Crossroads
The ACM Student Magazine

A Parallel Algorithm for DNA Alignment

by ***Thomas Royce*** and ***Rance Necaise***

Introduction to DNA

Deoxyribonucleic acid (DNA) is the hereditary molecule of life [**2**, **6**]. Information encoded within DNA confers physical characteristics such as haircolor, eye color, and susceptibility to certain diseases. By unraveling the secrets encoded within DNA, numerous biological and medical discoveries have been and continue to be made. One basic technique to glean information from a DNA molecule is to compare it to other DNA molecules. This comparison, generally called an alignment, is fundamental to the emerging field of bioinformatics revealing where sequences are the same and where are they different. The differences could be invaluable, for example, in helping to explain why individuals have different susceptibility to disease.

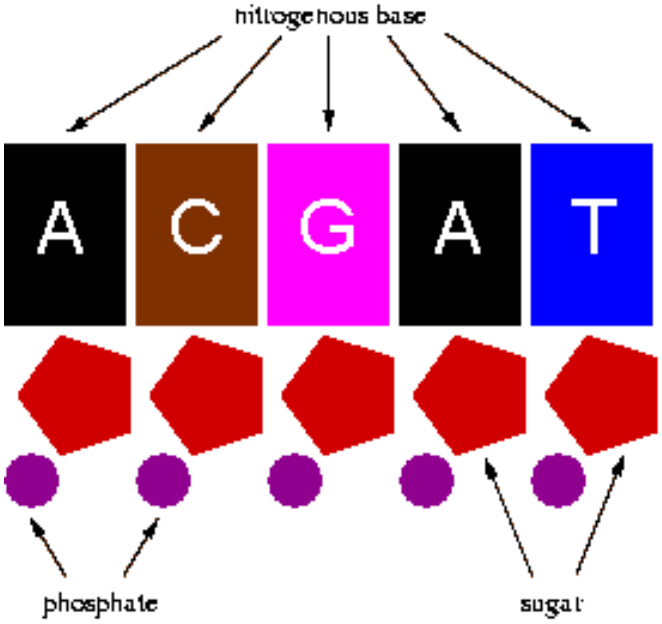

Figure 1: A DNA molecule.

For an algorithm to align two sequences, the DNA must first be represented in a way understandable to a computer. A deoxyribonucleic acid (DNA) molecule is made up of three parts: phosphates, sugars, and nitrogenous bases. As illustrated in **Figure 1**, the phosphates and sugars repeat continuously throughout the molecule. However, the nitrogenous bases differ from position to position and are, to a great extent, the information-containing portion of DNA. There are only four common nitrogenous bases in DNA. These bases are adenine, cytosine, guanine and thymine and can be represented as A, C, G, and T, respectively. A DNA molecule can therefore be expressed as a character string over the finite alphabet $\Sigma = \{A,C,G,T\}$.

The Alignment Problem

Using our representation of DNA as a string, we now formally state the alignment problem. An **alignment** between two input sequences, *s* and *t* over the alphabet $\Sigma = \{A,C,G,T\}$, expresses an equivalence relationship between the pair of sequences by generating two sequences *s'* and *t'* of equal length by inserting spaces (or gaps) into *s* and *t*. An **optimal alignment** is one that minimizes the number of times that gaps are inserted while simultaneously minimizing the number of times that mismatches occur (i.e. a C aligned with a T). There are various scoring schemes that give different penalties for adding gaps as opposed to mismatches, but the particular scheme used does not affect the core of the algorithm described here. To illustrate the idea of an alignment, consider the sequences given by s : taagaac and t : tgac. They could have an optimal alignment consisting of s' and t' below:

```
s': t a a g a a c
t': t - - g - a c
```

Given the scoring scheme of one for a match and zero otherwise, the above alignment would have a score of four. Note that other alignments can share the maximal score:

```
s': t a a g a a c
t': t - - g a - c
```

And other sub-optimal alignments can have lower scores (3 in the following instance):

```
s': t a a g a a c
t': - - t g - a c
```

With the sequencing of entire **genomes** (the entire set of DNA in an organism) becoming a reality (i.e. [**1**, **13**]), algorithms for large-scale alignments are needed. As the lengths of bacterial genomes tend to be in the millions of characters long and mammalian genomes into the billions, these alignment algorithms become an interesting challenge for theoretical computer scientists. The algorithm described here allows for large genomes to be aligned by making efficient use of

memory resources and by utilizing parallelization techniques.

Alignment Algorithms to Date

Standard alignment algorithms utilize dynamic programming methods [**9**, **14**]. The seminal work in this field is that of Needleman and Wunsch [**9**]. Understanding this work is important because it is subsequently used in the parallel algorithm and also because it clarifies the computational requirements of this traditional method. Note that other strategies beyond dynamic programming have been suggested including techniques as varied as hashing, Gibbs sampling, and suffix trees. For a review, see [**8**]. The method described next uses dynamic programming with the simple scoring scheme of one point for a match with no penalty for inserting spaces or gaps.

For sequences A and B of lengths m and n, respectively, first construct an *m x n* matrix M. Let A_i and B_j represent the i^{th} and j^{th} nucleotide of sequences A and B, respectively and let $M_{i,j}$ represent the cell of M that corresponds to both A_i and B_j. Assign a value to each $M_{i,j}$ based on the substitutability of nucleotide A_i for B_j. In the simplest case, assign '1' to $M_{i,j}$ if $A_i = B_j$ and assign '0' to $M_{i,j}$ otherwise.

Next, traverse M by considering cells starting at $M_{m,n}$ and following right to left and bottom to top until cell $M_{1,1}$ is reached. When considering cell $M_{i,j}$, add to the current value of $M_{i,j}$ the maximum value encountered over cells $M_{i+1,j+1}$, ..., $M_{i+1,n}$ and $M_{i+1,j+1}$, ...;, $M_{m,j+1}$. Also in cell $M_{i,j}$, include a pointer to the cell that was used for determining $M_{i,j}$'s new value.

When cell $M_{1,1}$ is reached, the optimal alignment can be reconstructed. This is done simply by finding the maximum score among the cells $M_{1,1}$, ..., $M_{1,n}$ and $M_{1,1}$, ..., $M_{m,1}$. From this cell, follow the pointers until there are no more to follow. If the pointer from cell $M_{i,j}$ points to cell $M_{i+1,j+1}$ then this represents an alignment of two nucleotides. Note that gaps need to be inserted in one of the two sequences to make this alignment if the pointer is not diagonal (**Figure 2**).

		T	A	A	G	A	A	C
T								
G								
A								
C								

(A)

		T	A	A	G	A	A	C
		1						
T								
G					1			
A			1	1		1	1	
C								1

(B)

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

(C)

		T	A	A	G	A	A	C
		4	3	3	2	2	1	0
T								
G		2	2	2	3	2	1	0
A		1	2	2	1	2	2	0
C		0	0	0	0	0	0	1

(D)

Figure 2: Dynamic programming sequence alignment. (A) Construction of dynamic programming matrix. (B) Scores for matches are added to matrix. (C) The score of 3 is obtained for the cell $M_{2,4}$ by adding its previous value of 1 to the largest value calculated in the lower-right sub-matrix. (D) The path representing an optimal alignment.

Unfortunately, this algorithm requires constructing an m by n matrix where m and n are the two input sequences' lengths. Clearly, this method is unsuitable for genome-scale alignments in which millions of characters are considered. Even if memory usage can be reduced [14], the asymptotic time complexities of alignment algorithms based on dynamic programming are quite large. For example, to align two complete human genomes using dynamic programming would take roughly 16000 years on a 850MHz x86 machine with infinite memory assuming just six floating point operations per cell in the matrix.

Faced with the challenge of whole-genome alignments, various heuristics [**3**, **4**] have been developed. Many such algorithms (including the one to be presented here) use the following approach.

Given two input sequences, first identify all substring matches between them that are at least of some predetermined length k (k = 50 works well for bacterial genomes). This can be done in a number of ways but some of the most successful implementations have utilized suffix trees [**3**] or hashing techniques [**4**]. With knowledge of all the substring matches of at least length k, it would be ideal to add them all to an alignment. Unfortunately, this cannot be done for the following reason:

Define a match sequence M(n) of input sequence n as the positional ordering in n of the substrings which have a match in the other input sequence. Consider two input sequences s and t. Let integers represent unique k-length substrings and let X denote any DNA sequence. If s = 123X45 and t = 12X453, then M(s) = 12345 which is not equal to M(t) = 12453. The alignment of substrings denoted by "1" and "2" can be added to a global alignment safely, but problems arise when "3" is added because "3" comes between "2" and "4" in s but after "5" in t (**Figure 3**). This problem can be avoided if maximum-size subsets M'(s) and M'(t) of M(s) and M(t) are taken in such a way that M'(s) is identical to M'(t). Such a subset for the above example would be 1245 for both M'(s) and M'(t) and can be found by applying algorithms to find the longest common subsequence of two sequences. With valid M'(s) and M'(t) at hand it is straightforward to add these matches to an alignment and then align the regions between the matches using a the dynamic programming method outlined earlier. Pseudocode for the heuristic blueprint follows:

- Identify short substring matches.
- Find longest common subsequence of matches.
- Add matches to the alignment.
- Align regions between matches via dynamic programming.

Note that while fast, this heuristic does not guarantee the global optimal alignment. Problems arise when attempting to align sequences with few k-length matches. Consider the extreme case of two divergent sequences that contain one similar stretch of DNA in common. This heuristic will align this similar stretch of DNA even if the sequence is at the beginning of one input and at the end of the other input. Essentially, the problem is of deciding the appropriate length substring matches to look for.

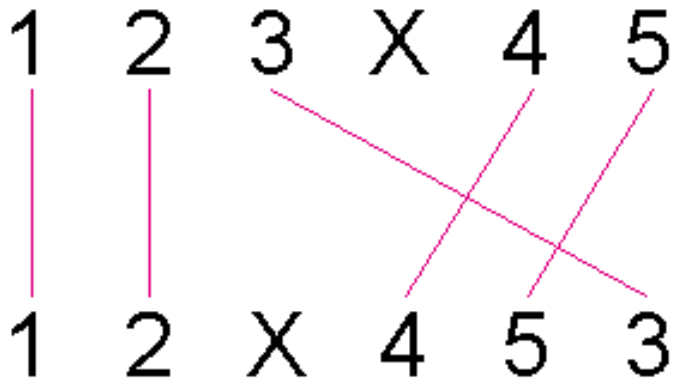

Figure 3: Illustration of why all k-length substrings cannot be added to an alignment. Note the crossing over that occurs if the '3' match is to be added. Therefore, the '3' match must be given up in favor of including the '4' and '5' matches. In this example, using the heuristic, substrings 1 and 2 are aligned followed by the dynamic programming alignment of '3X' and 'X' (note that these two Xs are just variables and need not be identical), followed by the alignment of substrings 4 and 5.

The New Parallel Algorithm

The new parallel algorithm follows closely the heuristic outlined above. Inputs are two strings s and t and a parameter k denoting the length of substring matches to find. Each node in the parallel system receives these inputs. Every substring of length k that belongs to either s or t is given an integer value v, calculated as follows. For each of the k characters in each substring, assign an integer value based on the character to integer mapping f(x) = {0 if x = A, 1 if x = C, 2 if x = G, 3 if x = T}. The value v for that k-length substring is then the sum of its k integers, calculated above, modulated by the number of nodes in the system. For example, a string ATCA would have v equal to zero if using a four node system (0 + 3 + 1 + 0 = 4 % 4 = 0) and one if using a five node system (0 + 3 + 1 + 0 = 4 % 5 = 1). Each node then looks for k-length substring matches among all of the substrings which have values of v that correspond to that node. (i.e. all substrings with a v value of 3 will be considered by node 3 of the parallel system). The matches are found by sorting the substrings lexicographically via a hashing scheme [7] and by then scanning these arrays for k-length matches. Note that by splitting up the substrings in this way all of the matches found by a sequential method are still guaranteed to be found in this

parallel algorithm. Also, this method of dividing up the substrings gives good load balancing characteristics. This concludes step (1) of the heuristic pseudocode.

The aforementioned longest common subsequence algorithm must be applied globally to all the matches found at each node. This is done by first sorting all of the matches based on their position in input sequence s via a sample sort algorithm [**12**] and consolidating the results onto a single node N. Node N finds the longest increasing subsequence (LIS) of these matches based on their position in input sequence t [**5**]. Note that the LIS algorithm simply takes as input an ordered sequence y and selects a maximum length subsequence y' such that y' is in increasing order. For example, the sequence y = {1,2,4,5,3} has an LIS of y' = {1,2,4,5}. The combination of sorting matches based on their position in one input sequence and performing LIS based on the other input sequence results in the longest common subsequence between the two sequences, step (2) in the pseudocode. Unfortunately, no parallel algorithm is known for the LIS problem but for highly homologous sequences it can be computed in linear time with regards to number of matches to be considered.

All of the remaining matches still reside on the root node N. The matches must be distributed amongst the system so that sequences falling between matches can be aligned in parallel. To do this, for each gap between matches, begin by calculating the product of the gap size relative to s and the gap size relative to t. Then take the square root of the product. Let the sum of these square roots be the total gap size (TGS). Note that the calculation of the products and the TGS are done only on the root node as this is the only node which has information about all matches.

Each processor is then distributed a subset of matches such that the subset's summed gap size (calculated in same manner as TGS) is roughly equal to the TGS divided by the number of nodes being used. Step (3) in the given pseudocode is then completed by adding the matches to an alignment residing on that node. This ensures reasonable load balancing. Alignments between the matches (pseudocode step (4)) are performed on each node for its given set of matches. Information about the first match on node i is also sent to node i-1 for all i > 0. This last information exchange is necessary so that the alignment between two nodes' neighboring matches can be computed. Put differently, the gaps in the matches that lie between two nodes i and i-1 are aligned on node i-1. The alignment calculated on each node is then gathered into a single global alignment on the given root node N to complete the parallel algorithm.

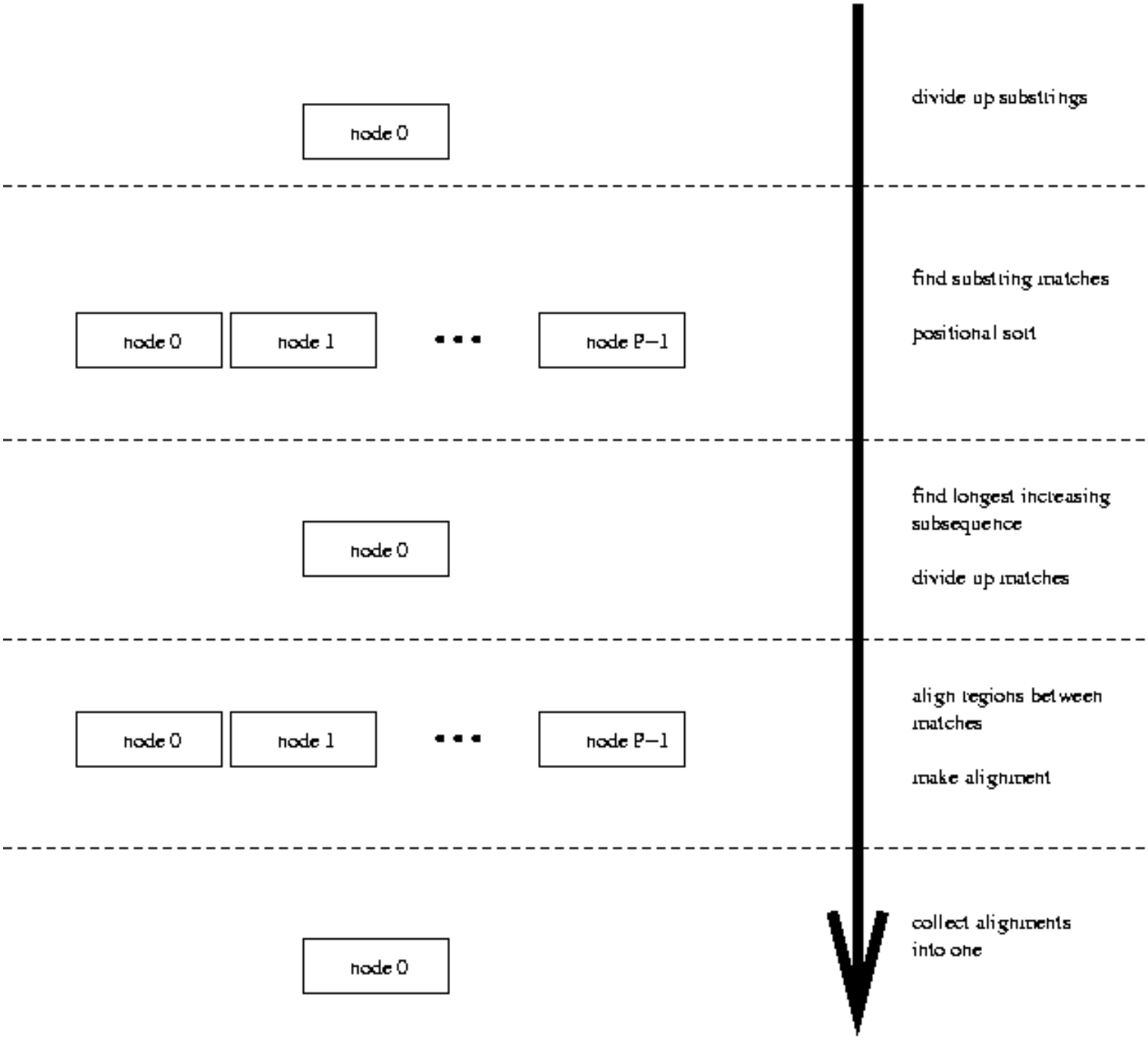

Figure 4: A flow chart of parallel algorithm's tasks.

Results

A proof of concept program was written in the C programming language using the Message Passing Interface (MPI) and compiled with the MPI compiler. Alignments were obtained using a 1.8 GHz x86 quad-processor with 2 GB of main memory and 10 GB of swap space. The following pairs of sequences were aligned to test the program. The boldface text to the left of the sequence descriptions will be the abbreviation used in subsequent analysis.

FISH	mitochondrial genomes from the Brook Trout and the Arctic Char (approx. 16 thousand characters per sequence)

TB	two strains of tuberculosis genomes (approx. 4.4 million characters per sequence)
ARAB	arabidopsis chromosome III and an artificial mutant (approx. 23.5 million characters per sequence)
AMPH	amphioxus mitochondrial genome and an artificial mutant (approx. 15 thousand characters per sequence)
AGRO	two genome sequences of Agrobacterium tumefaciens - one sequenced by Cereon Genomics, the other by U. of Washington (approx. 2 million characters per sequence)

The above sequences were downloaded in FASTA file format from Genbank [**10**]. The arabidopsis and amphioxus mutant sequences were obtained by running EMBOSS MSBAR [**15**] (a program that simulates mutations) on the original sequences obtained from Genbank. All resulting alignments were verified as being optimal or near optimal by comparing them with alignments obtained by applying the dynamic programming algorithm of Needleman and Wunsch when computational resources permitted. For large-scale alignments that did not permit using dynamic programming, verifications of correctness were obtained by comparing the results with those obtained from the MUMmer genome alignment system [**4**]. The alignment of arabidopsis sequences was not possible with MUMmer due to memory constraints so no validation was possible besides visual inspection for obvious gross errors.

The following table shows the results from running the algorithm on the previously itemized alignments with k=50. A comparison with the sequential algorithm offered in MUMmer is also given in the last column. All measurements were obtained by using the MPI clock functions and were taken three times. The average of those three times are presented in the table.

Table 1: Running times in seconds of the parallel algorithm.

Sequences	4 Nodes	2 Nodes	1 Node	MUMmer
FISH	0.1946	0.2879	0.5140	1.011
TB	15.6848	31.0019	70.9055	44.5433
ARAB	72.1245	105.1119	168.4927	unavailable
AMPH	0.0281	0.0412	0.0717	0.0513
AGRO	6.9203	9.9261	15.5762	26.0011

Final Remarks

The speed-up times are not optimal. That is, doubling the number of processors used does not halve the algorithm's running time. However, some speed-up is evident in the results. This work is an initial attempt at parallelizing the alignment problem for genome-scale alignments. It is likely that better solutions are not far off as the need for fast alignment solutions are increasingly becoming necessary in biological research. Also, measurements of memory usage show that approximately nine bytes of memory are used per input character in this system when run on one processor [**11**]. An additional four bytes per character are required for each additional node added to the parallel system. These memory requirements along with speed-ups from parallelization allow for the practicality of aligning whole genomes against one another in a timely fashion.

The problem presented here does not require much by way of biological foreknowledge for its understanding. It does however provide an example of the need for computer scientists to become involved with the emerging field of bioinformatics. Bioinformatics problems deal with large amounts of complex-structured data, making for interesting applications of theoretical computer science, randomized algorithms, artificial intelligence, and in this case, parallel computation. It is likely that new trends and techniques in computer science will be born from the field of bioinformatics.

References

1

Adams, M.D., S.E. Celniker, R.A. Holt, C.A. Evans, J.D. Gocayne et. al. "The Genome Sequence of Drosophila melanogaster." *Science* 287 (2000), 2185-2195.

2

Avery, O., C.M. MacLeod and M. McCarty. "Studies on the chemical nature of the substance inducing transformation of pneumococcal types. Induction of transformation by a desoxyribonucleic acid fraction isolated from Pneumococcus Type III." *Journal of Experimental Medicine*, 79 (1944), 137-158.

3

Chao, K.M., J. Zhang, J. Ostell and W. Miller. "A Local Alignment Tool for Very Long DNA Sequences." *Computer Applications in the Biosciences*, 11 (1995) 147-153.

4

Delcher, A., S. Kasif, R. Fleischmann, J. Peterson, O. White and S. Salzberg. "Alignment of Whole Genomes." *Nucleic Acids Research*, 27 (1999) 2369-2376.

5

Gusfield, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press. Cambridge, 1997.

6

Hershey, A. and M. Chase. "Independent function of the viral protein and nucleic acid fraction I growth of bacteriophage." *The Journal of General Physiology*, 36 (1952), 39-56.

7

Manber, U. and G. Myers. "Suffix Arrays: A new method for on-line string searches." In *Proc. of the First Ann. ACM-SIAM Symp. on Discrete Algorithms*, 1990.

8

Miller, W. "Comparison of genomic DNA sequences: solved and unsolved problems." *Bioinformatics*, 17 (2001), 391-7.

9

Needleman, S.B. and C.D. Wunsch. "A general method applicable to the search of similarities in the amino acid sequence of two proteins." *Journal of Molecular Biology*, 48 (1970), 443-453.

10

NCBI Genbank. **http://www.ncbi.nlm.nih.gov/Genbank/index.html**, February 1, 2002.

11

Royce, T.E. *A Parallel Algorithm for Approximating the Optimal Alignment of Two Genome-Scale DNA Sequences*. Honors Thesis. Washington and Lee University, 2002.

12

Shi, H. and J. Schaeffer. "Parallel sorting by regular sampling." *Journal of Parallel and Distributed Computing*, 14 (1992) 361-372.

13

Venter, J.C., et al. "The Sequence of the Human Genome." *Science*, 291 (2001), 1304-1351.

14

Waterman, M.S. and T.F. Smith. "Identification of common molecular subsequences." *Journal of Molecular Biology*, 147 (1981), 195-197.

15

Williams, G. MSBAR. European Molecular Biology Open Software Suite, 1999.

Biographies

Thomas Royce (**thomas.royce@yale.edu**) is a first-year graduate student in the combined program in Biological and Biomedical Sciences at Yale University.

Rance Necaise (**necaiser@wlu.edu**) is an assistant professor in the Department of Computer Science at Washington and Lee University.