Computational genomics on the grid

Chen, Chunxi

2007

Chen, C. X. (2007). Computational genomics on the grid. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2468>

<https://doi.org/10.32657/10356/2468>

Nanyang Technological University

Downloaded on 26 Apr 2023 21:35:43 SGT

Nanyang Technological University

**Computational Genomics
on the Grid**

Ph.D Thesis

by

Chunxi Chen

Supervisor: Dr. Bertil Schmidt

**Division of Information System
School of Computer Engineering
Nanyang Technological University
Singapore 639798**

January 2007

Abstract

The post-genomic era is characterized by large amount of data available from sequencing projects. The large size of biological data sets, inherent complexity of biological problems and the ability to deal with error-prone data all result in large run-time and memory requirements. Using parallel computing to solve computational genomics problems efficiently is therefore widely used approach.

Computational grids are a new parallel and distributed computing paradigm that gains high computing power by combing existing computational resources instead of building a new one. Using computational grids to solve computational genomics problems is a promising and challenging research field. Because computational genomics problems are typically of large-scale, there is a great potential that they can benefit from this type of processing power. However, to achieve good performance in a grid environment with its heterogeneous nature, careful resource scheduling and problem partitioning is needed to efficiently “gridize” an application.

In this thesis, we are investigating algorithms and techniques how to efficiently map computational genomics applications onto computational grid systems. In particular, we have studied the following two important applications:

- 1) Alignment of long genomic sequences;
- 2) Construction of genome-size suffix trees.

Comparing DNA sequences is a common and often repeated task in molecular biology. Existing algorithms still have problems with aligning long sequences. New solutions considering the tradeoff between speed and sensitivity are being demanded. We present an algorithm to compute the optimal and near-optimal non-intersecting alignments of two

sequences in linear space and quadratic time. In order to reduce its runtime significantly, we are using a hierarchical grid system as the computing platform. Static and dynamic load balancing approaches are investigated in order to achieve efficient mapping onto a hierarchical grid architecture. A new dynamic load balancing approach named scheduler-worker is proposed and evaluated. By comparing with other genome alignment tools we show that our approach can produce high quality results.

The suffix tree is a key data structure for biological sequence analysis, since it permits efficient solutions to many string based problems. Constructing large-size suffix trees is challenging because of high memory overheads and poor memory locality. Even though efficient suffix tree construction algorithms exist, their run-time is still very high for long DNA sequences such as whole human chromosomes. We have therefore developed a new linear time parallel algorithm for suffix tree construction. This algorithm uses a new data structure called the *common prefix suffix tree* (CPST). The main advantage of our new algorithm is that it can achieve linear speedups for both PC clusters and grid systems.

Acknowledgement

I would like to express my deepest gratitude and respect to the following people.

First and foremost, I would specially thank my supervisor, Dr. Bertil Schmidt, for his remarkable guidance, heuristic advice and encouragement in my research work. My research work benefits a lot from his enormous supportive efforts. Next, I would give my thankfulness to Dr. Stephen John Turner, Dr. Cai Wentong, Dr. Lee Bu Sung for their fruitful discussion with me. Their valuable advices, although critical sometimes, always inspired me. In addition, I also wish to say thanks to Mr. Lui Weiguo, Mr. Zeng Yi, Mr. Tang Min, Ms. Feng Yuhong, Ms. Wang Xiaoguang and all others who have supported my research work in one way or another.

Publications

- Book Chapters:

1. B. Schmidt, C.X. Chen, and W.G. Liu, Hierarchical Grid Computing for High Performance Bioinformatics, in Grids for Bioinformatics and Computational Biology, John Wiley & Sons, to appear in 2007.

- Journal Papers:

2. C.X. Chen, B. Schmidt, Constructing Large Suffix Trees on a Computational Grid, Journal of Parallel and Distributed Computing, in press, doi: 10.1016/j.jpdc.2006.08.004.
3. C.X. Chen, B. Schmidt, An Adaptive Grid Implementation of DNA Sequence Alignment, Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications, Vol. 21, No. 7, pp. 988-1003, July 2005.

- International Conference Papers:

4. C.X. Chen, B. Schmidt, Parallel Construction of Large Suffix Trees on a PC Cluster, Proceedings of Euro-Par 2005, Lisbon, Portugal, Springer, LNCS 3648, pp. 1227-1236, 2005. (Acceptance Rate: 31%)
5. C.X. Chen, B. Schmidt, Load Balancing for Hierarchical Grid Computing: A Case Study, Proceedings of 11th Annual International Conference on High Performance Computing (HiPC 2004), Bangalore, Springer, LNCS 3296, pp. 353-362, 2004. (Acceptance Rate: 22%)

6. C.X. Chen, B. Schmidt, Performance Analysis of Computational Biology Applications on Hierarchical Grid Systems, Proceedings of 2th International Workshop on Biomedical Computations on the Grid held in conjunction with the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), Chicago, IL, IEEE, 2004.
7. C.X. Chen, B. Schmidt, Computing Large-Scale Alignments on a Multi-Cluster, Proceedings of 5th IEEE International Conference on Cluster Computing (Cluster 2003), Hong Kong, pp. 38-45, 2003. (Acceptance Rate: 29%)
8. C.X. Chen, B. Schmidt, An Grid Implementation of Database Searching, Proceedings of 1st International BioEngineering Conference, Singapore, pp. 209-212, 2004.

Contents

Abstract	i
Acknowledgement	iii
Publications	iv
Contents	vi
List of Figures	xi
List of Tables	xix
1 Introduction	1
1.1 Overview	1
1.2 Motivation	2
1.3 Objectives	8
1.4 Approach and Methodology	9
1.5 Contributions	12

1.6	Synopsis of Thesis	13
2	Literature Review	15
2.1	Large-size Genome Analysis	15
2.1.1	Sequence Alignment	15
2.1.2	Constructing Large-size Suffix Trees	21
2.2	Grid Computing	27
2.2.1	Taxonomy of Parallel Computer Architecture	27
2.2.2	Introduction to Grids	31
2.2.3	Globus Toolkit—the De Facto Standard for Grid Computing . . .	37
2.2.4	MPICH-G2 – A Grid-Enabled MPI Implementation	40
2.3	Parallel Computing in Bioinformatics	43
2.3.1	Classification of Parallel Bioinformatics Applications	43
2.3.2	Fine-grained and Coarse-grained Parallel Approaches in Bioinformatics	45
2.3.3	BioGrid Projects	49
2.4	Summary	51
3	Hierarchical Grid Computing	52
3.1	Hierarchical and Heterogeneous Nature of the Grid	52
3.2	A Hierarchical Grid System	53
3.3	Efficiently Mapping Applications onto Grids	54

3.3.1	System-level Task Scheduling Strategies for Parallel Systems . . .	56
3.3.2	Progress and Problems of System-level Task Scheduling Strategies for Grids	57
3.3.3	Application-level Task Scheduling Strategies for Mapping Applica- tions onto Grids	58
3.3.4	Rules for Mapping Applications onto Grids	60
3.4	Parallel Computational Model	61
3.4.1	An Extension of the LogGP Model for a Heterogeneous Multi-Cluster	62
3.4.2	Time Analysis for Standard Master-Slave Paradigm	64
3.5	Summary	66
4	Aligning Long DNA Sequences	67
4.1	Pairwise Sequence Alignment	67
4.1.1	Smith-Waterman Algorithm	67
4.1.2	Optimal Alignments in Linear Space	70
4.1.3	Finding Near-optimal Alignments in Linear Space	72
4.2	Mapping the Application onto the Hierarchical Grid Architecture	76
4.2.1	Parallelization of Sequence Alignment	76
4.2.2	Mapping onto the Hierarchical Grid using Static Load Balancing .	79
4.2.3	Mapping onto the Hierarchical Grid using Dynamic Load Balancing	81
4.3	Performance Evaluation	86
4.3.1	Speedup and PDRDs	86

Ph.D Thesis	CONTENTS
4.3.2 Performance Comparison	89
4.3.3 Quality of Produced Alignments	90
4.4 Summary	97
5 Constructing Large-size Suffix Trees	99
5.1 Ukkonen's Linear Time Algorithm for Constructing Suffix Trees	99
5.1.1 High Level Description of Ukkonen's Algorithm	100
5.1.2 Suffix Extension Rules	100
5.1.3 Suffix Links and Skip/count Trick	101
5.1.4 Edge-label Compression	104
5.1.5 Two More Tricks	105
5.2 Constructing a Suffix Tree with CPSTs	107
5.2.1 CPST: Common Prefix Suffix Tree	107
5.2.2 Constructing CPSTs in Linear Time.	108
5.3 Constructing a Generalized Suffix Tree (GST) with CPSTs	118
5.4 Parallel Implementation of Constructing Large Suffix Trees using CPSTs	119
5.4.1 Input DNA Sequence	119
5.4.2 Prefix Distribution in DNA Sequences	119
5.4.3 Space Optimization	120
5.4.4 Mapping the Application onto the Hierarchical Grid Architecture	121
5.5 Performance Evaluation	122

5.5.1	Runtimes, Speedups and PDRDs on the Grid System	122
5.5.2	Performance for an Implementation with Data-servers	125
5.6	Adapting Operations on Suffix Trees to Distributed CPSTs	127
5.7	Summary	130
6	Conclusions and Future Work	132
6.1	Conclusions	132
6.2	Future Work	134
6.2.1	New Computational Model for Heterogenous Grid Systems	134
6.2.2	Multiple DNA Sequence Alignment	136
6.2.3	Space savings for CPSTs	137
6.2.4	Integrate CPST Constructing Algorithm with Important Bioinfor- matics Tools based on Suffix Trees	137

List of Figures

1.1	Growth rate of Genbank [9].	2
1.2	DNA Structure.	5
2.1	Global alignment between sequences S_1 and S_2	16
2.2	Edit distance between sequences S_1 and S_2	18
2.3	Local alignment between sequences S_1 and S_2	18
2.4	Dynamic programming matrix for aligning $S_1 = \text{GATATGTAT}$ and $S_2 = \text{GGATGATT}$. The score system is that assigns 1 to matches, -1 to mismatches and -2 for a space insertion or gap.	19
2.5	Searching 3-tuples for aligning $S_1 = \text{GATATGTAT}$ and $S_2 = \text{GGATGATT}$	20
2.6	The suffix tree of the sequence <i>cagca</i> \$.	22
2.7	The implicit suffix tree for the sequence <i>cagca</i>	23
2.8	Flynn's taxonomy of parallel computer architecture	28
2.9	MIMD's taxonomy	30
2.10	The layered grid architecture	33
2.11	Grid systems taxonomy.	35

3.1	The hierarchical parallel programming environment consisting of MPICH-G2, MPICH and Sun Grid Engine.	55
3.2	Master-slave paradigm	59
3.3	Hierarchical master-slave paradigm	60
3.4	Compromised master-slave paradigm	60
4.1	Example of the Smith-Waterman algorithm to compute the local alignment between two sequences ATCTCGTATGATG and GTCTATCAC.	69
4.2	Computing the similarity score in linear space.	70
4.3	Divide-and-Conquer method to compute the global alignment of sequence S_1 and sequence S_2 in linear space.	72
4.4	Find the midpoint.	72
4.5	Example of the computation of highest-scoring non-intersecting local alignments using $k=4$ and DNA sequences, substitution cost and gap penalties as in Figure 4.1.	75
4.6	Wavefront computation of the <i>similarity matrix</i> : (a) shift direction, (b) dependency relationship.	77
4.7	(a) Column-based division of an 8×8 matrix using 4 processors; (b) Wavefront computation for 4 processors, 8 columns and a 2×2 block size. The complete 8×8 matrix can then be computed in 7 iteration steps. Each processor P_i ($0 \leq i \leq 2$) except the last one (P_3) will send the cells in the most right column of the block it computes to the processor P_{i+1}	78
4.8	Parallel solution for determining the traceback in linear space on four processors, where SC_i stands for special column number i	79

4.9	Partitioning of the wavefront computation on the hierarchical grid architecture. The number of columns assigned to each cluster is determined by its performance to execute the Smith-Waterman algorithm using MPI. Inside a cluster, each processor (except the last one) will send the cells in the most right column of the block it computes to the next neighboring processor over the intra-cluster communication. The last processor in each cluster (except the last one, cluster3) will send the cells in the most right column of the block it computes to the first processor of the next neighboring cluster over the inter-cluster communication.	80
4.10	PDRD (<i>Performance degradation ratio under disturbance</i>) for the implementation using static load balancing.	81
4.11	Data transfer between <i>master</i> and <i>slave</i> for finishing a sub-job. The <i>slave</i> has to receive the upper row and left column from the <i>master</i> in order to start computing a new block. Afterwards the <i>slave</i> returns the right column and bottom row of this block to the <i>master</i>	83
4.12	Scheduler-worker technique.	84

- 4.13 Example of the scheduler-worker technique with the assumption that each cluster has only two processors. The *similarity matrix* is divided into blocks (denoted by the rectangles) consisting of cells. Each block has a certain number of rows of cells, while number of columns of cells in a block varies depends on the computing power of the processor computing the block. Each *worker* will report its computing performance to the *scheduler* every time it finishes a piece of work (such as at *send*). The *scheduler* judges whether a new job allocation is needed depending on all the computing performances. If a new job arrangement is needed, the *scheduler* will produce a new job allocation form and sends it to each *worker* (such as at *receive*). The *workers* implement the job rearrangement by exchanging data between *workers*. In this example, processors P_{01} is allocated with larger number of columns, some of which are computed by processor P_{00} and P_{10} before the *rearrange*. Therefore, processors P_{00} and P_{10} need to transfer the corresponding cells to processor P_{01} . These cells are indicated by bold lines (*a* and *b*) in the figure. In the same way, processor P_{11} need to transfer the cells on the bold line *c* to processor P_{10} and processor P_{20} need to transfer the cells on the bold line *d* to processor P_{11} 85
- 4.14 Execution times and speedups of the implementation, using static load balancing and without disturbance, on the grid system with different application-level inter-cluster bandwidths (8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s). 87
- 4.15 PDRD (*Performance degradation ratio under disturbance*) for the implementation using static load balancing. 88

4.16 Execution times and speedups of the implementation, using master-slave dynamic load balancing and without disturbance, on the grid system with different application-level inter-cluster bandwidths (8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s). The <i>similarity matrix</i> is divided into blocks of size 30000×30000 cells.	89
4.17 PDRD (<i>Performance degradation ratio under disturbance</i>) for the implementation using the master-slave technique.	90
4.18 Execution times and speedups of the implementation, using scheduler-worker load balancing and without disturbance, on the grid system with different application-level inter-cluster bandwidths (8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s).	91
4.19 PDRD (<i>Performance degradation ratio under disturbance</i>) for the implementation using the scheduler-worker technique.	92
4.20 Speedups of the implementation using scheduler-worker load balancing for aligning varying lengths of sequences.	92
4.21 Speedup comparisons of static, master-slave and scheduler-worker load balancing without disturbance for different application-level inter-cluster bandwidths (upper-left: 8MByte/s, upper-right: 1MByte/s, lower-left: 0.5MByte/s, lower-right: 0.3MByte/s).	93
4.22 PDRD comparisons of static, master-slave and scheduler-worker load balancing for different application-level inter-cluster bandwidths (upper-left: 8MByte/s, upper-right: 1MByte/s, lower-left: 0.5MByte/s, lower-right: 0.3MByte/s).	94

4.23	$Speedup \times (1 - PDRD)$ comparisons of static, master-slave and scheduler-worker load balancing for different application-level inter-cluster bandwidths (upper-left: 8MByte/s, upper-right: 1MByte/s, lower-left: 0.5MByte/s, lower-right: 0.3MByte/s).	95
4.24	Alignments of <i>M.pneumoniae</i> and <i>M.genitalium</i> by our grid-enabled system. The score scheme: match = 2, mismatch = -2, gapopen = -10, gapextension = -1.	98
4.25	Alignments of <i>M.pneumoniae</i> and <i>M.genitalium</i> by the block-partitioning method. The score scheme is the same as in Figure 4.24.	98
5.1	Extension $j \geq 1$ in phase $i + 1$. Walk up atmost one edge (labeled γ) from the edge of the path labeled $S[j - 1..i]$ to node A ; then follow the suffix link to B ; then walk down the path specifying substring γ ; then apply the appropriate extension rule to insert suffix $S[j..i + 1]$	103
5.2	The Suffix link and skip/count trick. In phase $i + 1$, substring γ has length ten. There is a copy of substring γ out of node A ; It is found three characters down the last edge, after four node skips are executed.	104
5.3	The left tree is a part of the suffix tree for sting $S = abcdefabcuvw$, with the edge-labels written explicitly. The right tree shows the edge-labels compressed.	105
5.4	The suffix tree of the sequence <i>accattgaagcgttaccagttat\$</i>	108
5.5	The CPSTs of the sequence <i>accattgaagcgttaccagttat\$</i> . using <i>compres</i> : <i>a</i> , <i>c</i> , <i>g</i> , and <i>t</i>	109
5.6	An example for the <i>compre suffix link</i>	110

5.7	$CPST(S, a)$, $CPST(S, c)$, $CPST(S, g)$, and $CPST(S, t)$ for the string $S = accattgaagcgttaccagttat\$$ with corresponding <i>compre</i> suffix links. . .	111
5.8	Illustration for Lemma 5.2.1.	112
5.9	Illustration for Lemma 5.2.2.	113
5.10	Inserting suffix 1 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$	115
5.11	Inserting suffix 2 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$	115
5.12	Inserting suffix 5 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$	116
5.13	Inserting suffix 6 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$	116
5.14	Inserting suffix 8 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$	117
5.15	Inserting suffix 9 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$	117
5.16	Illustration for Lemma 5.2.1. The $\nu(i)$ is a <i>valid</i> suffix for $CPST(S, \textit{compre})$. The insertion of the suffix $\nu(i)$ results in a new internal node with $CPST_pathlabel(A) =$ $S[\nu(i) + \textit{len}(\textit{compre})..l_i]$. The $\nu(i+1)$ and $\nu(j)$ are the first and last <i>valid</i> suffixes in the range (ν_i, l_i) . The $[\nu_j, l_j]$ is the next similar range as $[\nu_i, l_i]$	118
5.17	Data structure for CPST internal nodes.	121
5.18	Data structure for CPST leaf nodes.	121

5.19 Execution times (upper) and speedups (lower) of our implementation on the hierarchical grid system. The used sequences are substrings of the human chromosome NC_000001.4.	123
5.20 Execution times for using varying lengths of common prefixes. The used sequences are substrings of the human chromosome NC_000001.4. Five CPUs are used for four slaves (on four 2.60GHz CPUs) and one master (on another cluster).	124
5.21 PDRDs (Performance degradation ratio under disturbance) by an application running on one CPU that uses around 50% CPU time.	125
5.22 Data structure for CPST nodes in the implementation with <i>dataservers</i> . .	127
5.23 Runtimes of the implementation with <i>data-servers</i> for input strings with different lengths.	128
5.24 Work flow of the exact set matching using CPSTs.	129

List of Tables

1.1	Examples of applications based-on grids	7
2.1	Example applications of suffix trees in string computing.	24
2.2	Example applications of suffix trees in string computing.	24
2.3	Fine-grained Parallel Approach	46
2.4	Coarse-grained parallel approach in bioinformatics	48
4.1	Two pairs of Mycoplasma bacteria genomes which can't be found by PatternHunter II using its default score scheme: match = 1, mismatch = -1, gapopen = -5, gapextension = -1. The parameter '-0 64' is used in running PatternHunter II to optimize its alignments.	96
4.2	The top scores of alignments for aligning segments of length 1Mbp of the human chromosome X and the drosophila_melanogaster genome by our grid system and by PatternHunter II using the score scheme: match = 2, mismatch = -1, gapopen = -10, gapextension = -1. The parameter '-0 64' is used in running PatternHunter II to optimize its alignments.	97

5.1	The number of suffixes of the human chromosome NC_000001.4 of length 222,827,884 (only consider a , c , g and t) which start with 1-letter and 2-letter prefixes.	120
5.2	Operations on suffix trees and the kinds of traversals they require [26]. . .	129

Chapter 1

Introduction

1.1 Overview

The genomic sequence databases are growing at an enormous rate [9] (see Figure 1.1). This growth rate will continue since more and more genomic projects will be finished in the near future. As of September 2006, there are 422 published completed genomes, while 1665 genome sequence projects are ongoing worldwide [10]. As the data increases, so does the workload for processing, managing and sharing these data. Browsing and searching are expensive activities because extensive time is required to find appropriate information. Furthermore, with the continuing improvements in high-throughput genomic sequencing and the ever-expanding sequence databases, new advances in software programs for genomic sequence analysis are needed by the scientific community.

Computational grids are a new parallel and distributed computing paradigm that provides a resource for large scientific computing applications [66]. The large size of biological data sets, inherent complexity of biological problems and the ability to deal with error-prone data all result in large run-time and memory requirements. Therefore,

Figure 1.1: Growth rate of Genbank [9].

large-scale computational genomics problems can potentially benefit from that kind of computing power.

However, the development or adaptation of computational genomics applications for the grid environment is made challenging by the often heterogeneous nature of the resources involved and the facts that these resources have different computational power and may be connected by networks with widely varying performance characteristics. In order to achieve good performance in a grid environment, new parallel algorithms, careful resource scheduling and new problem partitioning are needed.

1.2 Motivation

For decades, biologists have tried to probe into how life evolved and how diseases or special phenotypes started by studying the flow of genetic information in a cell, this is, how

certain segments of DNA, the genes, are read and translated into proteins with specific function. Therefore, knowing the DNA structure, not only its biochemical structure but also the organization of the whole molecule as the genomes of an organism, would be the first step of such studies. Computational genomics, a subfield of computational biology that deals with the analysis of entire genome sequences would provide approaches for research on DNA. This field, established as independent discipline in the past few years, relies on extensive computational approaches to decode the information hidden behind billions of nucleotides.

DNA (deoxyribonucleic acid) is the genetic material. This is profoundly powerful statement to molecular biologists. To a large extent, it presents the answer to questions that have been wondered by philosophers and scientists for thousands of years: “what is the basis of inheritance?” and “what allows living things to be different from nonliving things?” Quite simply, it is the information stored in DNA that allows the organization of inanimate molecules into functioning, living cells and organizations that are able to regulate their internal chemical composition, growth, and reproduction. As a direct result, it is also what allows us to inherit our mother’s curly hair, our father’s blue eyes, and even our uncle’s too large nose. The various units that govern those characteristics at the genetical level, be it chemical composition or nose size are called *genes*.

Genes themselves contain their information as a specific sequence of nucleotides that are found in DNA molecules. Only four different base are used in DNA molecules: *guanine*, *adenine*, *thymine*, and *cytosine* (G, A, T, and C). Each base is attached to a phosphate group and a deoxyribose sugar to form a nucleotide (see Figure 1.2). The only thing that makes one nucleotide different from another is which nitrogenous base it contains. All of the information within each gene comes simply from the order in which those four nucleotides are found along lengthy DNA molecules. Complicated genes can

be many thousands of nucleotides long, and all of an organism's genetic instructions, its *genome*, can be maintained in millions or even billions of nucleotides.

A common theme throughout all biological systems and at all levels is the idea that structure and function are intimately related. DNA molecules within cells typically exist as double-stranded molecules was an invaluable clue as to how DNA might act as the genetic material. While the information content on each strand of a double-stranded DNA molecule is redundant and complementary. For every G on one strand, a C is found on its complementary strand and vice versa. For every A on one strand, a T is found on its complementary strand and vice versa. The interaction between G's and C's and between A's and T's is both specific and stable. Only the pairing between the nitrogenous bases G and C and the pairing between the nitrogenous base A and T have both the right spacing and interaction between their chemical groups to form stable *base pairs*.

Proteins are assembled from amino acids using information encoded in genes. Each protein has its own unique amino acid sequence that is specified by the nucleotide sequence of the gene encoding this protein. The process by which information is extracted from the nucleotide sequences of a gene and then used to make a protein is essentially the same for all living things on Earth and is described by the grandly named *central dogma* of molecular biology. Quite simply, information stored in DNA is used to make a more transient, single-stranded polynucleotide called RNA (ribonucleic acid) that is in turn used to make proteins. The process of making a RNA copy of a gene is called *transcription*. The messenger RNA (mRNA) serves as an intermediate between DNA and protein. Parts of the DNA are *transcribed* into transcripts that are processed to mRNA [6]. The process of converting that information from nucleotide sequences in RNA to the amino acid sequences that make a protein is called *translation* and is performed by a complex of proteins and RNA called *ribosomes*.

Figure 1.2: DNA Structure.

In a very real sense, any alignment between two or more nucleotide or amino acid sequences represents an explicit hypothesis regarding the evolutionary history of these sequences. As a direct result, comparisons of related protein and nucleotide sequences have facilitated many recent advances in understanding the information content and function of genetic sequences. For this reason, techniques for aligning and comparing sequences, and for searching sequence databases for similar sequences, have become cornerstone of bioinformatics. By aligning DNA sequences of entire genomes (i.e. including coding and non-coding regions), scientists will be able to identify important matched and mismatched regions. The “matches” may turn out to be functional homolog pairs, conserved regulatory regions or long repeats. Various sizes of “mismatches” can be easily detected in the process. SNP^{1.1} is one base pair mismatch in the middle of two matching regions. Large sized “mismatches” may be foreign fragments inserted into the genome

^{1.1}Single Nucleotide Polymorphism

by transposition, sequence reversal or lateral transfer from another organism. The information will help to detect important functional differences between pathogenic and non-pathogenic strains of the same species, figure out evolutionary distance between organisms, find out the regulatory regions of genes that may be preserved during evolution, and discover point mutations, deletions, reversions, insertions and duplications that may lead to diseases or special phenotypes.

The suffix tree is a compact trie of all suffixes over a string. It is a key data structure in the field of computational genomics, since it permits very efficient solutions to many string based problems. Examples include exact and approximate substring matching, the longest common substring problem and the maximal repetitive structures problem [74]. Consequently, many widely used large-scale computational genomics applications have achieved good performance using suffix trees, such as MUMmer [97], REPuter [98], and OASIS [113].

The output of sequence data from world-wide sequencing centers with constantly increasing sequencing capacities has been rising at an exponential rate for the past decade or two [17]. The first two publications of microbial whole genome sequencing projects were published in 1995. Only six and a half years later, there are almost 60 completed, annotated genomes available, along with draft analysis of several multi-cellular eukaryotes such as nematode, fly, man, weed, mouse, rat and fish. More still are currently underway. A summary of all completed and ongoing sequence projects can be found in [10]. With the continuing improvements in high-throughput genomic sequencing and the ever-expanding sequence databases, new advances in software programs for genomic sequence analysis are being demanded by the general scientific community.

Grid computing has been the most active research fields in the parallel and distributed computing community in the recent years. Distinguished from conventional parallel and

distributed computing, grid computing gains high compute power at low cost by combining existing computational resources instead of building a new one [66]. Thus, the computational grids can provide grid users with pervasive and inexpensive access to a wide variety of resources. Grid computing mainly focuses on resources sharing among geographically distributed sites and the development of innovative, high-performance oriented applications [66, 67]. A number of complex scientific applications have achieved good performance by using grid computing. We have categorized grid applications according to the category of grid systems they used (see Table 1.1). The research done in the thesis falls into the category “Distributed high performance computing”, since developing and adapting large scale applications on heterogenous grid systems are the most challenging.

Category		Characteristics	Examples
1	High throughput computing	Hunting for idle resources to increase aggregate throughput	Cryptographic problems [100] SETI@home [22]
2	Data intensive computing	Computation with information from many data sources	Digital sky survey [66] High-energy physics [110]
3	Distributed high performance computing	Large scale applications that need large amount of computing resources	DIS applications [114] Chemistry computation [118] Biological sequences analysis [42]
4	On demand computing	Local and Remote resources integration	Medical applications [124] Cloud detection [99]
5	Collaborative computing from multiple disciplines	Collaboration of multiple partners	Collaborative design [47]

Table 1.1: Examples of applications based-on grids

1.3 Objectives

The rapid increase of genomic sequences needs fast solutions to shorten the time of sequence analysis. Traditional approaches to sequence analysis techniques lead to long runtimes on sequential computers. Parallel and distributed computing is a widely used method to reduce this runtime significantly. The computational grid has emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale resource sharing innovative applications, and, in some cases, high-performance orientation. therefore, using computational grids to solve computational genomics problems is a promising and challenging research field. Our objectives are as follows.

- Many parallel algorithms for solving sequence analysis problems have been presented for different parallel architectures, such as systolic arrays, FPGAs ^{1.2} and SIMD^{1.3} architecture [128, 109]. However, computational grids have their own characteristics, such as heterogenous nature. Therefore, new sequence alignment algorithms have to be presented in order to run efficiently on the computational grids.
- Constructing suffix trees in parallel uses the idea of processing sub-trees independently. Once all the sub-trees have been constructed it is possible to merge them together to form a complete suffix tree. A similar approach has been recently used in [44] and [37]. Unfortunately, [44] only gives some actual experiments on a binary alphabet, which is not relevant in practice; [37] constructs suffix trees not in a cluster, but an SMP^{1.4} machine with quad CPUs and large memory. New data structures and the corresponding linear time algorithms have to be presented to

^{1.2}Field-Programmable Gate Array

^{1.3}Single Instruction Stream and Multiple Data Stream

^{1.4}Symmetric Multiprocessing

parallelize large-size suffix trees in grid environments.

- The development or adaptation of computational genomics applications for the hierarchical grid architecture is made challenging by the often heterogeneous nature of the resources involved and the facts that these resources have different computational power and may be connected by networks with widely varying performance characteristics [32]. Such concerns motivate new parallel communication patterns, and problem partitioning in parallel models.

1.4 Approach and Methodology

The above mentioned objectives have led our research work to be carried out in three directions:

- Parallel sequence alignment algorithms.
- Parallel suffix trees construction algorithm.
- Efficient mapping of parallel applications onto hierarchical grid systems.

Aligning long DNA sequences is a challenging task due to its high demands for computational power and memory. Dynamic programming based algorithms [117] can align long DNA sequences with high accuracy. However, since their complexities are quadratic with respect to the length of the two sequences this approach leads to a high computing time. One frequently used approach to speed up this time consuming operation is to introduce heuristics to the alignment algorithm [28, 41, 91, 92]. The main drawback of these solutions is that the more time efficient the heuristics, the worse is the quality of the result [122]. Some other tools, requiring large amount of memory, have shown to

achieve good execution times, but only at expense of accuracy. To find an alignment, they start by looking for perfect matches that can be further extended and joined together using a dynamic programming algorithm. For example, Chao et al. used a BLAST-like hashing scheme to identify exact k -mer matches and extend them to maximal-length matches [41]. Delcher et al. applied a data structure called suffix-tree to find out, in linear time, perfect matches of a given length, also called MUMs [30]. Maximal-length matches and MUMs are finally combined into local alignment chains by a dynamic programming step. However, neither of them apply the dynamic programming algorithm along the entire sequences. Thus these tools are only applicable to closely related genomes. The new generation of spaced seed-based approaches, PatternHunter [106] and PatterHunter II [101], has achieved comparative sensitivity with 3000 runtime savings in homology search, compared to methods based on dynamic programming. However, they don't try to compute long exact optimal alignments. Another approach to get high quality results in a short time is to use parallel processing [152]. The system presented in [109] parallelizes the dynamic programming calculation and is able to achieve a speedup of 41 on a 64-node PC cluster for aligning two DNA sequences of length 816,394 and 580,074.

Algorithms based on dynamic programming [117] have the highest accuracy among all the sequence alignment approaches. The Smith-Waterman algorithm [132] finds the most similar subsequences of two sequences (the local alignment) by dynamic programming. However, it imposes challenging requirements both on computer memory and run-time. Considering comparing two sequences with length of m and n , the memory and time complexity for Smith-Waterman algorithm is $\Theta(m \times n)$. For aligning two sequences of a few million base pairs in length this would lead to a memory requirement of several Terabytes. We extended Smith-Waterman algorithm in three directions: 1) to reduce the memory space complexity from quadratic to linear; and 2) to compute near-optimal

non-intersecting alignments since not only the optimal alignment but also sub-optimal alignments are biologically significant; and 3) to make it more suitable for efficient parallelization.

The first step for making use of suffix trees is to construct them. Several linear-time algorithms for suffix tree construction have been introduced (see [74] for a summary). Among them, Ukkonen’s algorithm is widely used. The key feature of Ukkonen’s algorithm is to make use of *suffix links*, which allow the incremental construction of suffix trees. Unfortunately, these algorithms are impractical for constructing large size suffix trees because of high memory overheads [50]. Considering constructing a suffix tree for human chromosome 1 of length around 240Mbp, this would lead to a memory requirement of about 3G bytes even using advanced space optimizing approaches [96]. In order to tackle the memory bottleneck problem, researchers have tried several approaches, such as space saving optimization [96], disk-based approaches [80, 57, 137], and new data structures [31, 45, 82, 86, 108]. However, all these approaches can’t solve the problem radically because of their own shortcomings and poor memory locality of suffix trees.

Our idea is to construct independent sub-trees [44, 87] and then merge these sub-trees together to form the complete suffix tree. We call this the *sub-tree idea*. Therefore, a new data structure is needed to define the criterion by which a standard suffix tree is divided into sub-trees. Moreover, the corresponding efficient algorithms for constructing these sub-trees separately are required. This task is challenging because of the lack of the relationships of nodes residing in different sub-trees which are widely used by existing algorithms to achieve linear time performance.

The computational grid is a hierarchical and heterogeneous environment. Resources shared in a grid system may reside in different administrative domains, run different software, be subject to different access control policies, and be connected by networks with

widely varying performance characteristics. Although some existing tools can hide some of the heterogenous characteristics of grid systems, special consideration is also required to cope with these factors: 1) the resources in the grid systems have different computational power; 2) these resources are shared among all users; and 3) the resources usually are connected by networks with widely varying performance characteristics. Therefore, a good strategy for efficient mapping parallel applications onto grid systems should comply with: 1) try to reduce data transfer over slow connections; and 2) the amount of work allocated to a processor should depend on the computational power that processor allocates to the application at that time in order to assure that no processor becomes the bottleneck.

1.5 Contributions

The contributions of our work are summarized below:

- We have developed an efficient parallel implementation of the dynamic programming algorithm in linear space for sequence alignment. In addition, our solution can compute near-optimal non-intersecting alignments since not only the optimal alignment but also sub-optimal alignments are biologically significant. Our system has high accuracy in aligning both related and unrelated genomes since it is based on dynamic programming. Performance experiments show that our implementation leads to significant runtime savings on a grid system consisting of several PC clusters.
- Heterogenous nature of grid systems motivates new dynamic load balancing techniques for efficient mapping applications onto grid systems. We have analyzed static

and dynamic load balancing approaches for mapping the sequence alignment application in order to achieve good performance. Based on this analysis, a new dynamic load balancing approach named scheduler-worker parallel paradigm is proposed and evaluated.

- We have presented a new data structure with a corresponding linear-time algorithm for constructing large suffix trees. This data structure is called *common prefix suffix tree* (CPST). All suffixes in a CPST share a common prefix. A standard suffix tree can be divided into a number of CPSTs. Each CPST can be tackled independently by one node in a parallel environment. We present an algorithm that permits a linear time construction of CPSTs. In order to map these independent tasks efficiently onto a heterogeneous hierarchical grid system, we use a dynamic load balancing strategy based on the master-slave paradigm. One of the difficulties in constructing large suffix trees using one or more clusters arises from the need to access the whole input sequence. Our solution is to set aside several *data-servers* which hold the whole sequence. Processes constructing CPSTs then can access the sequence through communicating with these *data-servers*.

1.6 Synopsis of Thesis

The rest of the thesis is structured as follows:

- **Chapter 2** presents a general survey of large-scale genome analysis, in particular alignment of genome sequences and construction of suffix trees. Moreover, we review grid computing and its application to bioinformatics.
- **Chapter 3** introduces hierarchical grid systems, and discusses their heterogeneous

characteristics. We also discuss the strategies on how to map applications onto such architecture in order to gain good performance.

- **Chapter 4** elaborates our parallel algorithm to align long DNA sequences and the performance evaluation on a hierarchical grid system.
- **Chapter 5** elaborates the new data structure and the corresponding algorithm for constructing large-size suffix trees. Implementation and evaluation are presented at the end of the chapter.
- **Chapter 6** concludes the achievement of our research work and suggests possible areas of future work.

Chapter 2

Literature Review

2.1 Large-size Genome Analysis

As more and more DNA sequences became available from the late 1970s, interest also increased in developing computer programs to analyze these sequences in various ways. Genome analysis intends to solve genome puzzles, such as determining coding regions, discovering regulatory signals, and deducing the mechanisms and history of genome evolution. In 1982, the Genomic Computer Group was founded in order to offer a set of programs for sequence analysis. From then on, more and more programs were catalyzed by the fast increase of available sequences.

2.1.1 Sequence Alignment

Sequence alignment is the procedure of comparing two (pairwise alignment) or more (multiple sequence alignment) sequences by searching for a series of substrings that are in the same order in the sequences. It is vital for discovering functional, structural, and

evolutionary information in biological sequences. “Similar” sequences in the parlance of analysis may have the same function. Comparative analysis of diverse genomes may provide clues about the mechanisms of evolution.

Introduction to Pairwise Sequence Alignment

To compare two sequences, we need an algorithm that gives a measure of similarity between them [132]. Generally speaking, this algorithm needs to align both sequences, i.e., place one sequence on top of the other, and insert spaces in arbitrary locations in such a way a score can be given to the alignment and this one will be optimal. The score can be simply calculated by assigning a value when symbols at the same location match, a different value when they are different and a third one when a space is introduced in one of the sequences [146]. We only require that no spaces can be inserted in both sequences at the same location. It is clear that by removing such spaces an equivalent alignment with better score will be produced.

For example, let consider two sequences $S_1 = \text{GATATGTAT}$ and $S_2 = \text{GGATGATT}$ and assume a score system that assigns 1 to matches, -1 to mismatches and -2 for a space insertion or gap. A possible alignment between the sequences is shown in Figure 2.1. To take the positions of the symbols into account we can slide one sequence along the other so as to allow more symbols to match. In the other words, we can slide one sequence along the other to get a better score.

sequence S_1:	G	A	T	A	T	G	T	A	T	-
sequence S_2:	-	G	G	A	T	G	-	A	T	T
score:	-2	-1	-1	1	1	1	-2	1	1	-2
total = -3										

Figure 2.1: Global alignment between sequences S_1 and S_2

In the example shown in Figure 2.1, the entire sequences are involved in the alignment

and so gaps were required at the beginning of S_2 and end of S_1 . This problem also known as the *global alignment* problem is often meaningful when comparing sequences of the same protein family.

From the biological point of view, the gap in the first column indicates a possible evolution from sequence S_1 to sequence S_2 (or from sequence S_2 to sequence S_1) by the deletion (or insertion) of nucleotide G. Similarly, the aligned pair at the second column, from left to right, can be understood as a mutation of A into G or, alternatively, of G into A.

From a mathematical point of view, the previous alignment can also be found by solving the equivalent *edit distance* problem. This consists of computing the edit distance between two given sequences, which is defined as the minimum number of *edit operations* (insertions, deletions and substitutions) needed to transform one sequence into the other, along with the sequence of steps that describes such transformation, also called *edit transcript*. Generally speaking, the more similar the sequences, the smaller the distance between them. Therefore, “finding the best alignment between two sequences” is equivalent to “finding the smallest distance between two sequences”. In other words, alignment and edit transcript are mathematically equivalent. Nevertheless, they are different from a modeling standpoint. An edit transcript focuses on the “process”, the series of events that turns one sequence into another, whereas an alignment just displays the result of such process.

Continuing with the example of sequences S_1 and S_2 . If we select the alphabet $\{I, D, S, M\}$ to represent insertions, deletions, substitutions, and matches (a match means no edit operation) with weights 2, 2, 1, 0 respectively, a transformation of sequence S_1 into S_2 can be described in terms of the following edit transcript (see Figure 2.2).

In many biological applications, local similarity is far more meaningful than global

sequence S_1:	G	A	T	A	T	G	T	A	T	-	
edit transcript:	<i>D</i>	<i>S</i>	<i>S</i>	<i>M</i>	<i>M</i>	<i>M</i>	<i>D</i>	<i>M</i>	<i>M</i>	<i>I</i>	
sequence S_2:	-	G	G	A	T	G	-	A	T	T	
score:	2	1	1	0	0	0	2	0	0	2	total = 8

Figure 2.2: Edit distance between sequences S_1 and S_2

similarity. For example, *motifs* or *domains* are structural or functional subunits well conserved among proteins even in different families that are usually surrounded by other less specific material. Therefore, determining these local regions of similarity turns out to be more interesting than a measure of the similarity between entire sequences. In other words, “one must be alert to regions of similarity even when they occur embedded in an overall background of dissimilarity.” [126]

From the previous example, we can see both sequences share a local region of high similarity (see the box in Figure 2.3).

sequence S_1:	G	A	T	<div style="border: 1px solid black; padding: 2px; display: inline-block;">A</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">T</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">G</div>	T	A	T	-
sequence S_2:	-	G	G	<div style="border: 1px solid black; padding: 2px; display: inline-block;">A</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">T</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">G</div>	-	A	T	T
score:				1	1	1				total = +3

Figure 2.3: Local alignment between sequences S_1 and S_2

When *local alignment* is considered, the edit distance and similarity problems are not always equivalent. The former minimizes an objective function whereas the latter tries to maximize it. As a result, optimal pairs in terms of edit distance would be matching strings under the similarity problem but because the objective function is minimized, they might be just one single character long and would not identify biologically important regions of similarity. Therefore, “a formulation such as local alignment, where matches contribute positively and mismatched and spaces contribute negatively, is more likely to find more meaningful regions of high similarity.” [132]

When local similarity is considered, the problem of sequence comparison becomes

more complex. The solution to the problem is not only the best local alignment but all alignments above a certain threshold.

Progress and Problems of Pairwise Sequence Alignment

Alignment of genomic sequences has been one of the most important tools in sequence analysis. Many sophisticated algorithms are available for aligning sequences and much research is going on.

The Smith-Waterman algorithm [132] uses a dynamic programming approach [139] to compute alignments. An alignment is generated by starting at the ends of the two sequences and attempting to match all possible pairs of characters between the sequences by following a scoring scheme for matches, mismatches, and gaps. This procedure generates a matrix of numbers that represents all possible alignments between the sequences (as shown in Figure 2.4). A highest score in the matrix indicates an optimal alignment. The Smith-Water algorithm has high accuracy. However it can also be slow due to the large number of all cells in the matrix when the sequences are long.

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

Figure 2.4: Dynamic programming matrix for aligning $S_1 = \text{GATATGTAT}$ and $S_2 = \text{GGATGATT}$. The score system is that assigns 1 to matches, -1 to mismatches and -2 for a space insertion or gap.

FASTA [103, 123, 149] is a program for rapid alignment of pairs of biological sequences. Rather than comparing individual residues in the two sequences, FASTA instead searches for matching sequence patterns or words, called k -tuples (as shown in Figure 2.5). These patterns comprise k consecutive matches in both sequences. The program then attempts to build a local alignment based on these word matches by the dynamic programming method. The algorithm used in FASTA is heuristic. Compared with the method based on dynamic programming, its speed is achieved through sacrificing accuracy.

Figure 2.5: Searching 3-tuples for aligning $S_1 = \text{GATATGTAT}$ and $S_2 = \text{GGATGATT}$.

Some new programs for aligning sequences are: ASSIRC [143], DIALIGN [115], DBA [85], MUMmer [30, 97], PipMaker/BlastZ [131], GLASS [33], BLAT [91], WABA [92], and LSH-ALL-PAIRS [38]. However, several recent comparative studies of genomes or large genomic segments are still using older methodologies to solve their particular problems [119, 105]. It may be that the newer alignment programs have not yet gained wide-spread exposure or acceptance by the general scientific community.

Until very recently, most of these algorithms were primarily designed for comparing small-size protein sequences or DNA sequences containing a single gene or operon. Therefore, they are incapable of producing accurate long alignments for comparing long

biological sequences. Dynamic programming based algorithms can align long sequences with high accuracy. However, they are too computationally demanding without special hardware. Some faster methods achieve their speeds through sacrificing accuracy. Although there exist some genome-length alignment tools, they have usually been designed with specific goals. Therefore, the new methods considering tradeoff between higher speed and increased sensitivity are demanded. In addition, with the continuously high-throughput genomic sequencing and the ever-expanding sequence databases, new advances in software programs for genome analysis are being demanded by biological scientists.

2.1.2 Constructing Large-size Suffix Trees

Introduction to Suffix Trees

Let S denote a string of size l . $S[i]$ is defined as the character at position i of S . $S[i..j]$ is defined as the substring of S from position i to j , $1 \leq i < j \leq l$. All suffixes S_i will then be $S[i..l]$, $1 \leq i \leq l$.

The *suffix tree* (ST) for an l -character string S is defined as follows.

- The ST is a rooted directed tree with exactly l leaves numbered 1 to l ;
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S ;
- No two edges out of a node can have edge-labels beginning with the same character;
- For any leaf i , the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i , i.e. it spells out $S[i..l]$.

As an example, the suffix tree for $S = cagca\$$ is shown in Figure 2.6. The path from the *ROOT* to leaf node 1 spells out the full string $cagca\$$, while the path to leaf node 2 spells out the suffix $agca\$$ which starts in position 2.

Figure 2.6: The suffix tree of the sequence $cagca\$$.

A terminating symbol that does not occur anywhere in the string (in the case of Figure 2.6, the symbol is $\$$) is usually added to the end of the string. The reason for this is explained as follows. Let's consider the case where one suffix of S matches a prefix of another suffix of S then no suffix tree obeying the definition stated above is possible, since the path for the first suffix would not end at a leaf. For example, if the last character of $cagca\$$ in Figure 2.6 is removed, creating string $cagca$, then suffix ca is a prefix of suffix $cagca$, so the path spelling out ca would not end at a leaf. To avoid this problem, we assume that the last character of S appears nowhere else in S . Then, no suffix of the resulting string can be a prefix of any other suffix. To achieve this in practice, a character is added to the end of S that is not in the alphabet that string S is taken from. In this thesis we use $\$$ for the terminating symbol.

The *implicit suffix tree* is also a widely used concept. An *implicit suffix tree* for prefix $S[1..i]$ of S is the the adapted suffix tree for $S[1..i]\$$ by removing 1) $\$$; and 2) unlabeled edges; and 3) internal nodes with only one child. The implicit suffix tree for Figure 2.6

is shown in Figure 2.7.

Figure 2.7: The implicit suffix tree for the sequence *cagca*.

Application of Suffix Tree in Bioinformatics

The suffix tree is a data structure that exposes the internal structure of a string in a deep way. It can solve many string computing problems with surprising efficiency. A collection of representative problems is shown in Table 2.1. Consequently, many widely used large-scale computational genomic applications have achieved good performance using suffix trees (see Table 2.2).

Progress and Problems of Constructing Large-size Suffix Trees

A first step to take advantage of suffix trees is to construct them. The naïve algorithm for constructing a suffix tree of a string S of length l is described as follows. The algorithm first enters a single edge for suffix S_1 (the entire string) into the tree; then it successively enters suffix S_i into the growing tree, for i increasing from 2 to l . We let T_i denote the intermediate tree that encodes all suffixes from 1 to i . In detail, T_1 consists of a single edge between the root of the tree and a leaf labeled 1. The edge is labeled with the string S . Tree T_{i+1} is constructed from T_i as follows: starting at the root of T_i find the longest path

String Computing Problems	Description
Exact string matching	Finding all occurrences of pattern p in string S
Exact set matching	Finding all occurrences from a set of patterns $\rho = \{\rho_1, \dots, \rho_n\}$ in a string S
Substring problem for a database of strings	Determining whether pattern p is a substring of any of string from a database of strings
Longest common substring of two strings	Finding the longest substring common to two given string S_1 and S_2
Pairs suffix-prefix matching	Given two strings S_i and S_j , any suffix of S_i that matches a prefix of S_j is called a <i>suffix-prefix match</i> of S_i, S_j
Maximal repetitive structure	Finding all maximal repetitive structure in a string This problem depends on how to define the maximal repetitive structure
k -mismatch	Given a pattern p , a string S , and a fixed number k , a k -mismatch of p is a $ p $ -length substring of S that matches at least $ p - k$ characters of p

Table 2.1: Example applications of suffix trees in string computing.

Applications	Description	References	Year
MUMmer	Rapidly aligning entire genomes	[30]	1999
REPuter	Compute and visualize repeats in whole genomes	[98]	1999
OASIS	Local alignments searches on biological sequences	[113]	2003
Weeder	Automatic discovery of conserved motifs in a set of related regulatory DNA sequences	[121]	2004

Table 2.2: Example applications of suffix trees in string computing.

from the root whose label matches a prefix of S_{i+1} . This path is found by successively comparing the matching characters in suffix S_{i+1} to characters along a unique path from the root, until no further match is possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character. At some point, no further matches are possible because no suffix of S is a prefix of any other suffix of S . When that point is reached, the algorithm is either at a node, let say ω , or it is in the middle of an edge. If it is in the middle of an edge, let say (μ, v) , then it breaks edge

(μ, v) into two edges by inserting a new node, called ω , just after the last character on the edge that matches a character in S_{i+1} and just before the first character on the edge that is mismatched. The new edge (μ, ω) is labeled with the part of the (μ, v) label that matched with S_{i+1} , and the new edge (ω, v) is labeled with the remaining part of the (μ, v) label. Then (whether a new node ω was created or whether one already existed at the point where the match ended), the algorithm creates a new edge $(\omega, i + 1)$ running from ω to a new leaf labeled $i + 1$, and it labels the new edge with the unmatched part of suffix S_{i+1} . Obviously, the time complexity of the naïve algorithms is $\Theta(m^2)$.

Weiner [148] gave the first linear time algorithm in 1973 and McCreight [112] gave a more space efficient linear time algorithm for constructing the suffix tree while processing the string from right to left in 1976. McCreight's algorithm was extremely difficult to understand. Only much later did Ukkonen [140, 141] give a simpler linear time algorithm that is just as space efficient as McCreight's algorithm. Ukkonen's algorithm has been the most widely used [74].

The linear time algorithms mentioned above are impractical for constructing large size suffix trees because of high memory overheads. For example, considering constructing the suffix tree of the human chromosome 1 of length around 240Mbp, this would lead to a memory requirement of about 3G bytes even using advanced space optimizing approaches [96]. Therefore, new suffix tree construction approaches are required because biological sequences typically have very large size and sequence datasets are growing at a rapid rate [17].

In order to tackle the memory bottleneck problem in constructing large suffix trees, researchers have tried several approaches. We have summarized these approaches into four categories:

- (1) **Space saving optimizations.** This approach exploits various kinds of redundancies in suffix trees to obtain more space efficiency[96]. However, the internal structure of suffix trees doesn't permit very significant space saving optimization without any sacrifice of suffix tree virtues. For example, if suffix tree construction doesn't use *suffix links*, an efficient vector can be used to present internal nodes. Unfortunately, this prohibits linear time construction. The paper in [86] claims that a suffix tree can be constructed in $15l-18l$ bytes of space, where l is the length of the input string. However, it doesn't provide any details. One of the best known results still requires $20l$ bytes in the worst case [96].
- (2) **Disk-based approaches.** Disk-based approaches [57, 80, 137] hold the suffix trees in second memory. Unfortunately, suffix tree construction has poor memory locality since it requires a semi-random walk over the tree as it is constructed [50]. Therefore, large-size suffix trees that will not fit in memory would take an unacceptably long time to construct and be accessed due to excessive page faulting. For example, the approach presented in [80], which is currently considered one of the best disk-based suffix tree construction algorithms, needs more than 32.5 hours to index a 286Mbp DNA sequence on a computer with 2GB RAM running at 300MHz.
- (3) **New data structures.** Another method is to develop alternative data structures which store less information than suffix trees and therefore have lower memory overheads. Some new data structures are *suffix array* [108], *level compressed trie* [31], *suffix binary search tree* [82], *suffix cactus* [86] and *PT-tree* [45]. This approach has the following two common shortcomings [96]. Firstly, they are specifically designed for certain applications and can not be adapted to other kinds of problems without severe performance degradation. Thus, they are not as versatile as suffix trees. Secondly, direct construction of these data structures is usually slower than suffix

tree construction.

- (4) **Constructing suffix trees in parallel.** This approach uses the idea of processing sub-trees independently. Once all the sub-trees have been constructed it is possible to merge them together to form the complete suffix tree. We call this the *sub-tree idea*. The paper by Kärkkäinen and Ukkonen [87] employs the *sub-tree idea* by presenting a data structure called *evenly spaced sparse suffix tree*. It contains every k^{th} suffix of the input string for some positive integer k . Unfortunately, the paper only includes theoretical research. The disk-based methods [80] and [84] also utilize the *sub-tree idea*. Parallel approach has been recently used in [44] and [37]. Unfortunately, [44] only gives some actual experiments on a binary alphabet, which is not relevant in practice; [37] constructs suffix trees not in a cluster, but an SMP^{2.1} machine with quad CPUs and large memory.

2.2 Grid Computing

2.2.1 Taxonomy of Parallel Computer Architecture

Flynn's taxonomy is the most widely-known classification scheme for parallel computers. Based on the scheme, a parallel computer's category is determined by the parallelism of its instruction and data streams. A process on the computers can be seen as executing a sequence of instructions (the instruction stream) that manipulates a sequence of operands (the data stream). The focus of Flynn's taxonomy is on the multiplicity of hardware used to manipulate the instruction and data streams [59]. A computer's hardware may support a single instruction stream or multiple instruction streams manipulating a single data

^{2.1}Symmetric Multiprocessing

stream or multiple data streams. Hence *Flynn's classification* results in four categories (as shown in Figure 2.8).

Figure 2.8: Flynn's taxonomy of parallel computer architecture

- **SISD**

The category SISD refers to computers with a single instruction stream and a single data stream. Conventional single processor computers fall into the SISD category. Even though it has only a single CPU executing a single instruction stream, a modern uniprocessor may still exhibit some concurrency of execution. For example, superscalar architectures support the dynamic identification and selection of multiple independent operations that may be executed simultaneously.

- **SIMD**

The category SIMD refers to computers with a single instruction stream but multiple data streams. The *processor array* [56] falls into this category, since it is a parallel computer with a single control unit executing one instruction stream, as well as multiple subordinate processors capable of simultaneously performing the same operation on different data elements.

- MISD

The category MISD refers to computers with multiple instruction streams but only a single data stream. There are few machines in this category, none that have been commercially successful or had any impact on computational science. One type of system that fits the description of an MISD computer is a systolic array [13], which is a network of small computing elements connected in a regular grid. All the elements are controlled by a global clock. On each cycle, an element will read a piece of data from one of its neighbors, perform a simple operation.

- MIMD

The category MIMD refers to computers with multiple instruction streams and multiple data streams. Multiprocessors and multicomputers fall into this category. Both of these architectures are based on multiple CPUs. Different CPUs can simultaneously execute different instruction streams manipulating different data streams.

Most contemporary parallel computers fall into the MIMD category. Hence the MIMD designation is not particularly helpful when describing modern parallel architectures. Memory architecture has a strong influence on the global architecture of MIMD machines and frequently determines the optimal programming model. Therefore, a further classification of MIMD which is based on the memory architecture is widely accepted. The classification results in three categories: *shared memory*, *distributed memory*, and *shared distributed memory* (as shown in Figure 2.9).

A parallel system is said to have a shared-memory architecture if any process, running in any processor, has direct access to any local or remote memory in the whole system. Otherwise, the system has distributed memory architecture. Shared memory architecture brings several advantages to bioinformatic applications. For instance, a single address map simplifies the design of parallel programs. In addition, there is no “time penalty”

Figure 2.9: MIMD's taxonomy

for communication between processes, because every byte of memory is accessible in the same amount of time from any CPU. However, shared memory does not scale well as the number of processors in the computer increases. Distributed memory systems scale very well, on the other hand, but the lack of a single physical address map for memory incurs a “time penalty” for inter-process communication. Parallel systems with a shared distributed memory design try to achieve the best of both shared and distributed memory architectures. A certain amount of memory physically attaches to each node (distributed architecture), but the hardware creates the image of a single memory for the whole system (shared architecture). In this way, the memory installed in any node can be accessed from any other node as if all memory were local with only a slight time penalty.

The availability of very fast processors in workstations, together with the widespread utilization of networks, led to the notion of a “virtual parallel computer” that connected several fast microcomputers by means of a fast LAN ^{2.2}. This distributed-memory system was called a “multicomputer”. Clusters of workstations (COWs) and beowulf-clusters [4] are good examples of multicomputer systems. Beowulf-clusters are popular since they are composed of ordinary hardware components (like any PC) together with public domain software (like Linux, PVM or MPI). Grid systems are the new paradigm of multicomputers.

Multicomputers bring several advantages to parallel computing: low cost (on average, one order of magnitude cheaper for the same computational power), maintenance (replacing fault nodes), scalability (adding new nodes), and code-portability. Some drawbacks also exist, such as it is not easy to manage the cluster as one integrated machine. In addition to this, current network technology has high latency and insufficient bandwidth to handle fast parallel processing. Special efforts are necessary to achieve good parallel performance on such kind of systems.

2.2.2 Introduction to Grids

Grid computing [64] has emerged as an important new parallel and distributed computing paradigm by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation.

^{2.2}Local Area Network

What is the grid?

In 1998, Ian Foster and Carl Kesselman attempted a definition of the *computing grid* in [61, 66]:

A computing grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.

This definition of the grid focuses on the access to computing resource, data, and services on demand. In 2001, the concept of *Virtual Organization (VO)* is proposed in [67]:

A set of individuals and/or institutions defined by the sharing rules is called a virtual organization (VO).

Based on the concept of *VO*, the definition of the grid is refined to address social and policy issues in [67]:

The real and specific problem that underlies the grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations.

Grid Architecture

The grid architecture is fundamental for the establishment, management and exploitation of dynamic, cross-organizational *VO* resource sharing. As defined in [66, 67], the layered grid architecture is shown in Figure 2.10.

Figure 2.10: The layered grid architecture

Components within each layer in Figure 2.10 share common characteristics and can be built on the capabilities and behaviors provided by any lower layer. For example, protocols at *Resource* and *Connectivity* layer are designed so that they can be implemented on top of a diverse range of resource types, defined at the *Fabric* layer. On the other hand, they can be used to construct a wide range of global services and application-specific behaviors at the *Collective* layer. The detail of each layer is described as follows:

- **Fabric: Interfaces to Local Control**

The grid *Fabric* layer provides basic grid protocols that enable grid applications to share resources, which can be, for example, computational resources, storage systems, network resources and sensors. *Fabric* components implement the local, resource-specific operations that occur on specific resources. These operations enable resource sharing operations at higher levels.

- **Connectivity: Communicating Easily and Securely**

The *Connectivity* layer defines core communication and authentication protocols required for grid-specific network transmission. Communication protocols enable the exchange of data between *Fabric* layer resources. Authentication protocols

provide cryptographically secured mechanisms for verifying the identity of users and resources.

- Resource: Sharing Single Resource

The *Resource* layer builds communication and authentication protocols of *Connectivity* layer to define protocols (and APIs, SDKs) for the secured negotiation, initiation, monitoring, control, accounting and payment of sharing operations on individual resources. *Resource* layer calls *Fabric* layer functions to access and control local resources.

- Collective: Coordinating Multiple Resources

While the *Resource* layer is focused on interactions with a single resource, the *Collective* layer in the architecture contains protocols and services (and APIs, SDKs) that are not associated with any specific resource but rather interactions across collections of resources.

- Application

The *Application* layer in the grid architecture comprises the user applications that operate within a *VO* environment. Applications are constructed in terms of services defined at any other layers.

Classification of Grid Systems

The emergence of a variety of new applications demands that grids support efficient data and resource management mechanisms (RMS^{2,3}) [66]. The architecture of the RMS in a grid system is mainly determined by the design objectives of this system [95]. The design objectives can be grouped into three different themes: improving application perfor-

^{2,3}Resource Management System

mance, data access, and enhanced services. Grid computing systems with similar design objectives usually have similar characteristics. Therefore, we can group grid systems into several categories as shown in Figure 2.11.

Figure 2.11: Grid systems taxonomy.

- Computational Grid.

The *computational grid* category denotes systems that have a higher aggregate computational capacity available for single application than the capacity of any constituent machine in the system. These can be further subdivided into *distributed supercomputing* and *high throughput* categories depending on how the aggregate capacity is utilized. A high throughput grid increases the completion of a stream of jobs. A distributed supercomputing grid executes the application in parallel on multiple machines to reduce the completion time of a job. Typically, applications that require a distributed supercomputer are grand challenge problems. Examples of grand challenge problems are computational genomics, fluid dynamics, weather modeling, nuclear simulation, molecular modeling, and complex financial analysis.

- Data Grid

The *data grid* category is for systems that provide an infrastructure for synthesizing new information from data repositories such as digital libraries or data warehouses that are distributed in a wide area network. *Computational grids* also need to

provide data services but the major difference between a *data grid* and a *computational grid* is the specialized infrastructure provided to applications for storage management and data access. In a computational grid the applications implement their own storage management schemes rather than use services provided by grids. Typical applications for these systems would be special purpose data mining that correlates information from multiple different data sources and then processes it further [88].

- Service Grid

The *service grid* category is for systems that provide services for more than one machine. This category can be further subdivided in *on demand*, *collaborative*, and *multimedia* grid systems. A *collaborative* grid connects users and applications into collaborative workgroups. These systems enable real time interaction between humans and applications via a virtual workspace. A *multimedia* grid provides an infrastructure for real-time multimedia applications. This requires supporting quality of service across multiple different machines whereas a multimedia application on a single dedicated machine can be deployed without QoS^{2.4}. An *on demand* grid category dynamically aggregates different resources to provide new services. A data visualization workbench that allows a scientist to dynamically increase the fidelity of a simulation by allocating more machines to a simulation would be an example of using service grids [23].

Most ongoing research activities developing grid systems fall into one of the above categories. Development of truly general-purpose Grid systems that can support multiple or all of these categories remains a hard problem.

^{2.4}Quality of Service

In our research work, distributed supercomputing systems, falls into the computational grid category, are used as the parallel infrastructure, since they can yield high computing power.

2.2.3 Globus Toolkit—the De Facto Standard for Grid Computing

The Globus Toolkit [11, 63] is fundamental enabling technology for the “grid”. From version 1.0 in 1998 to the 2.0 release in 2002 and now the latest 4.0 version, the Globus Toolkit has evolved rapidly into so-called “the de facto standard” for grid computing. A growing number of scientific projects have gained good performance by using this toolkit [133, 90, 144].

Globus Toolkit has the following components:

- Globus Security Infrastructure

Grid Security Infrastructure (GSI) [66] enables secure authentication and communication over an open network. GSI provides a number of useful services for the computing grid, including mutual authentication and single sign-on. GSI is based on public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol.

- Globus Resource Management System

The Globus Resource Management System (GRMS) [66] is developed to support resource allocation and reservation in the computing grid. The GRMS architecture is a layered system in which high-level global resource management services are layered on top of local resource allocation services.

GRMS has three main components:

- Resource Specification Language

The Globus Resource Specification Language (RSL) provides a common interchange language to describe resources. The various components of the Globus Resource Management architecture manipulate RSL strings to perform their management functions in cooperation with the other components in the system. The RSL provides the skeletal syntax used to compose complicated resource descriptions, and the various resource management components introduce specific $\langle attribute, value \rangle$ pairings into this common structure. Each attribute in a resource description serves as a parameter to control the behavior of one or more components in the resource management system.

- Grid Resource Allocation Manager

The Globus Resource Allocation Manager (GRAM) processes the requests for resources for remote application execution, allocates the required resources, and manages the active jobs. It also returns updated information regarding the capabilities and availability of the computing resources to the Monitoring and Discovery Service. GRAM provides an API for submitting and canceling a job request, as well as checking the status of a submitted job. The specifications are written by the user in the RSL, and is processed by GRAM as a part of the job request.

- Resource Broker/Co-allocator

Resource brokers are responsible for taking high-level RSL specifications and transforming them into more concrete specifications through a process called *specialization*. Transformations effected by resource brokers generate a specification in which the locations of the required resources are completely specified. Such ground requests can be passed to a resource co-allocator. Resource co-

allocator provides a co-allocation service, which coordinates ground requests that may span multiple GRAMs.

These three components of GRMS collaborate to realize the function of resource management of Globus. Applications submit resource requirements which are expressed in RSL. These resource requests are submitted to a resource broker then transformed to ground specifications. The resource broker gets resource information from the grid information services. The ground specifications and resource information are processed and the GRAMs that manage required resources are specified. Resource co-allocator transfers single request to individual GRAM. With the collaboration of multiple GRAMs, resource co-allocator allocates resources to applications to fulfill their resource requirements.

- Grid Information Services

Grid information services [66] are a vital part of any grid software infrastructure, providing fundamental mechanisms for discovering and monitoring, and thus for planning and adapting application behaviors.

Globus information Services have two basic components:

- A large, distributed collection of generic information providers provide access to information about individual entities, via local operations or gateways to other information sources. Information is structured in terms of a standard data model, taken from LDAP^{2.5}: an entity is described by a set of “objects” comprised of typed $\langle attribute, value \rangle$ pairs.
- Higher-level services collect, manage, index, and/or respond to information provided by one or more information providers. Globus distinguishes in par-

^{2.5}Lightweight Directory Access Protocol

ticular aggregate directory services, which facilitate resource discovery and monitoring for VOs^{2.6} by implementing both generic and specialized views and search methods for a collection of resources. Other higher-level services can use this information and/or information obtained directly from providers for the purposes of brokering, monitoring, troubleshooting, etc.

- GridFTP

GridFTP [66] is a high-performance, secured, reliable data transfer protocol optimized for high-bandwidth wide-area networks. It provides the following protocol features:

- GSI security on control and data channels;
- multiple data channels for parallel transfers;
- partial file transfers;
- third-party (direct server-to-server) transfers;
- authenticated data channels;
- reusable data channels; and
- command pipelining.

2.2.4 MPICH-G2 – A Grid-Enabled MPI Implementation

MPI and MPICH

Message Passing Interface (MPI) [72] is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.

^{2.6}Virtual Organization

It was designed for high performance on both massively parallel machines and on workstation clusters. The version MPI-1 [72] has been a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. The MPI-2 [71] is an extension to MPI-1. However, it has not gotten widely used by now.

MPICH [15, 89] is a popular implementation of MPI-1 standard with extensions to support the parallel I/O functionality defined in the MPI-2 standard. Its free distribution and wide portability have contributed materially to the adoption of the MPI standard by the parallel computing community. MPICH achieves its portability from its interfaces and layered architecture. At the top of the layered architecture is the MPI interfaces defined by the MPI standards. Directly beneath the interface is the MPICH layer, which implements the MPI interfaces. Much of the code in MPICH is independent of the networking devices or process management systems.

MPICH-G2

Computational grids typically involve heterogeneous collections of computers that may reside in different administrative domains, run different software, be subject to different access control policies, and be connected by networks with widely varying performance characteristics [62]. This makes it quite difficult to develop grid applications though some grid implementations, such as Globus Toolkit, are available. Few programmers take the time to master the intricacies of such grid environments, and thus often produce grid applications that are fragile, non-portable, and of poor performance. Therefore, some specialized grid-enabled tools, that hide mundane aspects of the heterogeneous grid environment without compromising performance, are required for application development in the grid environments. These tools may implement widely familiar programming models, such as message passing.

MPICH-G2 [16, 89] is an implementation of the complete MPI-1 [72] standard and MPI-2 [71] I/O standard. It hides heterogeneity of grids by using Globus Toolkit services for such purposes as authentication, authorization, executable staging, process creation, process monitoring, process control, communication, redirection of standard input and output, and remote file access. As a result a user can run MPI programs across multiple computers at different sites using the same commands that would be used on a parallel computer.

Before the startup of an MPICH-G2 application, the user needs to employ the *Grid Security Infrastructure* (GSI) [66] to obtain a proxy credential (pubic key) that is used to authenticate the user to each site. The user may also use the *Monitoring and Discovery Service* (MDS) [58] to select computers on the basis of, for example, configuration, availability, and network connectivity.

Once authenticated, the user can use the standard `mpirun` command to request the creation of an MPI computation. The MPICH-G2 implementation of this command uses the *Resource Specification Language* (RSL) [11] to describe the job. A RSL script may contain the description of resources (e.g., computers) and requirements (e.g., number of CPUs, memory, execution time) and parameters (e.g., location of executables, command line arguments, environment variables). Based on the information found in an RSL script, MPICH-G2 calls the Dynamically-updated Request Online Co-allocator (DUROC) to schedule and start the application across the various computers specified by users. The DUROC library itself uses the *Grid Resource Allocation and Management* (GRAM) [66] APIs and protocols to start and subsequently manage a set of subcomputations. Once an application has started, *Globus Access to Secondary Storage* (GASS) [63] will be used to direct standard output and error streams to the user's terminal and provide access to files regardless of location. Thus, masking essentially all aspects of geographical distribution

except those associated with performance.

On running of the application, MPICH-G2 selects the most efficient communication method possible between any two processes, using vendor-supplied MPI if available, or *Globus communication* (Globus IO) with *Globus Data Conversion* (Globus DC) for TCP/IP, otherwise.

MPICH-G2 is not the only grid-enabled implementation of MPI. Other implementations include PACX-MPI [69], Stampi [81] and MagPIe [93], etc. However, MPICH-G2 is the most widely used one due to its performance and the degree to which it hides and manages heterogeneity of grid environments. Many applications have achieved good performance by using MPICH-G2 [42].

2.3 Parallel Computing in Bioinformatics

2.3.1 Classification of Parallel Bioinformatics Applications

As motioned before, task scheduling strategies are vital for grid-based applications to achieve good parallel performance. One of key features that task scheduling strategies should consider is the computational characteristics of these applications. According to rate of free-dependent tasks, the data access pattern, and the task homogeneity, parallel Bioinformatics applications can be categorized into three computational patterns: *regular*, *semi-regular* and *irregular* [125, 138]. Detailed classifications are as follows.

Regular Computational Pattern

Bioinformatics applications with *regular computational pattern* can be divided easily into homogeneous subtasks and these subtasks has very few data dependency while they are

executing.

A good example for *regular computational pattern* is database searching [122]. Database searching is one of the most heavily used bioinformatics applications. It is also one of the most familiar applications to begin a discussion about parallelization in bioinformatics. Database searching has a very simple form as far as data flow is concerned, and the searching task can be divided easily into parts due to the elements in the database have no dependency, and thus a broad range of strategies have been proposed to apply parallel computing [94].

Semi-regular Computational Pattern

Bioinformatics applications with *semi-regular computational pattern* can be divided into homogeneous subtasks with somewhat difficulty because these subtasks has medium data dependency while they are executing. In order to parallelize efficiently such applications, sophisticated task scheduling strategies are required.

A good example for *semi-regular computational pattern* is aligning long biological sequences. The task (computing *similarity matrix*) [42] can be divided easily into the subtasks (a number of cells). The dependency relationship of these subtasks is that each cell (i,j) of the *similarity matrix* is computed from the cells $(i-1,j)$, $(i,j-1)$, $(i-1,j-1)$. Obviously, this calculation is based on the wavefront communication pattern. The parallel application with sophisticated task scheduling strategies has achieved good performance on a distributed memory computing architecture [42].

Irregular computational Pattern

Bioinformatics applications with *irregular computational pattern* are difficult to be divided into homogeneous subtasks because these subtasks have complex data dependency while they are executing. The high data dependency or differences among subtasks usually result in bad parallel performance.

A good example for *irregular computational pattern* is building phylogenetic trees [55]. The algorithm presented in [120] proceeds by adding sequences into a given tree topology in such a way that maximizes the likelihood topology. Once the new sequence is inserted, a local-optimization step is performed to look for minor rearrangements that could lead to a higher likelihood. Unfortunately, the reorganization task of one node is dependent on the reorganization of the previous nodes, due to the replacement of the current best-tree. Additionally, these rearrangement procedures result in high data transfer.

2.3.2 Fine-grained and Coarse-grained Parallel Approaches in Bioinformatics

Parallel computing enables a massive computational task to be divided into several subtasks that execute concurrently through different processors. In order to gain high parallel performance, the partition of subtasks needs to consider two key features: (a) *granularity*: the relative size of the subtask (fineness or coarseness of task division); and (b) *communication*: the data transfer among subtasks while they are executing. In general, any good parallel strategy represents a trade-off between reducing communication time and improving the computational load balance by adapting the *granularity*.

Fine-grained Parallel Approach

There are many fine-grained parallel implementations of applications in bioinformatics. Table 2.3 describes some widely known examples, such as DeCypher [5], Kestrel [14], SAMBA [73], and Systola 1024 [25]. These fine-grained parallel approaches usually depend on special hardware.

Name	architecture	applications	references
DeCypher	FPGA	S-W ^{2,7} , ClustalW, HMM	[5]
Kestrel	SIMD	S-W, HMM	[70], [14]
SAMBA	Systolic Array	S-W	[73]
SYSTOLA 1024	Instruction Systolic Array	S-W, HMM	[25], [128]

Table 2.3: Fine-grained Parallel Approach

- DeCypher [5] represents a class of technology known as *Configurable Computing*. A configurable computer is one in which tasks are accelerated by direct execution in hardware. Configurable computers utilize FPGA (Field Programmable Gate Array) integrated circuits. Some computational genomics applications, such as Smith-Waterman and ClustalW, have been parallelized efficiently on the DeCypher [5].
- Kestrel [14] is a 512-element linear parallel processor with 8-bit, single-instruction stream, multiple-data stream (SIMD) processing elements. The single-board system was designed and built at the University of California at Santa Cruz with particular emphasis on efficient high-throughput sequence analysis [70].
- SAMBA [73] is a 128 processor array for speeding up the comparison of biological sequences. It comprises a workstation (with its local disk), a systolic array made out of 128 VLSI full custom processors and a FPGA-Memory interface which fills the gap between a complete hardwired array of processors and a programmable Von Neuman machine [73].

- SYSTOLA 1024 [25] is a commercial product developed by using Instruction Systolic Array architecture (ISA) which is a massively parallel special purpose architecture. The SYSTOLA 1024 is an add-on-board for a PC to be used as an embedded system. A parallel solution using SYSTOLA 1024 for molecular sequence analysis is described in [128].

Coarse-grained Parallel Approach

In recent several years, clusters of workstations and PCs emerged as an important and rapidly expanding approach to parallel computing [152]. These systems often employ commodity processors and networking hardware as well as open source operating systems and communication libraries to deliver supercomputer performance at the lowest possible price. They also provide a large data storage capacity which allows efficient implementations of large-scale applications which are input and output (I/O) intensive. For these applications whose computational requirements exceed the resources available at even the largest supercomputing center, computational grids offer the promise of access to increased computational capacity through the simultaneous and coordinated use of geographically separated large-scale computers linked by networks. Through computational grids, the “size” of an application can be extended beyond the resources available at a particular location — indeed, potentially beyond the capacity of any individual computing site anywhere.

Clusters and grids based parallel computing can be grouped as coarse-grained parallel approach which uses the MIMD architecture. Table 2.4 shows some typical examples for coarse-grained parallel approach in bioinformatics, such as Bio-grid Computing Platform [145], EARTH [109], TurboBLAST [34], and Parallel Genehunter [46].

Name	applications	references
Bio-grid Computing Platform	S-W	[145]
EARTH	S-W	[109]
TurboBLAST	BLAST	[34]
Parallel Genehunter	linkage analysis of genetic marker data	[46]

Table 2.4: Coarse-grained parallel approach in bioinformatics

- Bio-grid computing platform [145] is a grid environment to manage and schedule resources and submit jobs for the applications that can be cut into a group of independent jobs with small-sized data input, such as the application of protein alignment analysis. It is called “embarrassingly parallel,” i.e., the job can be broken up into independent sub-jobs and each will be allocated to a separate resource.
- EARTH [109] supports multithreaded program execution models in which a program is viewed as a collection of threads whose execution ordering is determined by data and control dependence explicitly identified in the program. It has been used to develop a parallel implementation of the dynamic programming algorithm that, by using the collective memory of several nodes, meets the computer memory requirements of this kind of applications and is able to align any related genomes with reliable output in a reasonable time.
- TurboBLAST [34] is an accelerated, parallel deployment of NCBI BLAST. It delivers high-performance not by changing the BLAST algorithm, but by coordinating the use of multiple copies of the unmodified serial NCBI BLAST application. TurboBLAST is available for many parallel computing environments, from heterogeneous clusters of PCs, workstations, and Macintosh computers, to parallel supercomputers, to the worldwide computing grids.
- Parallel Genehunter [46] is a parallel implementation for performing multipoint linkage analysis of genetic marker data on large family pedigrees. It effectively

distributes both the computation and memory requirements of the analysis. It get high performance on Sandia National Laboratories's Cplant cluster which consists of several hundred DEC Alpha EV6 processors connected via Myrinet.

Although clusters and grids can yield high computing power at low cost, it is still challenging work to introduce clusters and grids in bioinformatics, since, in order to get high performance on clusters and grids, partitioning the computing task, balancing the load assigned to different processors and minimizing the communication overhead have to be considered felicitously.

2.3.3 BioGrid Projects

As more and more computation- and data-intensive bioinformatics applications benefited much from the grid technology, a new concept termed "BioGrid" arose in the biological community in recent years. BioGrid projects are to provide access to, and coordination of, the computing, data storage, and networking infrastructure required for researchers and educators throughput grid systems. Some of these projects are summarized as follows.

North Carolina bioinformatics Grid (NC BioGrid) [18] was established in December 2000 by the North Carolina Biotechnology Center to promote bioinformatics education, research, and development in North Carolina. The consortium consists of over 70 organizations — universities and colleges; biomedical, biotechnology and information technology companies; nonprofit institutions; and foundations. The NC BioGrid will accumulate the vast library of genomics, proteomics, and related data being created throughout the world, combine it with non-proprietary data from consortium members, and make this invaluable collection of data available to researchers and educators throughout North Carolina. The BioGrid will also make available high-performance computing hardware

and software to mine, analyze, and model this data and a high speed network to move the data sets among consortium sites.

BioGrid in Japan [1] aims to develop a computer grid technology that is able to satisfy all the IT needs for its world-class research activities in medicine and biology. The core institution carrying out this research project is the Cybermedia Center of Osaka University that is equipped with high capacity computational and network resources. Relevant research institutions and private enterprises also participate as a peripheral research group that are connected by high-speed networks. The objectives of this project are two-fold: 1) to facilitate pharmaceutical research and development based on biochemistry, information science, mathematical chemistry, physics, medicine and pharmacology; and 2) to develop a super computer network core technology to be used in other research fields with similar needs.

BioGrid at the Warsaw University in Poland [3] will develop interfaces to enable chemists and biologists to be able to submit work to HPC^{2.8} facilities. Its main task is to integrate selected applications with grid infrastructures and provide easy tools for non-experts in high performance computing. The tool sets and user interfaces for both simulations and visualization of biomolecules will be developed. As final step the computation portal will be established and tested with various biomolecular applications and a number of biomolecular projects and users.

BioGrid at National University of Singapore [2] was established in 2001. The grid system consists of several institutes in Singapore. The goal of the BioGrid is to develop high performance bioinformatics applications on computational grids. Finished and ongoing bioinformatics projects are 1) phylogenetic analysis on computational grids; 2) converting the curation and updating a series of Intron-Exon databases to the BioGRID comput-

^{2.8}High Performance Computing

ing platform for performance enhancement; and 3) *Burkholderia pseudomallei* genome analysis project and so on.

2.4 Summary

In this chapter, our research background is introduced in two areas and a survey of concepts and techniques is carried out accordingly. Section 2.1 presents the concept, significance, progress and problems of aligning biological sequences and constructing large-size suffix trees. The grid concept and several widely used grid implementation are described in Section 2.2. Section 2.3 reviews the application of parallel and distributed computing in Bioinformatics. These surveys motivate us to improve the performance of aligning pairwise long sequences and constructing large-size suffix trees by using computational grids.

Chapter 3

Hierarchical Grid Computing

3.1 Hierarchical and Heterogeneous Nature of the Grid

The computational grid enables resource sharing among geographically distributed sites all over the world. These sharing resources may reside in different administrative domains, run different software, be subject to different access control policies, and be connected by networks with widely varying performance characteristics. Therefore, computational grids are a hierarchical and heterogeneous environment [66]. Details are presented as follows.

- Each resource in a grid typically has grid infrastructure installation only in its control nodes. Grid users can submit tasks to these control nodes by using GRMS^{3.1}. There may be a LRMS^{3.2} [49, 134], e.g., PBS [21], LoadLeveler [12], LSF [20], NQE and Sun Grid Engine [24], in each geographical site shared in a grid. The LRMS

^{3.1}Grid Resource Management System

^{3.2}Local Resource Management System

will schedule the tasks in the control nodes to other nodes inside the local resource system. GRMS and LRMS constitute the hierarchical nature of the grid.

- Resources shared in the computing grid are commonly owned and controlled by different individuals or organizations in different sites. Administrators of these sites decide which resource to share and how to share. Therefore, grid users may meet different scheduling policies and security mechanisms when using resources shared in the Grid.
- Different sites may have different types of resources. Even the resources are of the same type, they may have different configurations, capacities and performance.
- The network links among different resources in a grid system usually have widely varying performance characteristics. Furthermore, the inter-resource connection is typically by one or two orders of magnitude slower than the intra-resource connection.

3.2 A Hierarchical Grid System

The grid system used in our research consists of three Linux PC clusters. The three clusters locate at two different research centers (PDCC: parallel and distributed computing center and BIRC: bioinformatics research center) at Nanyang Technological University. Two clusters are in PDCC (8 nodes for one cluster: Intel PIII 733MHz processor with 750MB memory; and 8 nodes for the other one: Intel(R) Xeon(TM) 2.60GHz processor with 1GB memory) and one cluster is in BIRC (8 nodes: Intel Itanium-1 processor 733MHz with 1GB memory). Each cluster is internally connected by a Myrinet (the intra-cluster connection) and an Ethernet switch is used as an inter-cluster connection.

The normal application-level bandwidth inside each cluster is about 190 MByte/s. The normal application-level inter-cluster bandwidth is about 8 MByte/s. In order to evaluate different inter-cluster bandwidths, we run an application, which only sends or receives data packages between the clusters. We can control the sizes and frequencies of the data packages to get different levels of intra-cluster bandwidth. The experimental testbed is similar to a real wide-area grid system.

The software architecture is shown in Figure 3.1. It can be divided into two layers. The upper layer is the MPICH-G2 [16, 11] layer that runs on the control node of each cluster. This allows (slow) inter-cluster communication. The lower one is the MPICH [15] layer that runs on all nodes within a cluster. This allows (fast) intra-cluster communication. Each cluster has SGE [24] installed. The SGE is a Distributed Resource Management (DRM) software. It can allocate parallel tasks from the control node to execution nodes inside a cluster. Each parallel task is firstly distributed to the control nodes of each cluster. Secondly, Sun Grid Engine allocates the task from the control node to another execution node within the cluster, complying with its scheduling strategy. Parallel processes can communicate via MPICH-G2 (between clusters) or with MPICH (within the same cluster). Applications based on MPI run in the grid systems just like inside a cluster from the view of a programmer.

3.3 Efficiently Mapping Applications onto Grids

Mapping applications onto parallel computing systems contains two kinds of things: 1) partitioning the large-size task into a few small-size subtasks; and 2) allocating these subtasks onto processors for parallel execution. The difficulty of partitioning an application is determined by its own computational characteristics. This difficulty, of course,

Figure 3.1: The hierarchical parallel programming environment consisting of MPICH-G2, MPICH and Sun Grid Engine.

can be reduced through improving the applications' algorithms. It is obvious that, the less dependence among subtasks, the easier to map efficiently these subtasks onto parallel systems. We have categorized bioinformatics applications into *regular*, *semi-regular*, and *irregular* in Section 2.3.1. The difficulty of achieving good parallel performance increases from *regular* category to *irregular* category. In order to achieve good parallel performance, the application partitioning also needs to consider the characteristics of target parallel environments. Next step to partitioning an task into subtasks is to allocate these subtasks onto parallel systems. Efficient subtasks allocating is to balance processor loads and thus to gain a short execution time. It can be performed at either system-level or application-level.

3.3.1 System-level Task Scheduling Strategies for Parallel Systems

The problem of task scheduling in parallel systems is the composite problem of deciding *where* and *when* an application should execute, i.e., on which processor and in what order the application processes will run. Both system and workload features need to be taken into consideration when designing a scheduling strategy. Although the scheduling problem is conceptually the same across different systems and parallel workloads, the feasibility and performance of possible solutions are very sensitive to the underlying architecture and workload changes. There is no such thing as policy portability across the architecture spectrum if we want to guarantee optimal performance. In other words, although the underlying physical architecture may be hidden from the user as run time environments give the illusion of the same parallel system, scheduling must be done with caution as solutions need to be carefully tailored to both hardware and software. Research in task scheduling for traditional parallel systems has generated a large amount of results under different assumptions about the type of parallel workload, the goals of scheduling, and the type of parallel systems [51, 52, 53, 54].

The complexity of solutions increases even more when additional conditions are taken into account. For example, consider the scheduling applications onto heterogeneous geographically dispersed grid systems. Heterogeneous characteristics of grid systems considerably contribute to the increase of the problem complexity.

3.3.2 Progress and Problems of System-level Task Scheduling Strategies for Grids

While computational grids bring unprecedented computing power for users, how to deliver this great computing power is still an elusive problem. One of the major puzzles is how to schedule applications into grid systems. Scheduling algorithms for traditional parallel systems can not be assumed for grid systems because of their heterogenous natures. Moreover, most of these algorithms are for dedicated systems. By dedicated, we refer resources in the system serve the application in a dedicated fashion. This is against the situation that grid systems are resource-sharing environments. Good schedules in a shared environment involve the integration of application specific information and system specific information [150].

A resource reservation strategy [65, 77] is proposed to address this problem. The shared resources are reserved in advance for one user's dedicated occupancy. It is useful for time urgent applications and could be applied on space-shared supercomputers because they can provide dedicated nodes for running the applications. However, it is not feasible on time-shared resources such as PCs and workstations in a heterogeneous system because private owners usually don't want their machine dedicated for grid tasks. Task scheduling in such a shared non-dedicated computing environment offers a big challenge.

A task scheduling mechanism based on determined prediction of machine availability has been investigated in [39, 75]. This method might be simple and good as it is useful for scheduling short-running applications in a single machine or a small system. However, due to the internal limitation of its determined prediction method, it is not suitable for scheduling distributed applications in a large system since the prediction errors caused by fluctuation of resource status increase with the number of resources running applications.

A probabilistic approach has been used to estimate the execution time of a parallel application in a heterogeneous computing system [102]. However, its model is based on the assumption that the execution time distribution of individual task is pre-known.

New models for scheduling jobs on grid systems are presented in some papers [49, 68, 83, 107, 135]. They almost make some infeasible assumptions. For instance, some of them make the assumption that there is no data dependence among tasks. Thus these strategies don't consider that the performance would suffer from data transferring on the slow connections in a grid system.

Research of grid task scheduling is still in its infancy [151]. There is no widely accepted task scheduling middleware by far. Grid tools, such as MPICH-G2, don't support adaptive job scheduling. Therefore, grid applications need special consideration in allocating subtasks onto processors in grid systems. The master-slave pattern, which is widely used in traditional parallel environments, is still a simple but effective programming paradigm in grid environments [111, 136].

3.3.3 Application-level Task Scheduling Strategies for Mapping Applications onto Grids

A good schedule for both data and computations is fundamental to obtain an efficient parallel implementation. In general, any parallel task scheduling strategy represents a trade-off between reducing communication time and improving the computational load balance.

The simple task scheduling strategy is based on a master/slave paradigm. Figure 3.2 illustrates its essence. One of the processors acts as the *master*, scheduling and dispatching blocks of tasks to the *slaves* which, in turn, perform the calculations specified by

the algorithm. When the slave completes one block, the master schedules a new block of tasks and repeats this process until all tasks have been computed. The master-slave paradigm is a widely used technique to implement dynamic load balancing. Some parallel bioinformatics applications have achieved good performance using this technique, such as fastDNAm1 [120] and HMMER [153].

Figure 3.2: Master-slave paradigm

The traditional master-slaver paradigm requires communication between *master* and *slavers*. If the communication uses the slow inter-cluster communication, such data transfer would cause bad parallel performance. Therefore, new load balancing techniques are demanded in the grid environment. The traditional master-slave paradigm can be improved to cope with the grid environment using following several ways.

- *Slaves* pre-fetch tasks from the *master* so as to overlap computations and communications. This approach relies on the assumption that these tasks are available previously.
- As the number of slaves increases, slaves can be divided into sets, each with a sub-master, in a hierarchical fashion. This approach, called as *hierarchical master-slave paradigm* (as shown in Figure 3.3), is quite suitable for multi-cluster grid systems, since each cluster can be equipped with one *sub-master* and thus the communication between *sub-master* and its *slaves* can use the fast intra-cluster connection.

Figure 3.3: Hierarchical master-slave paradigm

- Each slave node could manage its own pool of tasks and communicate with other slave nodes. This approach, called as *compromised master-slave paradigm* (as shown in Figure 3.4), reduce the data transfer between *master* and *slaves* by increasing communication among *slaves*. The efficacy of this approach depends on the assumption that the communication among slaves mostly use the fast intra-cluster connection.

Figure 3.4: Compromised master-slave paradigm

3.3.4 Rules for Mapping Applications onto Grids

Grid systems typically have a heterogeneous nature. In order to hide mundane aspects of the heterogeneous grid environment without compromising performance, some specialized tools have been developed [11, 16, 81, 93]. However, even with these tools, the following three aspects still have to be taken into account when running a parallel application in a

multi-cluster grid environment [42, 32].

- (1) Resources have different computational power.
- (2) Resources are shared, i.e. there are several users' tasks running at the same time, therefore, the effective CPU time of an application depends on the number of jobs running on the node at that time.
- (3) Resources in a grid system usually are connected by networks with widely varying performance characteristics. Furthermore, the inter-cluster connection is by one or two orders of magnitude slower than the intra-cluster connection.

In order to parallelize an application efficiently on a hierarchical multi-cluster grid architecture, the program should comply with the following rules:

- (1) Reduction of inter-cluster data transfer, since the inter-cluster link is usually very slow.
- (2) Amount of work allocated to a processor should depend on the computational power that processor allocates to the application at that time. This assures that no processor becomes the bottleneck.

3.4 Parallel Computational Model

Parallel computational model can accurately predict the performance of parallel systems without constructing the algorithms themselves. It is demanded by parallel application developers to analyze design changes in a quick and efficient manner. Historically, the

most widely used parallel models are PRAM ^{3.3} [60], VRAM ^{3.4} [35] and BSP ^{3.5} [142]. The LogGP model [27] is motivated by current technological trends in high performance computing towards networks of large-grained sophisticated processors. The LogGP model uses four parameters: latency, overhead, gap between consecutive messages and number of processors.

The heterogenous nature of grid systems claims for new computational models. These new models should take into account system heterogeneities both in computational nodes and in the communication networks. The LogGP is suitable to be extended as a new model for heterogenous parallel computing environments [36]. The reasons contains: the underlying architecture is very similar to a cluster; it removes the synchronization points needed in other models such as PRAM and BSP; and it allows modeling both short and long messages and assumes finite network capacity.

3.4.1 An Extension of the LogGP Model for a Heterogeneous Multi-Cluster

There are two main sources of heterogeneity in a heterogenous multi-cluster: the computational nodes and the communication subsystem. The nodes can have different processor, memory and Input/Output features. With respect to communications, three questions have to be considered: the network interface, structure and technology. Suppose there is a m -node multi-cluster $(N_1, \dots, N_m)$, in which both processors and network are heterogeneous. The model is based on the following parameters:

- Latency, L : Communication latency, depends both on network technology and topol-

^{3.3}Parallel Random Access Machine

^{3.4}Vector Random Access Machine

^{3.5}Bulk-Synchronous Model

ogy. In a heterogeneous multi-cluster, different network technologies can coexist. The network's topology defines the routing delays through the network. The *Latency Matrix* of a heterogeneous multi-cluster can be defined as a square matrix $L = \{l_{1,1}, \dots, l_{m,m}\}$, in which each element $l_{i,j}$ records the latency associated to the delivery of a message sent from node N_i to N_j .

- Overhead, o: the time needed by a processor to send or receive a message is referred to as *overhead*. The operations involved in sending or receiving messages are different, producing therefore different amounts of overhead. In consequence, the communication overhead must be broken down into two independent parameters : *sender overhead*, Os , and *receiver overhead*, Or . Additionally, due to system heterogeneity, these overheads depend on each node's computational power, resulting in different Os and Or values for different nodes. Rather than scalar parameters, vector parameters for both overheads are needed, resulting in a *sender overhead vector*, $Os = \{os_1, \dots, os_m\}$, in which each element os_i records the sender overhead for node N_i . Also, a *receiver overhead vector*, $Or = \{or_1, \dots, or_m\}$, records the receiver overhead for each node N_i .
- Gap between messages, g: This parameter depends on the nodes' performance for sending consecutive short messages. In the LogGP model the message size is an inherent parameter, since all of the messages considered in this model are short. Since this value depends on the network interface, it depends on the nodes' features just like the overheads do. Therefore, a *gap vector* $g = \{g_1, \dots, g_m\}$, can be defined in which each element g_i represents N_i 's gap for one byte long messages.
- Gap per byte, G: The Gap per byte depends on the network technology, in particular on the network capabilities to manage long messages, and it is fixed by the network bandwidth. On the other hand, the bandwidth in a heterogeneous multi-

cluster depends both on the network technology and topology. Due to the fact that in a heterogeneous network a message can cross different switches with different bandwidths, the total Gap for a particular message depends on its path. A *Gap matrix* $G = \{G_{1,1}, \dots, G_{m,m}\}$, can be defined, in which each element $G_{i,j}$ determines the Gap per byte for sending a long message from node N_i to node N_j . The inverse of the element $\frac{1}{G_{i,j}}$ is the available bandwidth to send long messages between this pair of nodes.

- Computational power, P_i : The number of nodes cannot be used in a heterogeneous model for measuring the system's computational power, because each node can have very different computational features. Therefore, it is necessary to account for each node's computational power, denoted by P_i , which can be defined as the amount of work finished by that processor during a unit time span for a specific application. P_i depends on the node's physical features (CPU organization and speed, memory and I/O capabilities, etc) but also on the particular algorithm implementation which is actually being processed. The original scalar P parameter can be replaced by a *computational power vector* $P = \{P_1, \dots, P_m\}$, in which each element P_i corresponds to the computational power of the node N_i .

3.4.2 Time Analysis for Standard Master-Slave Paradigm

Suppose there is a parallel application based on the standard master-slave paradigm. The functionality of its master process can be broken into three different phases: 1) first phase: after a short initialization process, the master sends a subtask to all of the slave processes; 2) second phase: the master begins a receive-send loop until there are no more subtask to be processed; 3) third phase: the master gathers all of the pending results. Here, a theoretical analysis of the application's response time is presented. This analysis

is based on the three phases of the master process.

- **First phase:** the response time of this phase is fixed by the time needed by the master to initialize the task, (Tr_m) and send a subtask, (the sender overhead Os_m), multiplied by the number of slaves N . In this phase the model allows the overlap of both computation and communication operations, since the only time involved is the sender overhead and not the whole sending time. This can be expressed as:

$$T_1 = Tr_m + Os_m \times N$$

- **Second phase:** in this case the response time is the time spent by the last slave to finish its work. Due to the heterogeneity, the number of subtasks for each slave has to be calculated. let's use NT_i denotes the number of subtasks for node N_i . The response time of this phase has several components as follows.

- (a) Sending and receiving overheads for master and slaves processes for the corresponding data size, (Os_m and Or_m indicate the sending and receiving overheads for the master; Os_{si} and Or_{si} stands for the sending and receiving overheads for the slave process i).
- (b) Latencies on sending the subtasks and the results, (L_{m-si}). Due to that the *latency matrix* is a symmetric matrix $L_{m-si} = L_{si-m}$.
- (d) Execution time for each subtask, obtained as the inverse of each node's computational power, $\frac{1}{P_{si}}$.

The sum of all these values gives the time for a slave to process a subtask. The response time for a slave to process all of its subtasks is expressed as:

$$T_{si} = (Tr_m + Or_{si} + \frac{1}{P_{si}} + Os_{si} + Or_m + L_{m-si}) \times NT_i$$

And thus, the total response time for the second phase is expressed as:

$$T_2 = \max_{i=1}^{i=N} T_{si}$$

- **Third phase:** the response time of the last phase is fixed by the reception time for the last results, one for each slave. Due to the differences in the overhead and the latency values of each node, the total response time of the third phase is calculated as:

$$T_3 = \sum_{i=1}^{i=N} (Os_{si} + L_{si-m})$$

3.5 Summary

In this chapter, we firstly analyze the hierarchical and heterogenous nature of grid systems. Secondly, the hierarchical grid system used in our research is presented. Thirdly, on the basis of discussing task scheduling strategies for grid systems, we introduce the master-slave paradigm and its extensions. At last, a parallel computational model for a heterogeneous multi-cluster is presented.

Chapter 4

Aligning Long DNA Sequences

4.1 Pairwise Sequence Alignment

4.1.1 Smith-Waterman Algorithm

Surprising relationships have been discovered between biological sequences that have little overall similarity but in which similar subsequences can be found. In that sense, the identification of similar subsequences is probably the most useful and practical method for comparing two sequences. The Smith-Waterman algorithm [132] finds the most similar subsequences of two sequences (the local alignment) by dynamic programming. The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another. Two elementary operations are used: substitution and insertion/deletion (also called a gap operation). Through series of such elementary operations, any segments can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken into as the measure of the distance between the segments.

Consider two strings S_1 and S_2 of length m and n . To identify common subsequences, the Smith-Waterman algorithm computes the similarity $H(i,j)$ of two sequences ending at position i and j of the two sequences S_1 and S_2 . The computation of $H(i,j)$ is given by the following recurrences:

$$H(i,j) = \max \begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ H(i-1,j-1) + sbt(S_1[i], S_2[j]) \end{cases}$$

$$E(i,j) = \max \begin{cases} H(i,j-1) - \alpha \\ E(i,j-1) - \beta \end{cases}$$

$$F(i,j) = \max \begin{cases} H(i-1,j) - \alpha \\ F(i-1,j) - \beta \end{cases}$$

where $1 \leq i \leq m, 1 \leq j \leq n$ and Sbt is a character substitution cost table. Initialization of these values are given by:

$$H(i,0) = E(i,0) = 0, 0 \leq i \leq m$$

$$H(0,j) = F(0,j) = 0, 0 \leq j \leq n$$

Multiple gap costs are taken into account as follows: α the cost of the first gap; β is the cost of the following gaps. Each position of the matrix H is a similarity value. The two segments of S_1 and S_2 producing this value can be determined by a traceback procedure.

The quadratic-space Smith-Waterman algorithm consists of two parts: computing the scores in the similarity and determining the optimal alignment. Figure 4.1 illustrates an example to compute the local alignment between two sequences $S_1 = \text{ATCTCGTAT-}$

GATG and $S_2 = \text{GTCTATCAC}$. The matrix $H(i, j)$ is shown for the computation with $\text{gap}(k) = 0 + 1 \times k$ and a substitution cost of +2 if the characters are identical and -1 otherwise. From the highest score (+10 in the example), a traceback procedure delivers the corresponding alignment (shaded elements), the two subsequences TCGTATGA and TCTATCA. Since the score (+10) is the highest one in the similarity matrix, the corresponding alignment is the optimal alignment.

	A	T	C	T	C	G	T	A	T	G	A	T	G
G	0	0	0	0	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	0	2	1	0	0	2	1	0
C	0	0	1	4	3	4	3	3	2	1	0	2	2
T	0	0	2	3	6	5	4	5	4	5	4	3	2
A	0	2	2	2	5	5	4	4	7	6	5	6	5
T	0	1	4	3	4	4	4	6	5	9	8	7	8
C	0	0	3	6	5	6	5	5	5	8	8	7	7
A	0	2	2	5	5	5	5	4	7	7	7	10	9
C	0	1	1	4	4	7	6	5	6	6	6	9	9

Figure 4.1: Example of the Smith-Waterman algorithm to compute the local alignment between two sequences ATCTCGTATGATG and GTCTATCAC.

Although it is able to report all possible alignments between two sequences, the Smith-Waterman algorithm imposes challenging requirements both on computer memory and run-time. Considering comparing two sequences with length of m and n , the memory and time complexity for Smith-Waterman algorithm is $\Theta(m \times n)$. For aligning two sequences of a few million base pairs in length this would lead to a memory requirement of several Terabytes. However, it is possible to reduce the memory space complexity from quadratic to linear by the algorithm described in the next subsection.

4.1.2 Optimal Alignments in Linear Space

The maximal score in the *similarity matrix* can be computed in linear space as follows. The value of cell (i, j) in the *similarity matrix* only depends on values of the cells $(i - 1, j - 1)$, $(i - 1, j)$ and $(i, j - 1)$. Thus, the i^{th} row can be computed by overwriting values for the $(i - 1)^{th}$ row in a left-to-right sweep. Let's take Figure 4.2 for an example. During the whole process of computing the similarity score, only $n+1$ cells need to be stored (signed with solid black points). The computation of the value of cell f only depends on a , e and b . After calculation of cell f , cell a is not required anymore. As the matrix is being built, we keep track of the maximum score as well as the index of the corresponding cell (i_{max}, j_{max}) . The indices i_{max} and j_{max} specify the end positions of the optimal local alignment of the sequences S_1 and S_2 . To find the starting positions i_{start} and j_{start} , the algorithm can be executed with the sequences in reversed order.

Figure 4.2: Computing the similarity score in linear space.

However, this approach will only find the maximal score, start and end points; it will

not find the actual alignment. Hirschberg [76] and Myers and Millers [116] presented a recursive divide-and-conquer algorithm for computing this alignment in linear space. The central idea is to find the “midpoint” of an optimal alignment using a “forward” and a “reverse” application of the linear space cost-only variation. Then an optimal alignment can be delivered by recursively determining the optimal alignments on both sides of this midpoint.

Figure 4.3 illustrates the main idea of the recursive divide-and-conquer method for computing the global alignment of sequence S_1 of length m and sequence S_2 of length n . Let $S_1[1..k]$ denote $S_1[1], \dots, S_1[k]$ and $\hat{S}_1[m..k]$ denote $S_1[m], \dots, S_1[k]$. $S_2[1..k]$ and $\hat{S}_2[n..k]$ are defined similarly. In order to find an optimal midpoint (i, j) to divide the problem into sub-problems, i is first set to $i = \lfloor n/2 \rfloor$ and then j can be determined as follows. The *similarity matrices* $S_1[1..m] \times S_2[1..i]$ and $\hat{S}_1[m..1] \times \hat{S}_2[n..i]$ are computed in linear space to get the values in the last row of the two matrices: $value_1[1..m]$ and $value_2[1..m]$ (see Figure 4.4). The j is then calculated by $j = \arg \max_{1 \leq k \leq m} \{value_1[k] + value_2[k]\}$. Once an optimal midpoint (i, j) is determined, the optimal alignment can be computed by (1) recursively finding an optimal alignment of $S_1[1..j]$ and $S_2[1..i]$, (2) recursively finding an optimal alignment of $S_1[j..m]$ and $S_2[i..n]$, and (3) concatenating these two partial optimal alignments. In this way, the divide-and-conquer method will recursively divide the given problem into smaller problems by finding its midpoint till a trivial subproblem is obtained.

Let's use $T(m \times n)$ denote the time required for finding the optimal midpoint for the problem of size $m \times n$. The time required for finding the midpoints for the subproblems of size $j \times i$ and $(m - j) \times (n - i)$ is $1/2 \times T(m \times n)$. Thus the cumulative time $T'(m \times n)$ for the recursive divide-and-conquer method is $(1 + 1/2 + 1/4 + 1/8 + \dots) \times T(m \times n) \leq 2 \times T(m \times n) \in O(m \times n)$.

Figure 4.3: Divide-and-Conquer method to compute the global alignment of sequence S_1 and sequence S_2 in linear space.

Figure 4.4: Find the midpoint.

4.1.3 Finding Near-optimal Alignments in Linear Space

The Smith-Waterman algorithm computes the highest-scoring or optimal local sequence alignment. However, there may be other biological important alignments with scores close to the optimal one. These near-optimal alignments may be alternate paths back through the dynamic programming matrix starting at the same high-scoring position. These alignments will use some of the same aligned base pairs (or amino acids for protein sequences) that were also used in the optimal alignment. Other alignments may use

the same base pairs but have them aligned differently. Yet a third kind of alternative alignment may use entirely different base pairs to achieve a range of possible alignments. For the comparison of long DNA sequences the detection of near-optimal (or high-scoring) non-intersecting local alignments is particular useful [41, 30]. Definition 4.1.1 gives the concept of non-intersecting.

Definition 4.1.1. Two local alignment paths in the similarity matrix are said to be non-intersecting if they do not share any cell in matrix H .

The Waterman-Eggert algorithm [147] finds a series of non-intersecting near-optimal local alignments that is widely used. Huang [79] has produced a linear-space version of this method using graph theory. Our approach is based on dynamic programming with some extensions to make it more suitable for efficient parallelization. It works as follows:

Definition 4.1.2. For each cell (i, j) in the similarity matrix H , define $st(i, j)$ to be the cell index of the starting position of the local alignment ending at (i, j) that has score $H(i, j)$.

The values $st(i, j)$ are calculated by the following recurrences:

$$st(i, j) = \begin{cases} (i, j) & \text{if } H(i, j) = 0 \text{ or } i = 0 \text{ or } j = 0 \\ st(i - 1, j - 1) & \text{if } H(i, j) = H(i - 1, j - 1) + sbt(S_1[i], S_2[j]) \\ st(i, j - Eg(i, j)) & \text{if } H(i, j) = E(i, j) \\ st(i - Fg(i, j), j) & \text{if } H(i, j) = F(i, j) \end{cases}$$

where:

$$Eg(i, j) = \begin{cases} 1 & \text{if } E(i, j) = H(i, j - 1) - \alpha \text{ or } j = 1 \\ Eg(i, j - 1) + 1 & \text{if } E(i, j) = E(i, j - 1) - \beta \end{cases}$$

$$Fg(i, j) = \begin{cases} 1 & \text{if } F(i, j) = H(i-1, j) - \alpha \text{ or } j = 1 \\ Fg(i-1, j) + 1 & \text{if } F(i, j) = F(i-1, j) - \beta \end{cases}$$

The start point $st(i_{max}, j_{max})$ of the maximal score at end point (i_{max}, j_{max}) can be easily computed in linear space using a strategy similar to the one shown in Figure 4.2. In order to determine k near-optimal non-intersecting alignments, the k highest similarity scores (together with the corresponding start/end points) that have different start points are stored during the linear-space computation of the *similarity matrix*. Thus, for the calculated k highest scores $H(i_1, j_1), \dots, H(i_k, j_k)$ holds $st(i_p, j_p) \neq st(i_q, j_q)$ for all $1 \leq p, q \leq k$. Theorem 4.1.1 shows that these alignments do not intersect for each $m \neq n$. Subsequently, the divide-and-conquer algorithm (see Section 4.1.2) is applied k times to compute the actual alignments from the lower right neighbor of $st(i_p, j_p)$ to (i_p, j_p) for $p = 1, \dots, k$. The detected alignments of this algorithm using $k = 4$ and the sequences and parameters of Figure 4.1 is displayed in Figure 4.5. The four calculated start/end points are $(1,3)/(8,11)$, $(0,5)/(6,12)$, $(4,0)/(9,5)$ and $(1,1)/(8,8)$. The corresponding alignments computed by the divide-and-conquer methods are displayed as shaded cells. Note that the algorithm only requires linear space, i.e. the complete matrix H does not need to be stored.

Theorem 4.1.1. *Given are two local alignment paths A_1 and A_2 in a similarity matrix with distinct start points $st(i_1, j_1)$, $st(i_2, j_2)$ and distinct end points (i_1, j_1) , (i_2, j_2) , i.e. $st(i_1, j_1) \neq st(i_2, j_2)$ and $(i_1, j_1) \neq (i_2, j_2)$. Then A_1 and A_2 are non-intersecting.*

Proof. Assume A_1 and A_2 are not non-intersecting. Then there exists at least one cell P in matrix H that is shared by both alignment paths. Therefore, it follows: $st(i_1, j_1) = st(p) = st(i_2, j_2)$, where $st(i_1, j_1) = st(i_2, j_2)$ contradicts the assumption.

□

		C=0	C=1	C=2	C=3	C=4	C=5	C=6	C=7	C=8	C=9	C=10	C=11	C=12	
		A	T	C	T	C	G	T	A	T	G	A	T	G	
R=0		0	0	0	0	0	0	0	0	0	0	0	0	0	
R=1	G	0	0	0	0	0	2	1	0	0	2	1	0	2	
R=2	T	0	0	2	1	2	1	1	4	3	2	1	1	3	2
R=3	C	0	0	1	4	3	4	3	3	3	2	1	0	2	2
R=4	T	0	0	2	3	6	5	4	5	4	5	4	3	2	1
R=5	A	0	2	2	2	5	5	4	4	7	6	5	6	5	4
R=6	T	0	1	4	3	4	4	4	6	5	9	8	7	8	7
R=7	C	0	0	3	6	5	6	5	5	5	8	8	7	7	7
R=8	A	0	2	2	5	5	5	5	4	7	7	7	10	9	8
R=9	C	0	1	1	4	4	7	6	5	6	6	6	9	9	8

Figure 4.5: Example of the computation of highest-scoring non-intersecting local alignments using $k=4$ and DNA sequences, substitution cost and gap penalties as in Figure 4.1.

Algorithm 4.1.1 describes the procedure of finding K highest scores and their start points, in linear space, with DNA sequences, substitution cost and gap penalties as in Section 4.1.1. We use a **struct** type called **Datatype** to store $m + 1$ points (the solid black points in Figure 4.2) that are needed for computing in linear space. the **Datatype** consists of two items: **value** and **startpoint** (**value** is the value of the point and the **startpoint** is its start point). Also, we use another **struct** type called **Topscore** to hold k highest scores and their start/end points. **Topscore** consists three items: **value**, **endpoint** and **startpoint**. In the algorithm, a , b , e , f are general points shown in Figure 4.2.

Algorithm 4.1.1. Finding Near-optimal Alignments

Vector **Datatype**: $RowData[0..m]$

Vector **Topscore**: $TopK[0..k - 1]$

```

FOR  $i \leftarrow 0$  TO  $m$  DO
   $RowData[j].value \leftarrow 0$ 
FOR  $j \leftarrow 1$  TO  $n$  DO {
  FOR  $i \leftarrow 1$  TO  $m$  DO {
    /*Determining the value*/

```

```

    vertical  $\leftarrow$  b.value - gap
    diagonal  $\leftarrow$  a.value + sbt( $S_1[i]$ ,  $S_2[j]$ )
    horizontal  $\leftarrow$  e.value - gap
    f.value  $\leftarrow$  max{vertical, diagonal, horizontal, 0}

    /*determining the start point*/
    IF vertical = max{vertical, diagonal, horizontal, 0} THEN
        f.startpoint = b.startpoint
    IF diagonal = max{vertical, diagonal, horizontal, 0} THEN
        f.startpoint = a.startpoint
    IF horizontal = max{vertical, diagonal, horizontal, 0} THEN
        f.startpoint = e.startpoint
    IF 0 = max{vertical, diagonal, horizontal, 0} and f.value  $\neq$  0 THEN
        f.startpoint = (i, j)

    /*updating k highest score points and corresponding start points*/
    IF f.value > TopK[k - 1].value THEN{
        IF  $\exists p \in (0..k - 1)$  and f.startpoint = TopK[p].startpoint THEN {
            IF f.value > TopK[p].value THEN
                REPLACE TopK[p] by f
        }
        ELSE {
            REPLACE TopK[k] by f
            make TopK[k - 1].value < TopK[k - 2] <  $\dots$  < TopK[0]
        }
    }
    e  $\leftarrow$  f
    a  $\leftarrow$  b
    b  $\leftarrow$  c
}

```

4.2 Mapping the Application onto the Hierarchical Grid Architecture

4.2.1 Parallelization of Sequence Alignment

Parallelization of long DNA sequence alignment consists of two parts:

- (1) Parallelization of the *similarity matrix* computation in order to calculate the k best start/end points.
- (2) Parallelization of the divide-and-conquer algorithm to calculate the actual alignments.

The parallelization of the *similarity matrix* calculation is based on the wavefront communication pattern. Figure 4.6 displays the dependency relationship: each cell (i, j) of the *similarity matrix* is computed from the cells $(i - 1, j)$, $(i, j - 1)$, $(i - 1, j - 1)$. The wavefront moves in anti-diagonals as depicted in Figure 4.6a, i.e. the shift direction is from north-west to south-east.

Figure 4.6: Wavefront computation of the *similarity matrix*: (a) shift direction, (b) dependency relationship.

Parallelization of the wavefront computation has been done in different ways depending on the particular parallel architecture being used [104]. On fine-grained architectures, the computation of each cell within an anti-diagonal is parallelized [129]. However, this technique is only efficient on architectures such as systolic arrays, which have an extremely fast inter-processor communication.

On coarse-grained architectures like homogeneous PC clusters it is more efficient to assign an equal number of adjacent columns to each processor as shown in Figure 4.7a. In order to reduce communication time further, matrix cells can be grouped into blocks.

Processor i then computes all the cells within a block after receiving the required data from processor $i - 1$. Figure 4.7 shows an example of the computation for 4 processors, 8 columns and a block size of 2×2 .

Figure 4.7: (a) Column-based division of an 8×8 matrix using 4 processors; (b) Wavefront computation for 4 processors, 8 columns and a 2×2 block size. The complete 8×8 matrix can then be computed in 7 iteration steps. Each processor P_i ($0 \leq i \leq 2$) except the last one (P_3) will send the cells in the most right column of the block it computes to the processor P_{i+1} .

Calculation of the actual alignment between a start/end point pair is only parallelized if the distance between these points is reasonably large. Aluru [29] presented a parallel solution of this problem using parallel prefix computations. However, this solution is only efficient for architectures with relatively fast inter-processor communication. The parallel solution for determining the optimal path on the hierarchical grid is as follows:

Let us define special columns (SC) as the last columns of the parts of the *similarity matrix* allocated to each processor, except the last processor. If we can identify the intersection of an optimal path with the special columns, we can split the problem into sub-problems and solve them sequentially on each processor using linear space algorithm (see Figure 4.8). The solutions of the sub-problems are then concatenated to get the optimal alignment. This is achieved as follows:

It remains to describe how the intersection of an optimal path with the special columns can be computed. We only store information on the special columns of a *similarity matrix* H . In addition, we store the most recently computed row of H in order to compute the next row. For each cell of H , the value of the cell and the $\langle SC \text{ number}, \text{row number} \rangle$ tuple of the entry in the closest special column to the left that lies on an optimal path from $H(0,0)$ to the entry are computed. Call such a tuple a *pointer* to the previous special column. This essentially gives the ability to perform a traceback through special columns, without considering other columns. Therefore, the whole traceback task can be divided into independent multiple subtasks. The computation for each subtask can be done in linear space in a similar way described in Section 4.1.2.

Figure 4.8: Parallel solution for determining the traceback in linear space on four processors, where SC_i stands for special column number i .

4.2.2 Mapping onto the Hierarchical Grid using Static Load Balancing

In this approach, the mapping has two levels of partitioning. Firstly, the matrix is divided into parts of adjacent columns equal to the numbers of clusters. Secondly, the part within each cluster is further partitioned. The computation is then performed in

the same way as shown in Figure 4.7b. This reduces the inter-cluster data transfer to a single column per iteration step. In order to avoid bottlenecks on the heterogeneous hierarchical grid architecture, the number of columns assigned to each cluster depends on its computational capabilities. The order of the partitioning depends on the inter-cluster bandwidths. Figure 4.9 displays the partitioning on our architecture.

Figure 4.9: Partitioning of the wavefront computation on the hierarchical grid architecture. The number of columns assigned to each cluster is determined by its performance to execute the Smith-Waterman algorithm using MPI. Inside a cluster, each processor (except the last one) will send the cells in the most right column of the block it computes to the next neighboring processor over the intra-cluster communication. The last processor in each cluster (except the last one, cluster3) will send the cells in the most right column of the block it computes to the first processor of the next neighboring cluster over the inter-cluster communication.

The static load balancing approach can achieve good performance under the condition that there is no disturbance from other applications. Figure 4.14 shows the speedups of different number of processors and different inter-cluster bandwidths. If the execution of a job is disturbed by another application in a node, it might become the bottleneck for the whole system. An experiment is therefore designed to measure the performance degradation.

In order to scale the extent to which a disturbance affects the application performance, we have defined PDRD (*performance degradation ratio under disturbance*) as [42, 43, 153]:

$$PDRD = \left(\frac{T' - T}{T} \right) \times 100\%$$

where T' denotes the execution time under disturbance and T denotes the execution time without any disturbance. Smaller PDRD values indicate a better robustness of the application to disturbance.

Figure 4.10 shows how is the effect of disturbance by an application running on the 2.60GHz CPU that uses around 50% CPU time.

Figure 4.10: PDRD (*Performance degradation ratio under disturbance*) for the implementation using static load balancing.

4.2.3 Mapping onto the Hierarchical Grid using Dynamic Load Balancing

A dynamic load balancing approach is to overcome the previously described problems. With dynamic load balancing, jobs are allocated dynamically to idle nodes in order to:

- (1) Keep all the nodes busy.
- (2) Minimize the overhead of the load balancer.

A dynamic load balancing mechanism can be implemented using two approaches: *system-level dynamic load balancing* and *application-level dynamic balancing*. The former uses system state information to make decision at runtime on how to dispatch workloads. The latter implements dynamic load balancing as part of the application.

The dominant programming paradigm for clusters is message passing using Message Passing Interface (MPI) [15]. With the goal of coupling several clusters in a computational grid environment, grid-enabled MPI solutions are developed by some institutions. By now there are four most widely used MPI implementations for grid environment, which are PACX-MPI [69], MPICH-G2 [89], Stampi [81] and MagPIe [93]. Unfortunately all these grid-enabled MPI implementations do not support *system-level dynamic load balancing* [78]. Therefore, the dynamic load balancing in our paper is an *application-level dynamic balancing*, which means balancing mechanism and balancer are implemented by users.

Traditional Master-Slave Paradigm

The master-slave paradigm is a widely used technique to implement dynamic load balancing. It is described in Section 3.3.3.

The implementation of long DNA sequence alignment on the hierarchical grid using this approach works as follows: the *similarity matrix* is divided into rectangular blocks as shown in Figure 4.11. The computation of a rectangular block is assigned by the *master* to an available *slave*. This requires the sending of the left column and upper row of the block to be computed to the *slave*. The *slave* then returns the right column and bottom row of the computed block to the *master*. Compared to static load balancing implementation, the advantage of this approach is its robustness under disturbance (see Figure 4.17). Unfortunately, it requires much more communication. For example, for a

Figure 4.11: Data transfer between *master* and *slave* for finishing a sub-job. The *slave* has to receive the upper row and left column from the *master* in order to start computing a new block. Afterwards the *slave* returns the right column and bottom row of this block to the *master*.

similarity matrix size of $100k \times 100k$ and a block size of 1250×1250 , the overall data to be transferred sequentially through inter-cluster link is around 240MBytes. This makes the approach very sensitive to the inter-cluster bandwidth (see Figure 4.16).

New Scheduler-Worker Technique

We present a new technique named scheduler-worker in order to achieve good robustness under disturbance as well as high performance for low inter-cluster bandwidth. The scheduler-worker technique can be regarded as an adjusting master-slave paradigm. The *workers* report their computing performances to the *scheduler* every time they finish a piece of work. The *scheduler* then produces a new job allocation form depending on every node's performance and broadcasts it to each *worker*. The new job allocations are implemented by exchanging data among *workers* (see Figure 4.12).

An example of this technique is shown in Figure 4.13. Initially, the *similarity matrix* is partitioned in the same way as in the static load balancing approach. During the computation each *worker* reports its performance to the *scheduler*. We define *NP* (*node*

Figure 4.12: Scheduler-worker technique.

performance) as:

$$NP_i = \frac{SB_i}{T}$$

where SB_i denotes the size of the block assigned to $worker_i$ and T denotes the time for finishing the computation of this block. NP-values describe the currently available computing power in a node.

The *scheduler* judges whether a new job allocation is needed depending on all the NPs. If there exists a disturbance in one node, the sizes of blocks will be rearranged. The equation is as follows:

$$SB_i = \frac{Size_x}{\sum_{j=0}^N \frac{NP_i}{NP_j}}$$

where SB_i is the size of the block allocated to $worker_i$, NP_i is the node performance of $worker_i$ and $Size_x$ is the size of Sequence X .

The *workers* will receive the job allocation form produced by the *scheduler*. If no new allocation is needed, the *workers* continue to compute without any interruption. If a new allocation is needed, the *workers* implement the rearrangement by exchanging data between *workers*.

The scheduler-worker technique has several advantages. Its speedup without distur-

Figure 4.13: Example of the scheduler-worker technique with the assumption that each cluster has only two processors. The *similarity matrix* is divided into blocks (denoted by the rectangles) consisting of cells. Each block has a certain number of rows of cells, while number of columns of cells in a block varies depends on the computing power of the processor computing the block. Each *worker* will report its computing performance to the *scheduler* every time it finishes a piece of work (such as at *send*). The *scheduler* judges whether a new job allocation is needed depending on all the computing performances. If a new job arrangement is needed, the *scheduler* will produce a new job allocation form and sends it to each *worker* (such as at *receive*). The *workers* implement the job rearrangement by exchanging data between *workers*. In this example, processors P_{01} is allocated with larger number of columns, some of which are computed by processor P_{00} and P_{10} before the *rearrange*. Therefore, processors P_{00} and P_{10} need to transfer the corresponding cells to processor P_{01} . These cells are indicated by bold lines (*a* and *b*) in the figure. In the same way, processor P_{11} need to transfer the cells on the bold line *c* to processor P_{10} and processor P_{20} need to transfer the cells on the bold line *d* to processor P_{11} .

bance is only slightly slower than the static load balancing implementation (see Figure 4.18). However, it achieves much better PDRD values (see Figure 4.19). It has only slightly worse PDRD values compared to the implementation using the master-slave approach. However, because of the significantly reduced data transfer, it achieves much

higher speedups for low inter-cluster bandwidths.

4.3 Performance Evaluation

4.3.1 Speedup and PDRDs

One of the goals of the paper is to investigate the performance of applications on grid systems with different inter-cluster connection bandwidth. In our experiments, intra-cluster application-level bandwidth is almost 190MByte/s. We investigate the speedups of each implementation on the grid system with 4 different application-level inter-cluster bandwidths: 8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s. How to construct this kind of test environment is described in Section 3.2. The method for computing speedup is [42, 43]:

$$\begin{cases} speedup = \frac{3 \times RT_{cluster1(1)} \times F_{cluster1(1)}}{RT_{real}} = \frac{3 \times RT_{cluster1(2)} \times F_{cluster2(1)}}{RT_{real}} = \frac{3 \times RT_{cluster3(1)} \times F_{cluster3(1)}}{RT_{real}} \\ F_{cluster1(1)} + F_{cluster2(1)} + F_{cluster3(1)} = 1 \end{cases}$$

where $RT_{cluster1(1)}$, $RT_{cluster2(1)}$ and $RT_{cluster3(1)}$ are the runtimes for one processor in cluster1, cluster2 and cluster3; $F_{cluster1(1)}$, $F_{cluster2(1)}$, $F_{cluster3(1)}$ are the fractions of tasks finished by each processor when three processors are used; RT_{real} is the runtime when multiple processors which equally distributed in each cluster are used.

The two sequences used in the experiments are the first 3Mbp subsequences on human chromosome X and drosophila melanogaster chromosome X. These genome sequences are downloaded from [7]. Using these two sequences, the runtimes for one processor in the three clusters are equal to 2,217,829 seconds, 604,853 seconds and 2,167,437 seconds respectively.

1. Performance of the Implementation using Static Load Balancing

The execution times and speedups of the grid-enabled DNA sequence alignment ap-

plication, using static load balancing and without disturbance, are shown in Figure 4.14. Figure 4.15 shows the PDRDs when one application is running in one 2.60GHz CPU and uses around 50% CPU time to disturb the execution of our application.

Number of Processors	time(seconds)			
	8MByte/s	1MByte/s	0.5MByte/s	0.3MByte/s
3	433585	487783	532127	585340
6	225131	260151	325189	418100
9	167240	198420	234136	266063
12	128646	144528	177375	220883
15	102691	114772	139366	169663
18	86717	96750	121945	142765

Figure 4.14: Execution times and speedups of the implementation, using static load balancing and without disturbance, on the grid system with different application-level inter-cluster bandwidths (8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s).

2. Performance of the Implementation using Master-Slave

The execution times and speedups of the grid-enabled DNA sequence alignment application, using master-slave load balancing and without disturbance, are shown in Figure 4.16. Figure 4.17 shows the PDRDs when one application is running in one 2.60GHz CPU and uses around 50% CPU time to disturb the execution of our application.

Figure 4.15: PDRD (*Performance degradation ratio under disturbance*) for the implementation using static load balancing.

3. Performance of the Implementation using Scheduler-Worker

The execution times and speedups of the grid-enabled DNA sequence alignment application, using scheduler-worker load balancing and without disturbance, are shown in Figure 4.18. Figure 4.19 shows the PDRDs when one application is running in one 2.60GHz CPU to disturb the execution of our application. The disturbance frequency is once every three minutes.

4. Speedups of the Implementation using Scheduler-Worker for Varying lengths of Sequences

We have conducted an experiment to show that our grid implementation using scheduler-worker load balancing is able to scale well for varying lengths of sequences. Figure 4.20 illustrates the speedups of our systems for aligning varying lengths of sequences (100kbp *times* 100kbp, 1Mbp *times* 1Mbp, and 5Mbp *times* 5Mbp). The inter-cluster bandwidth is 8MByte/s.

Number of Processors	time(seconds)			
	8MByte/s	1MByte/s	0.5MByte/s	0.3MByte/s
3	532127	557467	585340	594254
6	308073	418100	487783	557467
9	201841	316400	377639	450261
12	156090	249081	300174	377639
15	130075	201841	266063	325189
18	100920	169663	225130	278733

Figure 4.16: Execution times and speedups of the implementation, using master-slave dynamic load balancing and without disturbance, on the grid system with different application-level inter-cluster bandwidths (8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s). The *similarity matrix* is divided into blocks of size 30000×30000 cells.

4.3.2 Performance Comparison

Speedup comparisons of static, master-slave and scheduler-worker load balancing without disturbance are shown in Figure 4.21. It can be seen that static and scheduler-worker load balancing achieve better performance than master-slave load balancing. In particular, master-slave load balancing is very sensitive to inter-cluster bandwidth. This is due to the increased data transfer required by the strategy. The PDRD comparisons of using static, master-slave and scheduler-worker load balancing when one application is running

Figure 4.17: PDRD (*Performance degradation ratio under disturbance*) for the implementation using the master-slave technique.

in one 2.60GHz CPU to disturb the execution of our application are shown in Figure 4.22. It indicates that master-slave and scheduler-worker are more robust to disturbance than static load balancing. Figure 4.23 shows the $Speedup \times (1 - PDRD)$ comparisons of static, master-slave and scheduler-worker load balancing for different application-level inter-cluster bandwidths. These comparisons show that the scheduler-worker technique performs best with respect to the product of these two parameters (speedup and PDRD).

4.3.3 Quality of Produced Alignments

Quality of Produced Alignments

We have compared the alignments found by our system to the alignments found by PatternHunter II. Our system has the same sensitivity as the Smith-Waterman algorithm, since it computes exhaustively the similarity matrix. PatternHunter [19, 101, 106] is a new generation of spaced seed-based method for general-purpose homology searching. PatternHunter II claims to approach Smith-Waterman sensitivity at a speed 3000 times

Number of Processors	time(seconds)			
	8MByte/s	1MByte/s	0.5MByte/s	0.3MByte/s
3	450261	508991	557467	557521
6	229545	260151	334480	403683
9	167240	209050	243891	272251
12	131537	148187	185822	238914
15	102691	117068	142765	172158
18	89364	97556	125879	144528

Figure 4.18: Execution times and speedups of the implementation, using scheduler-worker load balancing and without disturbance, on the grid system with different application-level inter-cluster bandwidths (8MByte/s, 1MByte/s, 0.5MByte/s and 0.3MByte/s).

faster than Smith-Waterman [101].

PatternHunter II [101] uses the Smith-Waterman algorithm as a benchmark to evaluate its sensitivity. The experiment conducted in [101] aligns each mouse EST sequence to each human EST sequence. The sensitivity of PatternHunter II is evaluated as follows. If the Smith-Waterman algorithm finds a local alignment with score x for a pair of ESTs, the pair is regarded “found” by PatternHunter II if it finds a local alignment of score $\geq x/2$ for the same pair of ESTs. The sensitivity of PatternHunter II is then defined as $\frac{y'}{y}$, where y is the total number of aligned sequence pairs and y' is the number of pairs

Figure 4.19: PDRD (*Performance degradation ratio under disturbance*) for the implementation using the scheduler-worker technique.

Figure 4.20: Speedups of the implementation using scheduler-worker load balancing for aligning varying lengths of sequences.

found by PatternHunter II. The results in [101] show that the sensitivity of PatterHunter II exceeds 90% at a speed significantly faster than the Smith-Waterman algorithm.

Figure 4.21: Speedup comparisons of static, master-slave and scheduler-worker load balancing without disturbance for different application-level inter-cluster bandwidths (upper-left: 8MByte/s, upper-right: 1MByte/s, lower-left: 0.5MByte/s, lower-right: 0.3MByte/s).

However, this comparison is based on EST sequences, which are typically short (around 200bp long). Therefore, we have extended the comparison to much larger sequences by conducting two experiments. The first experiment compares closely related genomes and the second compares remotely related genomes. The experimental setups are as follows.

Figure 4.22: PDRD comparisons of static, master-slave and scheduler-worker load balancing for different application-level inter-cluster bandwidths (upper-left: 8MByte/s, upper-right: 1MByte/s, lower-left: 0.5MByte/s, lower-right: 0.3MByte/s).

- Experiment 1:

- Sequences: 12 sequences of the Mycoplasma bacteria family [8]. The average length of these sequence is around 1Mbp.
- Parameters: PatternHunter II default score scheme: match = 1, mismatch = -1, gapopen = -5, gapextension = -1. The parameter '-0 64' is used in running PatternHunter II to optimize its alignments.
- Results: For these closely related genomes, the sensitivity of PatternHunter II

Figure 4.23: $Speedup \times (1 - PDRD)$ comparisons of static, master-slave and scheduler-worker load balancing for different application-level inter-cluster bandwidths (upper-left: 8MByte/s, upper-right: 1MByte/s, lower-left: 0.5MByte/s, lower-right: 0.3MByte/s).

is 97%. There are only two pairs which can't be found by PatternHunter II.

The two pairs are shown in Table 4.1.

- Experiment 2:

- Sequences: First 3Mbp DNA sequence of human chromosome X and the drosophila_melanogaster genome, which are downloaded from [7]. Each of the 3Mbp DNA sequences is divided equally into three segments of length 1Mbp

each.

- Parameters: Since the genomes are relatively unrelated, we are using the following scoring scheme: match = 2, mismatch = -1, gapopen = -10, gapextension = -1. The parameter ‘-0 64’ is used in running PatternHunter II to optimize its alignments.
- Results: Table 4.2 shows the top scores of alignments by our system and PatternHunter II. It can be seen that none of the pairs are found by PatternHunter II.

Table 4.1: Two pairs of Mycoplasma bacteria genomes which can’t be found by PatternHunter II using its default score scheme: match = 1, mismatch = -1, gapopen = -5, gapextension = -1. The parameter ‘-0 64’ is used in running PatternHunter II to optimize its alignments.

<i>Sequence₁</i>		<i>Sequence₂</i>		top score by our system	top score by PatternHunter
name	length(bp)	name	length(bp)		
Mycoplasma_ genitalium	583k	Mycoplasma_ pneumoniae	821k	22461	6827
Mycoplasma.hyo- pneumoniae_232	872k	Mycoplasma.hyo- pneumoniae_7448	912k	464344	101610

Comparison to Block-Partitioning Method

Parallelizing large genome alignment can be achieved by partitioning long sequences into blocks with overlaps between adjacent blocks. This method has for example been used for human-mouse genome comparison in [130]. Each block of the first sequence is aligned independently to each block of the other sequence. Final alignments are then produced by merging alignments in overlapping areas. However, this method is likely to lose long optimal alignments compared to the Smith-Waterman wavefront partitioning method. We have conducted a corresponding experiment to compare our parallel method with the block-partitioning method. The sequences used are the M.pneumoniae genome of length

Table 4.2: The top scores of alignments for aligning segments of length 1Mbp of the human chromosome X and the drosophila.melanogaster genome by our grid system and by PatternHunter II using the score scheme: match = 2, mismatch = -1, gapopen = -10, gapextension = -1. The parameter ‘-0 64’ is used in running PatternHunter II to optimize its alignments.

$Sequence_1$	$Sequence_2$	top score by our system	top score by PatternHunter II
human_chro_x_segment_1	drosophila_chro_x_segment_1	9336	323
human_chro_x_segment_1	drosophila_chro_x_segment_2	6046	408
human_chro_x_segment_1	drosophila_chro_x_segment_3	16795	285
human_chro_x_segment_2	drosophila_chro_x_segment_1	6536	277
human_chro_x_segment_2	drosophila_chro_x_segment_2	6123	377
human_chro_x_segment_2	drosophila_chro_x_segment_3	10346	*
human_chro_x_segment_3	drosophila_chro_x_segment_1	9331	*
human_chro_x_segment_3	drosophila_chro_x_segment_2	6640	299
human_chro_x_segment_3	drosophila_chro_x_segment_3	16356	*

*: After using up about 500MB memory, PatternHunter II exits with the information: “PatternHunter has exceeded its memory limits”.

798kbp and the M.genitalium genome of length 567kbp. The alignments by our system are shown in Figure 4.24. In order to use the block-partitioning method, we divided the M.pneumoniae sequence into 3 equal-size blocks and the M.genitalium sequence into 2 equal-size blocks with 80kbp overlap between adjacent blocks. Figure 4.25 shows the alignments found by the block-partitioning method. It can be seen that the longest alignment found by our system (see Figure 4.24) can’t be found by the block partitioning method (See Figure 4.25).

4.4 Summary

In this chapter, we have described a dynamic programming algorithm to compute k non-intersecting near-optimal alignments in linear space. we have demonstrated how it can be efficient for aligning long DNA sequences with high accuracy. Three techniques to map this application onto a hierarchical grid system have been presented. We have

Figure 4.24: Alignments of *M.pneumoniae* and *M.genitalium* by our grid-enabled system. The score scheme: match = 2, mismatch = -2, gapopen = -10, gapextension = -1.

Figure 4.25: Alignments of *M.pneumoniae* and *M.genitalium* by the block-partitioning method. The score scheme is the same as in Figure 4.24.

studied the performance of these techniques under disturbance and for different levels of application-level inter-cluster bandwidths. The results show that the scheduler-worker technique performs best with respect to these two parameters.

Chapter 5

Constructing Large-size Suffix Trees

The suffix tree is a compact trie of all suffixes over a string. It is a key data structure in the field of bioinformatics, since it permits very efficient solutions to many string based problems. Examples include exact and approximate substring matching, the longest common substring problem and the maximal repetitive structure problem [74]. Consequently, many widely used large-scale computational genomic applications have achieved good performance using suffix trees, such as MUMmer [30], REPuter [98], and OASIS [113].

5.1 Ukkonen's Linear Time Algorithm for Constructing Suffix Trees

The first step to use suffix trees is to construct them. The most widely used algorithm for constructing suffix trees is the Ukkonen's algorithm [74]. The algorithm builds the suffix tree using implicit suffix trees. The definition of implicit suffix trees is presented in Section 2.1.2. We denote the implicit suffix tree of the string $S[1..i]$ by IST_i . Ukkonen's algorithm constructs an implicit suffix tree IST_i from each prefix $S[1..i]$ of string S of length l , starting from IST_1 and increasing i by one until IST_l is built. The true suffix tree for string S of length l is constructed from IST_l , and the time complexity for the

algorithm is $O(l)$.

5.1.1 High Level Description of Ukkonen's Algorithm

For a string S of length l , Ukkonen's algorithm is divided into l *phases*. In phase $i + 1$, IST_{i+1} is constructed from IST_i . Each phase $i + 1$ is further divided into $i + 1$ *extensions*, one for each of the $i + 1$ suffixes of $S[1..i + 1]$. In extension j of phase $i + 1$, the algorithm first finds the end of the path from the root labeled with substring $S[j..i]$. It then extends the substring by adding the character $S[i + 1]$ to its end, unless $S[i + 1]$ already appears there. So in phase $i + 1$ string $S[1..i + 1]$ is first put in the tree, followed by strings $S[2..i + 1]$, $S[3..i + 1]$, ..., $S[i + 1]$. Extension $i + 1$ of phase $i + 1$ extends the *empty* suffix of $S[1..i]$, that is, it puts the single character string $S[i + 1]$ into the tree (unless it is already there).

5.1.2 Suffix Extension Rules

To turn this high-level description into an algorithm, we must specify exactly how to perform a *suffix extension*. Let $S[j..i] = \beta$ be a suffix of $S[1..i]$. In extension j , when the algorithm finds the end of β in the current tree, it extends β to be sure the suffix $\beta S[i + 1]$ is in the tree. One of the following three extension rules may apply:

- **Rule 1:** In current tree, path β ends at a leaf. This is, the path from the root labeled β extends to the end of some leaf edge. To update the tree, character $S[i + 1]$ is added at the end of that leaf edge.
- **Rule 2:** No path from the end of string β starts with character $S[i + 1]$, but at least one labeled path continues from the end of β . In this case, a new leaf edge starting from the end of β must be created and labeled with character $S[i + 1]$. A

new internal node will also have to be created there if β ends inside an edge.

- **Rule 3:** Some path from the end of string β starts with character $S[i + 1]$. In this case the string $\beta S[i + 1]$ is already in the current tree, so we do nothing.

Using the suffix extension rules given above, once the end of a suffix β of $S[1..i]$ has been found in the current tree, only constant time is needed to execute the extension rules. Naively we could find the end of any suffix β in $O(\beta)$ time by walking from the root of the current tree. By this approach, extension j of phase $i + 1$ would take $O(i + 1 - j)$ time, IST_{i+1} could be created from IST_i in $O(i^2)$ time, and the suffix tree ST_m could be created in $O(m^3)$ time. Fortunately, the time can be reduced to $O(m)$ using a few implementation tricks.

5.1.3 Suffix Links and Skip/count Trick

Assume $x\alpha$ is a string over an alphabet Σ , where $x \in \Sigma$ and $\alpha \in \Sigma^*$. In a suffix tree, for an internal node A with path-label (from the suffix tree root to the node) $x\alpha$, there exists another node B with path-label α , then the pointer from A to B is called a *suffix link*. Although definition of suffix links does not imply that every internal node of an implicit suffix tree has a suffix link from it, it will, in fact, have one.

Recall that in phase $i + 1$ the algorithm locates suffix $S[j..i]$ of $S[1..i]$ in extension j , for j increasing from 1 to $i + 1$. Naïvely, this is accomplished by matching the string $S[j..i]$ along a path from the root in the current tree. Suffix links can shortcut this walk in each extension. The first two extension (for $j = 1$ and $j = 2$) in any phase $i + 1$ are the easiest to describe.

The end of the full string $S[1..i]$ must end at a leaf of IST_i since $S[1..i]$ is the longest string represented in that tree. That makes it easy to find the end of that suffix, and its

suffix extension is handled by Rule 1 of the extension rules. So the first extension of any phase is special and only takes constant time since the algorithm has a pointer to the end of the current full string.

Let string $S[1..i]$ be $x\alpha$, where x is a single character and α is a (possible empty) substring, and let $(A, 1)$ (A is a node) be the tree-edge that enters leaf 1. The algorithm next must find the end of string $S[2..i] = \alpha$ in the current tree derived from IST_i . The key is that node A is either the root or it is an internal node of IST_i . If it is the root, then to find the end of α the algorithm just walks down the tree following the path labeled α as in the naïve algorithm. But if A is an internal node, then A has a suffix link out of it to node B . further, since B has a path-label that is a prefix of string α , the end of string α must end in the subtree whose root is B . Consequently, in searching for the end of $x\alpha$ in current tree, the algorithm need not walk down the entire path from the root but can instead begin the walk from node B . This is the main point of including suffix links in the algorithm.

To describe the second extension in more detail. let γ denote the edge-label on edge $(A, 1)$. To find the end of α , walk up from leaf 1 to node A ; follow the suffix link from A to B ; and walk from B down the path (which may be more than a single edge) labeled γ . The end of the path is the end of α (See Figure 5.1). At the end of path α , the tree is updated followed the suffix extension rules. This completely describes the first two extensions of phase $i + 1$.

To extend any string $S[j..i]$ to $S[j..i + 1]$ for $j \geq 2$, repeat the same general idea: Starting at the end of string $S[j - 1..i]$ in the current tree, walk up at most one node to either the root or to a node A that has a suffix link from it; let γ be the edge-label of that edge; assuming A is not the root, traverse the suffix link from A to B ; then walk down the tree from B , following a path labeled γ to the end of $S[j..i]$; finally, extend the

Figure 5.1: Extension $j \geq 1$ in phase $i + 1$. Walk up at most one edge (labeled γ) from the edge of the path labeled $S[j - 1..i]$ to node A ; then follow the suffix link to B ; then walk down the path specifying substring γ ; then apply the appropriate extension rule to insert suffix $S[j..i + 1]$.

suffix to $S[j..i + 1]$ according to the extension rules.

In the extension $j + 1$ the algorithm walks down from node B along a path labeled γ . Directly implemented, this walk along γ takes time proportional to the number of characters on the edge. But a simple trick, called *skip/count* trick, will reduce the traversal time to something proportional to the number of the nodes on the path. It will then follow that the time for all the down walks in a phase is at most $O(m)$.

Let ρ denote the length of γ , and recall that no two labels of edges out of B can start with the same character, so the first character of γ must appear as the first character on exactly one edge out of B . Let ρ' denote the number of characters on the edge. If ρ' is less than ρ , then the algorithm does not need to look at any more of the characters on that edge; it simply skips to the node at the end of the edge. There it sets ρ to $\rho - \rho'$, sets a variable h to $\rho' + 1$, and looks over the outgoing edges to find the correct next

edge (whose first character matches character h of γ). In general, when the algorithm identifies the next edge on the path it compares the current value of ρ to the number of characters ρ' on that edge. When ρ is at least as large as ρ' , the algorithm skips to the node at the end of the edge, sets ρ to $\rho - \rho'$, sets h to $h + \rho'$, and finds the edge whose first character is character h of γ and repeats. When an edge is reached where ρ is smaller than or equal to ρ' , then the algorithm skips to character ρ on the edge and quits. Assumed that the γ path from B ends on that edge exactly ρ characters down its label. (See Figure 5.2)

Figure 5.2: The Suffix link and skip/count trick. In phase $i + 1$, substring γ has length ten. There is a copy of substring γ out of node A ; It is found three characters down the last edge, after four node skips are executed.

5.1.4 Edge-label Compression

With the edge-labels written explicitly, the edge-labels of a suffix tree for string S of length l might contain more than $O(l)$ characters in total. A simple, alternate scheme

exists for edge labeling. Instead of explicitly writing a substring on an edge of the tree, only write a *pair of indices* on the edge, specifying beginning and end positions of that substring of S (See Figure 5.3). Since the algorithm has a copy of string S , it can locate any particular character in S in constant time given its position in the string. Therefore, we may describe any particular suffix tree algorithm as if edge-labels were explicit, and yet implement that algorithm with only a constant number of symbols written on any edge.

Figure 5.3: The left tree is a part of the suffix tree for sting $S = abcdefabcuvw$, with the edge-labels written explicitly. The right tree shows the edge-labels compressed.

By using an index pair to specify an edge-label, only two numbers are written on any edge, and since the number of edges is at most $2m - 1$, the suffix tree uses only $O(m)$ symbols and requires only $O(m)$ space. This makes it more plausible that the tree can actually be built in $O(m)$ time.

5.1.5 Two More Tricks

Two more tricks will be explained. These tricks, plus with the suffix links and skip/count trick, will lead the algorithm to the desired linear time bound.

- **Rule 3 is a show stopper.** In any phase, if suffix tree extension rule 3 applies

in extension j , it will also apply in all further extension ($j + 1$ to $i + 1$) until the end of the phase. The reason is that when rule 3 applies, the path labeled $S[j..i]$ in the current tree must continue with character $S[i + 1]$, and so the path labeled $S[j + 1, i]$ does also, and rule 3 again applies in extension $j + 1, j + 2, \dots, i + 1$. Therefore, if rule 3 is used in extension j , then there is no need to explicitly find the end of any string $S[k..i]$ for $k > j$.

- **Once a leaf, always a leaf.** This is, if at some point in Ukkonen's algorithm a leaf is created and labeled j (for the suffix starting at position j of S), then the leaf will remain leaf in all successive trees created during the algorithm. This is true because the algorithm has no mechanism for extending a leaf edge beyond its current leaf. In more detail, once there is a leaf labeled j , extension rule 1 will always apply to extension j in any successive phase. So once a leaf, always a leaf. This observation permits another trick to be used. In phase $i + 1$, when a leaf edge is first created and would normally be labeled with substring $S[p..i + 1]$, instead of writing indices $(p, i + 1)$ on the edge, write (p, e) , where e is a symbol denoting "the current end". Symbol e is a global index that is set to $i + 1$ once in each phase. Therefore, the extension of leaf nodes can be achieved easily by increasing the variable e .

Ukkonen's algorithm can construct suffix trees in linear time with these implementation tricks [74]. Unfortunately, it is impractical for constructing large size suffix trees because of high memory overheads. For example, considering constructing the suffix tree of the human chromosome 1 of length around 240Mbp, this would lead to a memory requirement of about 3G bytes even using advanced space optimizing approaches [96]. The key feature of Ukkonen's algorithm is to make use of *suffix links*, which allows the incremental construction of suffix trees. However, *suffix links* incur a side effect that, internal nodes in a suffix tree have complicated connections (see Figure 5.4 for an ex-

ample). The requirement of maintaining these links over internal nodes in a suffix tree makes it impossible to decompose the construction based on Ukkonen's algorithm for distributed processors. Therefore new suffix tree construction approaches are required in order to construct suffix trees in parallel. Our solution is based on the idea that *suffix links* are constrained inside a part of suffix tree so that the part of suffix tree is possibly constructed independently. We named the new kind of links as the *compre suffix link* (see Definition 5.2.3). In the following section, a new data structure and the corresponding algorithm are described at length for constructing large size suffix trees in parallel.

5.2 Constructing a Suffix Tree with CPSTs

5.2.1 CPST: Common Prefix Suffix Tree

A way of dividing the problem (constructing suffix trees) into smaller sub-problems is to group the suffixes of a string first and then construct suffix trees for each suffix group. All suffixes of a string can be grouped according to the prefixes of each suffix. We define a *common prefix suffix tree* (CPST) of a string S to be a compact trie of a subset of the suffixes of the string which start with a same prefix (shown in Definition 5.2.1).

Definition 5.2.1. *Common prefix suffix tree (CPST):* For a given string S and a substring “*compre*” of S , a common prefix suffix tree, denoted as $CPST(S, \text{compre})$, is the compact trie of all suffixes of S which start with *compre*.

The suffix tree with corresponding *suffix links* for the string $S = \text{accattgaagcgttaccagttat}\$$ is shown in Figure 5.4. Figure 5.5 shows four CPSTs for the string $S = \text{accattgaagcgttaccagttat}\$$ and the common prefixes a , c , g , and t , i.e. $CPST(S, a)$, $CPST(S, c)$, $CPST(S, g)$, and $CPST(S, t)$. All CPSTs have fictitious connections to a root node. Considering the case *compre* = g , all suffixes in S starting with a g are $\text{gaagcgttaccagttat}\$, \text{gcgttaccagttat}\$,$

gttaccagttat\$, and *gttat*\$. The trie of all these suffixes are presented by the $CPST(S, g)$.

Figure 5.4: The suffix tree of the sequence *accattgaagcgttaccagttat*\$.

For a string $S \in \Sigma^*$ with $|\Sigma| = i$, the number of CPSTs for common prefixes of length l is i^l . For the DNA alphabet $\Sigma = \{a, c, g, t\}$ for example, the number of CPSTs with common prefixes length of 2 is $4^2 = 16$.

5.2.2 Constructing CPSTs in Linear Time.

A CPST is actually a subtree of a standard suffix tree. Once all the CPSTs of a given string have been constructed, the standard suffix tree of the string can be easily derived by concatenating the roots of every CPST with the virtual root of the standard suffix tree (shown in Figure 5.5).

Definition 5.2.2. Suffix chain: For each internal node of a suffix tree, there exists a directed chain of suffix links starting from this node and ending at the root node. This directed chain is called suffix chain.

$\textcircled{i}$: The suffix starting from location i of the input sequence

Figure 5.5: The CPSTs of the sequence $accattgaagcggtaccagttat\$$. using *compres*: a , c , g , and t .

For example in Figure 5.4, The *suffix chain* for internal node 3 is: $3 \rightarrow 4 \rightarrow 13 \rightarrow \text{root}$.

Definition 5.2.3. *Compre suffix link*: Given are two internal nodes A and B inside the same CPST. We define a *compre suffix link* from A to B , if A and B are part of the same suffix chain in the standard suffix tree, where A is before B and no other internal node of the CPST lies between A and B on this suffix chain. We also define a *compre suffix link* from B to the root of the CPST, if no other node inside the CPST is part of

the suffix chain between B and the root of the standard suffix tree.

Let's consider the *suffix chain* illustrated in Figure 5.6: $F \rightarrow A \rightarrow C \rightarrow D \rightarrow B \rightarrow E \rightarrow \text{root}$. Node A and B belong to $CPST_2$. We draw a *compre suffix link* from A to B , since the part of *suffix chain* between A and B ($A \rightarrow C \rightarrow D \rightarrow B$) doesn't contain any other nodes of $CPST_2$. Additionally, there is a *compre suffix link* from B to the root of $CPST_2$.

Figure 5.6: An example for the *compre suffix link*.

Figure 5.7 shows all *compre suffix links* for the example given in Figure 5.5.

In order to simplify the description, we use $s[i, j]$ to denote the substring of S starting at location i and ending at location j . $len(compre)$ denotes the length of *compre*. $CPST_pathlabel(N)$ denotes the characters on the path from the root to the internal node N . Furthermore, if suffix i starts with the prefix *compre*, we say suffix i is *valid* for $CPST(S, compre)$. The edge characters on a directed path from node A to a node B are denoted as $e(A, B)$.

The next two lemmata are the basis for efficient CPST construction using *compre suffix links*.

Lemma 5.2.1. *Given an internal node A of $CPST(S, compre)$ with $CPST_pathlabel(A) = S[l_1, l_2]$. Assume suffix l_3 is the next valid suffix for $CPST(S, compre)$ and $l_3 + len(compre) \leq$*

Figure 5.7: $CPST(S, a)$, $CPST(S, c)$, $CPST(S, g)$, and $CPST(S, t)$ for the string $S = accattgaagcgttaccagtat\$$ with corresponding compre suffix links.

l_2 . Then there exists an internal node B with $CPST_pathlabel(B) = S[l_3 + len(compre), l_2]$ and a compre suffix link from A to B .

Proof. The proof has two parts.

1. **Existence of B .** Since A is an internal node, there exists at least two substrings of S with $CPST_pathlabel(A) = S[l_1, l_2] = S[l_4, l_5]$. Hence, there are also two substrings of S with $S[l_3 + len(compre), l_2] = S[l_6, l_5]$. Therefore, there must be an

$$\text{pathlable}(A) = S[l_1, l_2]$$

$$\text{pathlable}(B) = S[l_3 + \text{len}(\text{compre}), l_2]$$

Figure 5.8: Illustration for Lemma 5.2.1.

internal node B with $CPST_pathlabel(B) = S[l_3 + \text{len}(\text{compre}), l_2]$.

2. **Existence of com-pre suffix link from A to B .** We show there exists a direct chain of *suffix links* from A to B by induction over $n = l_3 + \text{len}(\text{compre}) - l_1$. The claim then follows since A and B are inside the same $CPST$.

Basic Step: $n = 1$. Obviously, there is a directed *suffix link* from A to B .

Inductive Step: According to induction hypothesis, there is a directed chain of *suffix links* from A to a node C with $CPST_pathlabel(C) = S[l_3 + \text{len}(\text{compre}) - 1, l_2]$. Since there must also be a *suffix link* from C to B , it can be concluded that there is a directed chain of *suffix links* from A to B .

□

Lemma 5.2.2. *Given an internal node C with parent node A inside $CPST(S, compre)$. Assume there is a suffix link from A to another internal node B . Then there exists a node D below B with $e(A, C) = e(B, D)$ and there is a compre suffix link from C to D .*

Figure 5.9: Illustration for Lemma 5.2.2.

Proof. Let $CPST_pathlabel(A) = S[l_1, l_2]$ and $CPST_pathlabel(C) = S[l_1, l_4]$. Since there is a *compre suffix link* from A to B , it holds $CPST_pathlabel(B) = S[l_3, l_2]$, where l_3 is the next valid suffix for $CPST(S, compre)$ after l_1 and $l_3 < l_2$. Following Lemma 1, there exists an internal node D with $CPST_pathlabel(D) = S[l_3, l_4]$ and a *compre suffix link* from C to D . obviously, D is below B and $e(A, C) = s[l_2 + 1, l_4] = e(B, D)$.

□

Our algorithm constructs a CPST through orderly inserting *valid* suffixes for the CPST. In [74], the introduction of *suffix links* permits the usage of the skip/count trick. Based on this trick, Ukkonen's algorithm can achieve linear time complexity. the *compre suffix link* in CPSTs is the counterpart of *suffixlink* in standard suffix trees. We can locate the next node in the CPST by using the skip/count trick instead of traversing the

CPST from its root according to Lemma 2. Based on the definitions and lemmas above, the algorithm of constructing $CPST(S, compre)$ can be described as follows:

Algorithm 5.2.1. CPST construction algorithm.

Input: String $S = \alpha\$$, where $\alpha \in \Sigma^*$, $\$ \notin \Sigma$, and Σ is a finite alphabet.

Common prefix $compre \in \Sigma^*$ with $|compre| < |\alpha|$

Output: $CPST(S, compre)$

```

 $N = \text{number\_of\_valid\_suffixes}(S, compre);$ 
IF ( $N == 0$ ) RETURN ( $nil$ );
FOR  $i = 1$  TO  $N$  BEGIN
     $\nu(i) = \text{starting position of the } i^{th} \text{ valid suffix in } S;$ 
END
 $current\_node = CPST\_root;$ 
 $lemma2\_flag\_node = CPST\_root;$ 
FOR  $i = 1$  TO  $N$  BEGIN
    IF ( $(current\_node == CPST\_root) || (lemma2\_flag\_node == CPST\_root)$ )
         $new\_nodes\_info = \text{traversal}(current\_node, \nu(i), S);$ 
    ELSE
         $new\_nodes\_info = \text{skip\_count}(lemma2\_flag\_node, current\_node.edge\_labels);$ 
        IF ( $new\_nodes$  already exists)
             $new\_nodes\_info = \text{traversal}(new\_nodes\_info, \nu(j), S);$ 
         $create\_new\_nodes(new\_nodes\_info)$ 
         $create\_new\_CompreSuffixLink(new\_internal\_node, old\_internal\_node)$ 
         $current\_node = new\_internal\_node;$ 
         $lemma2\_flag\_node = current\_node.father.compre\_suffix\_link;$ 
    END
RETURN ( $CPST\_root$ );

```

We will be giving an example to show how Algorithm 1 works. In order to simplify the description, we consider the two-symbol alphabet $\Sigma = \{a, c\}$ and the input string $S = aaccaacaacac$. Figures from 5.10 to 5.15 show the processes of constructing $CPST(S, "a")$. For the *valid* suffix 1, 2, 5, 6 and 8, we need to traverse the CPST from its root in order to locate the new internal node because the *current_node* for inserting each of these suffixes is either the *CPST_root* or one of its children. In the step shown in Figure 5.13, a *compre suffix link* from node *A* to *B* is inserted. In the subsequent step shown in

Figure 5.14 the insertion of suffix 8 results in a new internal node C . Since the father of node C is the internal A and A has a *compre suffix link* to internal node B , we can use the skip/count trick to locate the next internal node. According to Lemma 2, there must exist an internal node D below B with $e(A, C) = e(B, D)$ (as shown in the Figure 5.15). The node D is the internal node produced by the insertion of the *valid* suffix 9. Obviously, we don't need to traverse the CPST from the root when inserting *valid* suffix 9. This is one of the key reasons why our algorithm can achieve the linear-time performance.

Figure 5.10: Inserting suffix 1 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$.

Figure 5.11: Inserting suffix 2 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$.

Theorem 5.2.1. *For an input sequence S and a substring *compre*, $CPST(S, \text{compre})$ can be constructed in linear time.*

Proof. Our algorithm constructs a $CPST(S, \text{compre})$ through orderly inserting *valid* suffixes. Assume that the insertion of the *valid* suffix $\nu(i)$ results in a new internal node A

Figure 5.12: Inserting suffix 5 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$.

Figure 5.13: Inserting suffix 6 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$.

with $CPST_pathlabel(A) = S[\nu(i) + len(compre)..l_i]$ (shown in Figure 5.16). The time complexity for this insertion is $O(l_i - \nu(i))$. For *valid* suffixes whose starting locations are in the range $[\nu_i, l_i]$ (such as suffixes $\nu(i+1)$ and $\nu(j)$ in Figure 5.16), we can use the skip/count trick [74] to insert them according to Lemma 2. The time complexity for these insertions using the skip/count trick are $O(m)$, where m is the number of suffixes whose starting locations are in the range $[\nu(i), l_i]$. Hence, the time complexity for inserting all

Figure 5.14: Inserting suffix 8 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$.

Figure 5.15: Inserting suffix 9 during the course of constructing $CPST(S, "a")$, where $S = aaccaacaacac$.

valid suffixes whose starting locations are in the range $[\nu_i, l_i]$ is $O(l_i - \nu(i)) + O(m)$. Obviously, this is linear to the length of the range. The whole input string is composed of these ranges. Therefore, the insertion of all *valid* suffixes can be accomplished in linear

time. □

Figure 5.16: Illustration for Lemma 5.2.1. The $\nu(i)$ is a *valid* suffix for $CPST(S, compre)$. The insertion of the suffix $\nu(i)$ results in a new internal node with $CPST_pathlabel(A) = S[\nu(i) + len(compre)..l_i]$. The $\nu(i+1)$ and $\nu(j)$ are the first and last *valid* suffixes in the range (ν_i, l_i) . The $[\nu_j, l_j]$ is the next similar range as $[\nu_i, l_i]$.

5.3 Constructing a Generalized Suffix Tree (GST) with CPSTs

A Generalized Suffix Tree (GST) is a suffix tree that combines the suffixes of a set $\{S_1, \dots, S_n\}$ of strings.

Our algorithms for constructing suffix trees can easily be adapted to construct GSTs. On constructing a GST for a set $\{S_1, \dots, S_n\}$ of strings, each string should be appended a different terminating symbol that does not occur anywhere in the set of strings. First build the CPSTs for S_1 using our linear time algorithm. Then starting at the roots of existing CPSTs, build the CPSTs for S_2 . When S_2 is fully processed, existing CPSTs will encode all the suffixes of S_1 and all the suffixes of S_2 . Using this way, the all CPSTs for the GST will be achieved.

5.4 Parallel Implementation of Constructing Large Suffix Trees using CPSTs

5.4.1 Input DNA Sequence

The DNA sequence used in this paper is human chromosome NC_000001.4 which is downloaded from [17]. The alphabet of actual DNA sequences consists of 16 characters, in which a , c , g , and t represent the four bases of DNA and r , y , w , s , m , k , b , d , h , and v represent undetermined base-pairs. In the paper, we only consider the determined bases a , c , g , and t . Hence, the human chromosome NC_000001.4 extracted by us is of length 222,827,884 bp.

5.4.2 Prefix Distribution in DNA Sequences

The purpose of presenting the new data structure called CPST is to divide a large-size suffix tree into a number of smaller size CPSTs first and then each CPST can be processed independently. This idea presumes that the suffix trees can be divided efficiently using CPSTs. However, this might not be possible for systematically biased sequences. Let's consider a worst case. For a sequence $S = aaaaaaaaaaaaaaaaaaaaaa$, all the suffixes of the sequence start with same prefix a . In such a case the idea of CPST is inefficient.

Fortunately, systematically biased DNA sequences rarely occur in practice. The appearance of the 4 symbols a , c , g , and t in actual DNA sequences is almost evenly distributed. This ensures that the number of DNA sequence suffixes starting with different possible prefixes are not severely imbalanced. Here we take human chromosome NC_000001.4 of length 222,827,884 as an example. Table 5.4.2 shows that the number of suffixes starting with different prefixes are well balanced. This means the suffix tree can be divided efficiently into sub-problems using CPSTs.

Table 5.1: The number of suffixes of the human chromosome NC_000001.4 of length 222,827,884 (only consider *a*, *c*, *g* and *t*) which start with 1-letter and 2-letter prefixes.

<i>compre</i>	Num of suffixes	<i>compre</i>	Num of suffixes
a	64875254	c	46493994
g	46483769	t	64974866
<i>compre</i>	Num of suffixes	<i>compre</i>	Num of suffixes
aa	21191409	ac	11189673
ag	15878823	at	16615349
ca	16200299	cc	12132633
cg	2256627	ct	15904435
ga	13313713	gc	9838754
gg	12121539	gt	11209763
ta	14169833	tc	13332934
tg	16226780	tt	21245318

5.4.3 Space Optimization

In order to reduce the memory required by our parallel suffix tree construction method, we are using two data structures, one for internal CPST nodes (see Figure 5.17) and one for CPST leaf nodes (see Figure 5.18). For a leaf node, we can take advantage of the fact that both the *son* and *compre_suffix_link* are equal to **NULL**. Hence, the leaf node representation requires less memory than the internal node representation.

Since we are using two types of nodes, a simple and efficient method for deciding whether a node pointer points to an internal node or a leaf node would be useful. Given a string S and a prefix *compre*, the number of leaf nodes of $CPST(S, \text{compre})$ is equal to the number of corresponding valid suffixes. Hence, a contiguous memory block, let's say $[address_1, address_2]$, for storing all leaf nodes can be allocated before the CPST construction process. During the CPST construction process, it can then be decided whether a node pointer points to a leaf node if it's address is in the range $[address_1, address_2]$.

```

struct INTERNALNODE
{
    unsigned long edge_start; //start location of the edge in the input string
    void * sons;              //pointer to son node
    void * right_sibling;     //pointer to right sibling node
    void * father;            //pointer to father node
    void * compre_suffix_link; //compre suffix link
}

```

Figure 5.17: Data structure for CPST internal nodes.

```

struct LEAFNODE
{
    unsigned long edge_start; //start location of the edge in the input string
    void * right_sibling;     //pointer to right sibling node
}

```

Figure 5.18: Data structure for CPST leaf nodes.

5.4.4 Mapping the Application onto the Hierarchical Grid Architecture

In Section 4.2 we evaluated three load balancing strategies for mapping a dynamic programming matrix calculation for pairwise sequence alignment onto a hierarchical grid architecture. However, suffix tree construction and pairwise sequence alignment have different characteristics, such as the rate of free-dependent tasks, the data access pattern, and the task heterogeneity. In particular, each CPST can be constructed independently.

The time required to construct an individual CPST depends on the length of the input string, the number of valid suffixes and the internal structure of the tree being constructed. Therefore, the use of a dynamic load balancing strategy is advantageous. We employ a traditional master-slave technique in our implementation. Assuming all slaves have access to the complete input string, the communication between *master* and *slaves* is minimal. Therefore, high efficiency can be achieved by having more tasks than number of *slaves*. The number of tasks in our suffix tree construction method can be

easily varied by adjusting the length of the common prefixes to the number of *slaves*.

5.5 Performance Evaluation

5.5.1 Runtimes, Speedups and PDRDs on the Grid System

We have used the hierarchical grid system described in Section 3.2 to evaluate the performance of our parallel implementation of large suffix tree construction. We have measured the execution times for varying number of processors per cluster and varying DNA sequence lengths. In our experiments, the length of *compre* is set to 3, i.e. the number of CPSTs is $4^3 = 64$. The individual tasks are mapped to processors using the dynamic load balancing strategy described earlier. Figure 5.19 shows the corresponding execution times and speedups. The method used for computing speedup is identical to the one used in Section 4.3.1.

Figure 5.20 shows the run-times while using different lengths of common prefixes: 1, 2 and 3. This is to investigate the overheads of constructing suffix trees using different number of CPSTs.

In order to scale the extent to which a disturbance affects the application performance, we have defined PDRD (*performance degradation ratio under disturbance*) in Section 4.2.2. Smaller PDRD values indicate a better robustness of the application to disturbance. Figure 5.21 shows how is the effect of disturbance by an application running on one CPU that uses around 50% CPU time.

Based on these measurements, we can make the following observations:

- (1) **Linear parallel speedups:** For constant sequence length and varying number of processors our implementation achieves linear speedups.

Figure 5.19: Execution times (upper) and speedups (lower) of our implementation on the hierarchical grid system. The used sequences are substrings of the human chromosome NC_000001.4.

Figure 5.20: Execution times for using varying lengths of common prefixes. The used sequences are substrings of the human chromosome NC_000001.4. Five CPUs are used for four slaves (on four 2.60GHz CPUs) and one master (on another cluster).

- (2) **Linear algorithmic complexity:** For constant number of processors and varying sequence length the overall runtime grows linearly.
- (3) **Low overhead for one CPST:** Similar run-times for using different number of CPSTs.
- (4) **Robustness:** By using a load balancing approach based on master-slave paradigm, the application achieves robustness under disturbance.
- (5) **Reduced memory consumption:** The measured average memory consumption of an individual CPST is $23 \times v(S, \text{compre})$ for the human chromosome NC_000001.4, where $v(S, \text{compre})$ denotes the number of valid suffixes for *compre* in *S*. The memory requirement would be $32 \times v(S, \text{compre})$ without using the space optimization.

Figure 5.21: PDRDs (Performance degradation ratio under disturbance) by an application running on one CPU that uses around 50% CPU time.

5.5.2 Performance for an Implementation with Data-servers

During the course of constructing CPSTs, the input string must reside in memory. This means every process in the parallel environment must allocate memory to hold the input sequence first and then only the remaining memory can be allocated to construct CPSTs. For a processor with 1G memory, the memory size which can be used by user applications is about 800M. If this kind of processors is used to construct suffix trees for a input string of length 500Mbp, the maximum memory size a processor could allocate for holding CPSTs is about 300M. Obviously the efficiency is low when the input sequence is very large. In order to solve the problem, we present a complement implementation by introducing a *data-server* concept.

If a PC cluster permits fast intra-cluster communication, it is possible that one or more nodes hold the whole input string while other nodes efficiently access contiguous

parts of the string by communication. We call this the *sharing input string idea*. Our implementation uses one or more *data-servers* which hold the whole input sequence. The processes constructing CPSTs (*constructors*) access the sequence through communication with these *data-servers*.

In order to decrease the communication between *dataservers* and *constructors*, we introduce the concept of *smallnode* and *largenode*. The communication between *dataservers* and *constructors* consists of two parts: 1) the *constructors* need to access the input string; and 2) the *constructors* need to get the edge labels of tree nodes. The communication in Case 1 has good efficiency since the *constructors* can get a whole block of the substring a time. The highly frequent and low efficient communication comes from Case 2, because the number of the nodes is large and the overhead of every communication is high. We classify the nodes of a CPST according to the lengths of their edge labels. The nodes whose edge-label lengths are larger than a criterion (*nodesize*) are called *largenodes*, or else called *smallnodes*. If the edge labels of the *smallnodes* are kept in their CPSTs, the access to these *smallnodes* doesn't need communication. Therefore, the communication in Case 2 will decrease.

The data structure for CPST nodes in this implementation is shown in Figure 5.22. The *smallnode_edge* is used to keep the characters on the coming edge of a *smallnode*. The *node_flag* is set to indicate the type, *largenode* or *smallnode*, of a node. Furthermore, for a *largenode*, it is assigned as the first character on the coming edge of the node. The addition of the *node_flag* in the data structure is attended to speedup traversing on CPSTs.

The implementation with *dataservers* supplies a way to construct parts of suffix trees for the input string whose length is very long, for example 500Mbp. It is particularly suitable for the clusters whose nodes have multiple processors, since the data transfer

```

typedef struct CPST_NODE
{
    unsigned long    edge_label_start;
    unsigned long    edge_label_end;
    struct CPST_NODE * sons;
    struct CPST_NODE * right_sibling;
    struct CPST_NODE * father;
    struct CPST_NODE * compre_suffix_link;
    unsigned char    * smallnode_edge;
    unsigned char    node_flag;
}

```

Figure 5.22: Data structure for CPST nodes in the implementation with *dataservers*.

between *data-servers* and *constructors* can use the fast inside-node communication. The cluster used to test the implementation performance is Compaq AlphaServer SC45 with 24 nodes, each node comprises four 1GHz Alpha processors with 1GB memory. The nodes are connected by a quadrics switch. Tests show that the application-level bandwidth inside a node is about 600MB/s. In the experiment, the length of *compre* is set to 3 and therefore the number of CPSTs is 64. The *nodesize* is set as 5. In order to make use of the very fast inside-node bandwidth, every node holds one *dataserver* and three *constructors*. Figure 5.23 shows the construction times for input strings with different lengths.

5.6 Adapting Operations on Suffix Trees to Distributed CPSTs

Our suffix tree construction method divides a standard suffix tree into multiple CPSTs. These CPSTs typically reside in different processors in a distributed grid environment. Gathering them into a single large-size suffix tree would introduce high communication cost and a large space consumption. A simple way to avoid these inefficiencies is to keep the CPSTs distributed. However, the drawback of this solution is that operations

Figure 5.23: Runtimes of the implementation with *data-servers* for input strings with different lengths.

designed for standard suffix trees have to be adapted to a collection of distributed CPSTs. Operations on suffix trees can be classified according their traversal type as follows [26]:

- a bottom-up traversal of the complete suffix tree;
- a top-down traversal of a subtree of the suffix tree;
- a traversal of the suffix tree using suffix links.

Table 5.2 shows some typical suffix trees operations with corresponding traversal type. In the following we briefly describe how one respective operation of each category can be efficiently adapted to a collection of distributed CPSTs.

- (1) **Exact set matching (top-down):** The exact set matching problem is to find all occurrences in text T of any pattern in a set of patterns $P = \{p_1, \dots, p_z\}$. The standard solution is to build the suffix tree Υ for T and then use this tree to successively search for all occurrences of each patten in P . Let us assume that we

Table 5.2: Operations on suffix trees and the kinds of traversals they require [26].

Operations on Suffix Trees	Type of tree traversal		
	Button-up	Top-down	Suffix-links
Super-maximal repeats	✓		
Maximal repeats	✓		
Maximal repeated pairs	✓		
Longest common substring	✓		
All-pairs suffix-prefix matching	✓		
Ziv-Lempel decomposition	✓		
Common substring of multiple strings	✓	✓	
Exact string matching		✓	
Exact set matching		✓	
Matching statistics		✓	✓

are using all *compres* of length of l for CPST construction and that all patterns in P are longer than l . The patterns in P can be partitioned according to their first l characters and send to the respective CPST. Each CPST can then independently search for its respective patterns. Assuming an uniform distribution of *compres* in T and P , this approach achieves maximal efficiency. Figure 5.24 shows the work flow of the operation of extract set matching on CPSTs.

Figure 5.24: Work flow of the exact set matching using CPSTs.

- (2) **Finding all maximal repeats (Button-up)**: Finding all maximal repeats in a string S can be solved by computing all left diverse nodes in the suffix tree Υ for S . Left diverse nodes can be identified by a button-up traversal of Υ considering the left character of any leaf of Υ (which is $S[i - 1]$ for the leaf representing suffix i). Obviously, this approach can be directly translated to a button-up traversal within each $CPST(S, compre)$. Since the traversal ends at the CPST-root, only maximal repeats of length larger equal than $len(compre)$ can be identified. However, in large-scale genome applications all repeats of interests certainly fulfill the requirement.
- (3) **Matching statistics (suffix links)**: Define $ms(i)$ to be the length of the longest substring of T starting at position i that matches a substring somewhere in S . These values are called the matching statistics of T w.r.t. S . Chang and Lawler [40] designed an algorithm to compute matching statistics in linear time using the standard suffix tree. Their approach can be efficiently adapted to distributed CPSTs as follows. $CPST(S, compre)$ is traversed in a simple left-to-right scan of T utilizing $compre$ suffix links, where only matching statistics for positions in T starting with the substring $compre$ are computed, i.e. only valid suffixes of T are considered. In each step, a valid suffix j of T is matched against the CPST until a mismatch occurs or all characters in T have been completely matched. This delivers the value $ms(j)$

5.7 Summary

In this chapter, we have introduced an efficient parallel algorithm for large-scale suffix tree construction using the CPST data structure. We have shown how a standard suffix tree can be divided into a number of CPSTs. Each CPST can then be processed independently by one cluster node. Our algorithm permits linear-time construction of CPSTs. It can

easily be adapted to construct generalized suffix tree. We have considered the space optimization methods without any sacrifice of suffix tree virtues. The solution through simplifying the leaf nodes make our implementation achieve the space complexity of 24.3 bytes per input character. Experiments on clusters and grid systems show that our implementation can achieve linear time and space complexity for a human chromosome DNA sequence.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The genomic sequence databases are growing rapidly, as shown in Figure 1.1. This growth rate will continue since more and more sequencing projects will be finished in the near future. As the data increases, so does the workload for processing, managing and sharing this data. Browsing and searching are expensive activities because extensive time is required to find appropriate information. Furthermore, with the continuing improvements in high-throughput genomic sequencing and the ever-expanding sequence databases, new advances in software programs for genomic sequence analysis are needed by the scientific community. Therefore, high performance computing is required to solve computational genomics problems efficiently today and even more in the near future. Computational grids are a new approach to gain high performance computing that combines existing computational resources instead of building a new one. Using computational grids to solve computational genomics problems is a promising and challenging research field, since large-scale computational genomics problems can benefit much from that kind of processing power.

However, to achieve good performance in a grid environment with its heterogeneous

nature, careful resource scheduling and problem partitioning is needed to efficiently “gridize” an application. In this thesis, we have investigated algorithms and techniques how to efficiently map bioinformatics applications onto computational grid systems. In particular, we have studied the following two important applications: 1) alignment of long genomic sequences; and 2) construction of large-size suffix trees.

Aligning long DNA sequences is a common and often repeated task in molecular biology. In this thesis, we have described a dynamic programming algorithm to compute k non-intersecting near-optimal alignments in linear space. we have demonstrated how it can be efficient for aligning long DNA sequences with high accuracy. Three techniques to map this application onto a hierarchical grid system have been presented. We have studied the performance of the these techniques under disturbance and for different levels of application-level inter-cluster bandwidths. The results show that the scheduler-worker technique performs best with respect to these two parameters. Through comparing with other genome aligning tools we show our approach can get high quality alignment.

The suffix tree is a key data structure for biological sequence analysis, since it permits efficient solutions to many string based problems. Existing linear time algorithms are impractical for constructing suffix trees for long DNA sequences because of high memory overheads and poor memory locality. In this thesis, we have introduced an efficient parallel algorithm for large-scale suffix tree construction using the CPST data structure. We have shown how a standard suffix tree can be divided into a number of CPSTs. Each CPST can then be processed independently by one cluster node. Our algorithm permits linear-time construction of CPSTs. Our implementation can achieve significant runtime savings on PC clusters and Grid systems.

6.2 Future Work

Our future work includes three parts. The first one is to present a new parallel computational model for heterogenous grid systems. In the future models, more sources of heterogeneity in general grid systems will be taken into account. Furthermore, their validity will be tested by performing experiments on real heterogeneous grid systems. The second one is to identify more applications in computational genomics that benefit much from the hierarchical grid architecture. This will include aligning multiple DNA sequences on the hierarchical grid architecture, since multiple sequence alignment is a very important part in the field of genomic sequence alignment. The third one is to apply more space saving techniques to our CPST construction algorithm and integrate the algorithm with some important bioinformatics tools based on suffix trees.

6.2.1 New Computational Model for Heterogenous Grid Systems

The need for solutions to complex computations based on grid computing has grown considerably. As a result, so has the need for models that can accurately predict the performance of parallel applications based on grid computing without constructing the algorithms themselves. Grid application developers have a need for analyzing design changes in a quick and efficient manner.

In Section 3.4, a parallel computational model has been introduced for a heterogeneous multi-cluster. This model takes into account two sources of heterogeneity in a heterogeneous multi-cluster: the computational nodes and the communication subsystem. The nodes can have different processor, memory and Input/Output features. With respect to communication, two questions have to be considered: the network interface and the network structure and technology. The model is based on the following five parameters:

1) *latency*: communication latency; 2) *overhead*: the time needed by a processor to send or receive a message is referred to as overhead. It consists of two independent parameters: *sender overhead*, and *receiver overhead*; 3) *gap between messages*: the gap for one byte long messages; 4) *gap per byte*: the available bandwidth to send long messages between nodes; and 5) *computational power*: the computational power for each node.

A future computational model for general grid systems should take into account more sources of heterogeneity. Furthermore, some experiments should be performed on some real grid systems to test the heterogeneous model's accuracy by comparing predictions and experimental results. Therefore, the future work on the parallel computational model consists of:

- (1) Computational resources in grid systems are shared, i.e. there are several users' tasks running at the same time. Therefore, the effective CPU time of an application depends on the number of jobs running on the node at that time. The future computational model will take into account this heterogeneity.
- (2) Grid systems usually contain some WAN ^{6.1} connections. The available bandwidth on these connections also varies over time. The future computational model will take into this heterogeneity.
- (3) Time analysis of parallel applications based on *master-slave*, *hierarchical master-slave* and *comprised master-slave* paradigms (as shown in 3.3.4) will be presented.
- (4) Practical experiments will be set up in order to test the model's accuracy by comparing predictions and experimental results.

^{6.1}Wide Area Network

6.2.2 Multiple DNA Sequence Alignment

In addition to pairwise sequence alignment, Multiple Sequence Alignment (MSA) is the other very important aspect in the field of genome sequence alignment. Compared to pairwise sequence alignment, MSA is an even more challenging task because its time and space complexities are in the order of the product of the length of the sequences. So many solutions for MSA can only compare short sequences, such as protein sequences. Since genomic sequences are much longer than protein sequences, aligning multiple genomic sequences is a much more difficult task.

There are some solutions available for aligning multiple sequences. MUSCLE is one of the latest methods [48].

MUSCLE

MUSCLE [48] is a new freely-distributed computer program to do MSA. It is a progressive method that can be described as three steps.

- (1) Pairwise alignment: pairwise alignment is done for each pair of sequences. These alignments are obtained by using k mer and the Kimura distance measures [48]. The scores caused by these alignments are stored in a distance matrix, which will be used to form a guide tree in the second step.
- (2) Guide tree construction: use the distance matrix obtained from the first step to build a guide tree. The guide tree is used to find out closely related sequences or a group of sequences that are aligned progressively in the last step to form the final MSA [127].
- (3) Profile alignment: produce final MSA by doing the pairwise alignment of sequences or group of sequences depending upon the topology of the guide tree.

Multiple Sequence Alignment on Grids

It is obvious that aligning pairwise sequences is an important part of aligning multiple sequences using MUSCLE. We have illustrated that the high demand for computational power, resulting from pairwise sequence alignment, can be satisfied by introducing the concept of grid. Therefore, MSA must be able to benefit much from computational grids. The remaining work to develop a grid-enabled MSA approach is to present a dynamic load balancing strategy and new parallel algorithms for mapping the other parts other than pairwise alignment onto grids.

6.2.3 Space savings for CPSTs

With introducing the idea of the small size leaf nodes, the space complexity of our implementation is 24.3 bytes per input character on average for human chromosome NC_000001.4. Further space optimization without suffering performance degradation is possible. For example, considerable space savings have been achieved by using a linked list representation [96].

6.2.4 Integrate CPST Constructing Algorithm with Important Bioinformatics Tools based on Suffix Trees

The suffix tree is a data structure that exposes the internal structure of a string in a deep way. It can solve many string computing problems with high efficiency. Consequently, many widely used large-scale computational genomic applications have achieved good performance using suffix trees, such as MUMmer [30, 97], REPuter [98], OASIS [113] and Weeder [121]. Among them MUMmer is the most popular tool for aligning long DNA sequences. Integrating the CPST algorithm with MUMmer would result in a very powerful sequence comparing tool.

MUMmer

MUMmer uses a combination of three ideas: suffix trees, the longest increasing subsequence (LIS) and Smith-Waterman alignment [30]. The novelty of the system derives from the integration of these ideas into a coherent system for large-scale genome alignment. The inputs to the system are two sequences, let's say Genome *A* and Genome *B*. The alignment process consists of the following steps.

- (1) Perform a maximal unique match (MUM) decomposition of the two genomes. This decomposition identifies all maximal, unique matching subsequences in both genomes. An MUM is a subsequence that occurs exactly once in Genome *A* and once in Genome *B*, and is not contained in any longer such sequence.
- (2) Sort the matches found in the MUM alignment and extract the longest possible set of matches that occur in the same order in both genomes. This is done using a variation of the well-known algorithm to find the LIS of a sequence of integers. Thus, we compute an ordered MUM alignment that provides an easy and natural way to scan the alignment from left to right.
- (3) Close the gaps in the alignment by performing local identifications of large inserts, repeats, small mutated regions, tandem repeats and SNPs.
- (4) Output the alignment including all the matches in the MUM alignment as well as the detailed alignments of regions that do not match exactly.

A Parallel MUMmer Implementation

MUMmer is packaged as three independent modules: suffix tree construction, sorting and extraction of the LIS, and generation of Smith-Waterman alignments for all the

regions between the MUMs. It is an open source project [30]. Since we have developed parallel suffix tree constructing algorithm and Smith-Waterman algorithm, it is possible to integrate these algorithms with MUMmer. This would result in a very powerful sequence alignment tool.

Bibliography

- [1] BioGrid in Japan. <http://www.biogrid.jp/>.
- [2] BioGrid in National University of Singapore.
<http://www.bic.nus.edu.sg/biogrid/index.html>.
- [3] BioGrid in Poland. <http://biogrid.icm.edu.pl/>.
- [4] ClusterWorld Conference and Expo. <http://www.beowulf.org/>.
- [5] Decypher Technology. <http://www.timelogic.com/>.
- [6] DNA, RNA and Protein Synthesis in Genetic Engineering and Society.
http://photoscience.la.asu.edu/Photosyn/courses/BIO_343/.
- [7] Ensembl Genome Browser. <http://www.ensembl.org/index.html>.
- [8] Genbank. <ftp://ftp.ncbi.nlm.nih.gov/genbank/genomes/Bacteria/>.
- [9] Genbank Growth. <http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html>.
- [10] Genomes Online Data. <http://www.genomesonline.org>.
- [11] Globus Project. <http://www.globus.org>.
- [12] IBM LoadLeveler. <http://www-128.ibm.com/developerworks/library/l-halinux3/>.
- [13] Instruction Systolic Array (ISA). <http://www.iti.fh-flensburg.de/lang/papers/isa/>.

-
- [14] Kestrel Parallel Processor in University of California, Santa Cruz.
<http://www.cse.ucsc.edu/research/kestrel/>.
 - [15] MPICH: A Portable Implementation of MPI. <http://www-unix.mcs.anl.gov/mpi/>.
 - [16] MPICH-G2 project in High-Performance Computing Laboratory of North Illinois University. <http://www3.niu.edu/mpi>.
 - [17] National Center for Biotechnology Information. <http://www.ncbi.nlm.nih.gov/>.
 - [18] North Carolina Bioinformatics Grid. <http://www.ncbiogrid.org/>.
 - [19] PatternHunter. <http://www.bioinformaticssolutions.com/products/ph/>.
 - [20] Platform LSF. <http://www.platform.com/products/LSF/>.
 - [21] Portable Batch System. <http://www.openpbs.org/>.
 - [22] SETI@home. <http://setiathome.ssl.berkeley.edu/>.
 - [23] SGI Enable Visulization on the Grid at Hokkaido University.
<http://www.gridtoday.com/03/0310/101160.html>.
 - [24] Sun Grid Engine. <http://gridengine.sunsource.net/>.
 - [25] Systola 1024 Processor Website. <http://goanna.cs.rmit.edu.au/heiko/Research.htm>.
 - [26] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Eplacing suffix trees with enhanced suffix array. *Journal of Discrete Algorithms*, 2:53–86, 2004.
 - [27] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the (logp) model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.
-

- [28] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
- [29] S. Aluru, N. Futamura, and K. Mehrotra. Biological sequence comparison using prefix computations. In *Proc. of the 13th IEEE International Parallel Processing Symposium*, pages 653–659, 1999.
- [30] A.L. Delcher and A. Phillippy and J. Carlton and S.L. Salzberg. Fast Algorithms for Large-scale Genome Alignment and Comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.
- [31] A. Andersson and S. Nilsson. Efficient Implementation of Suffix Trees. *Software Practice and Experience*, 25(2):129–141, 1995.
- [32] P. Aske, E.B. Henri, and F.H.H. Rutger. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. In *Proc. of the 5th IEEE HPCA '99*, pages 244–253, 1999.
- [33] S. Batzoglou, L. Pachter, J. Mesirov, B. Berger, and E.S. Lander. human and mouse gene structure: Comparative analysis and application to exon prediction. *Genome Res.*, 10:950–958, 2000.
- [34] R.D. Bjornson, A.H. Sherman, S.B. Weston, N. Willard, and J. Wing. TurboBLAST: A Parallel Implementation of BLAST Based on the TurboHub Architecture for High Performance Bioinformatics. In *HiCOMB*, 2002.
- [35] G.E. Blelloch. Vector Models for Data-Parallel Computing. *MIT Press*, 1990.

-
- [36] J.L. Bosque and L. Pastor. HLogGP: A new parallel computational model for heterogeneous clusters. In *Proc. of the 4th IEEE/ACM International Symposium on Cluster and the Grid*, pages 403–410, 2004.
 - [37] A.L. Brown. Constructing Chromosome Scale Suffix Tree. In *Proc. of the 2nd Asia-Pacific Bioinformatics Conference*, 2004.
 - [38] Jeremy Buhler. Efficient Large-Scale Sequence Comparison by Locality-Sensitive Hashing. *Bioinformatics*, 17(5):419–428, 2001.
 - [39] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proc. of Super Computer*, 2000.
 - [40] W.I. Chang and E.L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.
 - [41] Kun-Mao Chao, Jinghui Zhang, James Ostell, and Webb Miller. A local alignment tool for very long DNA sequences. *Computer applications in the Biosciences*, 11(2):147–153, 1995.
 - [42] C.X. Chen and B. Schmidta. An Adaptive Grid Implementation of Aligning Long DNA Sequences. *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Applications*, 2005.
 - [43] C.X. Chen and B. Schmidta. Constructing Suffix Trees in Parallel. *Constructing Large Suffix Trees on a Computational Grid*, to be appear.
 - [44] R. Clifford and M. Sergot. Distributed and Paged Suffix Trees for Large Genetic Databaseset. In *Proc. of the international conference on Computational methods in sciences and engineering*, 2003.
-

- [45] L. Colussi and A. De Col. A time and space efficient data structure for string searching on large texts. *Information Processing Letters*, 58(5):217–222, 1996.
- [46] G. Conant, A. Wagner, S. Plimpton, W. Old, and P. Fain. Parallel Genehunter: Implementation of a linkage analysis package for distributed memory architectures. In *HiCOMB*, 2002.
- [47] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann. Remote Engineering Tools for the Design of Pollution Control Systems for Commercial Boders. *International Journal Supercomputer Applications*, 10(2):208–218, 1996.
- [48] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.
- [49] Carsten Ernemann, Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. A Resource Management Architecture for Metacomputing Systems. In *Proc. of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, pages 39–46, 2002.
- [50] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottleneck in Suffix Tree Construction. In *Proc. of IEEE Annual Symposium on Foundations of Computer Science*, 1998.
- [51] D.G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. In *Research Report RC 87657, IBM T.J. Watson Research Center*, 1997.
- [52] D.G. Feitelson and L. Rudolph (Eds.). Job Scheduling Strategies for Parallel Processing. *Springer-Verlag, Lecture Notes in Computer Science*, 949, 1995.
- [53] D.G. Feitelson and L. Rudolph (Eds.). Job Scheduling Strategies for Parallel Processing. *Springer-Verlag, Lecture Notes in Computer Science*, 1162, 1996.

- [54] D.G. Feitelson and L. Rudolph (Eds.). Job Scheduling Strategies for Parallel Processing. *Springer-Verlag, Lecture Notes in Computer Science*, 1291, 1997.
- [55] J. Felsenstein. Phylogenies from Molecular Sequences: Inference and Reliability. *Annu. Rev. Genet.*, 22:521–565, 1988.
- [56] J. Fernando. Mapping Algorithms to Processor Arrays. In *Ph.D. Dissertation*, 1997.
- [57] P. Ferragina and R. Grossi. The string B-Tree: a new data structure for string search in external memory and its application. *Journal of the ACM*, 46(2):238–280, 1999.
- [58] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. of the 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375, 1997.
- [59] M.J. Flynn and K.W. Rudd. Parallel architectures. *ACM computing Surveys*, 28(1).
- [60] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. of the 10th ACM Symp. on the Theory of Computing*, pages 114–118, 1978.
- [61] I. Foster. What is the Grid? A Three Point Checklist. In *GRIDToday, July 20*, 2002.
- [62] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proc. of SC Conference*, 1998.
- [63] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

-
- [64] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.
- [65] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proc. of International Workshop on Quality of Service*, pages 181–188, 2000.
- [66] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc, 2004.
- [67] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal Supercomputer Applications*, 15(3), 2001.
- [68] Noriyuki Fujimoto and Kenichi Hagihara. Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid. In *Proc. of the 2003 International Conference on Parallel Processing (ICPP'03)*, 2003.
- [69] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computing Science*, 1998.
- [70] Leslie Grate, Mark Diekhans, David Dahle, and Richard Hughey. Sequence Analysis With the Kestrel SIMD Parallel Processor. In *Pacific Symposium on Biocomputing*, pages 323–34, 2001.
- [71] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference (Vol. 2)*. MIT Press, 1998.
-

- [72] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
- [73] P. Guerdoux-Jamet and D. Lavenier. SAMBA: Hardware Accelerator for Biological Sequence Comparison. *CABIOS*, 13(6), 1997.
- [74] D. Gusfield. *Algorithms on strings, trees and sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [75] Xiaoshan He, Xianhe Sun, and Gregor von Laszewski. QoS Guided Min-Min Heuristic for Grid Task Scheduling. *Journal of Computer Science and Technology, Special Issue on Grid Computing*, 18(4), 2003.
- [76] D.S. Hirschberg. A linear space algorithm for computing longest common subsequences. *Comm.ACM*, 18:341–343, 1975.
- [77] J.K. Hollingsworth and S. Maneewongvatana. Imprecise Calendars: an Approach to Scheduling Computational Grids. In *Proc. of the 19th IFIP International Conference on Distributed Computing Systems*, 1999.
- [78] Chao Huang, Orion Lawlor, and L.V. Kale. Adaptive MPI. In *Proc. of the 16th IFIP International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.
- [79] X. Huang and W. Miller. A time efficient, linear-space local similarity algorithm. *Advances in Applied Mathematics*, 12:337–357, 1991.
- [80] E. Hunt, M.P. Atkinson, and R.W. Irving. A Database Index to Large Biological Sequences. *The VLDB J.*, 7(3):139–148, 2001.

- [81] T. Imamura, Y. Tsujita, H. Koide, and H. Takemiya. An architecture of Stampi: MPI library on a cluster of parallel computers. *Lecture Notes in Computer Science*, 1908:200–207, 2000.
- [82] R.W. Irving. Suffix Binary Search Trees. In *Research Report in Department of Computer Science of University of Glasgow*, 1996.
- [83] M.A. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In *Proc. of the 7th IEEE Heterogeneous Computing Workshop*, pages 70–78, 1996.
- [84] R. Japp. Persistent Indexes for Data Intensive Applications. In *Proc. of the 20th British National conference on Databases*, 2003.
- [85] N. Jareborg, E. Birney, and R. Durbin. Comparative analysis of noncoding regions of 77 orthologous mouse and human gene pairs. *Genome Res.*, 9:815–824, 1999.
- [86] J. Kärkkäinen. Suffix Cactus: A Cross Between Suffix Tree and Suffix Array. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM95)*, pages 191–204.
- [87] J. Kärkkäinen and E. Ukkonen. Sparse Suffix Tree. In *Proc. of COCOON’96*, 1996.
- [88] N. Karonis, B. Toonen, and I. Foster. Distributed data mining on the grid. *Source Future Generation Computer Systems archive*, 18(8):1101–1112, 2002.
- [89] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.
- [90] K. Keahey, M.E. Papka, Q. Peng, D. Schissel, G. Abala, T. Araki, J. Burruss, E. Feibush, P. Lane, S. Klasky, T. Leggett, D. McCune, and L. Randerson. Grids

- for Experimental Science: The Virtual Control Room. In *Proc. of the Challenges of Large Applications in Distributed Environments (CLADE)*, 2004.
- [91] W.J. Kent. BLAT—the BLAST-like alignment tool. *Genome Res.*, 12(4):656–664, 2002.
- [92] W.J. Kent and W.J. Zahler. Conservation, regulation, synteny, and introns in a large-scale *C. briggsae*-*C. elegans* genomic alignment. *Genome Res.*, 10(8):1115–1125, 2000.
- [93] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjang. MagPIe: MPI’s collective communication operations for clustered wide area systems. In *Proc. of Ppopp’99*, pages 131–140, 1999.
- [94] M. Kopko, A. Ropelewski, and H. Nicholas. Multi-search – Parallel Sequence Database Searching Program. In <http://www.psc.edu/biomed/pages/dissem/multisearch.html>.
- [95] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(12):135–164, 2002.
- [96] S. Kurtz. Reducing Space Requirement of Suffix Trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.
- [97] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(R13), 2004.
- [98] S. Kurtz and C. Schleiermacher. REPuter: Fast Computation of Maximal Repeats in Complete Genomes. *Bioinformatics*, 15(5):426–427, 1999.

-
- [99] C. Lee, C. Kesselman, and S. Schwab. Near-Real-Time Satellite Image Processing: Metacomputing in C++. *IEEE Computer Graphics and Applications*, 16(4):79–84, 1996.
- [100] A. Lenstra. Factoring Integers Using the Web and the Number Field Sieve. In *Proc. of International Forum on Multimedia and Information Security*, pages 93–113, 1995.
- [101] Ming Li, Bin Ma, Derek Kisman, and John Tromp. PatternHunter II: Highly Sensitive and Fast Homology Search. *J. Bioinformatics and Computational Biology*, 2(3):417–440, 2004.
- [102] Y.A. Li and J.K. Antonio. Estimating the execution time distribution for a task graph in a heterogeneous computing system. In *Proc. of the 6th Heterogeneous Computing Workshop*, 1997.
- [103] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.
- [104] W.G. Liu and B. Schmidt. Parallel Design Pattern for Computational Biology and Scientific Computing. In *Proc. of Cluster 2003*, 2003.
- [105] J. Lund, F. Chen, A. Hua, B. Roe, M. Budarf, B.S. Emanuel, and R.H. Reeves. Comparative Sequence Analysis of 634Kbp of the Mouse Chromosome 16 Region of Conserved Synteny with the Human Velocardiofacial Region of Chr 22q11.2. *Genomics*, 63:374–383, 2000.
- [106] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

- [107] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *Proc. of the 8th Heterogeneous Computing Workshop*, 1999.
- [108] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [109] W.S. Martins, J.B. Cuvillo, W. Cui, and G.R. Gao. Whole Genome Alignment using a Multithreaded Parallel Implementation. In *Proc. of the 13th Symposium on Computer Architecture and High Performance Computing*, pages 10–12, 2001.
- [110] K. Marzullo, M. Ogg, A. Ricciardi, A. Amoroso, F.A. Calkins, and E. Rothfus. NILE: Widearea Computing for High Energy Physics. In *Proc. of the 7th ACM SIGOPS European Workshop*, pages 49–54, 1996.
- [111] S. Matsuoka and H. Casanova. Network-Enabled Server Systems and the Computational Grid. In *Global Grid Forum White Paper*, 2000.
- [112] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Jrnl. of Algorithms*, 23(2):262–272, 1976.
- [113] C. Meek, J.M. Patel, and S. Kasetty. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *Proc. of the 29th VLDB Conference*, 2003.
- [114] P. Messin, S. Brunett, D. Davis, T. Gottschalk, D. Curkendall, L. Ekroot, and H. Siegel. Distributed Interactive Simulation for Synthetic Forces. In *Proc. of the 11th International Parallel Processing Symposium*, 1997.

- [115] B. Morgenstern. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15:211–218, 1999.
- [116] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.
- [117] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol.*, 48(3):443–53, 1970.
- [118] J. Nieplocha and R. Harrison. Shared Memory NUMA Programming on the I-WAY. In *Proc. of the 5th IEEE Symposium on High Performance Distributed Computing*, pages 340–349, 1996.
- [119] M.R. Oggioni and G. Pozzi. Comparative genomics for identification of clone-specific sequence blocks in *Streptococcus pneumoniae*. *FEMS Microbiol Lett.*, 200(2):137–143, 2001.
- [120] G.J. Olsen, R. Hagstrom, and R. Overbeek. FastDNAm1: A Tool for Construction of Phylogenetic Trees of DNA Sequences Using Maximum Likelihood. *Comput. Appl. Biosci.*, 10:41–48, 1994.
- [121] G. Pavesi, P. Mereghetti, G. Mauri, and G. Pesole. WeederWeb: Discovery of Transcription Factor Binding Sites in a Set of Sequences from Co-Regulated Genes. *Nucleic Acids Research Web Server Issue 2004*, 32:W199–W203, 2004.
- [122] W.R. Pearson. Comparison of methods for searching protein sequence databases. *Protein Science*, 4(6):1145–1160, 1995.
- [123] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85:2444–2448, 1988.

- [124] C. Potter, R. Brady, P. Moran, C. Gregory, B. Carragher, N. Kisseberth, J. Lyding, and J. Lindquist. EVAC: A Virtual Environment for Control of Remote Imaging. *IEEE Computer Graphics and Applications*, 16(4):62–66, 1996.
- [125] A. Rodriguez, L.G. Fraga, E.L. Zapata, J.M. Carazo, and O. Trelles. Biological Sequence Analysis on Distributed-shared Memory Multiprocessors. In *Proc. of the 6th Euromicro Workshop in Parallel and Distributed Processing Conference*, 1998.
- [126] F.D. Russell. Searching through sequence databases. *Methods in Enzymology*, 183:99–110, 1990.
- [127] N. Saitoy and M. Nei. The neighbour-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4:406–425, 1987.
- [128] B. Schmidt, H. Schröder, and M. Schimmler. Protein Sequence Comparison on the Instruction Systolic Array. In *PaCT*, pages 498–509, 2001.
- [129] B. Schmidt, H. Schröder, and M. Schimmler. Hybrid architecture for bioinformatics. *Future Generations Computer Systems*, 18:855–862, 2002.
- [130] S. Schwartz, W.J. Kent, A. Smit, Z. Zhang, R. Baertsch, R.C. Hardison, D. Hausler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Res.*, 13:103–107, 2003.
- [131] S. Schwartz, Zheng Zhang, Kelly A. Frazer, Arian Smit, Cathy Riemer, John Bouck, Richard Gibbs, Ross Hardison, and Webb Miller. PipMaker—A Web Server for Aligning Two Genomic DNA Sequences. *Genome Res.*, 10(4):577–586, 2000.
- [132] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

-
- [133] B. Spencer, T.A. Finholt, I. Foster, C. Kesselman, C. Beldica, J. Futrelle, S. Gullapalli, P. Hubbard, L. Liming, D. Marcusiu, L. Pearlman, C. Severance, and G. Yang. NEESgrid: A Distributed Collaboratory for Advanced Earthquake Engineering Experiment and Simulation. In *Proc. of the 13th World Conference on Earthquake Engineering*, 2004.
 - [134] Achim Streit. A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. In *Proc. of the 8nd workshop on job scheduling strategies for parallel*, 2002.
 - [135] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests. In *proc. of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)*, 2002.
 - [136] A. Takefusa, S. Matsuoka, H. Casanova, and F. Berman. A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid. In *proc. of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01) table of contents*, 2001.
 - [137] S. Tata, R.A. Hankins, and J.M. Patel. Practical Suffix Tree Construction. In *proc. of the 30th VLDB Conference*, 2001.
 - [138] O. Trelles. On the Parallelisation of Bioinformatic Applications. *Briefings in Bioinformatics*, 2(2):181–194, 2001.
 - [139] M.A. Trick. A Tutorial on Dynamic Programming. In *<http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html>*, 1997.
 - [140] Esko Ukkonen. Constructing suffix trees on-line in linear time. In *Proc. of the 12th IFIP World Computer Congress on Algorithms, Software, Architecture*, 1992.
-

- [141] Esko Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.
- [142] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 22(8):103–111, 1990.
- [143] P. Vincens, L. Buffat, C. Andre, J.P. Chevrolat, J.F. Boisvieux, and S. Hazout. A strategy for finding regions of similarity in complete genome sequences. *Bioinformatics*, 14(8):715–725, 1998.
- [144] G. von Laszewski, M. Westbrook, I. Foster, E. Westbrook, and C. Barnes. Using Computational Grid Capabilities to Enhance the Ability of an X-ray Source for Structural Biology. *Cluster Computing*, 3(3):187–199, 2000.
- [145] Lizhe Wang, Wentong Cai, Lee Bu-Sung, Wei Jie, and Bertil Schmidt. Biogrid Computing – Parallel Computing for Protein Alignment Analysis. In *Proc. of the 6th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia 2002)*, 2002.
- [146] M.S. Waterman. General methods of sequence comparison. *Bull Math Biol.*, 46:473–500, 1984.
- [147] M.S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *Journal of Molecular Biology*, 197:723–728, 1987.
- [148] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. of the 14th IEEE Annual Symp. on Switching and Automata Theory*, pages 1–11, 1973.
- [149] W.J. Wilbur and D.J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Natl. Acad. Sci.*, 80:726–730, 1983.

- [150] Ming Wu and Xianhe Sun. A General Self-adaptive Task Scheduling System for Non-dedicated Heterogeneous Computing. In *Proc. of the IEEE International Conference on Cluster Computing*, 2003.
- [151] Ming Wu and Xianhe Sun. Memory Conscious Task Partition and Scheduling in Grid Environments. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing (in conjunction with SC 2004)*, pages 138–145, 2004.
- [152] Tieng K. Yap, Ophir Frieder, and Robert L. Martin. Parallel Computation in Biological Sequence Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–294, 1998.
- [153] W.R. Zhu, Y.W. Niu, J.Z. Lu, C. Shen, and G.R. Gao. A Cluster-Based Solution for High Performance Hmmpfam Using EARTH Execution Model. In *Proc. of IEEE International Conference on Cluster Computing (Cluster 2003)*, 2003.