Ordered Index Seed Algorithm for Intensive DNA Sequence Comparison

Dominique Lavenier

Ordered Index Seed Algorithm for Intensive DNA Sequence Comparison

Dominique Lavenier
IRISA / CNRS
Campus de Beaulieu
35042 Rennes, France
lavenier@irisa.fr

Abstract

This paper presents a seed-based algorithm for intensive DNA sequence comparison. The novelty comes from the way seeds are used to efficiently generate small ungapped alignments – or HSPs (High Scoring Pairs) – in the first stage of the search. W-nt words are first indexed and all the 4^W possible seeds are enumerated following a strict order ensuring fast generation of unique HSPs. A prototype – written in C – has been realized and tested on large DNA banks. Speed-up compared to BLASTN range from 5 to 28 with comparable sensitivity.

1 Introduction

Comparing DNA sequences is one of the basic tasks in computational biology, either for mining genomics database, or for filtering mass of data involved in the first steps of complex bioinformatics workflows. With the continuous increasing of genomic data, and the relative stagnation of microprocessor frequencies, faster algorithms for manipulating huge volume of data, and especially DNA banks, need to be investigated.

The first algorithm for comparing genomic sequences is due to Needleman and Wunsch in 1970 [1]. It is based on dynamic programming techniques and allows the user to globally align two sequences. In 1981, Smith and Waterman proposed a modified version for computing local alignments between two sequences [2]. The following year, this famous algorithm has been improved by Gotoh to include affine gap costs for better representing biological reality [3]. This family of algorithms is optimal: they provide the best alignments.

However, their quadratic complexity constitutes a serious barrier when dealing with large volume of data, due to the high computational power involved by dynamic programming techniques. In the late 80's, a powerful heuristic based on *seeds* as been introduced and rapidly adopted by the scientific community through the FASTA and BLAST software [4] [5].

Basically, this heuristic assumes that the two sequences (or part of the sequences) from which an alignment has been found share a common word of W characters. Thus, instead of scanning the whole search space (as it is done with dynamic programming), the search can only focus on these specific anchoring points. The consequence is a drastic reduction of the computation time, allowing biologists to run these algorithms on large DNA or protein banks. On another hand, this heuristic doesn't guaranty to detect the best alignments. Practically, it reports most of the alignments and satisfies common bioinformatics needs. In addition, the heuristic can be tuned by modifying the length of the seed according to a specified sensitivity.

To improve sensitivity, current research aims at selecting the best seeds: instead of considering a seed as a word of W contiguous characters, a word of W no necessarily consecutive characters may be considered. These seeds, referred as *spaced-seeds*, significantly increase the sensitivity. They have been implemented in software such as PatternHunter [8] or, in a more elaborate form, by Yass [11]. The next step is not only to consider one seed, by a set of several seeds which can be of different length and possibly made of different spaced-seeds (*multiple seeds*) [13]. Another seed family (*subset-seeds*) group different characters in the same set [12], still providing better expressiveness.

On this domain, the research activity is dynamic and contributes to significantly increase the sensitivity and, consequently, to narrow the gap with dynamic programming approach while offering short computation time.

This paper introduces a new way of manipulating seeds, not foccusing on a better sensitivity, but targeting a faster execution time. We address the problem of intensive DNA sequence comparison, such as the comparison of two DNA banks, or comparison of full genomes. Scans of DNA banks are not considered since, in this situation, the computation time is mostly determined by I/O capabilities of the system (fast access to mass storage).

Basically, the algorithm follows the traditional scheme of seed-based algorithms: hit detection, ungapped extension and gapped extension. The difference, compared to other approaches, is that these operations are globally and sequentially performed thanks to seed indexing and seed ordering techniques.

More precisely, seeds of the two banks are first indexed, allowing an immediate anchoring of there positions. From these seed positions, and following a seed criteria ordering (lowest to highest), an ungapped extension is started. During this process, if a seed smaller than the starting one is encountered, the process immediately stops. It means that this ungapped alignment has already been detected and doesn't need to be processed again.

Compared to usual seed-based algorithms, this approach provides shorter execution times. Reasons are:

- It extensively uses cache memory of processors: all the portions of sequence having the same seed are implicitly and simultaneously moved into the cache memory to perform fast pairwise comparisons (computation of ungapped alignments – or HSPs);
- Index ordered seed technique delivers unique HSPs avoiding complex data structure to be manipulated and saving time of search functions.

Compared to NCBI BLASTN, speed-up from 5 to 28 are obtained with comparable sensitivity. These results have been measured on a standard 3Ghz Cpre 2 Duo Intel processor with 2 Gbytes of memory when processing DNA banks ranging from 5 to 130 Mbp.

The rest of the paper is organized as follows: Section 2 describes the ordered index seed algorithm. Section 3 presents results from a C prototype software, named SCORIS-N. Section 4 concludes and gives perspectives of this work.

ORIS Algorithm

```
0   begin algorithm
1                                                  – step 1
2     index1 = index(bank1,W)
3     index2 = index(bank2,W)
4                                                  – step 2
5     for all 4^W possible seed s
6       for each seed s=s1 of index1
7         for each seed s=s2 of index2
8           hsp = extend(s1,s2)
9           if score(hsp) > S1
10            store hsp in T_HSP
11                                                 – step 3
12    T_ALIGN = ∅
13    for each hsp of T_HSP
14      if hsp ∉ T_ALIGN
15        align = extend_gap(hsp)
16        if score(align) > S2
17          store align in T_ALIGN
18                                                 – step 4
19    sort T_ALIGN
20    for each align of T_ALIGN
21      display(align)
21
21  end algorithm
```

Figure 1. Structure of the ORdered Index Seed (ORIS) algorithm. It proceeds in four separate steps: (1) indexing, (2) ungapped extension, (3) gapped extension, and (4) display of sorted alignments

2 Algorithm principles

The Ordered Index Seed algorithm (ORIS) proceeds in 4 successive steps as shown figure in 1. Step 1 indexes the two banks, step 2 performs hit extensions, step 3 computes gap alignments and step 4 displays alignments. Each step of the ORIS algorithm is now detailed in the following sections.

2.1 Step 1: Bank indexing

Bank indexing is directly performed from FASTA format input files. DNA sequences are stored together with an indexing structure to rapidly locate W character seeds. Figure 2 represents this structure.

A dictionary of 4^W entries stores the position of the first occurrence of each seed in both an integer array (INDEX) and a char array (SEQ). The SEQ array is filled with the bank of DNA sequences. The INDEX array is a structure linking the positions of identical seeds.

A seed S of W characters is simply encoded into an integer as follows:

$$codeSEED(S) = \sum_{i=0}^{i<W} 4^i \times codeNT(S_i)$$

where S_i is the ith character of the seed S. The function $codeNT$ provides the following 2-bit nucleotide code:

A	C	G	T
00	01	11	10

This encoding allows the seed to be ordered in a non ambiguous way. SA will be lower than SB if $codeSEED(SA) < codeSEED(SB)$.

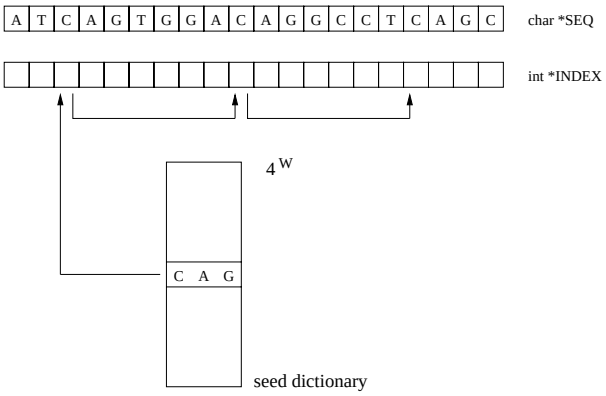

Figure 2. structure of the bank index

To eliminate non interesting alignments made of small repeats, a low complexity filter can be activated before indexing. In that case, W character words belonging to low-complexity regions are discarded from the index.

2.2 Step 2: Hit extensions

The goal of this step is to find all the HSPs (ungapped alignments) between two banks. It is done by considering all possible seeds, starting from the lowest one.

For a particular seed S, all the positions where it occurs in bank1 and in bank2 are examined. If X_1 and X_2 are respectively the number of occurrences in bank1 and bank2, then there are $X_1 \times X_2$ hit extensions to compute.

As we globally compute all the ungapped alignments before extending them with gap, we don't want multiple detections of the same ungapped alignment. As an example, consider the following ungapped alignment generated from the seed AACTGTAA:

```
... ATATGATGTGC AACTGTAA TTGCTCAGATTCTATG ...
    |||||||||| |||||||| ||||||||||||| ||
... ATATGATGTGC AACTGTAA TTGCTCAGGTTCTCTG ...
```

Latter on, when the seed AATTGCTC is considered, the same ungapped alignment will be generated:

```
... ATATGATGTGCAACTGT AATTGCTC AGATTCTATG ...
    ||||||||||||||||| |||||||| ||||||| ||
... ATATGATGTGCAACTGT AATTGCTC AGGTTCTCTG ...
```

To avoid this situation (generating and storing many times the same alignment), the extension process compares the code of each potential seed able to produce a hit with the code of the originate seed. If a seed with a lower code is encountered the process stops and no alignment is reported. The ungapped extension algorithm (left side only) is given below:

```
1   int extend_left(char *s1, char *s2, int length)
2   {
3     int maxi = SIZE_SEED;
4     int score = SIZE_SEED;
5     int L = SIZE_SEED;
6     char *ss1 = s1-1;
7     char *ss2 = s2-1;
8     int l = 0;
9
10    while ((maxi-score<XDROP)&&(l<length))
11      {
12        if (*ss1==*ss2)
13          {
14            score = score + MATCH;
15            maxi = max(score,maxi);
16            L++;
17            if ((L≥SIZE_SEED)
18              &&(codeSEED(ss1)≤codeSEED(s1)))
19                return -1
20          }
21          else {
22            score = score - MISMATCH;
23            L=0;
24          }
25        s1–; s2–; l++;
26      }
28    return maxi;
29  }
```

The `extend_left` function takes as input two char pointers (`s1`) and (`s2`) which are supposed to point to the first character of the seed on both sequences, and one integer value (`length`) specifying the maximum search space. The *while* loop computes a score and has three exiting possibilities:

1. the search space has been fully explored (`l>=length`);
2. the XDROP threshold has been reached (`maxi-score>=XDROP`);
3. the current seed code is lower than the strating seed code (line 18).

The two first conditions allow the function to return the maximun value of the score. The third condition is based on the comparison of a current seed value (`codeSD(ss1)`) with the code of the seed hit (`codeSEED(s1)`). A current seed code is valid only if a hit exists at that position: the previous `SIZE_SEED` characters must be identical on both sequences. This condition is tested thanks to the `L` variable which counts the number of identical successive characters (line 16). This variable is reset to zero when a mismatch occurs (line 23). The same type of function is applied to perform the right extension.

This simple test (comparison of the starting seed with a current potential seed) ensures that unique HSPs are generated. This is the key point of the ORIS algorithm. Without such a condition the same HSP would be produced in multiple copies, leading to add a costly procedure to suppress all the duplicates.

For large DNA banks, step 2 represents a high volume of computation. However, the way data are computed limits the execution time, mainly for two reasons:

- Even if the index structure represents hundred of mega bytes of data, the nested loops provide a very high data re-used, allowing the processor memory cache to be fully exploited. In that case, ALUs are fed with data at a very high rate, without latency due to cache misses.
- The ungapped extension procedure is fast since it is aborted as soon as the seed order condition becomes false.

Finally, step 2 stores all the HSPs overpassing a threshold value (figure 1, line 10). Actually, the storage is made by sorting the HSPs by diagonal number to optimize data access of the next step.

2.3 Step 3: Gapped alignments

From the set of HSPs computed during step 2, the goal of step 3 is to build gapped alignments. Such alignments are constructed starting from the middle of an HSP and performing an extension on both extremities by dynamic programming techniques. The extension is controlled by an XDROP value in order to stop when the score of the alignment significantly decrease. The final alignment consists in merging the right and left gapped extensions. Alignments are stored in the T_ALIGN structure and also sorted by diagonal number.

As a gapped alignment may contain several HSPs, including HSPs detected during the step 2, a test is done before starting an extension (line 14, fig 1). A gapped extension will be done only if an HSP does not belong to a gapped alignment previously computed and stored in the T_ALIGN structure. This test is fast since both HSPs and gapped alignments are sorted using the same criteria (diagonal number). To compute gapped alignments, HSPs are serially taken by increasing order of their diagonal number, leading to product alignments having the same increasing order, and nearly the same diagonal numbers. Thus, testing this condition does not involve time consuming search on the data structure due to the locality of the data (HSPs and gapped alignments).

At the end of this step, the T_ALIGN structure contains a complete set of the gapped alignments found between two DNA banks.

2.4 Step 4: Alignment display

The last step consists in producing an output file to display the results. The alignments are first sorted on the T_ALIGN structure according to a chosen criteria, for example the expected value attached to each alignment.

3 Experiments

The ORIS algorithm has been prototyped in C language for comparing two DNA banks ranging from short DNA sequences (such as EST) to full chromosomes of a few tens of Mbp. This section briefly presents the ORIS implementation, then compares performances with the NCBI BLAST software both on sensitivity and execution time.

3.1 ORIS Implementation

In the following, the ORIS implementation for comparing DNA sequences will be referred as SCORIS-N (Sequence Comparison using ORIS algorithm on Nucleotides). As we target intensive sequence comparison, the SCORIS-N program takes as input two DNA banks (FASTA format). A bank can either be a large set of DNA sequences or a single DNA sequence representing a full chromosome. The size of the bank which depends of the size of the available memory on the computer. The index structure required for storing a bank of size N (N is the number of nucleotides) is approximately equal to $5 \times N$ bytes. Comparing, for example, two chromosomes of 40 MBytes will require, at least, a free memory space of 400 MBytes.

An expected value is attached to each alignment in order to sort the results according to this criterion. The SCORIS-N program considers the size of the first bank and the size of the sequence from which the alignment is found in the second bank as parameters to compute the expected value.

As intensive sequence comparison is targeted, the output format – in the current version – does not report full the alignments. It only displays the alignment features as it is done in the `-m 8` option of BLASTN. Moreover, this output format is better suited for further automatic processing than the standard BLASTN output.

The program has been written in standard ANSI C, compiled with the gcc standard options and tested on a LINUX environment on a 3 GHz Core 2 duo Intel processor with 2 GBytes of memory (Dell Precision 390).

3.2 Data set

The following table gives the characteristics of the DNA banks we have used for testing SCORIS-N:

Bank Name	Origin	nb. seq	nb. nt (Mbp)
EST1	ESTs from GenBank	13013	6.44
EST2	ESTs from GenBank	11220	6.65
EST3	ESTs from GenBank	37483	14.64
EST4	ESTs from GenBank	34902	14.87
EST5	ESTs from GenBank	50537	25.48
EST6	ESTs from GenBank	53550	25.20
EST7	ESTs from GenBank	88452	40.08
VRL	Genbank gbvrl1	72113	65.84
BCT	misc. bacteria genomes	59	98.10
H10	Human chromosome 10	19	131.73
H19	Human chromosome 19	6	56.03

EST banks are made by randomly sampling the GenBank EST division.

3.3 Execution time

Execution times have been measured using the LINUX command `time` and by extracting the user time. To compare performance with the NCBI BLAST software (release 2.2.17, nov. 2007), blastall has been run as follows:

```
blastall -p blastn -d A -i B -o R -m 8 -e 0.001 -S 1
```

`A` and `B` are the two DNA banks. `R` is the output file storing the alignment features using the `m8` BLAST format. The expected value has been set to 10^{-3} which is a reasonable value in the context of intensive DNA sequence comparison. The `-S 1` option indicates that the search is done on a single strand only. Currently, the SCORIS-N prototype doesn't perform search on the complementary strand. The priority was first to demonstrate the efficiency of the algorithm. This option will come later, in a new release.

Figure 3 shows the execution time of BLASTN and SCORIS-N, when EST banks are compared to each other. It can be seen that SCORIS-N is much faster, and that the difference grows with the size of the banks.

The two following tables give speed-up as a function of the search space determined as the product of the size of two banks (in Mbp).

banks	search space (Mbp)	SCORIS-N exec. time (sec)	BLASTN exec. time (sec)	speed up
EST1 vs EST2	42.82	7.3	73.4	10.0
EST1 vs EST3	94.28	9.6	155.4	16.2
EST1 vs EST5	164.09	15.2	260.2	17.1
EST3 vs EST4	217.69	19.9	369.4	18.5
EST1 vs EST7	258.11	26.3	420.6	16.0
EST4 vs EST5	378.88	24.4	586.3	24.0
EST5 vs EST6	642.09	34.5	981.7	28.4
EST5 vs EST7	1021.23	54.3	1563.5	28.8
H19 vs VRL	3689	90	558	6.2
BCT vs EST7	3931	62	537	8.6
H19 vs BCT	5496	80	439	5.5
BCT vs VRL	6458	80	741	9.2
H10 vs VRL	8673	146	1266	8.6
H10 vs BCT	12922	145	965	6.6

When comparing large sequences, speed-up is less impressive, mostly because in that situation BLASTN performs well.

3.4 Sensitivity

Both SCORIS-N and BLASTN programs generate alignments using the same output format (`-m 8` option). This format provides the main characteristics of an alignment on a single text line such as its coordinates, its identity percentage, its length, its score, its expected value, etc. From this basic information, it is straightforward to compare two different output files to detect if identical alignments are generated or not.

We consider that two alignments are equivalent if they overlap of more than 80 %. Based on this metric, we define the following values:

- SC_{total} = number of alignments found by SCORIS-N
- BL_{total} = number of alignments found by BLASTN
- SC_{miss} = number of alignment missed by SCORIS-N compared to BLASTN
- BL_{miss} = number of alignment missed by BLASTN compared to SCORIS-N

These values are computed as follows: SC_{total} and BL_{total} are the number of alignments generated respectively by the programs SCORIS-N and BLASTN. If an alignment found by BLASTN is not found by SCORIS-N, then SC_{miss} is incremented. In the same way, if an alignment found by SCORIS-N is not found by BLASTN, then BL_{miss} is incremented.

We can then deduced the percentage of missed alignments according to a reference program:

$$SCORIS_{miss} = \frac{SC_{miss}}{BL_{total}} \times 100$$

$$BLAST_{miss} = \frac{BL_{miss}}{SC_{total}} \times 100$$

The two following tables report the percentage of missed alignments for various EST bank comparisons:

banks	BL_{total}	SC_{miss}	$SCORIS_{miss}$
EST1 vs EST2	34286	1137	3.31 %
EST1 vs EST3	34865	931	2.67 %
EST1 vs EST5	53426	1920	3.59 %
EST3 vs EST4	172325	4986	2.89 %
EST1 vs EST7	134170	4129	3.07 %
EST5 vs EST6	247533	9669	3.90 %
EST5 vs EST7	436635	15682	3.56 %

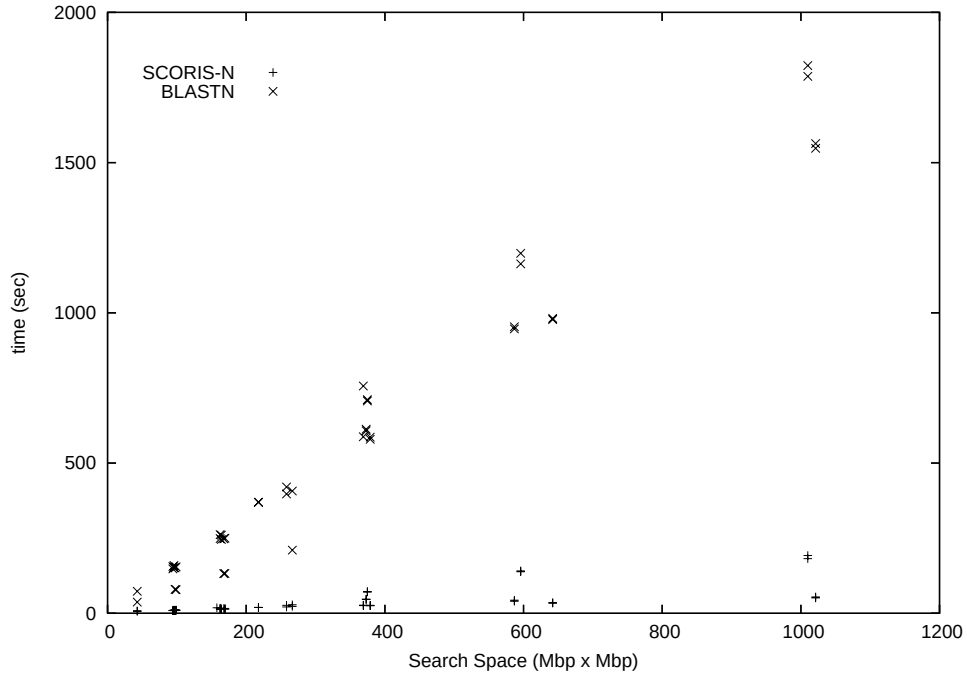

Figure 3. Execution time of SCORIS-N and BLASTN on the EST banks. The X axe is the search space determined as the the product of the size of each bank (in Mbp)

banks	SC_{total}	BL_{miss}	$BLAST_{miss}$
EST1 vs EST2	34412	952	2.76 %
EST1 vs EST3	34660	1050	3.02 %
EST1 vs EST5	52873	1625	3.07 %
EST3 vs EST4	173605	5892	3.39 %
EST1 vs EST7	138281	3816	2.74 %
EST5 vs EST6	248167	11733	4.72 %
EST5 vs EST7	438359	18124	4.13 %

The number of alignments generated by both programs are similar and missed alignments represent a small fraction of the total number of found alignments. A closer look to the missing alignments shows that they predominantly represent alignments with low score, with an expected value very close to the threshold value given by the user. There are probably slight differences in the computation of this information, leading to reject borderline alignments. Other causes of differences may come from:

- **the complexity filter:** the SCORIS-N low complexity filter presents some difference [14] with the dust filter included in BLASTN;
- **the gapped and ungapped extension** procedures have been rewritten and tuned for maximal performances. Small differences exist, especially for deciding if it is worth to continue the extension.

However, there are some alignments of interest found by BLASTN and not reported by SCORIS-N. These alignments both include a significant number of gaps near the anchoring seed and many substitution errors forbidding other 11-nt seeds to occur. In that specific cases, SCORIS-N is unable to generate valid HSPs.

To partially remedy this problem, an asymmetric indexing is done on 10-nt words. Asymmetric means that for one of the two input bank, only half words are considered. From a sensitivity point of view, this is a little bit more efficient than a 11-nt indexing. All 11-nt seeds are detected together with an average of 50% of the 10-nt seed anchoring.

The next tables compare sensitivity for large DNA sequences

banks	BL_{total}	SC_{miss}	$SCORIS_{miss}$
BCT vs EST7	2017	16	0.79 %
BCT vs VRL	1293	10	0.77 %
H10 vs VRL	490107	626	0.12 %
H19 vs VRL	530650	563	0.10 %
H10 vs BCT	0	0	-
H19 vs BCT	11	0	0.00 %

banks	SC_{total}	BL_{miss}	$BLAST_{miss}$
BCT vs EST7	1823	26	1.42 %
BCT vs VRL	1240	7	0.56 %
H10 vs VRL	503550	75	0.01 %
H19 vs VRL	543170	5	0.00 %
H10 vs BCT	0	0	-
H19 vs BCT	11	0	0.00 %

For this type of treatment, the difference between SCORIS-N and BLASTN is small. Again, the difference

mainly comes from short alignments having an expected value closed to the threshold value.

4 Conclusion and Perspectives

A new algorithm, called ORIS (ORdered Index Seed), targetting intensive DNA sequence comparison has been presented. It requires the banks to be indexed into the main memory of the computer before enumerating all the possible seeds in a well defined order. Ungapped extensions directly benefit of this scheme avoiding unecessary computations. The locality of the computation allows an intensive use of the memory cache, fastening the data access and, consequently the whole execution time.

A program, called SCORIS-N, has been prototyped in C to evaluate performances compared to the standard NCBI BLASTN program. Speed-up from 5 to 28 has been measured for comparable sensitivity. Next step is to provide a new version to minimize the difference between BLASTN and SCORIS-N. This is important since BLASTN is today the reference when comparing DNA sequences.

The SCORIS-N algorithm still need more investigation, such as:

- Comparing SCORIS-N with other programs which have also been designed for dealing with large DNA sequences and which also handle sequence indexing into main memory (BLAT [9], FLASH [6], BLASTZ [10]);
- Considering bigger treatments involving pairwise comparisons on larger sequences (full genomes) for monitoring the way the SCORIS-N algorithm behave. However, testing such computation will require systems having large memory;
- Testing SCORIS-N on genomes having a large number of repeat sequences. Generally, algorithm performances are not so good when dealing with these specific sequences.

The structure of the algorithm is also well suited for fine grained parallelism, especially step 2 and step 3. As a matter of fact, the outer loop of step 2 which considers all the possible 4^W seeds can be run in parallel since seed order prevents identical HSPs to be generated. The two inner loops can also be highly parallelized as the ungapped extensions refer to independent computations. There is then a very high potential of parallelism which can be exploited on the new generation of processors, especially on the multicore architectures.

Other possibilities is to deport step 2 and/or step 3 to dedicated hardware such as FPGA boards or GPU of new generations (Graphical Processing Units) which can now be programmed for general purpose computing (GP-GPU). Compared to the standard NCBI BLASTN program, the structure of SCORIS-N, composed of several distinct steps, makes it suitable for hardware accelerators: each step, and especially, step 2 and step 3, refers to very specific and highly parallel treatments.

Even if the first prototype of the SCORIS-N program provide significant speed-up on today processors, there are still plenty of room for further investigations on the new generations of processors. Particularly, we think that we can highly parallelize the ORIS algorithm with a very high scalability on various platforms ranging from multi-core architectures to stream processing paradigm (GP-GPU).

References

[1] S. Needleman, C. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J Mol Biol. 1970 48(3):443-53.

[2] T.F. Smith, M.S. Waterman. Identification of common molecular subsequences. J Mol Biol. 1981 Mar 25;147(1):195-7.

[3] O. Gotoh. An improved algorithm for matching biological sequences. J. Mol. Biol. 1982 162:705-708.

[4] W.R. Pearson, D.J. Lipman. Improved tools for biological sequence comparison. Proc Natl Acad Sci U S A. 1988 Apr;85(8):2444-8.

[5] S.F. Altschul, W. Gish W, W. Miller. E.W. Myers, D.J. , Lipman. Basic local alignment search tool, J Mol Biol. 1990 Oct 5;215(3):403-10.

[6] A. Califano, I. Rigoustsos, FLASH: A Fast Look-up Algorithm for String Homology, In Proc. of the 1st Int'l Conference on Intelligent Systems for Molecular Biology, Bethesda, MD, 1993.

[7] S.F. Altschul, T.L. Madden, A.A. Schffer, J. Zhang, Z. Zhang, W. Miller, D.J. Lipman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. 1997 Sep 1;25(17):3389-402.

[8] M. Li, B. Ma, D. Kisman, J. Tromp. PatternHunterII : Highly Sensitive and Fast Homology Search Bioinformatics, March 2002 18(3):440-445.

[9] W.J. Kent, BLAT: The BLAST-Like Alignment Too, Genome Research, 12 (4) 2002

[10] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, W. Miller, Human-Mouse Alignments with BLASTZ, Genome Res., Jan 2003; 13:103-107

[11] L. Noe, G. Kucherov. Yass: enhencing the sensitivity of DNA similarity search. Nucl. Acids Res, 2005, vol 33, Web Server Issue W540-W543.

[12] G. Kucherov, L. No, M. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. J. Bioinf. Comp. Biology, 4(2) 2006.

[13] J. Xu, D. Brown, M. Li, B. Ma. Optimizing multiple spaced seeds for homology search. Journal of computational biology : a journal of computational molecular cell biology, 13(7):1355-1358. 2006.

[14] A. Morgulis, E. M. gertz, A.A. Schaffer, R. Agarwala, A fast ans dymetric DUST implementation to mask low-complexity DNA sequences, Journal of Computational Biology, 13(5):1028-1040, 2006.

[15] P. Peterlongo, L. Noe, D. Lavenier, G. Georges, J. Jacques, G. Kucherov, M. Giraud. Protein similarity search with subset seeds on a dedicated reconfigurable hardware Workshop on Parallel Computational Biology, Gdansk, Poland, September 9-12, 2007