**Biological Sequence Assembly and Alignment**

**PREPARED AND PRESENTED BY STUDENTS OF**

* M.Sc. Computer and Information Science 4th Semester of University of Science and Technology College, University of Calcutta
* HIRAKJYOTI BANEERJEE
  + Roll No. – 91-CIS-091012
  + Registration No – 0007312 of 2006-2007
  + DIPENDU GHOSH
  + Roll No. – 91-CIS-091007
  + Registration No – 0000439 of 2006-2007

**Under the Supervision of:** DR. DEVADATTA SINHA

**INTRODUCTION**

Computing technologies have played an increasingly important role in biology since the launch of Human Genome Project (Carol and Robert,1996).Parallel computing, which acts as an effective way to speed up biological computing, has been used in many biological applications. Sequence assembly and sequence alignment are the most computing intensive parts of biological computing.

**SEQUENCE ASSEMBLY**

Sequence assembly, also called fragment assembly, refers to aligning and merging fragments of a much longer DNA sequence in order to reconstruct the original sequence. This is needed as DNA sequencing technology cannot read whole genomes in one go, but rather small pieces between 20 and 1000 bases, depending on the technology used. Typically the short fragments, called reads, result from shot gun sequencing genomic DNA, or gene transcripts(ESTs).

**SEQUENCE ALIGNMENT**

There are innumerable biological sequences with unknown structure and function. The alignment of these sequences to known sequences will yield insight into the unknown sequences if the two are similar. Sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Sequence alignment can be further divided into multiple sequence alignment (MSA) and pair wise sequence alignment. The main purpose of an alignment is to propose homologies between sites in two or more sequences, but it is also a necessary step in judging homology between sequences or genes.

**LARGE SCALE SEQUENCE ASSEMBLY**

Sequence assembly is used to recover the fragments that are broken from DNA sequences and assemble them into the original sequences. Currently, the most widely used approach for breaking DNA sequences is whole genome shotgun (WGS), which is less expensive and quicker than other approaches. The WGS fragments the genome into many pieces of various sizes. This fragmentation can be done in several ways, such as physically shaking the DNA and cutting it with restriction enzymes.

**EXAMPLE OF WGS**

* Genome:

ATGC**GTAG**CTGTAGTGATCGAGGTCCAA**GTAG**CTGT

* *Reads from first copy:*

**ATGCGTAG,** CTGTAGTG, ATCGAGGT, CCAAGTAG,

* *Reads from second copy:*

**GTAGCTGT,** AGTGATCG, AGGTCCAA, **GTAGCTGT**

Each copy is broken into many reads.There are only two copies of the genome, and all the reads are of same size. There is no sequencing error, and all the nucleic acids have been identified. From the previous example we can see that the reads can not be assembled from one copy into the original genome because of lack of the information about their relative positions, called “context” . We can see an example of this information by observing that the suffix of the first read from the first copy **“ATGCGTAG”**, and the prefix of the first read from the second copy **“GTAGCTGT”**, are the same. This overlap between the two reads can let them be joined into **“ATGCGTAGCTGT”.**

**PROBLEM FOR SEQUENCE ASSEMBLY**

A very challenging problem for sequence assembly is the “repeat” problem. That is, the assembler cannot distinguish well between the overlap and the repeats of reads. In the previous example we can see that **“GTAG”** is an overlap betweenthe two reads given, but it is also a repeat. Although the suffix of the first read from the first copy **“ATGCGTAG”** is the same as the prefix of the last read from the second copy, **“GTAGCTGT”,** this is *not* the overlap we have seen between the reads **“ATGCGTAG”** and **“GTAGCTGT”.** An assembly error will be produced if the two reads are joined together with repeat rather than overlap.

**EULER SEQUENCE ASSEMBLY**

The Euler sequence assembly approach was proposed by Pavel A. Pevzner . In the Euler sequence assembly approach, tuples are the minimal units to be assembled, rather than the reads. Tuples are generated from reads. Tuples from one read are all the substrings of that read with the same length. All the tuples generated form a debruijn graph. The vertices of the graph are the tuples. Supposing the length of a tuple is *l*, if the last *l-1* nucleotide acids of one tuple are the same as the first *l-1* nucleotide acids of another tuple, there will be a directed edge in the graph which connects these two adjacent tuples. The Euler assembly approach is to find all the Euler paths in the graph. The core of the Euler approach is the consistency analysis rule which solves the problems of path selection for branches when looking for Euler paths in a graph.

**PESA SEQUENCE ASSEMBLY ALGORITHM**

The PESA (Parallel Euler Sequence Assembly) proposes an effective parallelization of the Euler sequence assembly approach that includes data distribution and computation distribution. Tuples are generated from all the reads and stored in a distributed hash table. This table is evenly distributed over multiple computing nodes, and each node is responsible for its own part of the hash table. We use the djb2 hash algorithm to calculate the hash values for all tuple strings. Given a tuple string *s,* we calculate its hash value *h* = djb2(*s*). Supposing the number of computing nodes is *p* and the size of the hash table is *t;* The size of the partial hash table on each node is *t/p.* The number of the computing node to which *s* will be assigned is *h%(t/p).* Each tuple will be stored in the corresponding partial hash table on some computing node. After storing all the tuples in the hash table, we need to calculate the multiplicity of each tuple, which will determine how many times the tuple will appear in the final contigs. Only when the number of times that each tuple is visited equals its multiplicity will the assembly finish.

* The parallel assembly algorithm is described as follows:
* *Input:* hash table and reads
* *Output:* contigs

***1.*** *Take the first tuple t from the local hash table whose counter is bigger than 0. t is an initial contig.*

***2.*** *Look for tuples adjacent to t on the right. If there is only one such tuple, and this tuple is on the same computing node, join the tuple directly to t. If this tuple is on some other computing node, this computing node will communicate with the remote computing node to request this tuple. If the counter of this tuple is bigger than 1, it can be joined into the current contig. It is the responsibility of the remote computing node to decrease the counter for this tuple by 1. If the number of tuples adjacent to t is more than 1, apply a consistency analysis rule to determine if there exists one, and only one, tuple that can be joined to the contig. If so, join the tuple to the current contig if it is located on the same computing node. If it is located on another computing node, communicate with that node to join the tuple with the current contig, if possible.*

***3.*** *Check if there are requests from other computing nodes and serve them if found.*

***4.*** *Repeat (2) and (3) until the current contig cannot be extended any longer to the right because of no more tuples being available, counters of adjacent tuples becoming 0, or consistency analysis failing to determine which path the current contig should follow.*

***5.*** *Look for tuples adjacent to t on the left, and deal with these tuples in the same way as described in (2), (3), and (4).*

***6.*** *If there are tuples in the local hash table whose counters are bigger than 0, go to (1). Otherwise, the assembly process on this computing node finishes and the contigs generated will be sent to the master computing node.*

***7.*** *The master computing node merges the contigs from all the nodes into the final contigs.*

* *The counter* for each tuple is initialized to be the multiplicity of the tuple, which describes how many times the tuple will appear in the final assembly result.

**LARGE-SCALE PAIRWISE SEQUENCE ALIGNMENT**

* The Large-Scale Sequence Alignment Algorithms are of two types:

(a) **Pairwise Sequence Alignment.**

(b**) Large Smith-Waterman Pairwise Sequence Alignment.**

**PAIRWISE SEQUENCE ALIGNMENT**

A pairwise sequence alignment is a scheme of writing one sequence on top of another, where the residues in one position are deemed to have a common evolutionary origin. If the same letter occurs in both sequences, then this position has been conserved in evolution. If the letters differ, it is assumed that the two derive from an ancestral letter (which could be one of the two or neither). Homologous sequences may have different lengths, though, which is generally explained through insertions or deletions in sequences. A pairwise sequence alignment is a scheme of writing one sequence on top of another, where the residues in one position are deemed to have a common evolutionary origin. If the same letter occurs in both sequences, then this position has been conserved in evolution. If the letters differ, it is assumed that the two derive from an ancestral letter (which could be one of the two or neither). Homologous sequences may have different lengths, though, which is generally explained through insertions or deletions in sequences. The Smith-Waterman algorithm is the optimal algorithm for pairwise biological sequence alignment. Two sequences to be compared are placed on the top and at the left of a similarity matrix SM, and each element of SM is calculated using equation:

SM[ i , j-1 ] + gp

SM[ i-1 , j-1 ] + ss

SM[ i , j ] = max

SM[ i-1 , j ] + gp

0

The value of one element of SM is determined by the left, left upper andupper, elements. *gp* is the gap penalty for inserting a space into the sequence *ss* is the value obtained by comparing two letters*.*

**LARGE SMITH-WATERMAN PAIRWISE SEQUENCE ALIGNMENT**

The approach here is to parallelize the pairwise sequence alignment by distributing the computations along the diagonals of the similarity matrix because the computation of the elements along one diagonal is independent of that along other. But there is dependency between neighboring diagonals because the calculation of one element value relies on the values of its left, left upper and upper elements. So the parallel alignment is executed in a wavefront way, computing first the values along the first diagonal in parallel, then along the second diagonal in parallel, and so forth, through the last diagonal in parallel. While the parallel calculation goes on, selection for the element with the highest score is also conducted, so that only the elements in the last diagonal are required to be remembered. This greatly reduces the memory requirement. The computing granularity can be changed to achieve the best performance according to the size of the similarity matrix and number of computing nodes available. Larger granularity will reduce the communication overhead; this will improve the performance but, at the same time, it is more likely to incur load imbalance, which will degrade performance.

**LARGE-SCALE MULTIPLE SEQUENCE ALIGNMENT**

* The **Large-Scale Multiple Sequence Alignment** algorithms are of two types:

(a) **Multiple Sequence Alignment**

(b) **Large-Scale Clustal W Multiple Sequence Alignment**

**MULTIPLE SEQUENCE ALIGNMENT**

The aim of multiple alignment is to find the sites that are homologous in all the sequences. Most of the methods are based on a concept called progressive alignment. The methods work by constructing successive pairwise alignments; initially, two sequences are selected and aligned by pairwise methods, and the alignment is fixed. Then, a third sequence is selected and aligned with the first alignment, and this procedure is repeated until all sequences are aligned. Most methods use a “guide tree” to determine the order in which to add sequences. Clustal W is one of the most widely used multiple sequence alignment programs. It performs well with protein and protein-coding DNA sequences but is less suited to sequences like rRNA sequences.

* The algorithm proceeds as follows:

1. *Construct a distance matrix of all the n(n-1) sequence pairs.*

*2. Construct a guide tree by the neighbor-joining method based on the distances from the previous step.*

*3. Progressively align sequences at the nodes in order of decreasing similarity, using sequence-sequence, sequence-alignment, and alignment-alignment alignments.*

**LARGE-SCALE CLUSTAL W MULTIPLE SEQUENCE ALIGNMENT**

The parallelization of Clustal W is conducted in all its three stages. In the first stage we have two levels of parallelization. The first level of parallelization lies in the calculation conducted on the whole *n(n-1)* pairs of sequences . The second level of parallelization lies in the pairwise sequence alignment. For the second stage, the calculation of the minimal value of each row in the distance matrix can be parallelized. Finally, for the third stage, we exploit the parallelism that exists in the iterative loops. Different computing granularities can be adopted at different stages of the Clustal W algorithm in its implementation in a distributed computing environment. All sequence alignments conducted in the first stage of Clustal W are independent. They can be therefore parallelized with smaller granularity so as to achieve maximal parallelism. But, for the second and third stages, the granularity should be bigger because there are more dependencies among the computing nodes as smaller granularity will bring about greater communication overhead.

**LOAD BALANCING AND COMMUNICATION OVERHEAD**

Load balancing is a big problem for parallel computing. Load on the computing nodes with limited computing power will make other processes wait; thus, resources are wasted and performance is degraded. Since the communication overhead is relatively high in distributed systems, the interaction frequency between computing nodes should be as low as possible. Factors that should be considered by these load distribution algorithms are communication overhead, and heterogeneity. Computing nodes have to request the *l-tuples* from other nodes. If one request is sent per *l-tuple,* there will be too many communicationrequests that will bring about a large amount of overhead, incurred mainly by the communication startup. So, these single requests should have to be incorporated to a request set that will be sent once.

**CONCLUSION**

It is a worthwhile exercise to conduct large-scale biological sequence assembly and alignment by parallel computing to take advantage of its vast storage and computing capability. Effective task scheduling and good computing granularity will boost the performance of biological applications running on a distributed computing environment. The load balancing discussed in here is static, i.e., load distribution takes place at the beginning of calculation. In future, investigation can be done on dynamic load balancing to adjust the load assigned to the computing nodes dynamically to be adaptable to the fluctuation of the distributed computing environment.

**REFERENCES**

* **Biological Sequence Assembly and Alignment**

Wei Shi1, Wanlei Zhou1 and Yi-Ping Phoebe Chen1,2

1 School of Information Technology

Faculty of Science and Technology, Deakin University

221 Burwood Hwy, Burwood, Vic 3125, Australia

2 Australia Research Council Centre in Bioinformatics

* **BIOINFORMATICS**

*Srinivas Aluru*

Iowa State University

* <http://en.wikipedia.org/wiki/Sequence_assembly>
* <http://en.wikipedia.org/wiki/Sequence_alignment>