8 Biological Sequence Assembly and Alignment

Wei Shi[1], Wanlei Zhou[1] and Yi-Ping Phoebe Chen[1,2]

[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

8.1 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. They have benefited immensely from parallel computing, and will benefit more from further research on parallel biological computing. This chapter introduces recent developments in parallel sequence assembly and alignment.

Sequence assembly (Myers, 2002), also called fragment assembly, is used to recover fragments and build the original sequences. This is a very important step in DNA sequencing. Due to the large amount of biological data, it will take a very long time to assemble the fragments of a middle-sized genome such as rice. A parallel Euler sequence assembly approach is discussed in this chapter. This approach stores all the genomic data in the form of distributed hash table so as to assemble this data as a whole. This eliminates errors incurred by approximately partitioning the fragments into

groups and assembling them into groups, as in other approaches. Further, our system can be run on networks of workstations or an supercomputers. It is particularly suitable for those having no access to supercomputers but to other computing resources such as workstations and PCs that are connected by a local network. This is the first effort to parallelize the Euler sequence assembly algorithm to assemble a large-scale genome. Sequence alignment (Liming et al., 2000) is also an important research area in bioinformatics. 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 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. The use of pair wise alignments is usually aimed at latter, while multiple alignments are used to assess homology between sites in many sequences.

Pairwise sequence alignment has the following significance:

- It can used to categorize and classify the DNA or protein sequences. The comparison between an unknown sequence and a known sequence can yield an insight into whether the two sequence are in the same category.
- Evolutionary relationships can be obtained by pair wise sequence assembly that identify the common parts of the two sequences.
- The DNA sequence can be aligned with the protein sequence to determine the function of the DNA sequence. The function of the unknown DNA sequence can also be known by comparing this sequence to known DNA sequences.
- The three dimensional structures of the protein sequence can be predicted by comparing it with the sequence of known dimensional structures.
- Pair wise sequence alignment is helpful in sequencing new genes. Multiple sequence alignment has shown significance in the following scopes:
- The shared homological regions of multiple sequences can be identified by taking advantage of MSA, which is very important for research in genetic diseases.
- MSA is often used to determine the consensus sequence of several aligned sequences.

- Secondary and tertiary structures of an unknown sequence can be predicted by comparing this sequence with multiple known sequences.
- MSA is a preliminary step in molecular evolution analysis using phylogenetic methods for constructing phylogenetic trees.

We parallelize the most used sequence alignment tools. For pair wise sequence alignment, we parallelize the Smith-Waterman algorithm (Temple and Michael, 1981). We also parallelize the Clustal W (Julie et al., 1994) multiple sequence alignment tool. These parallelizations, not only speed up biological computing, but also reduce the overall memory requirement by making use of the parallel computing technologies we develop.

The introduction to parallel sequence assembly and alignment is given in Section 1. Section 2 discusses issues with the parallel sequence assembly, including the Euler approach, parallel algorithms, determination of coverage, and implementation. Parallel pair wise sequence alignment is introduced in Section 3. Smith-Waterman algorithm and its parallelization are described. In Section 4, we present the parallelization of the Clustal W multiple sequence alignment tool. Load balancing and communication overhead, which play important roles in parallel computing, are discussed in Section 5. This chapter concludes with a discussion on the areas of research that need be focused on in the future.

8.2 Large-Scale Sequence Assembly

8.2.1 Related Research

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 (Weber and Myers, 1997). 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. The following is an example that illustrates the basic idea behind WGS. In real genome sequencing, both the genome and the read will be much longer than the genome and the read in the example.

Example:
Genome:
ATGC**GTAG**CTGTAGTGATCGAGGTCCAA**GTAG**CTGT
Reads from first copy:
ATGCGTAG, CTGTAGTG, ATCGAGGT, CCAAGTAG,
Reads from second copy:
GTAGCTGT, AGTGATCG, AGGTCCAA, **GTAGCTGT**

This example gives a simple and ideal whole genome shotgun. 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. Each copy is broken into many reads. We cannot assemble the reads from one copy into the original genome because of lack of the information about their relative positions, called "context". At least two copies of the genome are therefore required to have context information available among the reads. 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"**. 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. **"GTAG"** is an overlap between the two reads above, but it is also a repeat as shown in the genome in the example (the nucleic acid in boldface). 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. It should be noted that in real genome sequencing the suffix and prefix of two reads with overlap or repeat are not necessarily exactly the same. Our example tries to explain this problem in a simple way.

The "repeat" problem had not been solved well until a new sequence assembly approach was proposed (Pavel et al., 2001). This approach reduces the sequence assembly problem to a variation of the classical Eulerian path problem, which has been demonstrated to be more accurate than other approaches. This Euler approach has a polynomial computing complexity rather than the exponential complexity of other approaches. Other sequence assembly programs fall into the "overlap-layout-consensus" paradigm, which was abandoned in Pevzner's approach. The following is a description for the three stages of the "overlap-layout-consensus" paradigm (Myers, 2002; and Kececioglu and Myers, 1995):

- Overlap – finding potentially overlapping fragments.
  The overlap problem is to find the best match between the suffix of one sequence and the prefix of another. If no sequencing errors occur, simply find the longest suffix of one string that exactly matches the prefix of another string. Because errors often occur in the process of sequencing, a common practice is to use the filtration method and to filter out pairs of fragments that do not share a significantly long common substring.
- Layout – finding the order of fragments.
  Many algorithms select a pair of fragments with the best overlap at every step. The selected pair of fragments with the best overlap score is checked for consistency. If this check is accepted, the two fragments are merged. At later stages of the algorithm, the collections of fragments (contig) – rather than individual fragments – are merged. The difficulty with the layout step is in deciding whether two fragments with a good overlap really overlap (i.e., their differences are caused by sequencing errors) or represent a repeat in a genome (i.e., their differences are caused by mutations).
- Consensus – deriving the DNA sequence from the layout.
  The simplest way to build a consensus is to report the most frequent character in the substring layout that is implicitly constructed after the layout step is completed. The weakness of this paradigm is that it cannot effectively solve the problem of fragment repeat, i.e., it cannot distinguish between fragment overlap and fragment repeat. It is a NP-hard problem to assemble the fragments under this paradigm. The software under this paradigm includes Phrap (Green, 1999), TIGR (Sutton et al., 1995), CAP3 (Huang and Madan, 1999), and Celera Assembler (Bonfield et al., 1995).
- Phrap – Phrap ("phragment assembly program", or "Phil's revised assembly program") is a program for assembling shotgun DNA sequence data. Some of its key features are: (1) allowing use of the entire read, not just the highest quality part; (2) using a combination of user supplied and internally computed data quality information to improve accuracy of assembly in the presence of repeats; (3) constructing contig sequences as a mosaic of the highest quality parts of reads.
- TIGR – The TIGR Assembler is a sequence fragment assembly program building contigs from small sequence reads. It uses a greedy algorithm and heuristics to build contigs, find repeat regions, and target alignment regions. Sequence overlaps are detected and scored using a 32-mer hash. Sequence alignment and merging is done using a Smith-Waterman algo-

rithm. Gap penalties and score values corresponding to the bases and their quality values are predefined and hard coded into the program.
- CAP3 – CAP3 uses base quality values in the computation of overlaps between reads, construction of multiple sequence alignments of reads, and generation of consensus sequences. It also uses forward-reverse constraints to correct assembly errors and link contigs.
- Celera Assembler – Celera Assembler accepts an overlap only if there is no other sequence in the genome with $\geq 94\%$ sequence identity. Accordingly, fewer true overlaps are accepted in the initial assembly stage. In later stages of the Celera assembly, contigs are linked together by using mate-pair information, and the resulting gaps are then filled by various methods that may use sequences not included in the initial stages.

It is well known that one of the challenges to biological computing is the large amount of biological data and the colossal computing capacity required to process this data. Although Euler sequence assembly algorithm has a lower computing complexity than other approaches, it still needs a lot of time to assemble those biological fragments for small or middle sized genomes. Parallel computing is an efficient way to solve computing-intensive problems, and sequence assembly has shown good parallelism that can be exploited (Rajkumar, 1999; Terry et al., 2003; and Wei and Wanlei, 2003). In fact, parallel computing has been used in the current sequence assembly, for example, in sequencing the genome of the human being in the Human Genome Project. Many computing nodes have participated in the process of assembling fragments from the human genome. But this is not "real" parallel computing for sequence assembly because, in this approach, all fragments have to be first partitioned into many groups whose size is suitable for assembling in a single computing node. Fragments from one group can be assembled only with other fragments within the same group. So, each computing node can assemble the fragment only from perspective of a group, and not the whole genome. The partition of the fragments is conducted sequentially and may produce errors. These errors will result in incorrect sequence assembly because the assembly is confined to individual groups and cannot cross these groups.

But, in our approach, the parallel sequence assembly is carried out by each computing node from the view of the whole genome. Each fragment can be visited by each computing node. Each node can assemble any fragment into its local assembly result. The assembly is conducted by all the computing nodes in parallel. This is a "real" parallel sequence assembly approach because it is genome oriented, not group oriented. We are the first to propose such a real parallel approach.

8.2.2 Euler Sequence Assembly

The Euler sequence assembly approach was proposed by Pavel A. Pevzner (Pavel et al., 2001). The main contribution of the Euler assembly approach is that it transforms the biological sequence assembly problem into an Euler path problem that has a polynomial solution, which is a solution to the notorious "repeat" problem.

In the Euler sequence assembly approach, tuples are the minimal units to be assembled, rather than the reads as in other approaches. Tuples are generated from reads. Tuples from one read are all the substrings of that read with the same length, which is normally 20. 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. Each path is in fact a contig. 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. The details for the consistency analysis rule can be found elsewhere (Pavel et al., 2001).

8.2.3 PESA Sequence Assembly Algorithm

Biological sequence assembly often costs a lot in computing time even for small or medium sized genomes because of the large magnitude of iterative computing. But most of the current assemblers are sequential programs. Biological data has to be partitioned before applying these programs to assemble the genome. The participation is conducted according to similarities. This process is not accurate. So errors could be introduced by the participation. These errors cannot be corrected by the assemblers. Thus, the sequential assembler cannot meet the requirements demanded by sequence assembly. The research on the parallel sequence assembler is just at its beginning. Our parallel sequence assembler is the first to use the parallel Euler approach. This section introduces the algorithm and implementation of our PESA sequence assembler.

Main idea

The PESA (Parallel Euler Sequence Assembly) algorithm we proposed is an effective parallelization of the Euler sequence assembly approach that includes data distribution and computation distribution. The data distribu-

tion is conducted first. Tuples are generated from all the reads and stored in a distributed hash table. A distributed hash table will take maximal advantage of the memory resources of a parallel computing platform. With more computing nodes or memory added to the computing platform, the hash table can accordingly become larger and accommodate more genome data. This table is evenly distributed over multiple computing nodes, and each node is responsible for its own part of the hash table. No single hash table containing all the data will eliminate the bottleneck for storing a large amount of genome data.

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 $h = \text{djb2}(s)$. 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. We use linked list to deal with the collision occurring in the hash table. Tuples with the same hash value will be put into the same linked list of the hash table in their processing order. 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. Multiplicities of all the tuples will be used to judge when the assembly process should finish. Only when the number of times that each tuple is visited equals its multiplicity will the assembly finish. This means all the tuples, not including their copies, have been assembled into contigs.

Based on the tuples stored in the local hash table, each computing node will start to assemble the tuples. 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. The details for the counter can be seen in Section 3.2. Our parallel sequence assembly approach will extend the contigs from both the right direction and left direction, as shown in (2) and (5). This will help increase the lengths of the contigs generated by individual computing nodes, and thus reduce the computation to be carried out by the merging process, which merges all the partial contigs from all computing nodes. The reduced computation is in fact distributed over each computing node and executed in parallel.

The computing nodes involved assemble the tuples according to their local hash tables and generate contigs in parallel. Each node needs to communicate with others because the tuples to be joined to the current contig will possibly lie in other computing nodes. Details of communications among the computing nodes can be seen from (2) and (3). After one tuple is processed, each node will serve the request from other nodes so as to ensure that other nodes will not waiting for a long time to receive the response. This will produce a better utilization of the parallel system.

Finally, the master computing node will collect all the contigs from the slave nodes to assemble them further into the final result. The master node has no difference from other nodes except for the merging process. The merging process will also use a consistency analysis rule to assemble the current contigs into longer ones. The process is very similar to the assem-

bly process discussed above. The tuples on both ends of every contig will be analyzed to see if they can be concatenated with other contigs. The merging process continues to assemble contigs until no more contigs can be joined.

Determination of coverage

The completion of the parallel Euler sequence assembly process depends on the coverage of the genome to be sequenced. Supposing the coverage of some genome is m, the number of a tuple that appears only once in the genome will ideally be m. The number of a tuple that appears more than once in the genome will ideally be a multiple of m. The number of a tuple appearing in the final assembly result should be the number of its appearing in the genome divided by m, i.e., multiplicity. We set a counter for each tuple that is initialized to its multiplicity. Each time a tuple is assembled, its counter will be decreased by 1. The parallel assembly process will finish when the counters of all tuples become zero.

In order to calculate the multiplicities of all the tuples, we have to know the coverage of the genome, which is in fact the number of copies of the genome to be sequenced. Generally, we can not get this information directly. In our approach, we calculated the coverage of the genome from statistics of all kinds of tuples with different numbers of their appearing in all the reads to be assembled. We define S_i as a set of tuples within which all tuples appear i times in all the reads. Among all the sets, there must exist a set $S_{coverage}$ in which all tuples have no repeats in the genome. The number of these tuples appearing in all the reads will be equal to the coverage of the genome. $|S_{coverage}|$ is larger than any other $|S_i|$, $i \neq coverage$ because in the genome the amount of tuples with multiplicity equal to 1 is more than the amount of tuples with multiplicity equal to 2, 3, or more. So, we can easily identify the set $S_{coverage}$ from all sets of tuples. The subscripting value of $S_{coverage}$ is just the coverage of the genome.

We compute statistics for four species from TIGR Benchmark (The Institute for Genomic Research, 2003). The result is shown as Figures 8.1 through 8.4 (the X axis represents the appearing number of tuples and the Y axis represents the number of tuples having some appearing number). From these figures, we can clearly see that there is a peak in each curve. The peak indicates the largest amount of tuples whose appearing number is just equal to the coverage of the genome. But there exists an error between experimental coverage of the genome and its real coverage, as discussed above. This is because some unqualified reads (too long or too short) have been removed from the chemical experiments for sequencing (Shamir and Tsur, 2002). So, the experimental coverage will be slightly smaller than

the real value. When implementing the system, we will augment the experimental coverage and use it to calculate the multiplicity of each tuple.

Our next work is to conduct large-scale experiments to demonstrate our parallel assembly approach. The experiments will be carried out at the Australian Partnership for Advanced Computing National Facility, which owns a powerful AlphaServer SC with more than 500 CPUs and 700 GB RAM and a Cluster with 600 CPUs and 150 GB RAM.

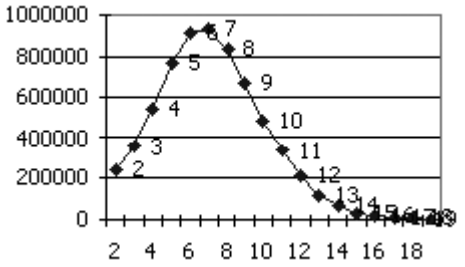

Fig. 8.1. Statistics of brucella suis

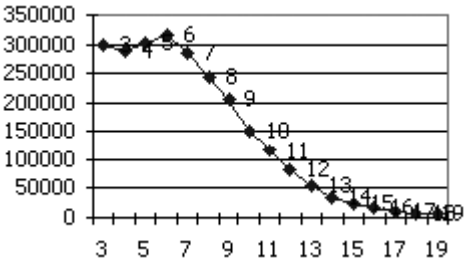

Fig. 8.2. Statistics of wolbachia sp

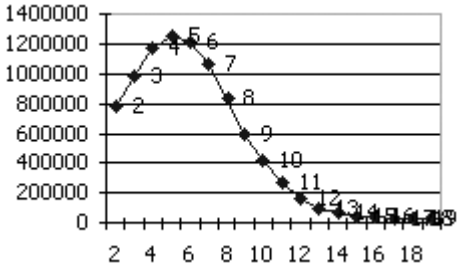

Fig. 8.3. Statistics of shewanella oneidensis

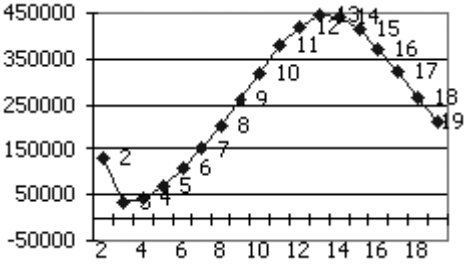

Fig. 8.4. Statistics of s*taphylococcus epidermis* RP62A

Implementation

The hash table is the most important data structure used by sequence assembly. The *l*-tuple to be accessed can be rapidly located by using the hash table. The *djb2* function, one of the best string hash functions, is used here. The hash table is evenly distributed over each computing node. Assuming that the size of the hash table is n and the number of computing nodes available is p (the computing nodes are numbered 0,1,2,…,p-1), the size of partial hash table on each node is $\lceil n/p \rceil$. For some *l*-tuple generated from some read, its hash value h can be calculated by the hash function with the string of *l*-tuple as input. This *l*-tuple will be assigned to the computing node numbered $h\%\lceil n/p \rceil$. A linear list is used to deal with the hash conflict. The hash table also contains the multiplicity of each *l*-tuple that determines how many times the *l*-tuple would appear in the assembly result. The repeat and non-repeat *l*-tuples will be assembled at the same time. The parallel assembly will finish when the multiplicities of all the *l*-tuples are decreased to zero.

8.3 Large-Scale Pairwise Sequence Alignment

8.3.1 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 se-

quences. Thus, a letter or a stretch of letters may be paired with dashes in the other sequence to signify such an insertion or deletion. In such a simple evolutionarily motivated scheme, an alignment mediates the definition of a distance for two sequences. One generally assigns 0 to a match, some negative number to a mismatch, and a larger negative number to an indel. By adding these values along an alignment, one obtains a score for this alignment. A distance function for two sequences can be defined by looking for the alignment that yields the minimum score. Luckily, using dynamic programming, this minimization can be effected without explicitly enumerating all possible alignments of two sequences.

The idea of assigning a score to an alignment and then minimizing over all alignments is at the heart of all biological sequence alignment. However, many more considerations have influenced the definition of the scores and made sequence alignment applicable to a wide range of biological settings. First, note that one may either assign a distance or a similarity function to an alignment. The difference lies more in the interpretation of the values. A distance function will define negative values for mismatches or gaps and then aim at minimizing this distance. A similarity function will give high values to matches and low values to gaps and then maximize the resulting score. The basic structure of the algorithm is the same for both cases. In 1981, Smith and Waterman showed that for global alignment, i.e., when a score is computed over the entire length of both sequences, the two concepts are in fact equivalent. Thus, it is now customary to choose the setting that gives more freedom in appropriately modeling the biological setting than one is interested in. In the similarity framework, one can easily distinguish among the different possible mismatches and also among different kinds of matches. For example, a match between two tryptophanes is usually seen to be more important than a match between two alanines. For amino acids, scoring matrices have been defined to assign a score to each possible pair of amino acids.

The Smith-Waterman algorithm (Temple and Michael, 1981) is the optimal algorithm for pairwise biological sequence alignment; it gives the optimal local alignment of two sequences in a mathematical sense. 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 (8.1):

$$SM\ [i, j] = \begin{cases} SM\ [i, j-1] + gp \\ SM\ [i-1, j-1] + ss \\ SM\ [i-1, j] + gp \\ 0 \end{cases} \tag{8.1}$$

The value of one element of *SM* is determined by the left, left upper and upper, elements. *gp* is the gap penalty for inserting a space into the sequence and *ss* is the value obtained by comparing two letters. It is often negative if the two letters are different. *gp* and *ss* can be reset by users. There will be one or more than one element with the maximum value after calculating values of all elements. Tracing back the path of each element it follows to get the maximum value, the algorithm gives one or more than one optimal local alignment between the two sequences.

8.3.2 Large Smith-Waterman Pairwise Sequence Alignment

Algorithm

The approach to parallelizing the pairwise sequence alignment is to distribute the computation along the diagonals of the similarity matrix, because the computation of element values 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 rememberered. 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. There will be a trade-off between the granularity and load balancing. 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.

Implementation

Supposing we align sequences X and Y with the lengths m and n, respectively, and partition the similarity matrix into blocks, as shown in Table 8.1. The number of columns of each block is k, and the number of rows is l. Each block will be assigned to a computing node. The blocks on different nodes will be calculated in parallel. A computing node needs to get $k+l+1$ elements from left, left upper, and upper blocks when it calculates some block, and $k+1$ elements are needed from another node. By partition-

ing the matrix into blocks, each computing node will calculate more data (a block) after receiving the adjacent elements at one time. So, the granularity is increased and communication frequency is reduced. The values of k and l should be set according to the network bandwidth available. But the granularity should not be too large or the parallelism would be damaged.

Table 8.1. The distribution of similarity matrix over computing nodes

Sequence X

Sequence Y						
	0	1	2	…	$\lceil m/k \rceil$-1	P1
	$\lceil m/k \rceil$	$\lceil m/k \rceil$ +1	$\lceil m/k \rceil$ +2	…	2$\lceil m/k \rceil$ -1	P2
	2$\lceil m/k \rceil$	2$\lceil m/k \rceil$+1	2$\lceil m/k \rceil$+2	…	3$\lceil m/k \rceil$ -1	P3
	…	…	…	…	…	…
	(t-1) $\lceil m/k \rceil$	(t-1) $\lceil m/k \rceil$+1	(t-1) $\lceil m/k \rceil$+2	…	t$\lceil m/k \rceil$-1	Pt
	t$\lceil m/k \rceil$	t$\lceil m/k \rceil$+1	t$\lceil m/k \rceil$+2	…	(t+1) $\lceil m/k \rceil$-1	P1
	…	…	…	…	…	…
	($\lceil n/l \rceil$-1) $\lceil m/k \rceil$	($\lceil n/l \rceil$-1) $\lceil m/k \rceil$+1	($\lceil n/l \rceil$-1) $\lceil m/k \rceil$+2	…	$\lceil n/l \rceil$ $\lceil m/k \rceil$-1	P$\lceil n/l \rceil$ % t

8.4 Large-Scale Multiple Sequence Alignment

8.4.1 Multiple Sequence Alignment

Multiple sequence alignment is known to be NP-complete. Given a number of sequences of symbols from an alphabet, the aim is to build an alignment matrix that maximizes some function. Gaps may be introduced between symbols, and in some multiple sequence alignment formulations, the objective function includes a measure of the number and lengths of gaps.

The aim of multiple alignment is to find the sites that are homologous in all the sequences. This is a very active area of research and numerous approaches are being proposed, with varying degrees of performance depending on the nature of the sequences aligned. 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 (Julie et al., 1994) is one of the most widely used multiple sequence alignment programs. This does not mean that it is ideal for all situations; 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 $n(n\text{-}1)$ 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

There are three main parameters that can (and should) be varied when using Clustal W; the substitution cost matrix, the gap opening cost, and the gap extension cost. It is also possible to provide a user guide tree, skipping directly to step three in the procedure.

8.4.2 Large-Scale Clustal W Multiple Sequence Alignment

Algorithm

The parallelization of Clustal W is conducted in all its three stages. For the first stage, we will have two levels of parallelization. The first level of parallelization lies in the calculation conducted on the whole $n(n\text{-}1)$ pairs of sequences. Because the calculation on different sequence pair is completely independent, we get a very high degree of parallelism. The second level of parallelization lies in the pairwise sequence alignment. For achieving load balancing across all the computing nodes, we will partition all the computing nodes into groups with similar computing capacities. In the first level of parallelization, all sequence pairs will be distributed into the computing groups according to each pair's computing load, which can be estimated in terms of the length of each sequence in the pair. The second level of parallelization will be achieved in each group, i.e., parallel pairwise sequence alignment for each pair, conducted in the group the pair of sequences belongs to.

For the second stage, the calculation of the minimal value of each row in the distance matrix can be parallelized. Then, the minimal value of the matrix can be calculated according to each computing node's result. Finally, for the third stage, we exploit the parallelism that exists in the iterative loops.

Implementation

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 theirfore 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. Smaller granularity will bring about greater communication overhead, which will offset the benefit gained from parallelism.

8.5 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.

As for parallel sequence assembly and alignment, load distribution algorithms developed before can still be used to improve their performance. But they should be modified so as to be as effective as possible for load balancing in parallel biological computing. Factors this should be considered by these algorithms are communication overhead, and heterogeneity.

The interaction among computing nodes in a parallel sequence assembly algorithm is not trivial. Computing nodes have to request the l-tuples from other nodes. If one request is sent per l-tuple, there will be too many communication requests 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. A similar situation occurs in sequence alignment, and a similar approach can be applied.

8.6 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. This chapter gives strategies for parallelization of the Euler sequence assembly, pairwise sequence assembly,

and multiple sequence assembly and their implementation. Effective task scheduling and good computing granularity will boost the performance of biological applications running on a distributed computing environment. Good preliminary experimental results have been achieved when conducting parallel sequence assembly and alignment on clusters. However we should do more to take advantage of parallel computing. The load balancing discussed in this chapter is static, i.e., load distribution takes place at the beginning of calculation. In future work, we will investigate 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

Bonfield, J.K., Smith, K.F., and Staden, R. (1995) A New DNA Sequence Assembly Program. Nucleic Acids Research 23:4992-4999.

Carol, A. Dahl, Robert, L. and Strausberg (1996.1) Human Genome Project: Revolutionizing Biology Through Leveraging Technology. Proceedings of SPIE - The International Society for Optical Engineering, pp 190-201.

Green, P. (1999) Documentation for Phrap and Cross-Match (Version 0.990319). http://www.genome.washington.edu/UWGC/analysistools/Phrap.cfm

Huang, X., Madan, A. (1999) CAP3: A DNA Sequence Assembly Program. Genome Research 9:868-877.

Julie, D.T., Desmond, G.H., Toby, J.G. and Clustal, W. (1994) Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice. Nucl. Acid Res. 22: 4673–4680.

Kececioglu, J. and Myers, E. (1995) Combinatorial Algorithms for DNA Sequence Assembly. Algorithmica 13:7-51.

Liming, C., David, J. and Evgueni, L. (2000) Evolutionary Computation Techniques for Multiple Sequence Alignment. Proceedings of the IEEE Conference on Evolutionary Computation. San Diego 7:829–835.

Myers, E.M. (2002) Toward Simplifying and Accurately Formulating Fragment Assembly. Journal of Computational Biology 2(2):275-290.

Pavel, A., Pevzner and Haixu, T. and Waterman, S. (2001) An Eulerian Path Approach to DNA Fragment Assembly. Proceedings of National Academy of Sciences of the United States of America 98(17): 9748-9753.

Rajkumar, B. (1999) High Performance Cluster Computing. Vol.1, Architecture and Systems, Vol.2, Programming and Applications, Prentice Hall, Upper Saddle River, New Jersey, USA.

Shamir, R. and Tsur, D. (2002.5) Large Scale Sequencing by Hybridization. Proceedings of the Annual International Conference on Computational Molecular Biology, pp 269-277.

Sutton, G., White, O., Adams, M. and Kerlavage, A. (1995) TIGR assembler: A New Tool for Assembling Large Shotgun Sequencing Projects, Genome Science and Technology 1:9-19.

Temple, F.S. and Michael, S.W. (1981) Identification of Common Molecular Subsequences. J. Mol. Bio. 147:195–197.

Terry A Braun, Todd E Scheetz, Gregg Webster, Abe Clark, Edwin M Stone, Val C Sheffield, Thomas L Casavant (2003) Identifying Candidate Disease Genes with High-Performance Computing. The Journal of Supercomputing 26(1):7-24

The Institute for Genomic Research (2003) Benchmark Data for Genome Assembly. http://www.tigr.org/tdb/benchmark/

Weber, J. and Myers, G. (1997) Whole Genome Shotgun Sequencing. Genome Research 7:401-409.

Wei, S and Wanlei, Z. (2003.12) Large-Scale Biological Sequence Assembly and Alignment by Using Computing Grid, Lecture Notes in Computer Science, Springer-Verlag.