DNA Physical Mapping on a Reconfigurable Platform

Adriano Idalgo and Nahri Moreano

Department of Computing,
Federal University of Mato Grosso do Sul, Brazil
adriano.idalgo@gmail.com, nahri@dct.ufms.br

Abstract. Reconfigurable architectures enable the hardware function to be implemented by the user and, due to its characteristics, have been used in many areas, including Bioinformatics. One application of Bioinformatics is the consecutive ones problem, which consists in finding a permutation of columns in a binary matrix, in such a way that all the ones in each row are consecutive. This matrix represents information about DNA fragments and probes, which allow the determination of the order of the nitrogenated bases that form the original DNA.

This work proposes a hybrid software/hardware system for solving the consecutive ones problem. Since this problem processes large volumes of data, the goal is to reduce its execution time, compared to a SW algorithm. We present and analyze several implementations, in the reconfigurable hardware, of sections of this algorithm, using a Virtex-II FPGA. Experiments performed using real chromosomes produced speedups of up to 29.62 and show potential for further optimizations exploiting dynamic reconfiguration.

Keywords: Consecutive ones problem, Reconfigurable architectures, Software/hardware partitioning.

1 Introduction

Reconfigurable computing characterizes the hardware in which the logic implemented is created and modified by the user and not by the manufacturer. It introduces many application possibilities that could not be developed using a hardware with fixed and predefined functionality. There is also a performance improvement potential of the application implemented on a reconfigurable hardware with respect to their implementation in software [1].

In another research area, the Bioinformatics, achievements have been reached recently on DNA mapping, using computational techniques to assist the sequencing task, which consists of identifying the order of base pairs in a chromosome. Due to technical limitations, the DNA is not directly sequenced and must be broken into fragments. One approach is to represent the information of fragments and probes as a binary matrix and to arrange the columns of the matrix so that all ones in each row are consecutive, in order to determine the order among the fragments. This problem is called the *Consecutive Ones Problem* [2].

R. Woods et al. (Eds.): ARC 2008, LNCS 4943, pp. 27–38, 2008.

Besides its vast practical application, DNA mapping requires manipulating huge volumes of data. In particular, the consecutive ones problem handles binary data. Such characteristics motivate the study and implementation, on reconfigurable devices, of solutions to this problem, in order to obtain a better performance in its execution.

In this work we present a hybrid Software/Hardware (SW/HW) solution for the consecutive ones problem. We developed several implementations, on a reconfigurable hardware, of operations with high execution frequency in a well-known algorithm [3] to the consecutive ones problem. These implementations exploit different trade-offs between computation and communication costs, on the processor/FPGA platform used. We performed an experimental evaluation, in order to analyze our solutions and compare them to the pure SW solution, using chromosomes of two living beings. The results show large performance gains and possibilities for improvements using reconfiguration.

This paper is organized as follows: Section 2 describes previous works with solutions for Bioinformatics problems using reconfigurable architectures. The consecutive ones problem and an algorithm for it are introduced in Section 3. Section 4 presents our hybrid implementations for this problem, while the results obtained through experiments are analyzed in Section 5. Finally, in Section 6 the conclusions and future extensions of this work are discussed.

2 Related Work

Most of works which solve Bioinformatics problems using reconfigurable architectures focus on the sequence alignment problem using the Smith-Waterman algorithm, which is based on the dynamic programming technique. The works in [4,5] implemented this algorithm on a FPGA, using processing elements which form a linear structure and operate in parallel. This processing elements compute the similarity between two sequences and compute in parallel all the values in each diagonal of the dynamic programming matrix.

The authors in [6] implemented the Smith-Waterman algorithm on a multi-FPGA network. In [7], a prefetching scheme for search in a sequence database is implemented, in order to accelerate the sequence alignment task on a FPGA, overlapping computation with communication.

The work in [8] presents a hybrid SW/HW system for the reconstruction of the phylogenetic tree of DNA sequences. This tree represents the evolution history of different organisms. A genetic algorithm is implemented in SW and a maximum likelihood function is implemented on a FPGA.

In [9] the authors designed a variant of BLAST, a well-known similarity search tool to compare DNA sequences. The goal was to build a specialized BLAST accelerator using a system with a general-purpose processor and a reconfigurable hardware (FPGA) associated with the disk controller.

There is not, to our knowledge, any previous solution to the consecutive ones problem in hardware, with either fixed or reconfigurable logic.

3 DNA Physical Mapping

Many DNA molecules are over millions of base pairs long, therefore too large to be sequenced as a whole. A physical map of a DNA contains the location of certain markers along the molecule. Given a sequenced fragment of the DNA, the physical map is used to locate the fragment in the DNA by matching markers in the fragment and the physical map.

In order to create a physical map, it is necessary to obtain several copies of the DNA, to use restriction enzymes to break each copy into fragments, and to clone each fragment, producing a collection of clones. Then we need to examine the clones for overlaps among them. The hybridization technique can be applied to obtain overlap information from the clones, using probes (short sequences) and verifying if each probe binds to each clone. If a probe hybridizes to two clones, then the clones overlap each other. Using the overlap information from the clones, it is possible to determine their relative order in the DNA.

3.1 Consecutive Ones Problem and Algorithm

The information produced by hybridization experiments with n clones and m probes can be modeled by a binary matrix M, $n \times m$, where the M_{ij} position says if probe j hybridized ($M_{ij} = 1$) or not ($M_{ij} = 0$) to clone i. M is said to own the *consecutive ones property* for rows (C1P), if all ones in each row are consecutive. In order to get the DNA physical map it is necessary to solve the consecutive ones problem, i.e., to find a permutation of columns (probes) such that all ones in each row (clone) are consecutive.

We describe briefly the polynomial time complexity algorithm proposed in [3] to this problem. Other solutions can be found in [10,11]. The algorithm uses certain criteria to separate the rows of M into components. If each component has the C1P, then M will also have this property. The algorithm performs the following steps: separate rows into components, permute the columns of each component, and join the components [2]. The following relations between rows and the number of intersections between their column sets are used to form the components and to guide the permutation of their columns.

Definition 1. *For each row i of M, let S_i be the set of columns k where $M_{ik} = 1$. Given two rows i and j three situations can arise:*

1. $S_i \cap S_j = \emptyset$;
2. $S_i \subseteq S_j$ *or* $S_j \subseteq S_i$;
3. $S_i \cap S_j \neq \emptyset$, *and none of them is a subset of the other.*

An undirected graph G_C is built, where each vertex in G_C corresponds to a row of M. There is an edge between vertices i and j if $S_i \cap S_j \neq \emptyset$, and none of them is a subset of the other. The components of M are the connected components of G_C. Thus, each component is a sub-matrix of M with the same number of columns and possibly fewer rows than M.

In order to permute the columns of a component, the rows are processed one by one. We permute the first two rows placing the exclusive 1s of one row in a direction (right or left), the exclusive 1s of the other in the opposite direction, and the intersection 1s in the middle. To insert a new row k it is necessary to find two previously placed rows, i and j, such that there are edges (k, i) and (i, j) in G_C. If $|S_j \cap S_k| < min(|S_j \cap S_i|, |S_i \cap S_k|)$, k is placed in the same direction of i (with respect to j); otherwise, k will have the opposite direction.

We represent the permutation solutions by associating a set of possible columns to each component column. These sets indicate which original matrix columns correspond to the component column. In the end, the column sets codify the order in which the columns must be arrange so that matrix M has all 1s consecutive in each row.

To join the components, a directed graph G_M is built, where vertices correspond to components of G_C. There is a directed edge from vertex α to β if, for every row i of component β, the set S_i is contained at least in one set S_j of component α. Since the relationship between components is given by edge direction, the topological order of G_M vertices indicates the order the components must be joined. The first component, α, is fixed, and to join another component β, the row l of β that has the leftmost 1 is chosen. Let c_β be this column. We must find all rows in α that contain S_l, and find the leftmost column c_α where all these rows have 1. Then, c_α and c_β are made the same column and the rows of β are joined to α. The final matrix with C1P is obtained after joining all components.

Fig. 1 illustrates the algorithm steps [2]. Matrix M in Fig. 1(a) represents hybridization results of 8 clones with 9 probes, and the corresponding graphs G_C, with four connected components, and G_M are shown in Figs. 1(b) and 1(c). Fig. 1(d) shows the result from the permutation of M.

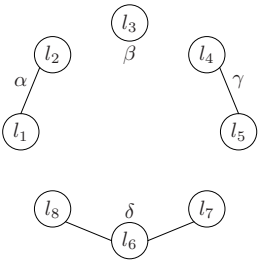
4 Hardware/Software Solution

We use, in our solutions to the consecutive ones problem, an hybrid architecture composed of general purpose processor with a reconfigurable component attached to it. The reconfigurable hardware (a FPGA) is used to implement a hardware accelerator for this application. Due to the characteristics of this architecture, the coupling between host processor and FPGA is weak, and the fastest available communication between them is through the network interface present in both devices. Therefore, the communication overhead can cause a major impact on the application total execution time, and consequently, the implementation strategy chosen for the communication is very important.

Initially, we implemented a SW solution to the problem, which uses only the host processor. Then, we profiled this implementation, determining the sections of the algorithm that consume most of the execution time. Based on these profile information, we performed the SW/HW partitioning. The application operations are divided, such that operations that can not be easily mapped on reconfigurable logic are executed on the host processor, and operations that can benefit from hardware implementation and consume substantial execution time are executed

	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}	{9}
l_1	1	1	0	1	1	0	1	0	1
l_2	0	1	1	1	1	1	1	1	1
l_3	0	1	0	1	1	0	1	0	1
l_4	0	0	1	0	0	0	0	1	0
l_5	0	0	1	0	0	1	0	0	0
l_6	0	0	0	1	0	0	1	0	0
l_7	0	1	0	0	0	0	1	0	0
l_8	0	0	0	1	1	0	0	0	1

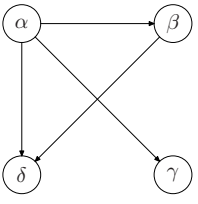
(a)



(b)

	{1}	{5, 9}	{4}	{7}	{2}	{6}	{3}	{8}
l_1	1	1	1	1	1	0	0	0
l_2	0	1	1	1	1	1	1	1
l_3	0	1	1	1	1	0	0	0
l_4	0	0	0	0	0	0	1	1
l_5	0	0	0	0	0	1	1	0
l_6	0	0	0	1	1	0	0	0
l_7	0	0	0	0	1	1	0	0
l_8	0	1	1	1	0	0	0	0

(d)



(c)

Fig. 1. (a) Matrix M and corresponding (b) graph G_C (with components α, β, γ, and δ), (c) graph G_M, and (d) final matrix

on the FPGA. Thus, some sections of the application code executed originally in software have been replaced by function calls for communication to the FPGA.

We selected for hardware implementation two critical operations: clones comparison and construction of the column sets for each component. For each operation, we developed different hardware implementations, producing several hybrid SW/HW solutions. The main goal is to reduce the execution time of the consecutive ones problem solution, when compared to its SW implementation.

The HW component of the hybrid solutions is composed of the main modules described below. In addition to these modules, it can use memory banks available on the board that contains the FPGA to increase the processing capacity.

- **Control:** Manages the data flow necessary for the other modules in the FPGA, so it is the most complex module. The control consists of a state machine which coordinates all steps of the HW operations.
- **Compare clones:** Compares two rows and counts their intersections.
- **Construct sets:** Constructs the column sets for each component.
- **Receive data:** Receives input data from the SW part of the application.
- **Send data:** Sends the operation results to the SW part of the application.

4.1 Comparing Clones

The comparison of rows (clones) of M determines whether two rows belong to same component and also the number of intersections between their column sets. The rows, represented as sequences of bits, are sent from the SW program to the FPGA. Given that the rows can be very long, the HW implementation of this operation performs comparisons and intersection counting in blocks of bits. In order to process two rows i and j, we operate the first block of i with the first block of j, then the second block of i with the second one of j, and so on. At each clock cycle, the circuit processes a pair of blocks, until the two rows are entirely operated. We use 32 bit-blocks.

The circuit of Fig. 2 shows how the row comparison operation is implemented in HW. At each cycle, the row comparator performs the following operations:

- Perform an AND operation with the two blocks, obtaining the auxiliary result R.
- In parallel:
  - If R is different from 0, the blocks have intersection. The relation result is set appropriately.
  - Perform a XOR operation with the first block and R. If the result is different from 0, the first block is not contained by the second one. The relation result is set accordingly.
  - Perform a XOR operation with the second block and R. If the result is different from 0, the second block is not contained by the first one. The relation result is set accordingly.
  - Count the number of 1s in R and accumulate the partial result.

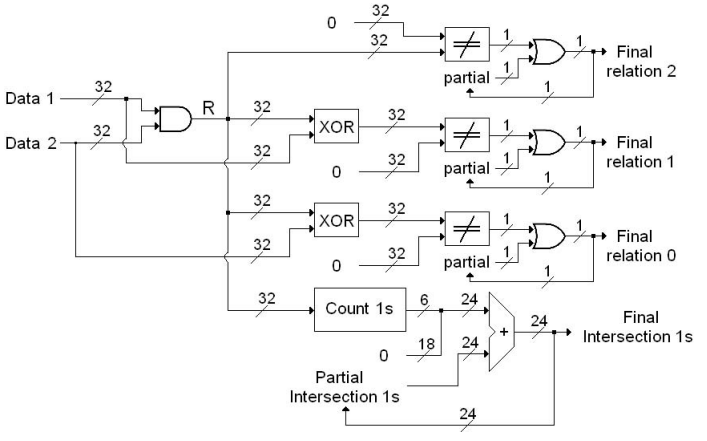


Fig. 2. Circuit for clone comparison

4.2 Constructing Column Sets

After permuting the component columns, the construction of the column sets of each component is performed processing the components by columns. When scanning column c, if the position corresponding to row i contains 1, it indicates that at least one element from the set S_i of columns of row i belongs also to the column set of c. If this is the first 1 found, the set S_i is copied to this column set. Otherwise, only the columns belonging to both sets will be part of the column set of c. If the position corresponding to row i contains 0, the columns belonging to S_i do not belong to this column set. So any element in S_i should be removed from the column set of c.

The column set construction operation implemented in HW uses memory banks of the FPGA board to store the matrix M, the permuted component stored by columns, and the indexes of the rows of M that belong to the component. The set constructor uses the same strategy of the clone comparator and process the input data in blocks. The input data to the set constructor are a block from the component column and a block from the column set of a row. The output is the block from the column set of the component.

The circuit of Fig. 3 shows how this operation is implemented in HW. The constructor receives a column block every 32 clock cycles and a row block each cycle. It maintains the partial result P and an index indicating the position to be accessed in the column. After receiving a row block, the constructor verify the column current position and performs the following operations:

- If this position has 0, the block of the row set is inverted (operation NOT) and perform an AND operation with this result and P.
- If this position has 1, perform an AND with the block of the row set and P.
- The result is stored in P, and the column index is decremented.

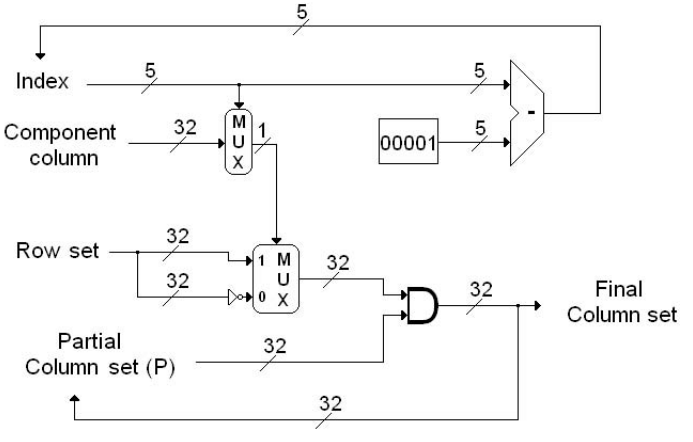


Fig. 3. Circuit for column set construction

4.3 Hybrid Implementations

We developed five different hybrid solutions using several implemented versions of the two operations (clone comparison and set construction), with increasingly functionality and control complexity, and exploring different trade-offs between computation and communication costs.

SW/HW 1: Demand Sending and Clone Comparison
This solution implements in HW only the row comparison operation. There is no use of memory banks and the communication is stream-based. The SW determines which pairs of rows must be compared and for each pair, sends a package containing the two rows to be compared. The rows are processed in HW in a pipelined way: while some blocks are unpacked, others are compared.

SW/HW 2: Complete Sending and Demand Clone Comparison
The second solution also implements in HW only the row comparison operation, but it uses a memory bank to store the entire matrix M. The SW initially sends the entire M to the HW. Then the SW determines which pairs of rows must be compared and for each pair, sends a package containing only the two row indexes (rather than the two entire rows). The packages are smaller and the replicated sending of the same row is eliminated (when it is compared to several distinct rows). Therefore the communication overhead can decrease.

SW/HW 3: Complete Sending, Demand Clone Comparison, and Set Construction
This solution is similar to the previous one, but it includes the column set construction operation in HW. Since more memory banks are used, the control becomes more complex accordingly.

SW/HW 4: Complete Clone Comparison, and Set Construction
The fourth solution implements in HW both the row comparison and the column set construction operations. The SW initially sends the entire M to the HW, which performs all row comparisons (each row is compared to every row). The SW does not need to send a comparison request for each pair of rows. The goal is to reduce the communication overhead when there are many row comparisons. The set construction control is not changed with respect to the previous implementation.

SW/HW 5: Parallel Clone Comparison and Parallel Set Construction
The last solution also implements in HW both operations and introduces the exploitation of parallelism in their execution. There are in the HW two modules for row comparison and another two for set construction. Additional memory banks are used and matrix M is stored in an interleaved way in two banks. This way, both row comparison modules (as well as the set constructors) can work in parallel and provide twice the performance for each operation.

5 Experimental Results

The platform used to implement our solutions consists of a computer with a general purpose processor Athlon 64 with 2.2 GHz clock frequency, 2 GB of RAM, and a 100 Mb/s network connection, and a multimedia board from Xilinx [12] containing a Virtex-II XCV2000 FPGA operating at 50 MHz clock frequency and five memory banks of 2 MB each one. The pure SW implementation, as well as the SW parts of the hybrid solutions were developed in the C programming language. The HW parts of the hybrid solutions were developed using the VHDL hardware description language and synthesized with the Xilinx/ISE tool.

We used chromosomes from two living beings to generate the clones and probes, which in turn produced the binary matrices used in the experiments. The first was the chromosome 5 of *Arabidopsis thaliana* (a plant of the mustard family), and the second was a chromosome 2 contig of *Homo sapiens*, both obtained from NCBI [13]. For the first chromosome, 3,285 clones and 4,096 probes were generated, and for the second chromosome, 2,881 clones and 4,096 probes.

For each input matrix, the pure SW algorithm and the five hybrid solutions described in Subsection 4.3 were executed. Table 1 shows the results for the various solutions, applied to the matrix formed with the *Arabidopsis thaliana* data. For each operation implemented in HW (row comparison and set construction), we show the time spent only on FPGA processing and the total time spent performing the operation. The latter includes communication, FPGA processing, and SW processing times. The number of row comparisons performed is also shown. Finally, the table shows the total execution time of the solution and the speedup of each hybrid implementation with respect to the SW algorithm.

The results show that, for the *Arabidopsis thaliana* data, a few comparisons between rows are necessary, indicating that the matrix has few components. Therefore, the comparison of all row pairs (performed by implementations SW/HW 4 and 5) offers no benefits because it makes a far greater number of comparisons than are actually necessary. For this operation, the best solutions are implementations SW/HW 2 and 3.

Considering the set construction operation, the first two hybrid implementations perform the set construction in SW, while implementations SW/HW 3 and 4 use one single constructor module in HW, and reduce in more than 16 times

Table 1. Results for *Arabidopsis thaliana* chromosome (execution times in seconds)

Implementation	Row Comparison			Set Construction		Total time	Speedup wrt. SW
	Number of comparisons	Time in FPGA	Total time	Time in FPGA	Total time		
SW	7,879	–	0.70	–	571.45	573.11	–
SW/HW 1	7,879	0.04	1.30	–	589.34	591.45	0.97
SW/HW 2	7,879	0.04	0.67	–	572.59	574.11	1.00
SW/HW 3	7,879	0.04	0.63	34.45	35.17	36.67	15.63
SW/HW 4	5,393,970	27.72	29.95	34.45	35.16	65.89	8.70
SW/HW 5	5,393,970	13.92	16.14	17.22	17.94	34.86	16.44

Table 2. Results for *Homo sapiens* chromosome (execution times in seconds)

Implementation	Row Comparison			Set Construction		Total time	Speedup wrt. SW
	Number of comparisons	Time in FPGA	Total time	Time in FPGA	Total time		
SW	2,080,334	–	214.83	–	272.42	487.73	–
SW/HW 1	2,080,334	10.65	351.53	–	280.55	633.56	0.77
SW/HW 2	2,080,334	11.07	137.08	–	248.28	386.47	1.26
SW/HW 3	2,080,334	11.15	124.12	15.11	15.61	145.53	3.35
SW/HW 4	4,148,640	21.32	23.06	15.11	15.59	39.02	12.50
SW/HW 5	4,148,640	10.70	12.44	7.55	8.03	20.84	23.40

the total time spent on this operation. In the last implementation, the introduction of a new set constructor module provided a performance gain around 100% when compared to the previous solution.

We can see from the SW solution results that, for this matrix, most of the total execution time is spent on the set construction operation. We can develop a SW/HW implementation 6, combining *Complete Sending and Demand Clone Comparison* (as SW/HW 3) *and Parallel Set Construction* (as SW/HW 5). This solution produces the best results for the *Arabidopsis thaliana* matrix, with 19.35 seconds of total execution time and yielding a speedup of 29.62 with respect to the SW algorithm.

Table 2 presents the results obtained with the second matrix, containing the *Homo sapiens* data. We can see that, for this matrix, it is necessary to compare many more row pairs, and consequently the time spent on the row comparison operation represents a considerable portion of the SW algorithm total execution time. Thus, the implementation of this operation in HW has enabled all hybrid implementations (except the first one) a substantial performance gain when compared to the SW solution. The approach used in solution SW/HW 1 sends to the FPGA a message with the pair of rows, for each row comparison, and produces a negative impact on the communication time. Solutions SW/HW 2 and 3 send all M rows once and, for each comparison, send only the row indexes, reducing the communication time and yielding an average performance gain of 65%, for this operation, with respect to the SW algorithm. Implementation SW/HW 4, which makes all comparisons and does not send row indexes, reduces even further the communication time and produces a performance gain of 832% in the time spent on this operation, compared to the SW algorithm. The last hybrid solution nearly doubles this gain, using two row comparison modules. Therefore, we can conclude that if the number of comparisons is significant, it is better to perform all comparisons than to send messages with comparison requests.

The results from Table 2 also show significant performance gains for implementations that perform in FPGA the set construction operation. The use of parallel row comparison and set construction (solution SW/HW 5) produced the hybrid implementation with the best results, for this matrix, yielding a speedup of 23.4 when compared to the SW algorithm. Both operations benefit from parallelism, because their data can be partitioned and processed independently.

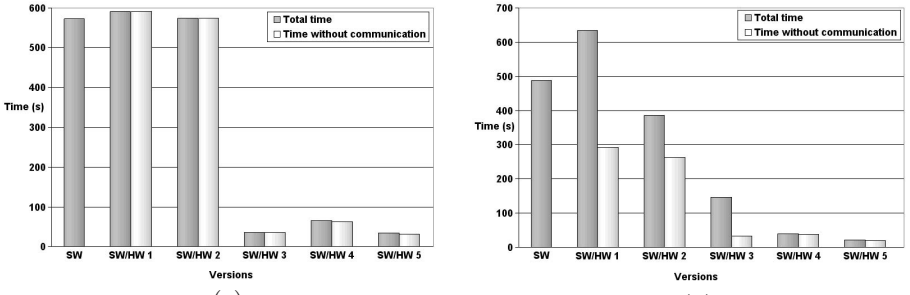


Fig. 4. Execution time for (a) *Arabidopsis thaliana* and (b) *Homo sapiens* chromosomes

Fig. 4 shows the total execution time for all implementations and for both chromosomes. In order to analyze the impact of communication overhead in each hybrid solution, the figure also shows the execution time without the communication overhead, but including both FPGA and SW processing. We can see that, for the *Homo sapiens* chromosome, some hybrid solutions can benefit from a platform with a stronger coupling between processor and FPGA, and consequently lower communication overhead.

6 Conclusions and Future Research

This paper presented several SW/HW solutions to the consecutive ones problem, used in DNA physical mapping. We developed these solutions on a platform with a processor and a reconfigurable component attached to it, and also implemented the pure SW algorithm. The hybrid solutions perform in hardware operations that most contribute to the SW algorithm total execution time. We performed experiments with real chromosomes in order to compare the performance of these solutions to the SW implementation, and analyze different trade-offs between communication overhead and computation.

The results showed the reconfigurable device capability in efficiently execute the operations, clone comparison and set construction, and both provided performance gains. These operations perform repetitive tasks that require a continuous data flow and present a considerable amount of data parallelism, features that are benefited from the implementation in hardware.

We achieved drastic performance improvements using approaches with reduced communication and with pipeline and parallelism. The hybrid implementations yielded speedups up to 29.62, with respect to the SW algorithm, for the *Arabidopsis thaliana* chromosome. For the *Homo sapiens* chromosome, we produced speedups up to 23.4.

We can further improve these results using an architecture with stronger coupling and a reconfigurable component without the strict limitations of low clock frequency and small memory capacity. This way, communication time can be reduced and more parallelism can be exploited. The solutions developed employ

at most two hardware modules, for each operation, working in parallel, due to memory limitations. However, the application offers more parallelism potential and there was still plenty of area available in the FPGA.

From the experimental evaluation we conclude that the choice of the best SW/HW solution may be different for each input matrix with clones and probes data. A challenging approach is to identify which characteristics of this matrix influence it to have a few or many components and affect the demand of a few or many clone comparisons. Based on this investigation, we can dynamically reconfigure the FPGA to implement the best solution to that matrix.

References

1. Bondalapati, K., Prasanna, V.K.: Reconfigurable Computing Systems. Proceedings of the IEEE 90(7), 1201–1217 (2002)
2. Setubal, J.C., Meidanis, J.: Introduction to Computational Molecular Biology. PWS Publishing Company (1997)
3. Fulkerson, D., Gross, O.: Incidence Matrices and Interval Graphs. Pacific Journal of Mathematics 15(3), 835–855 (1965)
4. Jacobi, R.P., Rincón, M.A., Carvalho, L.G., Llanos, C.H., Hartenstein, R.W.: Reconfigurable Systems for Sequence Alignment and for General Dynamic Programming. In: Proceedings of the Brazilian Workshop on Bioinformatics, pp. 25–32 (2004)
5. Oliver, T., Schmidt, B., Maskell, D.: Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs. In: Proceedings of the International Symposium on Field-Programmable Gate Arrays, pp. 229–237 (2005)
6. Regester, K., Byun, J.H., Mukherjee, A., Ravindran, A.: Implementing Bioinformatics Algorithms on Nallatech-Configurable Multi-FPGA Systems. Xcell Journal Online (53), 100–103 (2005)
7. Meng, X., Chaudhary, V.: An Adaptive Data Prefetching Scheme for Biosequence Database Search on Reconfigurable Platforms. In: Proceedings of the ACM Symposium on Applied Computing, pp. 140–141 (2007)
8. Mak, T.S.T., Lam, K.P.: Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA. In: Proceedings of the IEEE Computational Systems Bioinformatics Conference, pp. 512–514 (2004)
9. Krishnamurthy, P., Buhler, J., Chamberlain, R.D., Franklin, M.A., Gyang, K., Jacob, A., Lancaster, J.: Biosequence Similarity Search on the Mercury System. In: Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors, pp. 365–375 (2004)
10. Hsu, W.L.: A Simple Test for the Consecutive Ones Property. In: Ibaraki, T., Iwama, K., Yamashita, M., Inagaki, Y., Nishizeki, T. (eds.) ISAAC 1992. LNCS, vol. 650, pp. 459–468. Springer, Heidelberg (1992)
11. Booth, K.S., Lueker, G.S.: Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-tree Algorithms. Journal of Computer and System Sciences 13(3), 335–379 (1976)
12. Xilinx: Xilinx Multimedia Board (2005), `http://www.xilinx.com`
13. NCBI - National Center for Biotechnology Information: NCBI (2007), `http://www.ncbi.nlm.nih.gov/`