Approximate String Matching in DNA Sequences

Lok-Lam Cheng David W. Cheung Siu-Ming Yiu
Department of Computer Science and Infomation Systems,
The University of Hong Kong, Pokflum Road, Hong Kong
{llcheng,dcheung,smyiu}@csis.hku.hk

Abstract

Approximate string matching on large DNA sequences data is very important in bioinformatics. Some studies have shown that suffix tree is an efficient data structure for approximate string matching. It performs better than suffix array if the data structure can be stored entirely in the memory. However, our study find that suffix array is much better than suffix tree for indexing the DNA sequences since the data structure has to be created and stored on the disk due to its size. We propose a novel auxiliary data structure which greatly improves the efficiency of suffix array in the approximate string matching problem in the external memory model. The second problem we have tackled is the parallel approximate matching in DNA sequence. We propose 2 novel parallel algorithms for this problem and implement them on a PC cluster. The result shows that when the error allowed is small, a direct partitioning of the array over the machines in the cluster is a more efficient approach. On the other hand, when the error allowed is large, partitioning the data over the machines is a better approach.

1. Introduction

DNA sequences, holding the code of life of every living organism, can be considered as strings over an alphabet of four characters, {A, T, C, G}, called bases. Mammalian DNA sequences can be very long. For example, a human genome (DNA) contains around 3Gbp. Searching patterns in the DNA sequences databases is usually the first and a crucial step in DNA related research.

Besides the human genome, researchers are also interested in DNA sequences of other species. There are a lot of other DNA sequencing projects that are being carried out in different laboratories to determine the DNA sequences of different species. In Jun 2002, according to GenBank [1], one of the largest public DNA sequences database, the total number of bases stored is over 20Tbp. So, an efficient searching tool which is flexible enough to allow users to locate patterns in different genomes is desired.

Currently, biologists mainly use BLAST [1], a popular DNA pattern searching engine, to locate patterns in DNA sequences. Basically, BLAST searches a pattern by scanning the sequences sequentially with the help of heuristics and filtering technique to speed up the searching. FASTA [10, 20] is another popular utility that also uses sequential searching with heuristics technique. In [26], Williams proposed a new genomic databases system named CAFE and show that CAFE is faster and more accurate than FASTA and BLAST. Since these approaches are heuristics, all of them cannot guarantee that all significant matches are found. We call this case, missing some matches, as *match-miss*.

In addition to using heuristics, there are some works which are without match-miss for approximate pattern searching. E. Hunt [8] proposed to partition suffix tree of the DNA sequence into different sub-trees, such that each sub-tree can be built in main memory only. In section 4.1, we will show the experiments which compare Hunt's suffix tree and suffix array and we found that suffix array performs much better than Hunt's suffix tree.

In this paper, we focus on a practical solution for indexing and approximate pattern searching without match-miss in huge size of DNA sequences data. The following is the summary of our contributions:

1. We show that suffix array is much more efficient than suffix tree in external memory model (i.e. that the main memory is not large enough to store the whole indexing structure)

2. We propose a novel auxillary indexing structure named *quick lookup table* which can improve the searching efficiency of suffix array.

3. We propose two novel approaches of parallel computing for indexing and searching DNA sequences in PC clusters which can reduce the index construction time and querying time.

[1] http://www.ncbi.nlm.nih.gov/Database/index.html

2. Background and Related Work

Searching patterns in DNA sequences can be considered as the traditional approximate string matching (ASM) problem:

> Given a character string S, a string q, and an error bound k.
> Find all substrings, s, in S such that $ed(s,q) \leq k$, where $ed(s,q)$ is the edit distance between s and q. Edit distance is defined as "the minimal number of inserations, deletions and substitutions to make two strings equal"

In the context of bioinformatics, S can be human DNA sequences consisting 4 types of characters and the length of S can be 3Gbp. The length q can be several thousands. The error bound k can be 10% of the length of q.

2.1. Linear Scanning Algorithms

For the ASM problem, there are many works have been done before. [14] provides a very good survey of ASM algorithms and indexing structures. A simple $O(|S||q|$ *Dynamic programming* algorithm [22] can solve the problem. However, in the context of searching the human DNA with $|S| = 3G$, the performance of the algorithm is unsatisfactory. On the other hand, the ASM problem can be mapped to the *non-deterministic automaton model* and it gives the worst-case time algorithm $O(|S|)$ (for details, refer to [23, 27]). [28] improves the cost automaton simulation to $O(\lceil |q|/w \rceil |S|)$ by *bit-parallelism* technique where w is the number of bits to represents a word in CPU. However, it is only suitable for short queries.

2.2. Candidate Selection Techniques

Candidate Selection Technique is quite efficient practically. Instead of performing a dynamic programming over S, a set of potential candidates are selected. Then, each candidate will be verified to see if it is a matched string. For example, [2, 13] proposed to partition a query into j pieces. Based on the observation that at least one of these pieces appear with at most $\lfloor k/j \rfloor$ errors in any occurrence of the matching region, the problem can be solved as follows. We partition q into $q_1 q_2 .... q_j$. For each q_i, we search for matches sub-strings in $|S|$ with at most $\lfloor k/j \rfloor$ errors. The matched sub-strings are our candidates. We call this as *candidate selection phase*. After that, for each candidate, we verify with the neighborhood to check whether it match with q within k error and this called *verification phase*.

BLAST [1] and FASTA [10, 20] also use this technique. They use heuristics to purge out candidates that will not *possibly* be match strings for the query. However, the heuristics may purge regions that cover matches. So, both methods are not 100% accurate.

2.3. Index Structures for ASM

One special property of DNA sequences is that it cannot be broken into words, unlike normal text. So, inverted files [6], String B-trees [5] and prefix index [9] may not be good choices. Moreover, q-grams [15, 19] are not very suitable for low similarity [14]. However, suffix trees [24, 25] and suffix arrays [11] have been shown to be useful and efficient in solving ASM problems in [16].

2.3.1. Suffix Tree. A suffix tree is a compact representation of a trie corresponding to the suffixes of a given string where all nodes with one child are merged with their parents. For an extensive description of suffix tree and its applications in biological problems, please refer to [7].

A straightforward approach to construct a suffix tree from S is to insert the suffixes one by one into the suffix tree which takes $O(|S|^2)$ time. To speed up the construction process, we can make use of *suffix links* [12, 24]. With the help of suffix links, the complexity of construction time can be reduced to $O(|S|)$. Interested readers can refer to [7, 12, 24] for the details.

Unfortunately, the size of suffix tree can be very large if S is very long. For the human DNA, $|S|$ is 3Gbp and the size of suffix tree is more than 48 GBytes. So, main memory algorithms for constructing suffix tree are impractical.

2.3.2. Hunt's Version of Suffix Tree. D. R. Clark et al. [3] proposed an algorithm for maintaining a modified suffix tree construction, called *Partitioned Compact Pat Tree*, on secondary storage such that the number of disk access for searching and updating is minimized, and the Partitioned Compact Pat Tree is converted from *Compact Pat Tree*. However, how to build a Compat Pat Tree for very large sequence with limited memory is not mentioned.

Recently, Hunt et al proposed a method that makes use of external memory for constructing suffix trees [8]. The idea of Hunt's algorithm is to partition the suffix tree into sub-trees, such that each sub-tree can be constructed in main memory to eliminate the IO. Basically, they group the suffices according to their prefixes. Roughly speaking, they fix a length k, suffices with the same prefix of length k go to the same sub-tree.

The drawback of the tree partition forces them to abandon the use of suffix links. Thus, the construction time is $O(|S|^2)$ in worse case and $O(|S| \lg |S|)$ in average case.

2.3.3. Suffix Array. Suffix Array [11] is basically a sorted list of all the suffices in S in lexicographical order. One advantage of suffix array over suffix tree is that the size of a suffix array is much smaller than that of a suffix tree. Suffix array is highly related to suffix tree. We can use a suffix array to simulate a suffix tree [16]. Basically, nodes of suffix tree correspond to intervals in suffix array. So, each time

the suffix tree algorithm is at a given node, its suffix array simulation is at a given interval. In suffix tree, given a node v, we just need to follow the pointer from v to find v's children. This takes $O(1)$ time. However, in suffix array, given a node v, we need to take $O(\lg n)$ time to find v's children by binary search, where n is the size of interval representing node v.

2.3.4. ASM Algorithms on Suffix Tree/Suffix Array. ASM algorithm over suffix trees or suffix arrays have been studied in [16, 18]. For simplicity, we name this algorithm as "ASMDFS" (**A**pproximate **S**tring **M**atching with **DFS**). We will describe the algorithm roughly. For a given query q and error k, we want to find all substrings s in S with $ed(q,s) \leq k$. The algorithm starts from the root node. We descend recursively by every branch of the suffix tree in a DFS manner up to a limited depth, $q+k$. For every visited node, we get the path-label [2] x. If $ed(q,x) \leq k$, report all the leaves of the current subtree as answers. For the details of the algorithm, please refer to [16, 18].

2.3.5. Candidate Selection Technique on ASMDFS (CSTASMDFS). However, we do not apply the ASMDFS on suffix tree or suffix array directly because it involves many number of node accesses and greatly degrades the performance. As suggested in [16], a candidate selection approach, discussed in Section 2.2, should be used. We call this method as *Candidate Selection Technique on ASMDFS* (CSTASMDFS). All the algorithms in section 3 will be based on this algorithm.

CSTASMDFS algorithm is divided into two phases: *Candidate Selection Phase* and *Verification Phase*. For a given query q and error k, we partition q into j pieces. [16] have discussed how to choose the optimal value of j. In candidate selection phase, we apply ASMDFS on each q_i with error $\lfloor k/j \rfloor$ and find certain numbers of candidates. In verification phase, we verify all the candidates found in previous phase and determinate whether it is a match.

In [16], it shows that suffix array and suffix tree perform very well in ASM problem with the CSTASMDFS algorithm and also shows that suffix array performs better than suffix tree in some cases. However, the study only concentrated on main memory model. In this paper, we are focus on external memory model index structures for DNA.

3. Our Approach

In order to be more precisely to express our idea, we first define some notations and functions:

- For a given DNA sequence S, define $S[i]$ as the ith character of S and $|S|$ is the length of S.
- to convert a DNA sequence S into an integer value, we define the $val(S)$ function:
  $val(S) = \sum_{i=1}^{|S|} val(S[i]) \times 4^{|s|-i}$;
  $val(\text{"}a\text{"}) = 0$, $val(\text{"}c\text{"}) = 1$, $val(\text{"}g\text{"}) = 2$, $val(\text{"}t\text{"}) = 3$
- to convert an integer value back to a DNA sequence S, we define the $str_m(x)$ function:
  $str_m(x) = S$ if $val(S) = x$ and $|S| = m$
- given a DNA sequence S, we define $suf(j,S)$ as the jth suffix of S.
- given a suffix array A of a DNA sequence S, define $A[i]$ is an integer of the ith element and represents the suffix $suf(A[i],S)$.
- given a sequence S, we define $pre(m,S)$ which returns the prefix of S with length m. (i.e. the first m characters of S)

[2] Path-label of node x is the string concatenating all the label of the edges travelling from root node to x.

3.1. Indexing Large DNA sequences for External Memory

Suffix tree and suffix array have been shown to be good for ASM problem [16]. So, our research focuses on the two indexing structures. There are not many related works about suffix array index on external memory model. Besides Hunt's suffix tree, [4] studies the construction of suffix arrays in external memory. It assumed that the sequence data S is also stored in external memory. However, in our situation, we assume the DNA sequence S is stored in the main memory and the suffix array is stored in external memory. The algorithms proposed in [4] are not optimized for our situation and involve too many IOs, e.g. external sort. So, in section 3.1.1, we propose a new algorithm without external sort to avoid IOs. One may argue that the assumption is not valid if S is too large and cannot fit into main memory. So, in section 3.3, we propose the DP parallel computing approach to solve this problem.

3.1.1. Suffix Array Construction in External Memory. Since the main memory is not enough to store the whole suffix array, we build it part by part. Assuming the main memory can only support sorting M suffices. Logically, we divide the suffix array into k parts (p_1, p_2,..., p_k) and ensure the size of each part is not more than M. In order to actually divide the suffix array into k parts, we first need to build a statistical information array B which stores some statistical information about the prefixes of suffices of the DNA sequence S. We choose a number m, the length of prefix in every suffices we want to investigate (the appropriate value of m is around 8 to 12). The size of B is 4^m. $B[i]$ stores the number of suffices with prefix equals $str_m(i)$ in the DNA sequence S . To build $B[i]$, scanning S once, for each suffix s, we increase $B[val(pre(m,s))]$ by 1.

For p_i, it only contains suffices with certain prefixes. (i.e. suffices with same prefix of length m go to same part.) Specifically, for any s in p_i, $a_{i-1} \leq val(pre(m,s)) < a_i$

where $a_0 = 0$, $a_{i-1} < a_i$ and $a_k = 4^m$. To ensure size of $p_i \leq M$, we need to ensure $\sum_{j=a_{i-1}}^{a_i-1} B[j] \leq M$. The function of a_i is to define the range and size of the parts. To find a_i, we can use Algorithm 1.

Algorithm 1 Find_ai(B,M,m)

1. a_0=0; i=0
2. while $a_i < 4^m$
3. $sum = 0$; i++; a_i=a_{i-1}
4. while $sum + B[a_i] < M$ and $a_i < 4^m$
5. $sum = sum + B[a_i]$
6. a_i++
7. return $\{a_0, a_1, ..., a_k\}$

After found all the a_is for the parts, we can build the suffix array of S(Algorithm 2) . The for-loop at lines 2-4 is to find all the indexes of the suffices that belong to p_i and store to the array A. Then, sort the suffices representing in A (line 5) and store this part to hard-disk appending on the previous part, if any (line 6). Then, go back to line 1 for next part until all parts have been sorted and stored.

3.1.2. Suffix Tree VS Suffix Array. In this section we will compare the suffix tree and suffix array data structures. Since our goal is to solve the ASM problem on DNA sequences, we will focus on the performance of CSTASMDFS algorithm, mentioned in section 2.3.5. The algorithm involves DFS traversing on suffix tree. Since suffix array can simulate suffix tree, the algorithm can also be applied on suffix array by simulation . One may think that DFS on suffix tree must be faster than the simulation on suffix array. However, it is not the case for external memory model of suffix tree and suffix array. It is mainly due to the following reasons:

1. Nearly every node access causes a random IO on hard-disk in suffix tree. In suffix tree, the nodes are randomly distributed. In other words, the parent node is normally not located with it's children within same disk block. So, when the algorithm traces the pointer from parent node to it's children, it often requires to load a new disk block from the hard-disk. However, in the situation of suffix array, the random IO only occurs at the simulation of top level nodes with binary searching. When the simulation is down to deeper nodes, the interval to represent the nodes are smaller and normally within a disk block. So, no random IO is required.

2. The system cache can help to improve the performance for external memory index structure. Since the size of suffix array is smaller than suffix tree, suffix array can utilitize more effectively the system cache. So, less cache miss in suffix array can improve the overall performance.

In our experiment, it shows that suffix array is 10-30 times more efficient than suffix tree in CSTASMDFS algorithm. For more details, please refer to section 4.1.

Algorithm 2 SA_Ext_Build (S,m,a_0,a_1,...,a_k)

1. For i=1 to k
2. for j=1 to $|S| - m$
3. if $a_{i-1} \leq val(pre(m, suf(j,S))) < a_i$
4. append j in A
5. sort A in the order of the suffices for each element it represented
6. write out A to hard-disk and append on previous part if any

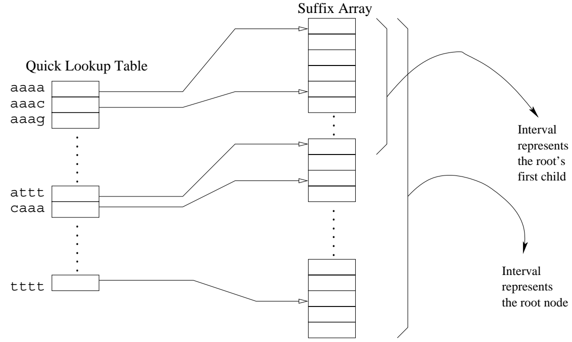

Figure 1. Example of *Quick Lookup Table*

3.2. Quick Lookup Table (QLT)

As analyzed in section 3.1.2, suffix array performs better than suffix tree on CSTASMDFS algorithm in external memory model. However, there are still some room for improvement. During the simulation of suffix tree at top level nodes, it induces random IO due to the binary searching on large intervals. So, we introduce the *quick lookup table* (QLT) such that binary searching is not required for top level nodes simulation.

A QLT, T, is an integer array with size 4^m and $T[i]$ stores a position, j, for the suffix array A, such that starting from $A[j]$ up to $A[T[i+1]-1]$, all the suffices in the range must have prefix equal to $str_m(i)$. For a chosen value m (normally between 8 and 12), it can avoid binary searching for the simulation of nodes with depth up to m.

In Figure 1, it shows an example of a QLT, T, used to accelerate the suffix tree simulation in a suffix array A. In the example, the value of m is 4. $T[0]$ corresponds the prefix "aaaa" pointing to $A[1]$ and $T[1]$ corresponds "aaac" pointing to $A[5]$. So, we can deduce that the suffices in the first 4 entries, $A[1..4]$,must start with "aaaa". For simulating the suffix tree, the root node is represented by the interval of the whole suffix array (Figure 1). Then, for the DFS algorithm, we need to find the interval representing the first child of the root node. Without the QLT, we need to use binary search to find the entry, $A[i]$, in the suffix array, such that the suffix in $A[i]$ starting with "a" and the suffix in $A[i+1]$ starting with "c". However, with the help of QLT, we can just follow the pointer of the "caaa" entry in the QLT to find the interval. So, we can use the same technique to find the children of

other nodes in the simulation of nodes with depth up to m.

So, with the help of QLT, we can greatly reduce the random IO of the suffix tree simulation in suffix array. Theoretically, the larger m is, the better performance can be achieved. However, larger m means larger table and larger main memory requirement. For, the construction of the QLT, it can be directly converted from the statistical information array B (discussed in section 3.1.1) by assigning $T[i+1]$ as the value of $B[i]$.

3.3. Parallel Computing Approaches

Although suffix array with QLT provides a very good way to solve the ASM problem, the response time for ASM problem in a very large DNA sequence (e.g. 3Gbp in Human) may not be acceptable. So, parallel algorithms should be considered. In this section, we will propose two parallel computing approaches for PC clusters – *Index Partitioning* (IP) and *Data Partitioning* (DP). In order to distinguish between nodes in suffix tree and nodes in PC Clusters, we call the nodes in suffix tree be *treenodes* and the nodes in PC clusters be *PCnodes*.

3.3.1. Index Partitioning (IP) Algorithm. The idea of IP comes from Hunt's tree partition algorithm [8]. However, we partition the suffix array rather than partition the suffix tree. In IP algorithm, we partition the suffix array into sub-arrays and each PCnode in the cluster is responsible for building one of the sub-arrays. For searching a query, each PCnode applies the CSTASMDFS algorithm to find the matching results and report back to the users.

Specifically, assuming that there are N PCnodes in the cluster, we partition the suffix array into N approximately equal sized sub-arrays, SA_1,SA_2...SA_N. For SA_i, it only contains suffices with certain prefixes, similar to the parts p_i in Section 3.1.1 (i.e. suffices with same prefix of length m go to same sub-array). To be more precise, for any suffices s in SA_i, $b_{i-1} \leq val(pre(m,s)) < b_i$ where $b_0 = 0$, $b_{i-1} < b_i$ and $b_N = 4^m$ and $\sum_{j=b_{i-1}}^{b_i-1} B[j] \approx |S|/N$. The array B is the statistical information array mentioned in Section 3.1.1. The function of b_i is to define the range and size of SA_i. To find b_i, we can use Algorithm 3 which is very similar to Algorithm 1.

Algorithm 3 Find_bi(B,S,N)

1. b_0=0;$b_N = 4^m$; $sum = 0$
2. for i = 1 to N-1 do
3. $\quad b_i$=b_{i-1}
4. $\quad$ while $sum + B[b_i] > i \times |S|/N$ and $b_i < 4^m$
5. $\quad\quad sum = sum + B[b_i]$
6. $\quad\quad b_i$++
7. return {b_0,b_1,...,b_N}

For the sub-array construction, the SA_i will be assigned to the PCnode$_i$ and we apply similar technique mentioned in section 3.1.1 to build the SA_i in external memory model. Specifically, in PCnode$_i$, SA_i is further divided into k_i parts, p_{i_1}, p_{i_2}, ...,$p_{i_{k_i}}$. For any suffix s in p_{i_j}, $a_{i_{j-1}} \leq val(pref(m,s)) < a_{i_j}$ where $a_{i_0} = b_{i-1}$, $a_{i_{j-1}} < a_{i_j}$ and $a_{i_{k_i}} = b_i$. To ensure the size of $p_{i_j} \leq M$, we need to ensure $\sum_{k=a_{i_{j-1}}}^{a_{i_j}-1} B[k] \leq M$. The usage of a_{i_j} is to define the the range and size of the parts. To find the a_{i_j}, it is similar to algorithm 1 but we need to change "0" in line 1 to "b_{i-1}" , and the 4^m in lines 3 and 4 to "b_i".

After building the sub-arrays in each PCnode, it is ready for answering query. For a given query q and error k, both q and k will be boardcasted to every PCnodes. For every PCnode, the PCnode$_i$ carries out the CSTASMDFS algorithm, mentioned in section 2.3.5, on its own sub-array SA_i and return the results back to user.

You may notice that there will be some duplicated results, since the same match may be found on more than one PCnodes. We explain this by an example. Assume that there are two PCnodes in the PC cluster and the suffix array A is divided into two sub-arrays: SA_1 and SA_2. SA_1 contains the suffices starting with "aaaaa" and SA_2 contains the suffices starting with "ttttt". Moreover, the DNA sequence S contains the sequence "aaaaattttt" at position p. The query q is "aaaaattttt" with k = 1 and is divided into two subqueries: $q_1 = aaaaa$ and $q_2 = ttttt$. PCnode$_1$ contains SA_1, it finds a match of q_1 at position p in S in candidate selection phase and will locate the candidate at region $[p-1...p+11]$. In the verification phase, PCnode$_1$ will report that there is a match at p. Similarly, PCnode$_2$ contains SA_2, it will also find the same candidate at region $[p-1...p+11]$ using q_2 in candidate selection phase. So, PCnode$_2$ will also report that there is a match at p after verification phase.

3.3.2. Data Partitioning (DP) Algorithm. Besides partitioning the index, we can also partition the data (the long DNA sequence) S. The idea of Data Partitioning (DP) algorithm is every simple. Assume that there are N PCnodes in the cluster, we divide S into N sub-sequences, S_1, S_2, ... S_N. Each PCnode is responsible for one of the sub-sequences, i.e. the PCnode$_i$ get the S_i and builds the suffix array A_i for S_i by directly applying the algorithm described in Section 3.1.1. Then, A_i will be stored in PCnode$_i$.

To answer a query, we directly apply the CSTASMDFS algorithm to solve the problem. Given a query q and an error k, at PCnode$_i$, we apply the CSTASMDFS algorithm to search the pattern q with error k on its local suffix array A_i. The results will be reported to the users.

However, there may be some matches locating across the two sub-sequences. We called this case the cross boundary case. So, for each cross boundary, between sub-sequences S_i and S_{i+1}, we pick the sequence s contains the last $|q|+k-1$ of S_i and concatenate with first $|q|+k-1$ of S_{i+1}. We

use the dynamic programming algorithm to verify whether there exists a match in s.

4. Experiments

In this section, we will describe the experiments related to the algorithms mentioned in section 3. In all the experiments, the DNA sequences are human genome downloaded from DNA Data Bank Japan (DDBJ) and the queries used in all the experiments are randomly picked from the DNA sequences.

4.1. Suffix Tree vs Suffix Array

In this part, we evaluate the performance of Hunt's suffix tree [8], suffix array without QLT and suffix array with QLT (m=12) based on the external memory model. In the experiment, the size of the DNA sequence is 350Mbp, and we use a PC with P4 2GHz CPU and 512M RAM. The OS is Mandrake Linux 8.2 with kernel 2.4.19. The index size of the Hunt's suffix tree and suffix arrays is 7.8G and 1.4G respectively, and the index building time is 3.5hr and 0.89hr respectively. Then, we tested different query lengths (100, 250, 1000) and two error rates (10%, 5%). For each query type, we got the statistics by issuing a batch of 20 queries. The algorithm for ASM problem used in this experiment is the CSTASMDFS (referring to section 2.3.5). To investigate the performance for external memory model, the indexes are located in the hard-disk and the algorithm only load the blocks of the indexes if necessary . In the case of Hunt's suffix trees, there are total 30 sub-trees. We run CSTASMDFS on the sub-trees one by one and collect all the results. The performance of the indexes is shown in Figure 2(a). We found that suffix array with QLT performs the best. It is faster than Hunt's suffix tree 10 to 30 times and faster than suffix array about 100%. It is because the random IO effect discussed in section 3.1.2. This can be verified in Figure 2(b), which shows the percentage of IO time for each index structure. We found that Hunt's suffix tree requires much more IO time than others.

4.2. Performance of Quick Lookup Table (QLT)

In this section, we will investigate the performance of QLT. In this experiment, we use a suffix array with different size of QLT for 350Mbp DNA sequence on same machine used in the previous experiment. We tested the suffix array with different size of QLT (m=0, 4, 6, 8, 12) and different type of queries – two different query lengths (100, 1000) and two different error rates (10%, 5%). Referring to Figure 3, we found that the QLT can improve the performance of suffix array by around 100%. Moreover, it is good enough for m between 8 and 12. We probably cannot further reduce the random IO for simulation of treenodes with $m > 8$.

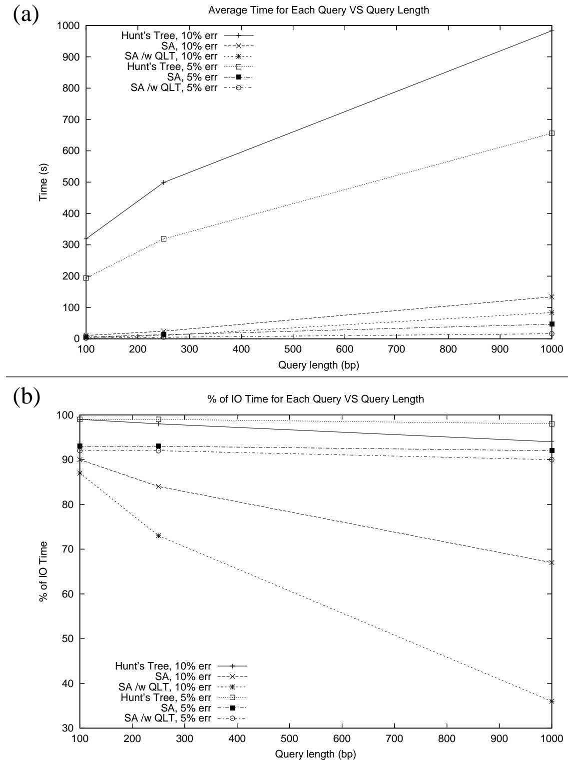

Figure 2. Performance of Hunt's Suffix Tree and Suffix Array (with and without QLT)

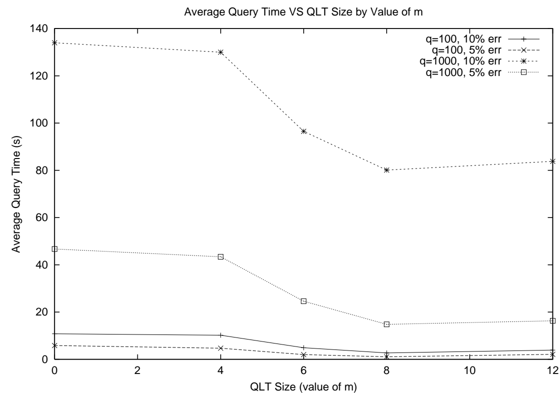

Figure 3. Performance of QLT

4.3. IP vs DP

In this section, we will present the experimental results about the two parallel computing approaches for DNA indexing – IP (Index Partitioning) and DP (Data partitioning). The experiment was carried out in a PC Cluster with 5 PC-nodes, 1 PCnode for master PCnode and the remaining 4 PCnodes for indexing and answering the queries. Each PC-

node contains a dual PIII 800MHz CPU with 1G RAM and the OS is RedHat Linux 7.2 with kernel 2.4.7.

In this experiment, we first build the index structures with QLT ($m = 12$) for the two parallel approaches and both of them required around 1.5 hour to construct the indexes. We tested different query lengths – 100, 250 and 1000 and different error rates – 5%, 7.5% and 10%. For each type of queries, we get the statistics by issuing a batch of 50 queries. Figure 4(a) shows the performance for candidate selection phase, the average time for each PCnode required to finish the candidate selection phase for one query. We found that DP is always slower than IP in candidate selection phase. The reason is due to the number of treenode access for the DFS in the candidate selection phase and this can be verified in Figure 4(b) which shows the average number of treenode access for DFS in candidate selection phase. We found that the more number of treenode access, the more time required in the candidate selection phase.

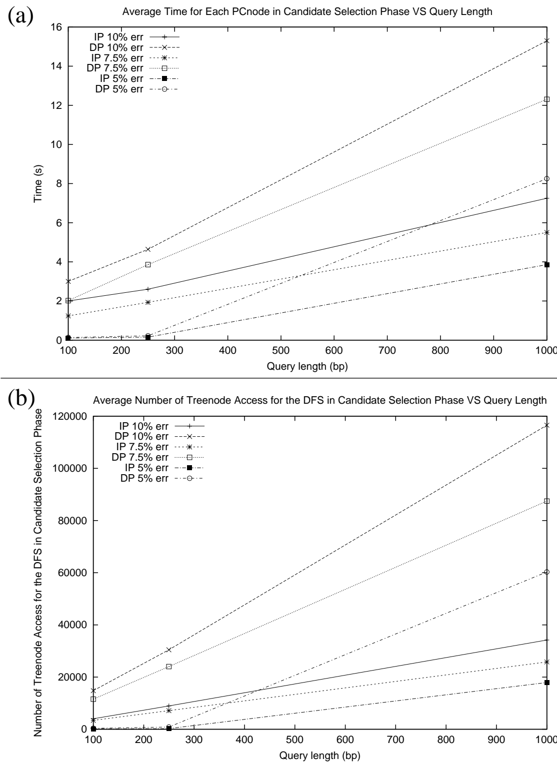

Figure 4. Candidate Selection Phase

Figure 5(a) shows the average time required for each PCnode in verification phase. We found that DP performs better than IP because the length of candidate regions of DP is shorter. Figure 5(b) shows the average total length of candidate regions for each PCnode per query and we found that the length of candidate regions is related to the time for verification phase.

Figure 6 shows the average response time for different

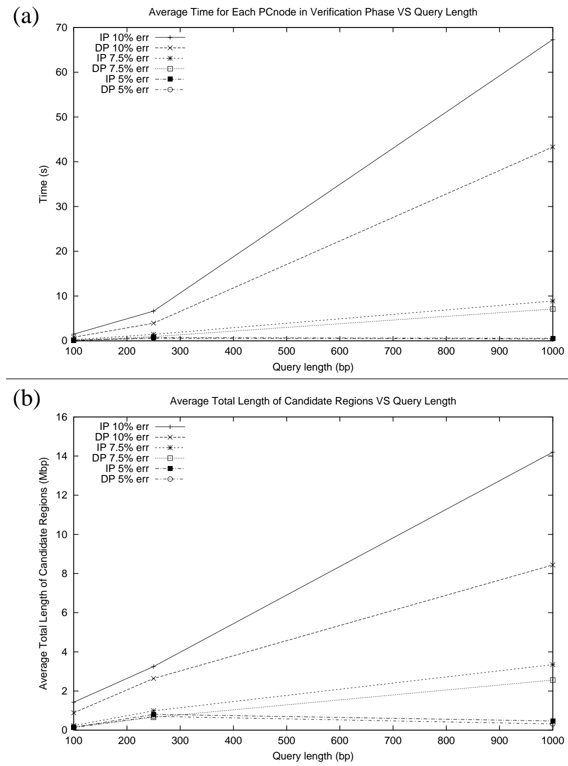

Figure 5. Verification Phase

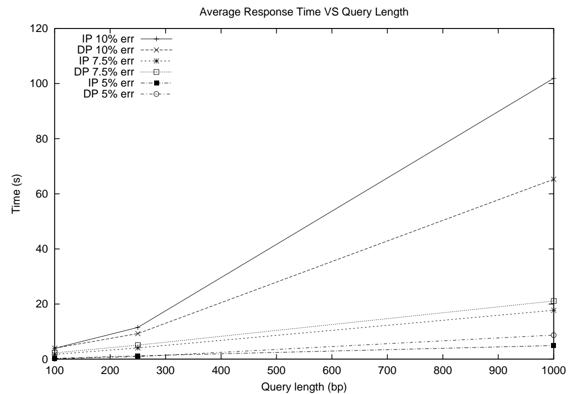

Figure 6. The average response time of the two parallel computing approach

query types on the two parallel computing approaches. We found that, DP performs better than IP when the error rate is high (10%). However, the IP performs better than DP in low error rate (5%). Roughly speaking, at higher error rate, verfication phase dominates the overall performance (comparing Figure 4(a) and Figure 5(a)). IP generates more candidates than DP in each PCnode and causes IP requiring more verification time in verification phase.

5. Conclusion and Future Works

In bioinformatics, researchers often want to do approximate string matching on huge DNA sequences. In this paper, we want to solve the approximate string matching (ASM) problem in a practical way. So, we consider the use of a suffix array or suffix tree index which are suggested to be very good for ASM problem in [16]. In the experiment, it shows that suffix array performs much better than suffix tree for external memory model. Moreover, we introduce the quick lookup table (QLT) which can greatly improve the performance of suffix array. Besides, due to the fact that DNA sequences are very long, it may not be practical to use a single machine for solving the problem. So, we propose two parallel approaches which are IP (Index partitioning) and DP (data partitioning). Generally speaking, DP is better than IP if the error k of ASM problem is large. Moreover, DP is more scaliable in terms of the length of DNA sequence data.

From the experiments, we found that the bottleneck is at verification phase when the error value, k, is high. We may consider some better candidate selection or verification techniques to reduce the number of candidates or verification time. One method may be the *Hierarchical Verification Algorithm* suggested in [17].

In IP, there is duplication of work in verifying candidates because same candidates may be found in more than one PCnodes in the cluster. So, it is possible to develop methods to reduce the duplication of work.

In bioinformatics, people may use other metrics (e.g. a score matrix) to measure the similariry of patterns in the ASM problem. It may be feasible to extend our CSTAS-MDFS algorithm to support some of these metrics such as scoring matrix for the ASM problem by applying the idea in [21].

References

[1] S. Altschul, W. M. W. Gish, E. W. Myers, and D. Lipman. A basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[2] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[3] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. of the 7th annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391, 1996.

[4] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[5] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *ACM*, 46(2):236–280, 1999.

[6] H. Gary and N. Prywes. Outline for a multi-list organized system. In Annual Meeting of the ACM, 1959. paper 41, ACM, New York.

[7] D. Gusfield. *Algorithms on Strings, Trees, and Sequenes: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[8] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequence. In *Proc. VLDB Conference*, 2001.

[9] H. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *ACM SIGMOD Conference on Management of Data*, pages 403–414, 2000.

[10] D. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[11] U. Manber and G. Myers. Suffix arrays: A new method for on-line string string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[12] E. M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the A.C.M.*, 23(2):262–272, 1976.

[13] G. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–317, 1991.

[14] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[15] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998.

[16] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[17] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. *Algorithmica*, 30(4):473–502, 2001.

[18] G. Navarro, E. S. R. Baeza-Yates, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[19] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proc. on CPM*, number 1848, pages 350–363, 2000.

[20] W. R. Pearson and D. J. Lipman. Improved tools fr biological sequence comparison. In *Proc. Natl. Academy Science*, volume 85, pages 2444–2448, 1988.

[21] E. Rocke. Using suffix tree for gapped motif discovery. In *Proc.* 11th *Annual Symposium on CPM*, number 1848, pages 335–349, 2000.

[22] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[23] E. Ukkonen. Finding approximate patterns in strings. *J. Algo.*, 6:132–137, 1985.

[24] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[25] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[26] H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):63–78, 2002.

[27] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conf*, pages 153–162, 1992.

[28] S. Wu and U. Manber. A fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.