a discrete symbol, the profiles can be aligned like sequences with the pairwise alignment accelerator. The novel traceback capabilities of the pairwise alignment accelerator combined with the extended discrete profile formalism advance the capabilities and performance of MSA.

Five dimensions in profile space are defined to represent profiles that contain nucleotide and gap character frequencies. From this space, a selection algorithm determines a discrete number of sample points and corresponding symbols to represent nearby points. Since the pairwise alignment of discrete symbols requires comparative weights, a substitution table is calculated that reflects the cost of substituting one of these sample points with another. In the case of pairwise sequence alignment, the substitution table reflects the cost of substituting one character with another. The discrete symbols associated with each sample point become the indices into the substitution table and the codebook for a quantization algorithm.

For the accelerator to sustain maximum performance, the host system must supply reduced profiles at the accelerator's input data rate. A new quantization technique is used for this reduction on the host to reach the needed performance. For each continuous profile position, the reduction algorithm searches for a nearby sample point and then returns the corresponding discrete symbol. Efficient host-side data structures support high-throughput quantization.

7.2 Future Work

Several Qnet features not explored in this work are worth investigation. For example, Qlinks are a potential interface to partial reconfiguration areas. If Qlink locations are locked, attached modules could be dynamically reconfigured. The sub-microsecond latency achieved in the PowerPC benchmark system should be realizable in other processor implementations with a similar interface design. Since DAP is applicable to distributed systems without reconfigurable computing, a software implementation of DAP may benefit scatter/gather operations on a conventional cluster system. Finally, DAP may deliver graphics commands to a GPU with better performance and flexibility. Through Qnet and a PCI Express bridge, the FPGA could offload specific functions of the graphics application.

The advantages of the presented pairwise alignment method are also applicable to accelerating local alignment and semiglobal alignment. A general-purpose accelerated alignment library that consists of both local and global methods may be applied to bioinformatics programs with minimal effort. Adding support for affine gaps and position specific gap costs would extend the usefullness of the library to a greater number of applications. SSE and GPU versions of the pairwise alignment algorithm would benefit users without reconfigurable hardware.

Discrete profile alignment could be extended to support proteins that have a larger alphabet by adding more dimensions to profile space. Since proteins have an amino acid alphabet of 20 characters, instead of 4 like DNA, an alphabet compression scheme would be necessary to reduce the number of characters and the corresponding dimensionality of profile space to a practical number. Reducing the alphabet to six classes based on physico-chemical properties, as done in MAFFT, would only require 6 dimensions for the classes plus 1 for gaps. Another extension which might improve quality involves adding more profile space dimensions to support position specific gap costs, secondary structure information, residue chemical properties, or other position specific information. Nevertheless, adding too many dimensions may severely impact performance.

Currently, discrete profile alignment uses sample points that are distributed on a regular grid in profile space. However, varying the distribution of sample points may improve alignment quality or reduce the number of sample points needed to maintain quality. Reducing the number of sample points is beneficial because it allows for a more efficient and higher performing FPGA implementation. Quality might also be improved during progressive alignment by using a different set of sample points for each level of the guide tree. At levels closer to the root, group-to-group alignment includes a greater number of sequences. Hence, using more sample points for these levels may improve quality by reducing the quantization error.

The general applicability of discrete profile alignment could be demonstrated with SSE and GPU versions of our pairwise alignment algorithm. Users with only commodity hardware could then realize the performance benefits of discrete profile alignment, although the speedup of these versions is anticipated to be much less than the speedup of the FPGA version demonstrated here. A performance comparison between SSE, GPU, and FPGA accelerated versions would provide insights into the strengths and limitations of each acceleration method.

Another area of investigation is to apply the coarse-grained parallelism of cluster supercomputers and the fine-grained parallelism of reconfigurable computing to multiple sequence alignment. Since this work only uses a single core and one accelerator, a cluster with reconfigurable computing could provide an estimated 20–30x speedup beyond this work.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the spring joint computer conference*, pages 483–485, New York, NY, USA, 18-20 April 1967. ACM.

[2] S. Bailey and T. Talpey. The architecture of direct data placement (DDP) and remote direct memory access (RDMA) on internet protocols. RFC 4296, December 2005. http://www.ietf.org/rfc/rfc4296.txt.

[3] Khaled Benkrid, Ying Liu, and AbdSamad Benkrid. A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(4):561–570, April 2009.

[4] Ajay V. Bhatt. Creating a PCI Express interconnect. White paper, Technology and Research Labs, Intel Corporation, 2002.

[5] V. Michael Bove, Jr., Mark Lee, Yuan-Min Liu, Christopher McEniry, Thomas Nwodoh, and John Watlington. Media processing with field-programmable gate arrays on a microprocessor's local bus. In *Proceedings of SPIE Media Processors*, volume 3655, pages 14–20, San Jose, CA, USA, January 1999.

[6] Nicolas Bray and Lior Pachter. MAVID: Constrained ancestral alignment of multiple sequences. *Genome Research*, 14:693–699, 2004.

[7] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder, editors. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.

[8] Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, October 1988.

[9] Hyrum Carroll, Wesley Beckstead, Timothy O'Connor, Mark Ebbert, Mark Clement, Quinn Snell, and David McClellan. DNA reference alignment benchmarks based on tertiary structure of encoded proteins. *Bioinformatics*, 23(19):2648–2649, 2007.

[10] Kridsadakorn Chaichoompu and Surin Kittitornkun. Multithreaded ClustalW with improved optimization for Intel multi-core processor. In *International Symposium*

on Communications and Information Technologies, ISCIT '06, pages 590–594, 18-20 October 2006.

[11] Patrick Chain, Stefan Kurtz, Enno Ohlebusch, and Tom Slezak. An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105–123, June 2003.

[12] James Cheetham, Frank Dehne, Sylvain Pitre, Andrew Rau-Chaplin, and Peter J. Taillon. Parallel CLUSTAL W for PC clusters. In *Computational Science and Its Applications, ICCSA 2003*, 2003.

[13] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the Design Automation Conference (DAC'01)*, pages 684–689, Las Vegas, Nevada, USA, 18-22 June 2001.

[14] Frédéric Delsuc, Henner Brinkmann, and Hervé Philippe. Phylogenomics and the reconstruction of the tree of life. *Nature Reviews Genetics*, 6(5):361–375, May 2005.

[15] Xi Deng, Eric Li, Jiulong Shan, and Wenguang Chen. Parallel implementation and performance characterization of MUSCLE. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[16] Pedro C. Diniz and Joonseok Park. Data reorganization engines for the next generation of system-on-a-chip FPGAs. In *Proceedings of the 10th International Symposium on Field-Programmable Gate Arrays (FPGA'02)*, Monterey, California, USA, 24-26 February 2002.

[17] Justin Ebedes and Amitava Datta. Multiple sequence alignment in parallel on a workstation cluster. *Bioinformatics*, 20(7):1193–1195, 2004.

[18] Sean R. Eddy. SQUID – C function library for sequence analysis. compalign source, 2009.

[19] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[20] Elizabeth W. Edmiston, Nolan G. Core, Joel H. Saltz, and Roger M. Smith. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, 17(3):259–275, 1988.

[21] Tarek El-Ghazawi. Is high-performance, reconfigurable computing the next supercomputing paradigm? In *Proceedings of the ACM/IEEE SC'06 Conference*, Tampa, Florida, November 2006.

[22] Isaac Elias. Settling the intractability of multiple alignment. *Journal of Computational Biology*, 13(7):1323–1339, September 2006.

[23] Eleazar Eskin. From profiles to patterns and back again: A branch and bound algorithm for finding near optimal motif profiles. In *Proceedings of the eighth annual international conference on Resaerch in computational molecular biology, RECOMB '04*, pages 115–124. ACM, 2004.

[24] Philippe Faes, Bram Minnaert, Mark Christiaens, Eric Bonnet, Yvan Saeys, Dirk Stroobandt, and Yves Van de Peer. Scalable hardware accelerator for comparing DNA and protein sequences. In *Proceedings of the First International Conference on Scalable Information Systems (INFOSCALE '06)*, Hong Kong, 29 May-1 June 2006. ACM.

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

[26] Michael Feldman. IBM cuts cell loose. HPCwire, 24 November 2009.

[27] Da-Fei Feng and Russell F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[28] Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, December 1937.

[29] Shea N. Gardner, Marisa W. Lam, Nisha J. Mulakken, Clinton L. Torres, Jason R. Smith, and Tom R. Slezak. Sequencing needs for viral diagnostics. *Journal of Clinical Microbiology*, 42(12):5472–5476, December 2004.

[30] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005.

[31] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, December 1982.

[32] Osamu Gotoh. Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments. *Journal of Molecular Biology*, 264(4):823–838, 1996.

[33] Osamu Gotoh. Multiple sequence alignment: Algorithms and applications. *Advances in Biophysics*, 36:159–206, 1999.

[34] Catherine Grasso and Christopher Lee. Combining partial order alignment and progressive multiple sequence alignment increases alignment speed and scalability to very large alignment problems. *Bioinformatics*, 20(10):1546–1556, 2004.

[35] X. Guan and E. C. Uberbacher. A multiple divide-and-conquer (MDC) algorithm for optimal alignments in linear space. Technical Report ORNL/TM-12764, Oak Ridge National Lab., June 1994.

[36] Jaap Heringa. Two strategies for sequence comparison: profile-preprocessed and secondary structure-induced multiple alignment. *Computers & Chemistry*, 23(3-4):341–364, 1999.

[37] Dzung T. Hoang and Daniel P. Lopresti. FPGA implementation of systolic sequence alignment. In Herbert Grünbacher and Reiner W. Hartenstein, editors, *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, pages 183–191. Springer-Verlag, Berlin, 1992.

[38] Ricardo P. Jacobi, Mauricio Ayala-Rincón, Luis G.A. Carvalho, Carlos H. Llanos, and Reiner W. Hartenstein. Reconfigurable systems for sequence alignment and for general dynamic programming. *Genetics and Molecular Research*, 4(3):543–552, September 2005.

[39] Nachiket Kapre, Nikil Mehta, Michael deLorimier, Raphael Rubin, Henry Barnor, Michael J. Wilson, Michael Wrighton, and André DeHon. Packet switched vs. time multiplexed FPGA overlay networks. In *The 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, 24-26 April 2006.

[40] Kazutaka Katoh, Kazuharu Misawa, Kei ichi Kuma, and Takashi Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research*, 30(14):3059–3066, July 2002.

[41] Kazutaka Katoh and Hiroyuki Toh. Recent developments in the MAFFT multiple sequence alignment program. *Briefings in Bioinformatics*, 9(4):286–298, 2008.

[42] Jens Kleinjung, Nigel Douglas, and Jaap Heringa. Parallelized multiple alignment. *Bioinformatics*, 18(9):1270–1271, 2002.

[43] Timo Lassmann, Oliver Frings, and Erik L. L. Sonnhammer. Kalign2: high-performance multiple alignment of protein and nucleotide sequences allowing external features. *Nucleic Acids Research*, 37(3):858–865, 2009.

[44] Ivica Letunic, Richard R. Copley, Birgit Pils, Stefan Pinkert, Jörg Schultz, and Peer Bork. SMART 5: domains in the context of genomes and networks. *Nucleic Acids Research*, 34(suppl 1):D257–D260, 2006.

[45] Kuo-Bin Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[46] Yiming Li and Cheng-Kai Chen. Parallelization of multiple genome alignment. In *High Performance Computing and Communications, HPCC 2005*, volume LNCS 3726, pages 910–915. Springer-Verlag Berlin Heidelberg, 2005.

[47] Xu Lin, Zhang Peiheng, Bu Dongbo, Feng Shengzhong, and Sun Ninghui. To accelerate multiple sequence alignment using FPGAs. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)*, page 5, July 2005.

[48] R. Lipton and D. Lopresti. Comparing long strings on a short systolic array. In Will Moore, Andrew McCabe, and Roddy Urquhart, editors, *Systolic Arrays*, pages 363–376. Hilger, 1987.

[49] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In *High Performance Computing, HiPC 2006*, volume LNCS 4297, pages 363–374. Springer Berlin / Heidelberg, 2006.

[50] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, May 2009.

[51] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. MSA-CUDA: Multiple sequence alignment on graphics processing units with CUDA. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'09)*, pages 121–128. IEEE Computer Society, July 2009.

[52] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. In *IEEE*

International Symposium on Parallel & Distributed Processing, IPDPS 2009, pages 1–8, May 2009.

[53] Scott Lloyd and Quinn Snell. Qnet: A modular architecture for reconfigurable computing. In *Proceedings of the 2008 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'08)*, pages 259–265, July 2008.

[54] Scott Lloyd and Quinn O. Snell. Hardware accelerated sequence alignment with traceback. *International Journal of Reconfigurable Computing*, 2009:10, 2009. Article ID 762362.

[55] Jiancong Luo, Ishfaq Ahmad, Munib Ahmed, and Raymond Paul. Parallel multiple sequence alignment with dynamic scheduling. In *International Conference on Information Technology: Coding and Computing, ITCC 2005*, volume 1, pages 8–13, April 2005.

[56] C. Macken, H. Lu, J. Goodman, and L. Boykin. The value of a database in surveillance and vaccine selection. *International Congress Series*, 1219:103–106, October 2001.

[57] Terrence S. T. Mak, Pete Sedcole, Peter Y. K. Cheung, and Wayne Luk. On-FPGA communication architectures and design factors. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'06)*, pages 1–8, August 2006.

[58] Mentor. *Handel-C Language Reference Manual.* Mentor Graphics Corp., 2007. http://www.mentor.com/.

[59] Mentor. *DK Design Suite Software Product Description.* Mentor Graphics Corp., 2008. http://www.mentor.com/.

[60] Dmitri Mikhailov, Haruna Cofer, and Roberto Gomperts. Performance optimization of Clustal W: Parallel Clustal W, HT Clustal, and MULTICLUSTAL. SGI ChemBio, Silicon Graphics, Inc., 2001.

[61] Guilherme L. Moritz, Cristiano Jory, Heitor S. Lopes, and Carlos R. Erig Lima. Implementation of a parallel algorithm for protein pairwise alignment using reconfigurable computing. In *Reconfigurable Computing and FPGA's (ReConFig)*, pages 99–105. IEEE, September 2006.

[62] MPI. *MPI: A Message-Passing Interface Standard.* MPI Forum, November 2003. http://www.mpi-forum.org/.

[63] Robert Muth and Udi Manber. Approximate multiple string search. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, volume LNCS 1075, pages 75–86. Springer Berlin / Heidelberg, 1996.

[64] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences : CABIOS*, 4(1):11–17, 1988.

[65] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[66] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March/April 2008.

[67] Cédric Notredame. Recent progresses in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131–144, 2002.

[68] Cédric Notredame, Desmond G. Higgins, and Jaap Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–217, September 2000.

[69] Tim Oliver, Bertil Schmidt, Douglas Maskell, Darran Nathan, and Ralf Clemens. High-speed multiple sequence alignment on a reconfigurable platform. *International Journal of Bioinformatics Research and Applications (IJBRA)*, 2(4):394–406, 2006.

[70] Tim Oliver, Bertil Schmidt, Darran Nathan, Ralf Clemens, and Douglas Maskell. Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. *Bioinformatics*, 21(16):3431–3432, 2005.

[71] OMG. *Common Object Request Broker Architecture: Core Specification.* Object Management Group, Inc., March 2004. http://www.omg.org/.

[72] OpenMP. *OpenMP Application Program Interface.* OpenMP Architecture Review Board, May 2008.

[73] Joonseok Park and Pedro Diniz. An external memory interface for FPGA-based computing engines. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, 2001.

[74] Thilo Pionteck, Carsten Albrecht, Roman Koch, Erik Maehle, Michael Hübner, and Jürgen Becker. Communication architectures for dynamically reconfigurable FPGA

designs. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–8, March 2007.

[75] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, December 2004.

[76] Andrew Rambaut and Nicholas C. Grassly. Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Computer Applications in the Biosciences : CABIOS*, 13(3):235–238, June 1997.

[77] Tirath Ramdas and Gregory Egan. A survey of FPGAs for acceleration of high performance computing and their application to computational molecular biology. In *TENCON 2005 IEEE Region 10*, pages 1–6, November 2005.

[78] Vipin Sachdeva, Michael Kistler, Evan Speight, and Tzy-Hwa Kathy Tzeng. Exploring the viability of the Cell Broadband Engine for bioinformatics applications. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007*, pages 1–8, March 2007.

[79] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[80] Craig Sanderson. Simplify FPGA application design with DIMEtalk. *Xcell Journal*, Winter(51):104–107, 2004.

[81] Tom Slezak, Tom Kuczmarski, Linda Ott, Clinton Torres, Dan Medeiros, Jason Smith, Brian Truitt, Nisha Mulakken, Marisa Lam, Elizabeth Vitalis, Adam Zemla, Carol Ecale Zhou, and Shea Gardner. Comparative genomics tools applied to bioterrorism defence. *Briefings in Bioinformatics*, 4(2):133–149, June 2003.

[82] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[83] Robert R. Sokal and Charles D. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38(22):1409–1438, 20 March 1958.

[84] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1(1):107, October 2008.

[85] Koji Tajima. Multiple DNA and protein sequence alignment on a workstation and a supercomputer. *Computer Applications in the Biosciences : CABIOS*, 4(4):467–471, 1988.

[86] Guangming Tan, Shengzhong Feng, and Ninghui Sun. Parallel multiple sequences alignment in SMP cluster. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)*, July 2005.

[87] Guangming Tan, Liu Peng, Shengzhong Feng, and Ninghui Sun. Load balancing and parallel multiple sequence alignment with tree accumulation. In *Euro-Par 2006 Parallel Processing*, volume LNCS 4128, pages 1138–1147. Springer Berlin / Heidelberg, 2006.

[88] Julie D. Thompson, Desmond G. Higgins, and Toby 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*, 22(22):4673–4680, 1994.

[89] Julie D. Thompson, Patrice Koehl, Raymond Ripp, and Olivier Poch. BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark. *Proteins: Structure, Function, and Bioinformatics*, 61(1):127–136, 2005.

[90] Julie D. Thompson, Frédéric Plewniak, and Olivier Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[91] Keith D. Underwood, K. Scott Hemmert, and Craig Ulmer. Architectures and APIs: Assessing requirements for delivering FPGA performance to applications. In *Proceedings of the ACM/IEEE SC'06 Conference*, Tampa, Florida, November 2006.

[92] Tom VanCourt and Martin C. Herbordt. Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135–145, March 2007.

[93] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. Accelerating multiple sequence alignment with the Cell BE processor. *The Computer Journal*, 53(6):814–826, 2010.

[94] Susana Vinga and Jonas Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(4):513–523, 2003.

[95] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 256–266, New York, NY, USA, 19-21 May 1992. ACM.

[96] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, Winter 1994.

[97] Ting Wang and Gary D. Stormo. Identifying the conserved network of cis-regulatory sites of a eukaryotic genome. *Proceedings of the National Academy of Sciences of the United States of America*, 102(48):17400–17405, November 2005.

[98] Stefan Washietl, Ivo L. Hofacker, and Peter F. Stadler. Fast and reliable prediction of noncoding RNAs. *Proceedings of the National Academy of Sciences of the United States of America*, 102(7):2454–2459, February 2005.

[99] Andreas Wilm, Indra Mainz, and Gerhard Steger. An enhanced RNA alignment benchmark for sequence alignment programs. *Algorithms for Molecular Biology*, 1:19, 2006.

[100] Adrianto Wirawan, Bertil Schmidt, and Chee Keong Kwoh. Pairwise distance matrix computation for multiple sequence alignment on the Cell Broadband Engine. In *Computational Science – ICCS 2009*, volume LNCS 5544, pages 954–963. Springer-Verlag Berlin Heidelberg, 2009.

[101] Xilinx. *EDK Concepts, Tools, and Techniques.* Xilinx, Inc., 2007. http://www.xilinx.com/.

[102] Xilinx. *Multi-Port Memory Controller (MPMC) Product Specification.* Xilinx, Inc., 2009. http://www.xilinx.com/.

[103] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search with FPGAs. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 271–282, 2002.

[104] Jaroslaw Zola, Xiao Yang, Adrian Rospondek, and Srinivas Aluru. Parallel T-Coffee: A parallel multiple sequence aligner. In *Proceedings of the ISCA 20th International Conference on Parallel and Distributed Computing Systems*, pages 248–253, 24-26 September 2007.