Systolic Filter for fast DNA Similarity Search

P. Guerdoux-Jamet - D. Lavenier

Abstract

This paper presents a systolic filter for speeding up the scan of DNA databases. The filter acts as a co-processor which performs the more intensive computations occurring during the process. Validation has been done on a based FPGA prototype board tightly connected to a workstation and has shown that the filter may boost the performances of the machine by a factor ranging from 50 to 400 over current workstations.

1 Introduction

The scan of DNA databases is a fundamental task in molecular biology. This operation consists of identifying those sequences in the DNA database which contain at least one segment sufficiently similar to some segments of a query sequence.

The computational complexity of this operation is proportional to the product of the length of the query sequence and the total number of nucleic acids in the database. In general, segment pairs (one from a database sequence and one from the query sequence) may be considered similar if many nucleotides within the segment match identically.

Presently, softwares such as BLASTN [1] or FASTA [9] are extensively used to perform the scan of the DNA database. They have been designed to run on standard computers (i.e. Von Neuman machine) and include software techniques for speeding up the process. These techniques are based on heuristics which can be tuned by setting external parameters.

The search sensitivity depends mainly of these parameters. In general, a low sensibility implies a short computation time (a few minutes), while a high sensibility involves very long computation time (a few hours).

One could think that, in the future, the increasing power of the micro-processors would decrease the computation time. Unfortunately, the banks of DNA sequences grow in syze by approximatively 50 % per year and there is no reason to expect this progression to change in the next few years.

Actually, the DNA databases and the micro-processor performances grow approximately at the same speed. As an example, the graph of the figure 1 shows two growing curves: the dark line represents the size of GenBank [2] (in million of nucleotides) since 1986; the dash line measures performance (in MIPS) of the fastest available 80x86 processor at introduction [5]. The two curves follow nearly the same exponential progression. Thus, biologists will continue to have the dilemma of getting incomplete results in a short time or waiting a long time for satisfying solutions.

This paper propose a hardware approach for solving this dilemma: a systolic filter which, when plugged into the extension slot of a PC or a workstation, boosts considerably the performances of the machine.

The next section presents the filter algorithm. Section 3 details the architecture of the filter and its implementation on a FPGA based support while section 4 gives performance

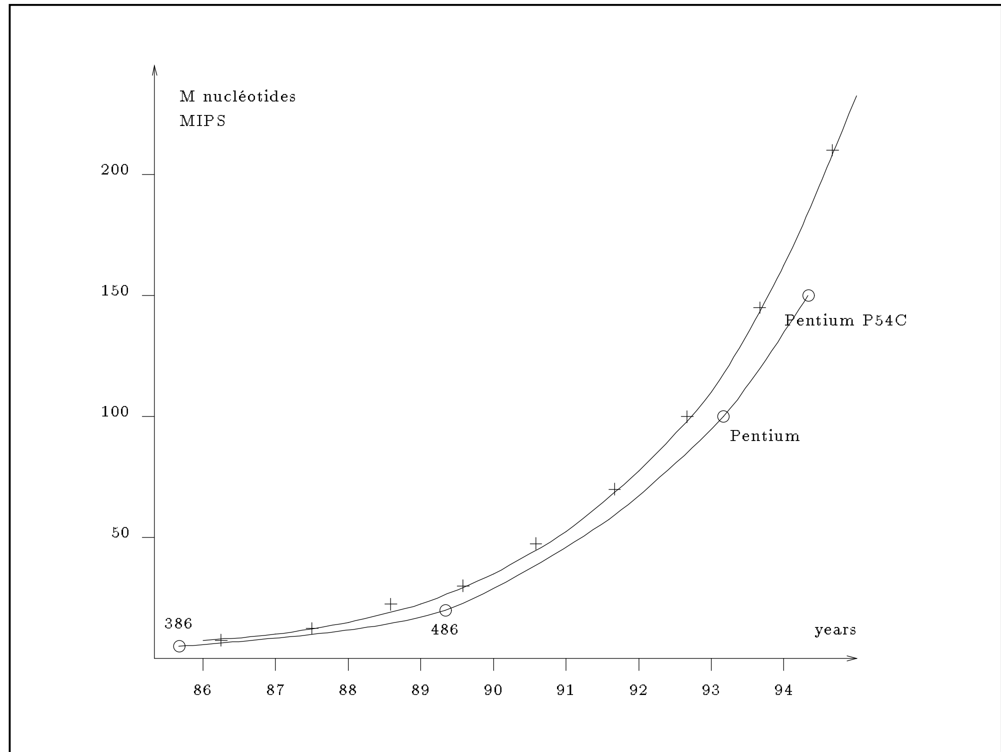

Figure 1. size of GenBank versus 80x86 power

figures. Section 5 describes the design methodology we follow before concluding on some perspectives of this work.

2 Algorithm

The problem is to locate similar segments between two DNA sequences. Given two segments $Sg0$ and $Sg1$, having both n characters, it is generally accepted that the similarity S is measured as follows:

$$S = \sum_{i=1}^{i=n} \mathbf{SUB}[Sg0(i)][Sg1(i)]$$

where $Sg(i)$ represents the ith character of the segment Sg and **sub** a substitution table which indicates the cost of replacing one character by another. In the case of DNA sequences, the alphabet consists of 4 letters, A,C, G and T and the substitution table may not be used. Only costs for matches and mismatches are considered. The similarity is then calculated as follows:

$$S = \sum_{i=1}^{i=n} \quad \mathbf{if}\ (Sg0(i) = Sg1(i))\ \mathbf{then}\ M\ \mathbf{else}\ N$$

where M is the cost of a match and N the cost a mismatch. M is a positive number while N is a negative number. Only the segments which have a cost bigger than a threshold value T are considered as similar.

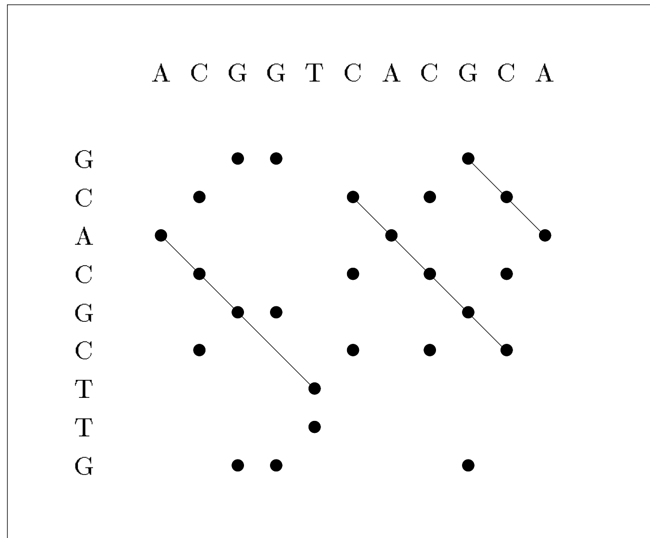

Figure 2. detection of similar segments

The detection of such similar segments between two sequences $S0$ and $S1$ implies to compute the similarity of all possible subsequences of $S0$ with all possible subsequences of $S1$. This is achieved by calculating a $n0 \times n1$ matrix Tx such as:

$$Tx[i][j] = MAX \begin{cases} Tx[i-1][j-1] + (\textbf{if } (Sg0(i) = Sg1(j)) \textbf{ then } M \textbf{ else } N) \\ 0 \end{cases}$$

with $0 < i \leq n0$ and $0 < j \leq n1$

The maximum operation keeps the value of $Tx[i][j]$ positive or null; the effect of this operation is to promote positive scores vis-a-vis negative scores: a negative score indicates no similarity and is not an interesting information. The scores which are memorized in $Tx[i][j]$ and which are greater than the threshold value represent the areas at S0(i) and S1(j) where similarities occur. Two similar segments of length k are detected between a local maximum score $S(i,j)$ located at $Tx[i][j]$ and the first zero score located at $Tx[i-k][j-k]$.

Figure 2 gives a graphic representation of the computation done on the matrix Tx; dark lines represent the pieces of diagonals where the score has overstepped the threshold value. Generally, a first step consists in storing only the diagonal numbers and the maximum score which has been calculated on these diagonals. The algorithm used is the following :

```
for (i=1; i<=n0; i++)
  for (j=1; j<=n1; j++)
    {
      D[n0-i+j] = D[n0-i+j] + (if (S0[i]==S1[j]) ? M : N );
      if (D[n0-i+j]<0)
        then D[n0-i+j]=0;
        else Smax[n0-i+j]=max(Smax[n0-i+j],D[n0-i+j]);
    }
```

The `Smax` table memorize the maximum score of the $n0 + n1 - 1$ diagonals. Only the diagonals which have a score greater than the threshold value will be then considered to precisely locate similar segments.

This is the basis of the algorithm used by software such as BLASTN or FASTA for scanning the database and detecting similar segments. For speeding up the process they do not performe the computation on simple characters (which correspond to the nucleotides) but on words of several characters. Since the complexity of the algorithm is mainly determined by the length of the two sequences, a 8 to 1 compression, for example, will reduce drastically the computation time; the drawback is that the sensibility will be equally reduced.

The filter we propose performs a character-by-character comparison. The idea is to hardwire a simpler version of the previous algorithm: instead of computing the maximum score of a diagonal, we just detect that the score has overstepped a threshold value T. The algorithm becomes :

```
for (i=1; i<=n0; i++)
  for (j=1; j<=n1; j++)
    {
      D[n0-i+j] = D[n0-i+j] + (if (S0[i]==S1[j]) ? M : N );
      if (D[n0-i+j]<0)
        then D[n0-i+j]=0;
        else R[n0-i+j] = R[n0-i+j] || (D[n0-i+j] >= T);
    }
```

From the boolean table `R` one can easily detect the interesting diagonals, compute the exact score and locate precisely similar segments.

3 Architecture and implementation

A simple dependence analysis of the above nested loop shows that the computation of each diagonal can be performed in parallel and supported on the systolic linear network depicted on the figure 3. Each processor is responsible for the computation of one diagonal score.

The computation done by a processor corresponds to the body of the nested loop:

```
initialization :
      int  d = 0;
      bool r = false;

systolic cycle :
      d = (c0==c1) ? (d + M) : (d + N);
      d = max (d,0);
      r = r || (d>=T)
```

The data (c0,c1) are 2-bit encoded and advance through the register network every systolic cycle. The architecture of a processor is mainly based on an adder which adds a positive or a negative value according to the equality or the inequality of the input data. When the output of the adder oversteps the threshold value T, the flag `r` is set. When the output becomes negative, the adder is reset to zero.

The hardware platform we use for validating the filter is the PRL-DEC Perle-1 board [3] developed by the DEC Paris Research Lab. It is based on the PAM (Programmable Active Memory) [4] concept: like a RAM memory module, a PAM is attached to the system bus of a host computer. The processor can write into, and read from the PAM. Being an active hardware co-processor however, the PAM processes data between write and read instructions. The specific processing is determined by the content of its *configuration memory*.

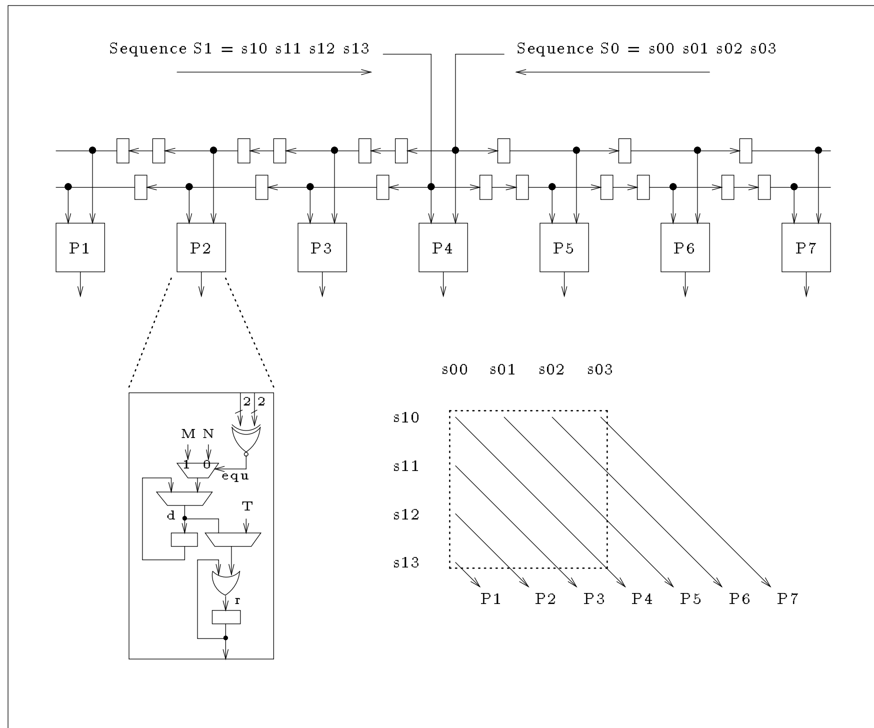

Figure 3. architecture principle of the filter

The Perle-1 board is built around a large array of bit-level configurable logic cells surrounded by local RAM banks. The central computational array consists of a 4×4 matrix of Xilinx XC3090 programmable gate arrays [12]. The host bus interface is a TurboChannel interface delivering a 100 MBytes/s bandwith.

The complexity of a processor depends on the M, N and T parameters. The higher are these values, the higher is the quantity of hardware resources required, and the smaller is the number of processors which can be fitted into a single FPGA component. We choose arbitrarily to make the design with the lowest possible parameters, that is: $M = 1, N = -1$ and $12 \leq T \leq 15$. In that case, the architecture of a processor can be simplified: the adder is replaced by a counter and the threshold detector by a 4-input AND gate connected to the adder output.

16 such processors have been implemented into a Xilinx 3090, leading to a total of 256 processors in the Perle-1 board. Actually, we implement only a half-array of the array shown in the figure 3. The reason is that, in most cases, the size of the sequences are longer than the size of the array and partitioning is needed. A half-array is better suited for partitioning than a complete array.

The synopsis of the design implemented on Perle-1 is represented by the figure 4. The data coming from the host workstation – via a FIFO – are compressed in order to reduce the host/array bandwith. The DCP module will uncompress the data.

The result data sent to the workstation are the numbers of the processors for which an overstep has occurred. The ADR module performs this task. The data are pushed sequentially to the OUT FIFO as soon as the processors have finished their computation.

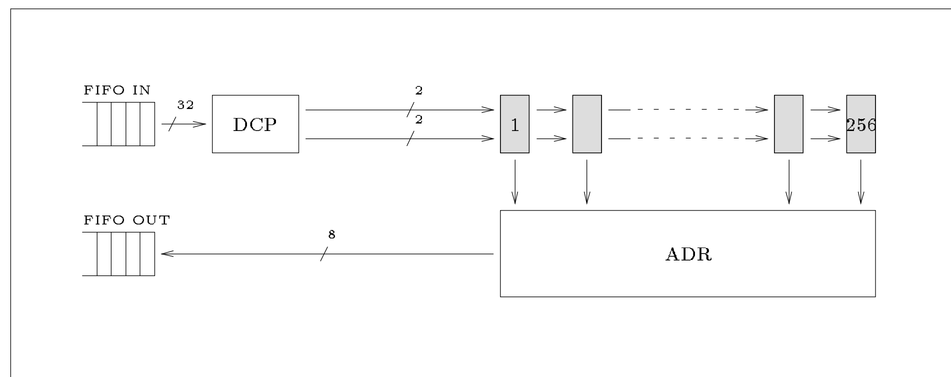

Figure 4. synopsys of the systolic filter implemented on Perle-1

4 Performance

To measure performances, we developed a prototype software, called HScan, in which the filter has been integrated and acts as a co-processor. This software produces all the segment pairs, with a score greater than a threshold value, between a DNA query sequence and a DNA database. HScan works as follows:

```
read (QS);                                          /* read query sequence */
NbSeqDB = open (DB);                                     /* open data base */
for (i=1;i<=NbSeqDB;i++)
 {
   Read (DBS);                          /* read the ith database sequence */
   NbDiag = filter(QS,DBS,M,N,T,NumDiag);
   for (j=0;j<NbDiag;j++)
    {
     ComputeSSP(QS,DBS,NumDiag[j],M,N);
    }
 }
```

The procedure `filter()` takes as parameters two DNA sequences (`QS` and `DBS`), the match and mismatch costs (`M` and `N`) and the threshold value (`T`); it produces a list of diagonals (`NumDiag`) whose the score has overstepped the threshold value. The procedure `ComputeSSP()` computes the real score and locates the similar segment pairs.

The execution time has been compared with the execution time of BLASTN (version 1.4) and FASTA (release 17) running on a SPARC 10 workstation. The parameters of these softwares have been set to produce the same results (i.e. $M = 1$ and $N = -1$) and the threshold value has been set to the value computed by BLAST.

More precisely, the measures have been done using the UNIX command `time`. Thus, this is the total elapsed time, as it can be feel by the user. BLASTN has been called with the following parameters:

```
time blastn <database> <query> M=1 N=-1 W=1 top
```

It specifies that the cost of a match is 1 (`M=1`), the cost of a mismatch is -1 (`N=-1`), the size of a word is one (`W=1`), i.e. the comparison is done character by character, and that the search is restricted to the positive strand of the sequences (`top` parameter). Setting the parameter `W` to 1 ensures a more sensitive search and, consequently, is more time consuming. The result is a list of significant similar segment pairs.

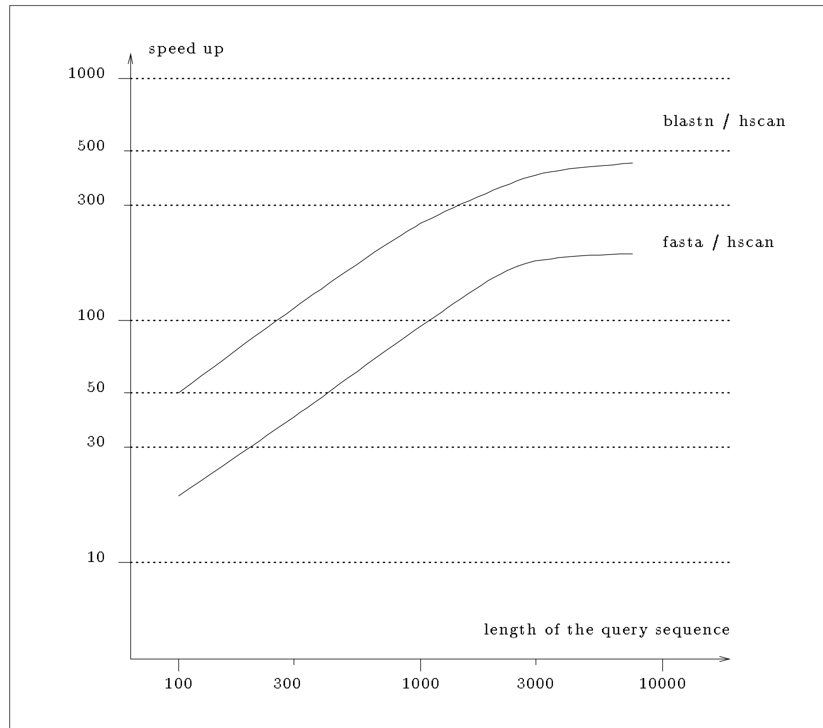

Figure 5. speed up between HScan which integrated the systolic filter and the softwares BLASTN and FASTA running on a SPARC-10 workstation

FASTA has been called with the following parameters:

```
time fasta -s <matrix> -b 50 -d 0 <query> <database> 1
```

It specifies that the cost of a match and the cost of a mismatch are taken from a substitution table (`-s <matrix>`), that the result consists in the 50 "best" sequences which contains the "best" similar segment pairs (`-b 50`) and that the size of a word is one (last parameter = 1). As FASTA cannot select only similar segment pairs which have a score greater than a threshold value, we ask for the 50 "best" sequences to be outputed.

The diagram of the figure 5 shows the speed-up compare to BLASTN and FASTA for different lengths of the query sequence. It must be point out that the longer the query sequence, the better is the speed-up. This is particularly interesting since the size of DNA sequences becomes larger due to advanced techniques in molecular cloning and DNA sequencing techniques.

We measure also the speed-up obtained with different sizes of the filter and for different lengths of the query sequence as shown on the figure 6. We took as a reference the execution time of HScan with a 32-cell filter for 5 query sequences of length 20, 100, 300, 1000 and 8000. Then we determined the execution times of HScan with a 64, 128, 192 and 256 cell filter.

For short query sequences a filter larger than 128 cells does not provide important speed-up. On the other hand, for long query sequences, a larger filter will provide better performance. This last point must also be considered for future applications where comparisons of complete genomes are envisaged. In that case, the size of the query sequence represents several tens or hundreds of millions of nucleotides (the human genome has about 3.3 billions nucleotides).

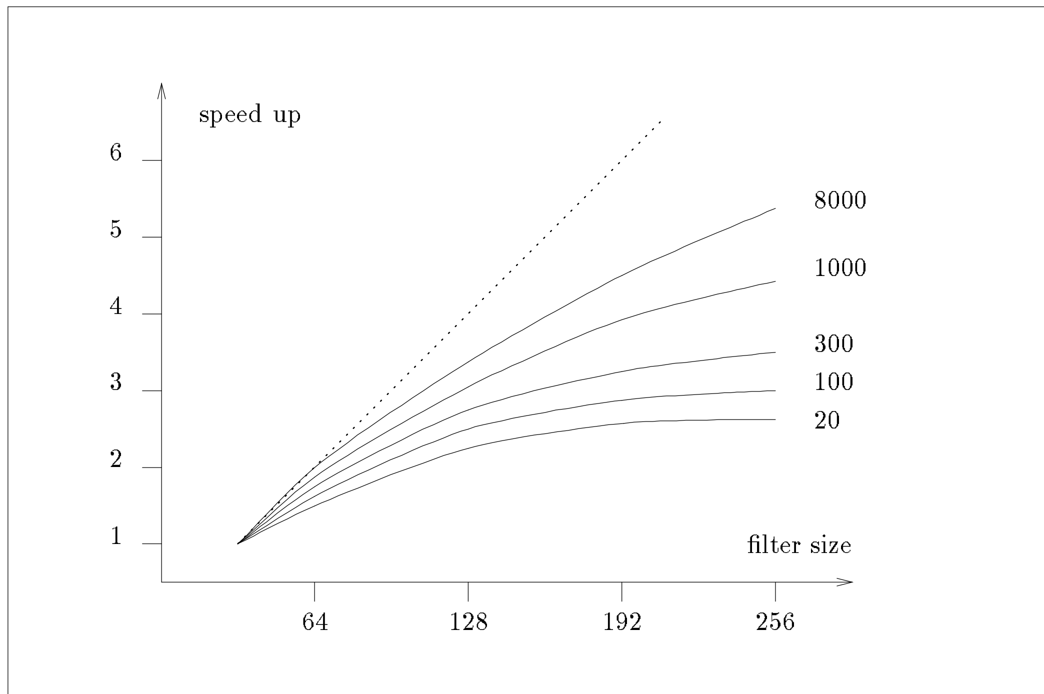

Figure 6. speed-up obtained with different sizes of the filter. The reference is the execution time of HScan with a 32 cell filter

5 Design methodology

If the systolic filter provides an attractive solution for the execution-time / sensibility tradeoff, it appears also as a rich investigation support for experimenting the design implementation of FPGA prototype. The approach we test uses a software environment based on a pivot language, the C-stolic language, [10] which is a tool we developped to parallelize systolic algorithms. The design steps of the filter can be sumarized as follows:

1. profiling of existing software to locate the time consumming part;
2. substituing this part by a C-stolic procedure which parallelizes the computation described in section 2;
3. validating this new version by intensive simulations on real data set;
4. generating the hardware architecture for the FPGA Perle-1 board from the C-stolic description.
5. measuring the performance: if improvements are possiblem goto 2.

Step 1

Step 1 show up that more than 98 % of the execution time was spent in the detection of similar segments when maximal sensitivity is asked. As mentionned earlier, it consists in a nested loop whose the computation complexity is directly proportionnal to the sequence length. This analysis does not required specific tools and has been done using available profilers.

Step 2

This step consists in writing a C-stolic procedure which parallelizes the detection of similar segments on a systolic structure. C-stolic is a systolic programming language which simplifies the programmer's task by making explicit the data-flow of systolic algorithms, and by exposing the data delivery mechanism. The language captures the essential features of systolic machines in a simple C-like programming model; the C-stolic language principle may be rapidely understood by the two main following extensions:

data structures: the C language storage classes are extended to make the difference between scalar variables and variables located on the systolic array. Hence, a new storage class specifier, the `systolic` class, is defined to reserve variables on each processor and to control the set of operations which are allowed on these objects.

communications: News operators are introduced to match the hardware architecture and to express the tight coupling between neighboring cells. The programming model assumes a SIMD execution mode and synchronous communications. Correct use of this model implies that the emission of a value by a processor in one direction is followed by the reception of the data sent by the neighboring processor located on the opposite direction. The global effect of the these operators is a shift of the network variables, which reflect the data-flow characteristics of systolic algorithms.

As an example, the C-stolic procedure for detecting the similar segments on a linear systolic array may be expressed as follows:

```
 1: SimSegDetect(S0,S1,l0,l1,Res)
 2: char *S0;
 3: char *S1;
 4: int  Nb;
 5: int  R[NB_PROC];
 6: {
 7:   int i,j,k;
 8:   systolic int c0, c1, tc0, d, r, tmp;
 9:   r=0; d=0;
10:   for (i=0; i<(max(l0,l1)+NB_PROC); i++) {
11:     c1 => c1 : S1[i];
12:     tmp = c0;     c0 => tc0 : S0[i];     tc0 = tmp;
13:     d = (c0 == c1) ? d + MATCH : d - MISMATCH;
14:     d = (d <  0) ? 0        : d;
15:     r   = r || ( d >= THRESHOLD );
16:   }
17:   j=0;
18:   for (i=0; i<NB_PROC; i++) {
19:     r : k =< r ;
20:     if (k==1) Res[j++]=i;
21:   }
22:   Res[j]=-1;
23:  }
```

The procedure takes, as parameters, two DNA sequence (and their lengths) and computes a table (`Res`) which memorizes the number of the diagonales where similar segments has been detected. Line 8 declares six systolic variables which represente the internal processor variables.

Line 11 expresses, with a condensed syntax (operator `=>`), a left to right communication: it means that processor P_i send the character `c1` to P_{i+1} and receives (on the same chanel) another character from P_{i-1}; it indicates also that the leftmost processor of the systolic array get a new character from the sequence `S1`. The following line (line 12) simulates a two register stage connection between processors.

The last loop (line 18 to 21) collects the results and stores them into the table `Res`. Line 19 is a right to left transfer (operator `=<`) for outputing the `r` value computed on each processor. the variable `k`, which is a scalar variable, is assigned on each systolic cycle by the leftmost processor;

Step 3

Once the C-stolic procedure has been substituated to the time consumming part of the code, simulation had to be performed for testing whether the parallelization was correct or not. In the present case, we had to decide if the results produced by the systolic filter, and then refined by the host computer, were valid compared to those produced by existing software. This can be achieved only on real data (biological data bases) and requires expensive simulation ressources.

The C-stolic compiler has been designed to generate code for different parallel machines. More precisely, it works as follows:

1. from the C-stolic code, it separates the parallel operations (systolic computation) from the scalar ones (I/O management);
2. a C program is generated for each process with communication and synchronization operations;
3. a compilation is done separately on each C program with the native compiler of the taget parallel machine.

Thus, the simulation can be performed on parallel machines leading both to a short execution time and the validation of the algorithm. Presently, code for the MasPar [8], the iPSC/2 [6], the iWarp [11] [7] and the ArMen machine may be generated.

Step 4

The design of the filter architecture has been guided by the code produced by C-stolic compiler since it separates clearly the computation done on each systolic cycle and the computation performed outside the network, i.e. the way the data had to be send to or receive from the network. These two files have been used as a starting point for generating the complete architecture of the filter.

The transcription into hardware has been done manually since no tools have been developped to fill this gap. The C-stolic description facilitates this task but does not provide a zero-fault design methodology. In spite of using the programming tools provided with the Perle-1 board, this task can be assimilated to the writing of assembly code: the test may be very tedious and take a very long time.

Step 5

This final step has the charge of deciding if the architecture which has been designed is satisfying. It must test if maximal performances are reached and, more precisely, if a good load balancing between the host and the hardware structure is achieved.

Actually, this step may confirm the initial intuition or, on the other hand, detect unsuspected misfunctionning. As an example, the first architecture filter we have in mind was more sophisticated: we planed to design a systolic network capables of computing the maximum score of all the diagonals and to give, as result to the host station, the definitif scores.

This design was completely unbalanced: the host station was idled most of the time since the systolic filter had the charge of doing the complete computation. Further more, the cell complexity was leading to implement a small network, thus an important overhead calculation due to partitionning.

The final version is much more well balanced in the sens that the host station participates actively to the filter computation and allows to have a larger network. The overall performance is then better.

This design methodology is typically a co-design approach. Presently, it suffers of the non automatization of step 4 which implies a relatively long design cycle. Ongoing researches focuss on this aspect: they aims at producing automatically an hardware description, from C-stolic, for rapid systolic co-processor prototyping.

6 Conclusion

DNA similarity search is an computation intensive-task which may take a few hours on standard workstations when using common softwares parametrized for high sensitivity. We have proposed a systolic filter which reduces the execution time by detecting potential areas where similarities may occur.

An implementation of a 256 cell filter on a FPGA prototype board has shown that this mechanism can boost the performances of standard workstations by a factor ranging from 50 to 400 according to the length of the query sequence.

Further work includes implementation of more complex cells allowing parametrized match and mismatch costs; the current +1 and -1 values may not be suited to biological reality. As an example the BLAST default parameters are +5 and -4 and the FASTA default parameters are +4 and -3. Consequently, the cell adder and the threshold detector will be larger, and fewer cells will be fit onto a single FPGA component.

16×3090 XILINX FPGAs have been used to implement a 256 cell filter. The use of update technology (the XC4000 logic cell array family, for example) should decrease the numbers of components and provide the ability to design a small board which could easily be plugged into the extension slots of standard machines.

Another implementation approach is to fit the filter into an ASIC dedicated to the DNA similarity search. We are currently designing such a chip: the first investigations show that a 256-cell filter can be integrated on a reasonable silicon area and would provide a low cost hardware accelerator. Furthermore, the systolic nature of the filter simplifies the design of a cascadable chip for larger filters.

Beyond the work realized for this specific application, we propose a design methodology for automating the implementation of systolic arrays: the algorithm is first parallelized using the C-stolic language, it can be then executed on parallel machines for intensive simulations, and finally synthetized for FPGA hardware platform. The design loop needs to be improve, especially for shortening the synthesis step which is still too long because of the human intervention. Our current efforts address mainly this aspect of the work.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Biol. Mol*, (215):403–410, 1990.

[2] D. Benson, D. J. Lipman, and J. Ostell. Genbank. *Nucl. Acids Res.*, 21(13):2963–2965, 1993.

[3] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McWhirter J. McCanny and E. Swartzlander, editors, *Systolic Array Processors*, pages 301–309, Pretice Hall, 1989.

[4] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories : a performance assessment. In F. Meyer auf der Heide, B. Monien, and A.L. Rosenberg, editors, *Parallel Architectures and their efficient use*, pages 119–130, Lecture notes in Computer Science, Springer-Verlag, oct 1992.

[5] R. Halfhill. 80x86 wars. *Byte*, 74–88, June 1994.

[6] *iPSC/2 and iPSC/860 Manual Set.* Intel Scientific Computers, 1990.

[7] Annaratone M., Arnould E., Gross T., Kung H. T., Lam M., Menzilcioglu O., and Webb J. A. The warp computer: architecture, implementation, and performance. *IEEE Transactions on Computer*, C-36(12):1523–1538, dec 1987.

[8] J. R. Nickolls. The Design of the MasPar MP-1 : A Cost Effective Massively Parallel Computer. In *COMPCON*, IEEE, feb 1990.

[9] W. R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85:3244–3248, 1988.

[10] F. Raimbault and D. Lavenier. Relacs for Systolic Programming. In *ASAP 93*, pages 132–135, IEEE Computer Society Press, Venice, Italy, Oct 1993.

[11] Borkar S., Cohn R., Cox G., Gleason S., Gross T., Kung H.T., Lam M., Moore B., Peterson C., Pieper J., Rankind L., Tseng P. S., Sutton J., Urbanski J., and Webb J. Iwarp: an integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, IEEE Computer Society and ACM SIGARCH, nov 1988.

[12] Xilinx. *The Programmable Gate Array Data Book.* 2100 Logic Drive, San Jose, CA 95124 USA, 1992.