Project 2: Multithread LCS Final Report

December 14, 2017

Sabbir Ahmed

1 Description

The longest common subsequence (LCS) problem is to be implemented for the assignment. The objective of this project is to design, analyze, and implement a multithreaded version of the LCS length algorithm. A memoized or bottom-up approach. In either case, the goal is to design an algorithm that makes efficient use of multiple processors.

Analysis is required once the LCS length algorithm is designed. The work $T_1(m, n)$ and the span $T_\infty(m, n)$ are to be computed, where m and n are the lengths of the input sequences X and Y, respectively. Finally, the parallelism is to be computed along with an estimate of a range of parameters (m, n, and P) for which linear or near-linear speed-up may be expected.

The last part of the project is to implement the LCS length algorithm in C or C++ using OpenMP on UMBC's maya cluster and measure its performance empirically using 1, 2, 4, and 8 processors (optionally, it may be tested on 16 processors; most maya nodes have eight cores, but some have 16). Test data and LCS lengths of various sizes are to be generated to demonstrate the performance characteristics of the algorithm. The algorithm to recover the LCS must also be implemented, but this need not be multithreaded.

2 Initialization

The matrix is initialized to store the LCS before computation. Since the implementation utilizes a two-dimensional array allocated on the heap to represent the matrix, initialization was possible in $\Theta(m)$ time. Since Milestone 2, the initialization loop was parallelized to bring the runtime down to $\Theta(\lg(m))$. This method is faster than iterating through all the cells of the matrix to assign each to a placeholder. Algorithm 2.1 provides the implementation used to initialize the matrix.

Reading the sequences from the one-line input files, truncating them if necessary and storing them into buffers all take constant time operations.

Algorithm 2.1: Initialization of the Longest Common Subsequence Matrix

```
1  lcs_matrix = pointer array(m + 1);
2  parallel for i = 0 to m + 1
3      lcs_i = pointer array(n + 1);
```

3 Serial Algorithm

Milestone 1 required implementation of the serial LCS length algorithm and its analysis. Table 1 visualizes the serial implementation of computing the LCS length of sample strings using the algorithm described in equation (15.9) of *Introduction to Algorithms* [1].

Table 1: LCS matrix constructed when comparing GCGTCA and ACGAA

	∅	G	C	G	T	C	A
∅	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1
C	0	0	1	1	1	1	1
G	0	1	1	2	2	2	2
A	0	1	1	2	2	2	3
A	0	1	1	2	2	2	3

In Milestone 2, it was observed that the value of the cells depended on the cells above, left or top-left of itself. The cells in the other directions have no effect. This property allowing for subproblems to be independently solvable builds the foundation for the algorithm for the parallel implementation to find the LCS length.

Algorithm 3.2 provides the pseudocode for computing the length of the LCS. The serial algorithm utilizes the bottom-up approach to compute the LCS.

Algorithm 3.1: Serial Longest Common Subsequence Length

```
1  function LCS-LENGTH(X, Y, m, n)
2      // allocate (m + 1) × (n + 1) LCS matrix
3      lcs = new matrix[1, 2, .., m + 1][1, 2, .., n + 1]
4      for i = 0 to m
5          for j = 0 to n
6              // if upper-leftmost cell, content is 0
7              if (i == 0 or j == 0)
8                  lcs_{i,j} = 0
9              // if equal, content is the left anti-diagonal value incremented by
                   1
10             else if (X_{i-1} == Y_{j-1})
11                 lcs_{i,j} = lcs_{i-1,j-1} + 1
12             // maximum between previous row and previous column values
13             else
14                 lcs_{i,j} = max(lcs_{i-1,j}, lcs_{i,j-1})
15     return lcs_{m,n} // the last value of the matrix is the length
```

3.1 Time Complexity - Work

The running time of the serial implementation provides the work, T_1, of the algorithm. Simple analysis of the algorithm provided in the snippet Algorithm 3.2 suggests a non-linear running time from the nested loop. The inner loop (line 5) iterates n times with several constant conditional checks, while the outer loop (line 4) iterates m times. The runtime is therefore:

$$T_1(m, n) = \Theta(mn)$$

The work for the algorithm, $\Theta(mn)$, is quadratic if $m = n$.

3.2 Printing the Longest Common Subsequence

After computing the length, the matrix may be used to print the LCS itself. Printing the subsequence is done in a serial implementation since the project does not emphasize its analysis. Since its implementation is not necessary for the scope of the project, the LCS printing functionality will be used for debugging purposes only. Algorithm 3.3 provides the pseudocode used to print

the LCS.

Algorithm 3.2: Serial Longest Common Subsequence Printing

```
1  function SERIAL-LCS-PRINT(X, Y, m, n, lcs)
2      lcsstr = new string
3      cursor = lcs_{m,n} // cursor of the matrix
4      i = m, j = n // init from the bottom-rightmost cell
5      while (i > 0 and j > 0)
6          // if current character in X[] and Y[] are same
7          if (X_{i-1} == Y_{j-1})
8              lcsstr_{cursor-1} = X_{i-1} // result gets current character
9              i--, j--, cursor-- // decrement i, j and cursor
10         // find the larger of two and go to that direction
11         else if (lcs_{i-1,j} > lcs_{i,j-1})
12             i--
13         else
14             j--
15     print lcsstr
```

4 Parallel Algorithm

Milestone 2 required implementation of the parallel LCS length algorithm. This version of the implementation uses the same LCS matrix to generate the same cell values. Table 2 revisits the matrix generated by the serial implementation and highlights the independent subproblems.

Table 2: LCS matrix constructed when comparing GCGTCA and ACGAA with the anti-diagonal subproblems highlighted

	∅	G	C	G	T	C	A
∅	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1
C	0	0	1	1	1	1	1
G	0	1	1	2	2	2	2
A	0	1	1	2	2	2	3
A	0	1	1	2	2	2	3
	[1]	[2]					

The parallel implementation must generate the same values in the cells using these subproblems to avoid race conditions between threads.

To iterate through the cells in these anti-diagonal independent subproblems, the algorithm must consider the constraint that the lengths of the anti-diagonal subproblems do not monotonically increase. The lengths increase until the anti-diagonal label as [1] (highlighted in ▢) and decreases after the anti-diagonal labeled as [2] (highlighted in ▢).

To avoid issues with bounds of the lengths of the anti-diagonals, the parallel algorithm splits the matrix into two halves visualized in Table 3.

Table 3: LCS matrix constructed when comparing GCGTCA and ACGAA highlighting its two halves used for the parallel algorithm.

	∅	G	C	G	T	C	A
∅	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1
C	0	0	1	1	1	1	1
G	0	1	1	2	2	2	2
A	0	1	1	2	2	2	3
A	0	1	1	2	2	2	3

Algorithm 4.4 provides the pseudocode for computing the length of the LCS.

Algorithm 4.1: Parallel Longest Common Subsequence Length

```
1  function P-LCS-LENGTH(X, Y, m, n)
2      // first half of the matrix
3      for i = 1 to n
4          parallel for j = 1 to i
5              if Y_{i-j} == X_{j-1}
6                  lcs_{i-j+1,j} = lcs_{i-j,j-1} + 1
7              else if lcs_{i-j,j} ≥ lcs_{i-j+1,j-1}
8                  lcs_{i-j+1,j} = lcs_{i-j,j}
9              else
10                 lcs_{i-j+1,j} = lcs_{i-j+1,j-1}
11     // second half of the matrix
12     anti-diagonal_len = 0
13     for i = 2 to m
14         // if the anti-diagonal length is not at its maximum
15         if anti-diagonal_len < (m - n)
16             anti-diagonal_len++  // increment the maximum anti-diagonal length
17         parallel for j = i to (n + anti-diagonal_len)
18             if Y_{n-j+i-1} == X_{j-1}
19                 lcs_{n-j+i,j} = lcs_{n-j+i-1,j-1} + 1
20             else if lcs_{n-j+i-1,j} ≥ lcs_{n-j+i,j-1}
21                 lcs_{n-j+i,j} = lcs_{n-j+i-1,j}
```

```
22              else
23                  lcs_{n-j+i,j} = lcs_{n-j+i,j-1}
24      return lcs_{m,n} // the last value of the matrix is the length
```

4.1 Time Complexity - Span

The running time for the parallel implementation provides the span, T_∞ of the algorithm. Computing the runtime of the parallel implementation with an undefined number of processors will yield the span of the algorithm.

The two loops handling their corresponding half of the LCS matrix consist of symmetric operations with identical runtimes of varying sizes, m and n. Therefore, computing the runtime for one of these loops is sufficient.

The inner **parallel** loop spawns all its $\text{ITER}_\infty(j)$ concurrently with an unbounded number of processors. Since $\max_{1 \le j \le i} \text{ITER}(j) = \Theta(i)$, (congruently $\max_{i \le j \le n+d} \text{ITER}(j) = \Theta(i)$ for the second half of the matrix), the **parallel** loop is $\Theta(i)$ [2].

The overhead for the loops come from the outer loops iterating for the total number of anti-diagonals. The number of anti-diagonals is congruent to the sum of the lengths of the sequences minus the $\varnothing$ character, since the implementation skips the first iterations to optimize space. The runtime is therefore:

$$\begin{aligned} T_\infty(m,n) &= \Theta(m+n-1) \\ &= \Theta(m+n) \end{aligned}$$

The span for the algorithm, $\Theta(m+n)$, is linear if $m = n$.

4.2 Parallelism and Speedup

Parallelism is the ratio of the work to the span of the algorithm, T_1/T_∞. It represents the maximum possible speedup on P processors. Computing the parallelism for the LCS algorithm:

$$\begin{aligned}\frac{T_1}{T_\infty} &= \frac{\Theta(mn)}{\Theta(m+n)} \\ &= \Theta\left(\frac{mn}{m+n}\right)\end{aligned}$$

If $m = n$,

$$\begin{aligned}&\to \lim_{n\to\infty} \Theta\left(\frac{n^2}{2n}\right) \\ &\approx \Theta(n)\end{aligned}$$

Speedup is similar to parallelism, where it is the ratio of runtime of the serial execution to the parallel execution with P processors, T_1/T_P. Speedup is bounded by the parallelism of the algorithm,

$$\begin{aligned}\frac{T_1}{T_P} &\leq \frac{T_1}{T_\infty} \\ &\leq \Theta\left(\frac{mn}{m+n}\right) \\ &\leq \Theta(n)\end{aligned}$$

The speedup for the algorithm appears near-linear for P processors.

Another useful relationship between parallelism and the number of processors is parallel slackness, $(T_1/T_\infty)/P$. Linear speedup is achievable while usefully increasing the number of processors if

$$\begin{aligned} 1 &< \frac{T_1/T_\infty}{P} \\ P &< \frac{T_1}{T_\infty} \\ P &< \Theta\left(\frac{mn}{m+n}\right) \end{aligned}$$

Since this project measured the algorithm's performance using up to 16 processors, linear speedup is achievable if

$$\begin{aligned} P &< \Theta\left(\frac{mn}{m+n}\right) \\ 16 &< \frac{mn}{m+n} \end{aligned}$$

Therefore, if $m = n \geq 32$, linear speedup is achievable.

5 Performance Evaluation

The performance of the parallel implementation was measured on the maya nodes using 1, 2, 4, 8 and 16 processors. Slurm Workload Manager was used along with a Bash script to generate different combinations of P processors with various sequence lengths, m and n. Each combinations ran 5 times so that their mean execution times could be computed for analysis.

5.1 Execution Times

Once all the execution times generated, their means were calculated for each combinations of P, m and n. The execution times where $m = n$ were separated from those with $m \neq n$. Figure 1 plots the execution times of the algorithm against the sequence lengths.

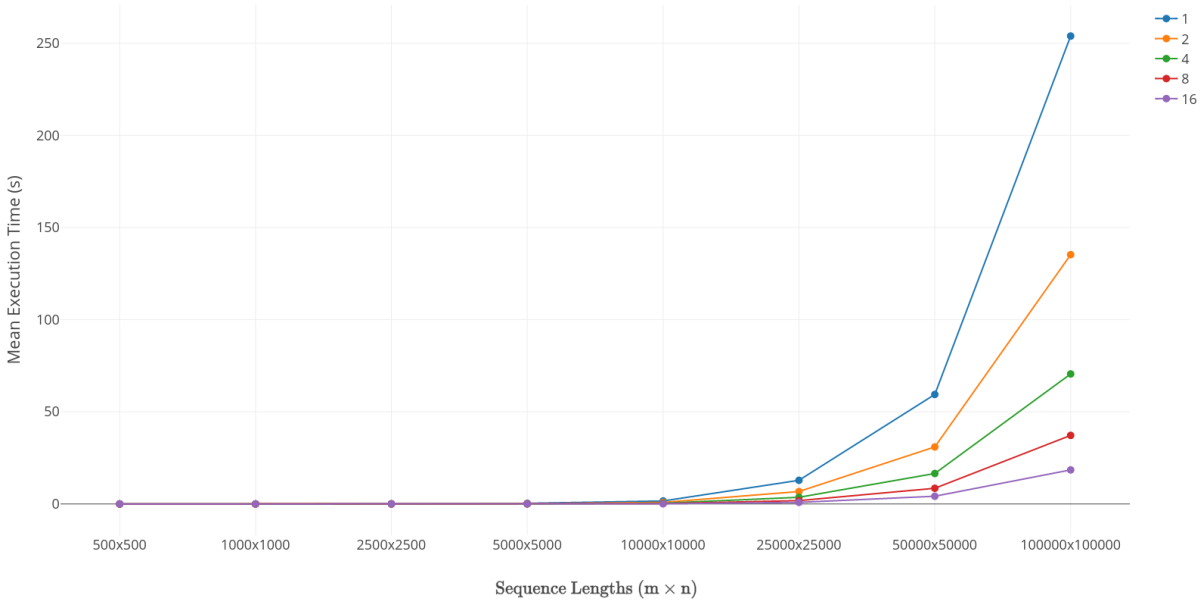

Figure 1: $\log_{10}(t_{exec})$ against sequence lengths, m, n, where $m = n$.

Note the lengths of the sequences do not necessarily follow a constant pattern (they increase in factors of {2, 2.5, 2, 2, 2.5, 2, 2}), due to timing constraints of the job queue of Slurm. Also note that since the execution times increase exponentially, differentiating between the functions becomes difficult. Figure 2 provides a semi- logarithmic plot of the same data, representing the dependent variable in a logarithmic scale.

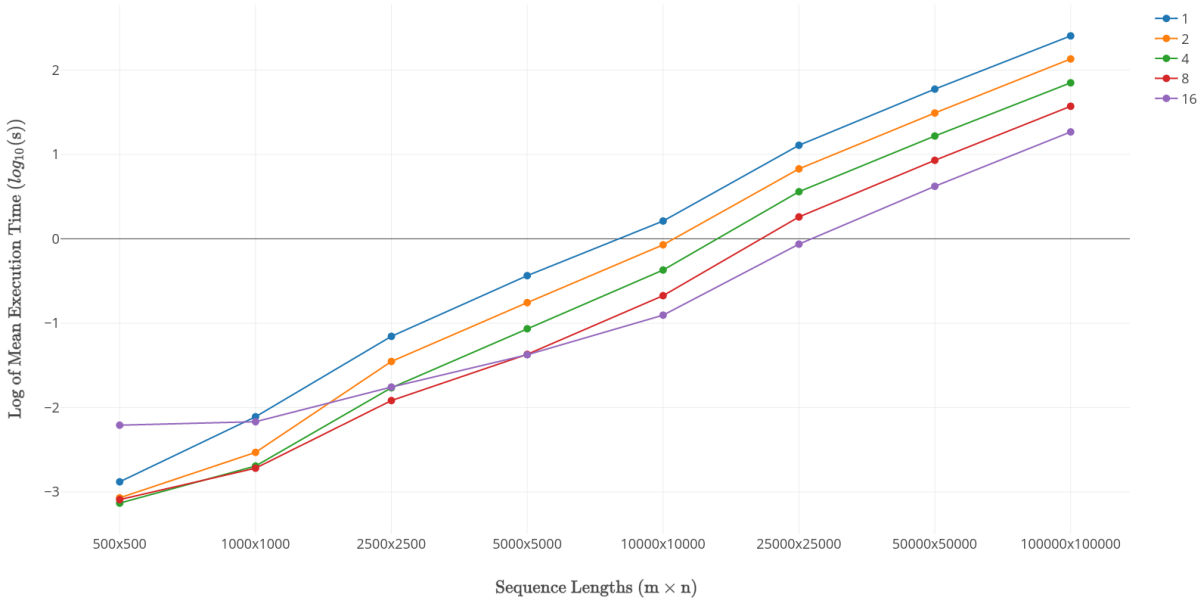

Figure 2: $\log_{10}(\mathrm{t_{exec}})$ against sequence lengths, m, n, where $m = n$.

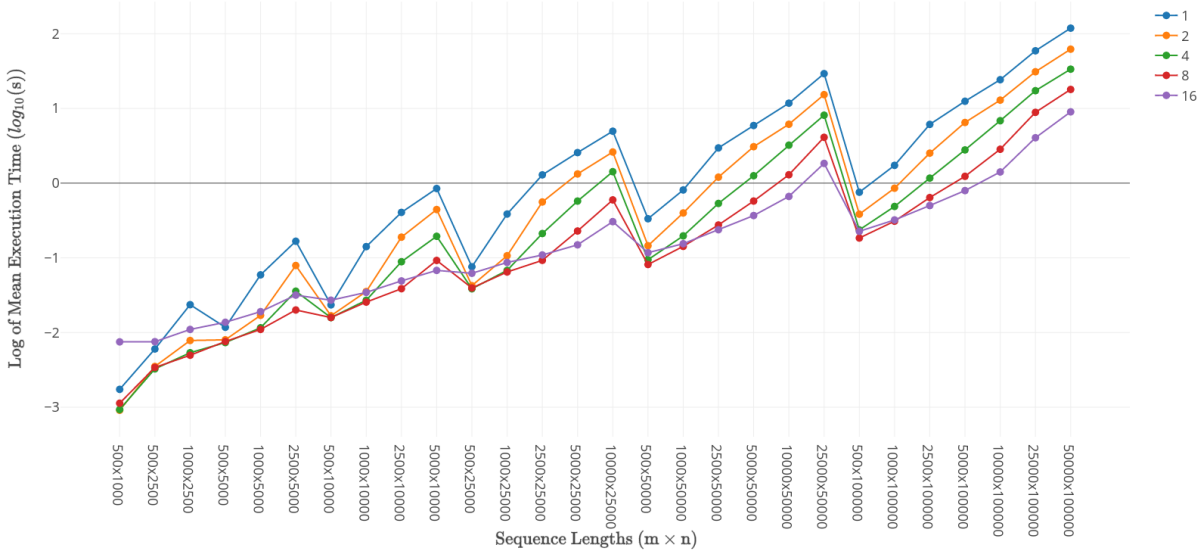

Figure 3: $\log_{10}(\mathrm{t_{exec}})$ against sequence lengths, m, n, where $m \neq n$.

To measure the speedup per processor, the ratio of execution times of T_1 to T_P for $P = \{1, 2, 4, 8, 16\}$ were computed.

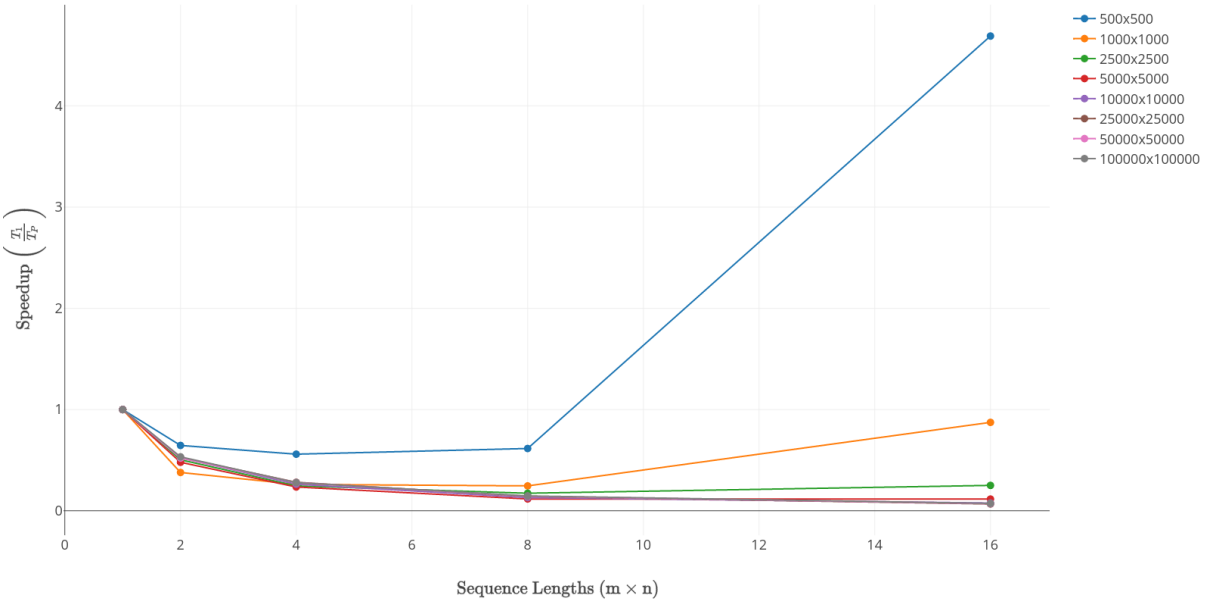

Figure 4: Speedup $\left(\frac{T_1}{T_P}\right)$ against number of processors, P, where $m = n$.

Since the speedup, $T_1/T_P \ll P$, the parallel algorithm is said to have possible linear speedup.

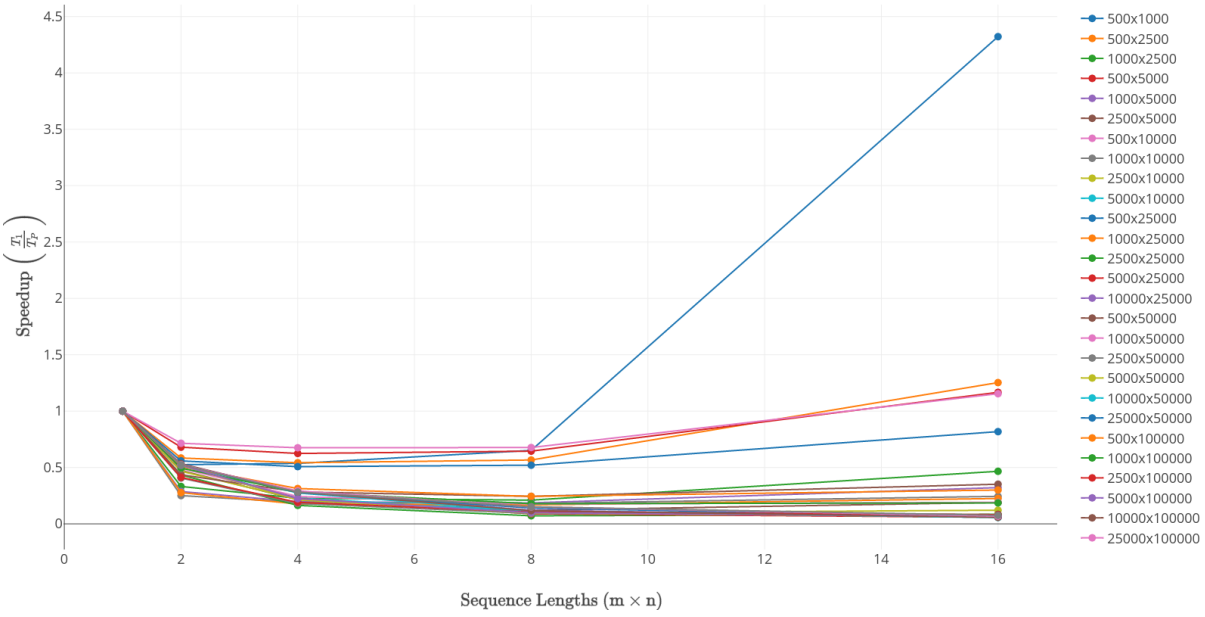

Figure 5: Speedup $\left(\frac{T_1}{T_P}\right)$ against number of processors, P, where $m \neq n$.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[2] X. Huang. A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 18(3):223–239, Jun 1989. URL `https://doi.org/10.1007/BF01407900`.