Suffix trees: How to do Google search in bioinformatics?

Curriculum module designer: Ananth Kalyanaraman (ananth@eecs.wsu.edu)

This document contains information pertaining to the programming project and the different ways in which it can be designed. The document is presented in the form of FAQs with the hope that different questions about the project can be addressed in a specific manner. In a separate document (titled "Algorithm_Description") the the parallel algorithms for the programming project under different programming models are provided. C++/MPI-implementations are also provided in the Project Source folder.

Q) In what order should the instructor use the lecture slides?

Primary lecture slides: Lecture1_CompBioPrimer_BWcurr: Introductory slides on computational biology for CS majors

Lecture2_SuffixTree_BWcurr_part1: Introduces students to various string-based operations typical in bioinformatics applications, including a highly popular database search algorithm (BLAST). The slides also introduce students to a simple string data structure (lookup table) used in BLAST.

Lecture3_SuffixTree_BWcurr_part2: Introduces various tree-based string data structures including tries, compacted tries, PATRICIA trees and Suffix Trees. The slides define these data structures, highlights their strengths and use-cases, and provides some example applications for these advanced data structures.

Lecture4_PatternMatching_Project: Provides slides for presenting the project specification to the students and parallel algorithms that can be used in the project solutions. A more detailed description of these algorithms can be found in the "Algorithm Description" document that are included in the module folder.

Supplementary slides:

SupplementarySlides_MolBioPrimer_BWcurr: More detailed introductory slides on computational biology and some biological concepts for CS majors. This is an optional set of slides that the instructor can choose to cover, depending on the level of interest for students in computational biology and bioinformatics.

SupplementarySlides_Example_GlobalAlignment: Slides showing an animation of the dynamic programming based optimal global alignment computation algorithm by Needleman-Wunsch.

Q) What is the goal of this project and what are expected learning outcomes?

The primary goal of this project is to introduce the problem of pattern matching and have the students design, implement and evaluate different algorithmic approaches to solve this problem. The pattern matching problem is the problem of checking to see if a set of query sequences occur as substrings in a given string database (e.g., genome).

Upon successful completion, the students should:

a) Be able to design and analyze algorithms for the problem of pattern matching (and extend the ideas to other closely related string matching problems);
b) Have acquired extensive experience in using string data structures (both tree-based and array-based) and their APIs for matching problems;
c) Be able to identify the main challenges in implementing a parallel solution to string matching.
d) Be able to design and implement parallel approaches to support scalable processing of multiple queries and large sequence databases.
e) Be able to identify the tradeoffs among the different string matching approaches using different data structures/techniques.
f) Be able to identify the primary challenges in translating the techniques from theory to practice.
g) Be able to identify the kind of pattern matching techniques that are better suited for different practical settings (use-cases).
h) Be able to differentiate between exact matching to inexact matching techniques.
i) Be able to identify and appreciate real world applications that can benefit from the use of scalable pattern matching techniques.

Q) Are there other variants of the problem that can be implemented?

Yes, there are multiple variants of the original problem. Given a query sequence:

(i) The first simple variant is that the pattern matching routine returns a boolean (true, if the query is found as a substring, and false otherwise)
(ii) A second variant is that, in addition to returning the boolean, the function also returns the frequency of the query in the database – i.e., how many times does each query occur as a substring in the database.
(iii) A third variant is to expect the function to return not just the count but also the exact Genomic positions that contain the query occurs as a substring.
(iv) A fourth (albeit independent) variant could be to allow for some errors while matching a query against the input genome/database. In other words, treat the problem as one of inexact matching.

Q) What is contained in the "ProjectSource" folder? What implementations are provided?

The project source folder contains multiple codebases. The Appendix explains the algorithms implemented by the code in the source folder. All programs are written in C++ and MPI.

Here is a brief outline of the folder structure:

a) pmatch_naive: This folder contains the Naïve search algorithm.
b) pmatch_st: This folder contains the suffix tree based search algorithm.
c) "scripts": The scripts folder contains all the Perl scripts for use in the project. The scripts can be used to generate synthetic inputs (by simulating DNA sequencing) and collect statistics about FASTA formatted sequence files. A Readme.txt is provided to help use the scripts
d) data: This folder contains numerous FASTA files which could be used as test and experimental inputs (both database and queries).

Q) What kind of background should the students have to work on this project?

a) Students should have a strong programming experience in C/C++. Additional knowledge of Standard Template Libraries (STL) could be a plus but not mandatory.
b) Students should be familiar with the following topics:
    a. Data structures including different types of tree data structures
    b. Design of algorithms
    c. Complexity analysis and asymptotic notation
    d.
c) Students need not have any biological background.
d) Students should be comfortable in at least one of the standard parallel programming environments such as MPI, MapReduce or OpenMP.

Q) Does the project have a multiscale component – either in terms of data or in terms of computation?

Yes, the ProjectSource folder has two subfolders:

a) pmatch_naive: implements a naïve searching algorithm that is better suited for smaller input sizes (both in the length of the genome and number of queries).
b) pmatch_st: implements a suffix-tree searching algorithm that is better suited for use-cases where the genome (big or small) needs to be preprocessed only once (or a few times infrequently), while there are large batches of queries coming over a period of time.

Runtime analysis of these two algorithms is provided in the Algorithm Description document.

Both algorithms can be run on a single processor, or on multiple processors, providing the project with multiple computation scales. Sample MPI implementations are included, and ideas for parallelizing under other frameworks such as OpenMP and MapReduce are provided in the Algorithm Description document.

Also, the project allows for three different test plans when it comes to performance analysis (as listed in the sample job script, sub.sh, in the test folders of these codebases):

a) Studying runtime as a function of genome size (keeping #queries and number of processors fixed)
b) Studying runtime as a function of number of queries (keeping genome and number of processors fixed)
c) Studying runtime as a function of number of processors (keeping genome and #queries fixed)

Numerous input files (both genomes and query files) are provided as part of the package (please see under the “data” folder). These can be used to test to scale the performance of different implementations. Also on the “scripts” folder, Perl scripts are provided (see readgenerator.pl) to generate arbitrarily sized inputs. Please refer to the readme file in that folder for more details.

The above setting also allows instructors to introduce the concepts relating to parallel performance – viz. speedup, efficiencty, strong and weak scaling; and follow that up with an experimental plan that would allow the students to test them. The instructors can ask the students to represent the experimental results generated from the above three test plans in the form of a table – for instance, a table the parallel runtimes for different input sizes (number of queries: 1K,2K,4K,8K,16K, etc.) over multiple processor sizes (core counts: 1,2,4,8,16, etc.). From this table the students can plot speedup, efficiency, and identify if the code exhibits strong or weak scaling.

Q) How do I know if the students have designed their algorithm well? Are there reference algorithms provided along with the project?

Yes, please refer to the Appendix section below.

Q) Are there extensions and other variants of this project that can be pursued?

Yes, please refer to the Appendix section below.

Q) How long is the project expected to take and is it a team work based project?

The expected duration for implementation for both the projects (naïve and suffix-tree based) is roughly 2 weeks, assuming the students are proficient in writing C/C++ programs and have basic programming experience in MPI/OpenMP/MapReduce.

It is desirable that the students work in teams of 2 or 3.

Q) From the ProjectSource folder, what are the project components that the students should be asked to implement and what parts should be provided to them (as libraries) for use in the project?

<u>For the naïve implementation:</u> It can be expected that the entire codebase can be implemented by the students. Alternatively, if the instructor wants to have the students save some time in reading the FASTA file and focus more on the implementation details of pattern matching, then the instructor can provide the FastaReader library (API: fasta_reader.h, and implementation in fasta_reader.cpp) which the students should be able to use (as shown in pmatch_naive.cpp and pmatch_st.cpp) to load the input sequence files, both genome and queries.

<u>For the suffix tree-based implementation:</u> The students are expected to be provided with the suffix tree data structure API and implementation in SuffixTree.h and SuffixTree.cpp respectively. They need to use this library in order to build the suffix tree on the genome and then use the tree, using its API functions which are well documented (refer to the public methods in SuffixTree class), in order to implement pattern matching (as shown pmatch_st.cpp).

Q) What is the practical relevance of solving the pattern matching problem? Are there real world applications (both in bioinformatics and outside) that can benefit from an efficient implementation for this problem?

Yes, very briefly, the formulation of pattern matching that is targeted here has direct applications in genome read-to-reference mapping and genome searching applications. More specifically, the genome read-to-reference mapping problem is a common theme in resequencing projects, where a biologist has: i) a reference (already sequenced) genome that is representative of a species, and alongside ii) new raw reads (sequences) collected from specific individuals of that species (or a closely related one). The goal is to find the minor variations (called single nucleotide polymorphisms [SNPs]) in the individual's genome relative to the reference.

The second problem is genome searching, where a biologist has sequenced a new genic sequence but wants to locate it along the source genome or in genomes from closely related species (this is common in comparative genomics projects).