Bit-Parallel Algorithm for the Block Variant of the Merged Longest Common Subsequence Problem

Agnieszka Danek and Sebastian Deorowicz

Abstract. The problem of comparison of genomic sequences is of great importance. There are various measures of similarity of sequences. One of the most popular is the length of the longest common subsequence (LCS). We propose the first bit-parallel algorithm for the variant of the LCS problem, block merged LCS, which was recently formulated in the studies on the whole genome duplication hypothesis. Practical experiments show that our proposal is from 10 to over 100 times faster than existing algorithms.

Keywords: sequence comparison, genome, longest common subsequence.

1 Introduction

Nowadays, the amount of data in genomics is huge and grows fast. To gain from data we have to analyze them, e.g., look for some common patterns, similarities, etc. The genomes are usually stored as sequences of symbols. There are many tools for sequence comparison, e.g., sequence aligning, edit distance, longest common subsequence (LCS) [2, 9]. In the LCS problem, for two input sequences, we look for a longest possible sequence being a subsequence of both input ones. (A subsequence can be obtained from a sequence by removing zero or more symbols.) Of course, the more close the length of an LCS to the length of the input sequences, the more similar they are.

There are many variants of the LCS measure, e.g., constrained longest common subsequence [6, 15], longest mosaic common subsequence [11], longest common increasing subsequence [16]. The current paper focuses on the recently presented, merged longest common subsequence (MLCS) problem [10, 14]. Here, the inputs are three sequences: T, A, B. What is looked for is a sequence P, which is a

Agnieszka Danek · Sebastian Deorowicz
Institute of Informatics, Silesian University of Technology,
Akademicka 16, 44-100 Gliwice, Poland
e-mail: {agnieszka.danek,sebastian.deorowicz}@polsl.pl

subsequence of T and can be split into two subsequences P' and P'' such that P' is a subsequence of A and P'' is a subsequence of B.

The MLCS problem was formulated during the analysis of the *whole genome duplication* (WGD) followed by massive gene loss hypothesis. An evidence of WGD appearance in yeast species was given in [13]. Kellis et al. show that *S. cerevisiae* arose by duplication of eight ancestral chromosomes after which massive loss of genes (nearly 90%) took place. The deletions of genes were in paired regions, so at least one copy of each gene of the ancestral organism was preserved. The proof of that was based on the comparison of DNA of two yeast species, *S. cerevisiae* and *K. waltii* that descend directly from a common ancestor and diverged before WGD. These two species are related by 1:2 mapping satisfying several properties (see [13] for details), e.g., each of the two sister regions in *S. cerevisiae* contains an ordered subsequence of the genes in the corresponding region of *K. waltii* and the two sister subsequences interleaving contain almost all of *K. waltii* genes. Solving the MLCS problem for the three sequences (two regions in one species and one region in the other species) one can check whether such a situation (WGD) happened.

Dynamic programming (DP) algorithms are a typical way of solving various problems of comparison of sequences. For a number of the mentioned sequence comparison problems faster methods exist. Some of them make use of the bit-level parallelism [3, 8]. Its main idea is simple. The neighbor cells in a DP matrix (e.g., the cells within a single column) often differ by 0 or 1. A single bit rather than a whole computer word is sufficient to represent such difference. Thus, w (computer word size; $w = 64$ is typical nowadays) cells can be stored in a single computer word. The key aspects are the computations that must be done for such 'compacted' representation. E.g., Allison and Dix [1] presented the equations for the LCS problem in which a complete computer word is computed in a constant time, which means w times speedup over the classical DP algorithm. Some of other successive approaches following this way are [5–7, 12].

The paper is organized as follows. Section 2 contains the definitions and information about known solutions. In Section 3 we propose a bit-parallel algorithm. Section 4 compares the proposed and the known algorithms experimentally. The last section concludes the paper.

2 Definitions and Background

The sequences $T = t_1t_2\ldots t_r$, $A = a_1a_2\ldots a_n$, $B = b_1b_2\ldots b_m$ are over an alphabet Σ, which is a subset of integers set. W.l.o.g. we assume that $m \leq n$. The size of the alphabet is denoted by σ. $S_{i..j}$ is a component of S, i.e. $S_{i..j} = s_is_{i+1}\ldots s_j$.

For any sequence S, S' is a subsequence of S if it can be obtained from S by removing zero or more symbols. A *longest common subsequence* of two sequences is a sequence that is a subsequence of both sequences and has the largest length (note that there can be more than one such a sequence). A *merged longest common subsequence* (MLCS) of T, A, B is a longest sequence $P = p_1p_2\ldots p_z$ being a subsequence of T, such that $P' = p_{i_1}p_{i_2}\ldots p_{i_k}$, where $1 \leq i_1 < i_2 < \ldots < i_k \leq z$, is a subsequence of A and P'' obtained from P by removing symbols at positions $i_1, i_2, \ldots, i_k$ is a subsequence of B. Alternatively we can say that MLCS of T, A, B is a longest common subsequence of T and any sequence that can be obtained by merging A and B.

A *block merged longest common subsequence* (BMLCS) of T, A, B, and two block constraining sequences of indices $E^A : e^A_0 < e^A_1 < e^A_2 < \ldots < e^A_{n'}$, $E^B : e^B_0 < e^B_1 < e^B_2 < \ldots < e^B_{m'}$, where $e^A_0 = 0$, $e^A_{n'} = n$, $e^B_0 = 0$, $e^B_{m'} = m$, is a longest sequence $P = p_1 p_2 \ldots p_z$ satisfying the block constrains. This means that $r' < n' + m'$ indices $e^P_0 \leq e^P_1 \leq e^P_2 \leq \ldots \leq e^P_{r'}$, where $e^P_0 = 0$, $e^P_{r'} = z$, of the following properties can be found. For each valid q, the component $P_{e^P_q+1..e^P_{q+1}}$ is a subsequence of $A_{e^A_{u(q)}+1..e^A_{u(q)+1}}$, so-called *A-related component*, or a subsequence of $B_{e^B_{v(q)}+1..e^B_{v(q)+1}}$, so-called *B-related component*. The function u maps the components of P and A and is defined only for qs of A-related components. Similarly the function v maps the components of P and B and is defined only for qs of B-related components. Both u and v are monotonically growing functions. Alternatively, BMLCS problem is an MLCS problem in which the merging of A and B only at block boundaries is allowed. The BMLCS problem is a generalization of the MLCS problem as when the sequences E^A and E^B contain all valid indices of A and B, BMLCS reduces to MLCS. The elements of E^A and E^B are called end-of-block (EOB) markers.

Bitwise operations used in the paper are: & (bitwise and), | (bitwise or), $\sim$ (negation of each bit), ^(bitwise xor), $<<$ (shift to the left by given number of bits). The notation 0^i means i 0 bits, while 1^i means i 1 bits. The computer word size is denoted by w. For any bit vector W, $W^{[i]}$ means ith bit of W.

The BMLCS problem was introduced in [10]. The authors proposed two algorithms for it working in $O(r(mn' + nm')$ and $O(nm + nn'm')$ time. Faster solution, proposed by Peng et al. [14], solves the problem in $O(\min\{zr(n' + m'), r(n'm + m'n)\})$ time.

3 The Algorithm

Points of departure of our work are the algorithms from [7, 10], so below we give the necessary concepts of them. Huang proposed a DP algorithm computing a 3-dimensional matrix, but due to the properties of the DP recurrence only a fraction of them is computed (see Eq. (1)). Roughly speaking, only the parts related to the EOB markers are defined.

$$L(i,j,k) = \max \begin{cases} \max \begin{cases} L(i-1,j-1,k)+1, & \text{if } t_i = a_j, \\ L(i,j-1,k), & \text{if } t_i \neq a_j, \\ L(i-1,j,k-1)+1, & \text{if } t_i = b_k, \\ L(i,j,k-1), & \text{if } t_i \neq b_k, \\ L(i-1,j,k), & \text{if } t_i \neq a_j \wedge t_i \neq b_k, \end{cases} & \text{if } j \in e^A \wedge k \in e^B, \\ \max \begin{cases} L(i-1,j-1,k)+1, & \text{if } t_i = a_j, \\ L(i,j-1,k), & \text{if } t_i \neq a_j, \\ L(i-1,j,k), & \text{if } t_i \neq a_j, \end{cases} & \text{if } j \notin e^A \wedge k \in e^B, \\ \max \begin{cases} L(i-1,j,k-1)+1, & \text{if } t_i = b_k, \\ L(i,j,k-1), & \text{if } t_i \neq b_k, \\ L(i-1,j,k), & \text{if } t_i \neq b_k, \end{cases} & \text{if } j \in e^A \wedge k \notin e^B, \end{cases} \tag{1}$$

In [7] the simpler variant of the BMLCS problem, i.e., the MLCS was solved in a bit-parallel way. In this problem a similar 3-dimensional DP matrix is computed. Due to the lack of constrains on the positions of merging between input sequences the complete matrix must be calculated. As shown in [7] for such a case the differences between some cells of the matrix are no larger than 1. Therefore, a bit-vector representation of matrix L is possible. Firstly a two-dimensional matrix M of vectors of integers was defined as follows:

$$i \in M(j,k) \quad \text{iff}\; L(i,j,k) - L(i-1,j,k) = 1 \quad \text{for}\; 1 \leq i \leq r. \tag{2}$$

For each pair (j,k) this matrix stores only the indices i at which the value of the corresponding cells in matrix L is larger by 1 than its 'lower' neighbor. The integers are from range $(0,r]$. Thus, they can be represented as a vector $W(j,k)$ of r bits in such a way that $W(j,k)^{[i]} = 0$ means that $i \in M(j,k)$ and $W(j,k)^{[i]} = 1$ that $i \notin M(j,k)$. In [7] the necessary operations that can be used to mimic the operations made on L for the bit-vector-based representation W are shown.

Figure 1 shows an example of matrix L computed according to Eq. (1) and the related bit-vector representation of our algorithm.

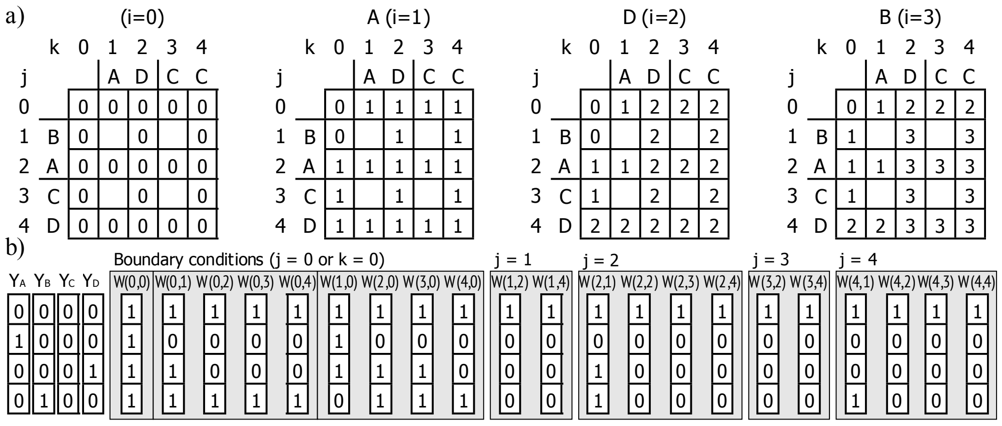
a)
(i=0)
A (i=1)
D (i=2)
B (i=3)
k 0 1 2 3 4
j
A D C C
B A C D
b)
Boundary conditions (j = 0 or k = 0)
j = 1
j = 2
j = 3
j = 4
YA YB YC YD
W(0,0) W(0,1) W(0,2) W(0,3) W(0,4) W(1,0) W(2,0) W(3,0) W(4,0)
W(1,2) W(1,4)
W(2,1) W(2,2) W(2,3) W(2,4)
W(3,2) W(3,4)
W(4,1) W(4,2) W(4,3) W(4,4)

Fig. 1 Example of computations of the BMLCS for T=ADB, A=BACD, E_A=$\{0,2,4\}$, B=ADCC, E_B=$\{0,2,4\}$ with **(a)** a classical DP algorithm **(b)** bit-vector representation W

In this work we use the same bit-vector representation of L. The key difference between the MLCS and BMLCS problems is that the DP recurrence must take care of the EOB markers. The total number of bit vectors that must be computed is $\Theta(m'n + n'm)$ (c.f. Eq. (1), where L is undefined for pairs of indices which both are not EOB markers).

The computation of boundary conditions for the BMLCS problem is exactly the same as for the MLCS problem. Let us now focus on a computation of a single bit vector W for some pair (j,k). At the beginning we precompute the vectors $W'(j-1,k)$ representing the 'influence' on the current bit vector from the sequence A and $W''(j,k-1)$ from the sequence B (exactly as in [7]). Then this two temporary bit vectors are used to calculate the value of $W(j,k)$. There are three possibilities here:

1. only j is end-of-block marker, i.e., $j \in E^A$, $k \notin E^B$,
2. only k is end-of-block marker, i.e., $j \notin E^A$, $k \in E^B$,
3. both j and k are end-of-block markers, i.e., $j \in E^A$, $k \in E^B$.

The cases 1 and 2 are simple since according to Eq. (1) if only j is an EOB marker the values of $L(x, j-1, k)$, for any x, has no influence on the value of $L(i, j, k)$. Thus, the same holds for $M(j-1,k)$ and $W(j-1,k)$. Similar can be said when only k is an EOB marker. Therefore, we have $W(j,k) = W''(j,k-1)$ for case 1 and $W(j,k) = W'(j-1,k)$ for case 2.

The most interesting is the case 3. Here both $W'(j-1,k)$ and $W''(j,k-1)$ must be taken into account. For simplicity let us think of the alternative representation of them: $M'(j-1,k)$ and $M''(j,k-1)$. To compute $M(j,k)$ we should process these vectors in parallel and for the first pairs of integers put the smaller one to $M(j,k)$. Then, do the same for the second pair and so on. It is easy for integer vectors but not for bit-vector alternative. We can scan the bit vectors bit by bit and mimic the operations on M, but the main idea behind the bit-parallel algorithms is to process computer words (containing w bits) at once.

Let us now switch to the bit-vector representation. At the beginning we compute $U \leftarrow W'(j-1,k) \& W''(j,k-1)$ and $V \leftarrow W'(j-1,k) ^\wedge W''(j,k-1)$. The 1 bits in U indicate the positions at which there are no 0 in $W'(j-1,k)$ nor $W''(j,k-1)$, thus no 0s are possible in W at these positions. 1s in V indicate the positions at which 0 is present in exactly one of $W'(j-1,k)$ and $W''(j,k-1)$. Then we compute the $W^*(j,k)$ bit vector by processing the bits of V one by one. If $V^{[i]} = 1$ we know that exactly one of $W'(j-1,k)^{[i]}$ and $W''(j,k-1)^{[i]}$ is 0, so we must decide whether set the ith bit of W to 0 or 1. Thus, we maintain two counters of zeros seen when processing $W'(j-1,k)$ and $W''(j,k-1)$ bit by bit: z^A and z^B. If for some i we have $W'(j-1,k)^{[i]} = 0$ we increment z^A by 1. If z^A is now larger than z^B we set the ith bit of $W^*(j,k)$ to 0, otherwise we set it to 1. Similar is made when $W''(j,k-1)^{[i]} = 0$. During the scan over V we do not care about the positions for which there are 0s in both $W'(j-1,k)$ and $W''(j,k-1)$, as in this case $W(j,k)$ at this position must be 0, which will be guaranteed later. When $W^*(j,k)$ is ready, we include the 0s from U by:

$$W(j,k) \leftarrow ((W'(j-1,k) | W''(j,k-1)) \& W^*(j,k)) | U. \tag{3}$$

This last computation can be easily made computer word by computer word. The computation of $W^*(j,k)$, as described above, needs bit by bit scan. Fortunately, such description was only for clarity. Below, we show a practical approach. In a real implementation all bit vectors are stored as arrays of w-bit long computer words. Thus, when we need to compute some word of $W^*(j,k)$ representation we have some values of counters z^A and z^B after processing previous words. The value of the word of $W^*(j,k)$ depends on the corresponding words of $W'(j-1,k)$ and $W''(j,k-1)$ and $z^A - z^B$. (This difference can be bounded by w, since when it is larger one of $W'(j-1,k)$ and $W''(j,k-1)$ has no influence on the result.) Therefore, there are $2^w \times 2^w \times (2w+1)$ possible inputs. Precomputing the results for all the possible inputs will be to costly, so we take some smaller value $w' < w$ and treat all computer

Algorithm 1. Pseudocode of the BMLCS length computing algorithm. The array Y of r-bit long vectors store for index c the values 1 only at positions at which the symbol c appears in T.

```
// Boundary conditions
$W(0,0) \leftarrow 1^r$                                   // bit vector of r 1s
for $j$ from 1 to $n$ do
    $U \leftarrow W(j-1,0)$ & $Y^{[A[j]]}$
    $W(j,0) \leftarrow (W(j-1,0)+U) \mid (W(j-1,0)-U)$
for $k$ from 1 to $m$ do
    $U \leftarrow W(0,k-1)$ & $Y^{[B[k]]}$
    $W(0,k) \leftarrow (W(0,k-1)+U) \mid (W(0,k-1)-U)$
// Main computations
for $j$ from 1 to $n$ do
    for $k$ from 1 to $m$ do
        if $k \in E^B$ then                     // k is an end-of-block marker
            $U' \leftarrow W(j-1,k)$ & $Y^{[A[j]]}$
            $W'(j-1,k) \leftarrow (W(j-1,k)+U') \mid (W(j-1,k)+U')$
        if $j \in E^A$ then                              // j is an EOB marker
            $U'' \leftarrow W(j,k-1)$ & $Y^{[B[k]]}$
            $W''(j,k-1) \leftarrow (W(j,k-1)+U'') \mid (W(j,k-1)+U'')$
        if $j \in E^A$ and $k \notin E^B$ then                // only j is an EOB marker
            $W(j,k) \leftarrow W''(j,k-1)$
        if $j \notin E^A$ and $k \in E^B$ then                // only k is an EOB marker
            $W(j,k) \leftarrow W'(j-1,k)$
        if $j \in E^A$ and $k \in E^B$ then       // both j and k are EOB markers
            $U \leftarrow W'(j-1,k)$ & $W''(j,k-1)$
            $V \leftarrow W'(j-1,k)$ ^ $W''(j,k-1)$
            $z^A \leftarrow 0; z^B \leftarrow 0$
            for $i$ from $1$ to $r$ do
                if $(V >> i)$ mod $2 = 1$ then
                    if $(W'(j-1,k) >> i)$ mod $2 = 0$ then
                        $z^A \leftarrow z^A + 1$
                        if $z^A > z^B$ then
                            $W^*(j,k) \leftarrow W^*(j,k)$ & $(\sim (1 << i))$
                    else
                        $z^B \leftarrow z^B + 1$
                        if $z^B > z^A$ then
                            $W^*(j,k) \leftarrow W^*(j,k)$ & $(\sim (1 << i))$
            $W(j,k) \leftarrow ((W'(j-1,k) \mid W''(j,k-1))$ & $W^*(j,k)) \mid U$
// Determination of the result
$\ell \leftarrow 0; V \leftarrow W(n,m)$
while $V \neq 0^r$ do
    $V \leftarrow V$ & $(V-1)$; $\ell \leftarrow \ell + 1$
return $\ell$
```

words of W^* (and the vectors needed for processing) as arrays of subvectors of size w'. In this case the memory requirements are $2^{w'} \times 2^{w'} \times (2w'+1)$ computer words, which could be quite small, e.g., when setting $w' = 8$.

Most operations on bit vectors can be made computer word by computer word in a constant time per word. The only slower operation is the computation of W^*, which is made only for $\Theta(m'n')$ pairs of indices (j,k). Thus the worst-case time complexity of our algorithm is $\Theta((mn' + m'n)\lceil r/w \rceil + m'n' \lceil r/w' \rceil)$. The speedup over the algorithm by Huang [10] is $\Theta(w')$.

A complete pseudocode of the proposed algorithm is given in Algorithm 1. The result, which is the length of the BMLCS, is the number of 0s in $W(n,m)$.

4 Experimental Results

The proposed algorithm was compared to the algorithms known from the literature. Figure 2 shows the results for various sets of input parameters. The examined algorithms are Huang [10], Peng [14], Our (our algorithm with bit by bit scan when computing vectors W^*), Our-LUT (our algorithm with lookup table for faster computation of vectors W^*). From Fig. 2a we see that for small number of blocks

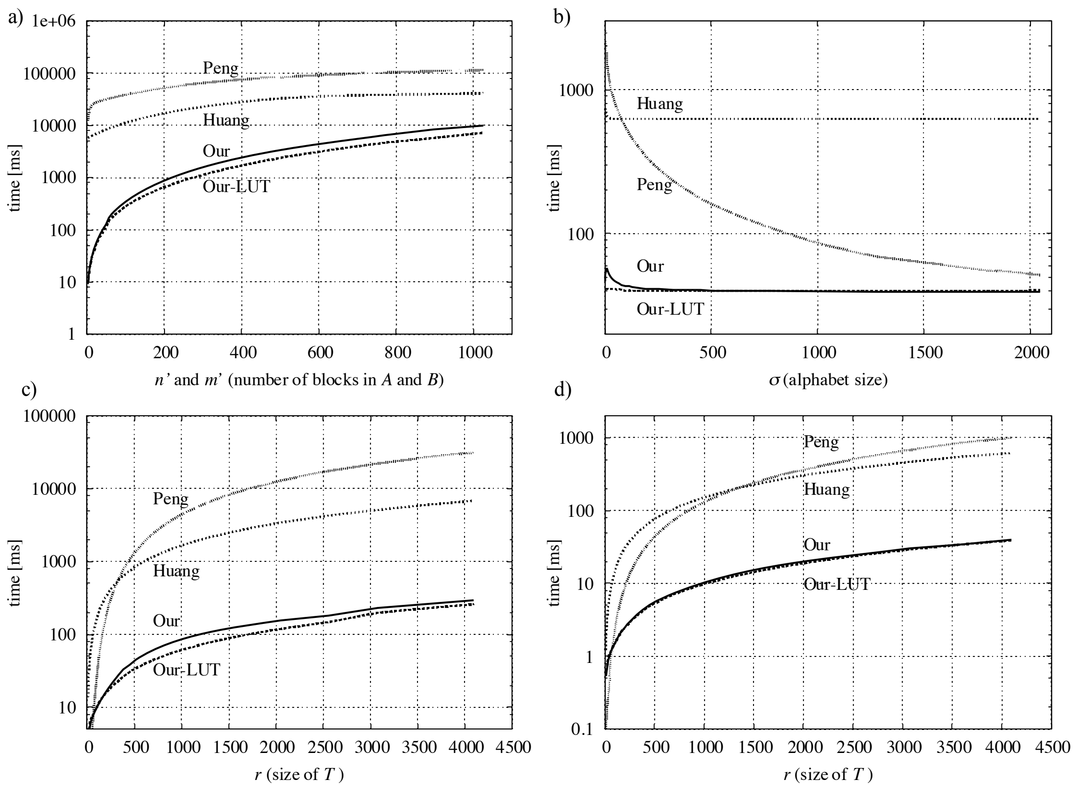

Fig. 2 Experimental comparison of the BMLCS length computing algorithms for various sets of input parameters: a) $T = 2560$, $A = B = 1024$, $\sigma = 4$, $w = 4$; b) $T = 1024$, $A = B = 256$, $n = m = 100$, $w = 4$; c) $A = B = 512$, $n = m = 100$, $\sigma = 4$, $w = 4$; d) $A = B = 128$, $n = m = 50$, $\sigma = 128$, $w = 4$

the advantage of Our algorithm over Peng and Huang is huge (even over 100 times). The gain of using a lookup table is negligible for small number of blocks and significant for large number. This is caused by the fact, that when the number of blocks is large, the W^* vectors are frequently computed. Figure 2b presents the influence of the alphabet size. For small alphabets (e.g., 4-symbol DNA code) our approaches are about 10–50 times faster than Huang and Peng. The last two figures (Fig. 2c and Fig. 2d) show how the algorithms perform for changing size of sequence T for two sets of sequences A and B of different sizes.

In all examined cases we outperformed the literature algorithms significantly in speed (usually 10 times and sometimes even more than 100 times). The gain from a lookup table is noticeable in many cases, however, it depends on the relative number of computations of vectors W^*.

Table 1 shows that for real data sets used in [10] our bit-parallel algorithm is more than 240 times faster than Huang's algorithm and more than 500 times faster than Peng's algorithm. Such large speedups are mainly due to few blocks and the small size of the DNA alphabet ($\sigma = 4$). As we can see in Fig. 2, for small number of blocks (a) and for small alphabets (b) we are much faster than the literature algorithms.

Table 1 Experimental comparison of computations times for real data sets: dodA data set is six-exon DNA sequence of *A. muscaria* dodA gene (gi:2072623) ($r = 1629$, $n = 942$, $m = 687$, $n' = 7$, $m' = 6$), p&d data set is a part of DNA sequences of *Drosophila melanogaster* ($r = 6000$, $n = 2480$, $m = 1756$, $n' = 3$, $m' = 3$)

Data set	Huang[ms]	Peng[ms]	Our[ms]	Our-LUT[ms]	Huang/Our	Peng/Our
dodA	2 388.56	5 444.61	9.69	9.64	246.52	561.94
p&d	55 452.99	171 847.63	49.42	51.40	1 122.19	3 477.64

5 Conclusions

We proposed the first bit parallel algorithm for solving the block merged longest common subsequence problem. The practical experiments show that the algorithm is from 10 to over 100 times faster than the known algorithms. The worst-case time complexity of the method is $\Theta((mn' + m'n)\lceil r/w \rceil + m'n'\lceil r/w \rceil)$ which is $\Theta(w')$ times better than the dynamic programming algorithm by Huang.

Acknowledgements. This research was supported by the European Union from the European Social Fund (grant agreement number: UDA-POKL.04.01.01-00-106/09) and by Polish National Science Center upon decision DEC-2011/03/B/ST6/01588.

References

1. Allison, L., Dix, T.I.: A bit-string longest-common-subsequence algorithm. Information Processing Letters 23(6), 305–310 (1986)
2. Apostolico, A.: General pattern matching. In: Atallah, M.J., Blanton, M. (eds.) Algorithms and Theory of Computation Handbook, ch. 13, pp. 1–22. CRC Press (1998)
3. Baeza-Yates, R.A., Gonnet, G.H.: A new approach to text searching. Communications of the ACM 35(10), 74–82 (1992)
4. Crawford, T., Iliopoulos, C.S., Raman, R.: String matching techniques for musical similarity and melodic recognition. Computing in Musicology 11, 71–100 (1998)
5. Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J., Reid, J.F.: A fast and practical bit-vector algorithm for the longest common subsequence problem. Information Processing Letters 80(6), 279–285 (2001)
6. Deorowicz, S.: Bit-parallel algorithm for the constrained longest common subsequence problem. Fundamenta Informaticae 99(4), 409–433 (2010)
7. Deorowicz, S., Danek, A.: Bit-parallel algorithm for the merged longest common subsequence problem. International Journal of Foundations of Computer Science (to appear)
8. Dömölki, B.: An algorithm for syntactical analysis. Computational Linguistics 3, 29–46 (1964)
9. Gusfield, D.: Algorithms on Strings, Trees, and Sequences—Computer Science and Computational Biology. Cambridge University Press (1997)
10. Huang, K.S., Yang, C.B., Tseng, K.T., Ann, H.Y., Peng, Y.H.: Efficient algorithms for finding interleaving relationship between sequences. Information Processing Letters 105(5), 188–193 (2008)
11. Huang, K.S., Yang, C.B., Tseng, K.T., Peng, Y.H., Ann, H.Y.: Dynamic programming algorithms for the mosaic longest common subsequence problem. Information Processing Letters 102(2-3), 99–103 (2007)
12. Hyyrö, H.: Bit-parallel LCS-length computation revisited. In: Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004), pp. 16–27 (2004)
13. Kellis, M., Birren, B.W., Lander, E.S.: Proof and evolutionary analysis of ancient genome duplication in the yeast saccharomyces cerevisiae. Nature 428(6983), 617–624 (2004)
14. Peng, Y.H., Yang, C.B., Huang, K.S., Tseng, C.T., Hor, C.Y.: Efficient sparse dynamic programming for the merged lcs problem with block constraints. International Journal of Innovative Computing, Information and Control 6(4), 1935–1947 (2010)
15. Tsai, Y.T.: The constrained longest common subsequence problem. Information Processing Letters 88(4), 173–176 (2003)
16. Yang, I.H., Chien-Pin, H., Chao, K.M.: A fast algorithm for computing a longest common increasing subsequence. Information Processing Letters 93(5), 249–253 (2005)