Data Distribution and Task Scheduling for Distributed Computing of All-to-All Comparison Problems

A THESIS SUBMITTED TO
THE SCIENCE AND ENGINEERING FACULTY
OF QUEENSLAND UNIVERSITY OF TECHNOLOGY
IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yi-Fan Zhang

Principal Supervisor: Professor Yu-Chu Tian

Associate Supervisor: Professor Colin Fidge

Associate Supervisor: Doctor Wayne Kelly

Science and Engineering Faculty

Queensland University of Technology

2016

Data Distribution and Task Scheduling for Distributed Computing of All-to-All Comparison Problems

A THESIS SUBMITTED TO

THE SCIENCE AND ENGINEERING FACULTY

OF QUEENSLAND UNIVERSITY OF TECHNOLOGY

IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Yi-Fan Zhang

Principal Supervisor: Professor Yu-Chu Tian

Associate Supervisor: Professor Colin Fidge

Associate Supervisor: Doctor Wayne Kelly

Science and Engineering Faculty

Queensland University of Technology

2016

Copyright in Relation to This Thesis

© Copyright 2016 by Yi-Fan Zhang. All rights reserved.

Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

QUT Verified Signature

Signature:

Date: 13/1/2016

To my family

This is a three year journey of life. I have gained much and also lost some. I believe everything starts from some points but when I finally get my PhD degree, I just could not remember my original willings and can only look forward.

We are led into a wonderful world, we meet one another here, greet each other and wander together for a brief moment. Then we lose each other and disappear as suddenly and unreasonably as we arrived. Life is not always what we expect, while it is still worth fighting for.

For my family who give me understanding and support; for my supervisors who give me patient and guide; for my friends who give me comfort and encourage, thank you all.

Abstract

Distributed computing using a distributed data storage architecture has been widely applied in solving large-scale computing problems due to its cost effectiveness, high reliability and high scalability. It decomposes a single large computing problem into multiple smaller ones and then schedules each of these to distributed worker nodes. Its performance largely depends on the data distribution, task decomposition, and task scheduling strategies. Significantly degraded performance may result from inappropriate data distribution, poor data locality for the computing tasks, and unbalanced computational loads among the distributed system. An inappropriate data distribution consumes excessive storage space. Poor data locality means that the data required by a particular worker is not available locally and creates overheads associated with rearranging data between the nodes at run time. Load imbalances lengthen the overall computation time. New approaches are needed to deal with all of these issues for distributed computing of large-scale problems with distributed data.

All-to-all comparison is a type of computing problems with a unique pairwise computing pattern. It involves comparing two different data items from a data set for all possible pairs of data items. All-to-all comparison problems are widely found in various application domains such as bioinformatics, biometrics and data mining. For example, in data mining, clustering algorithms use all-to-all comparisons to derive a similarity matrix to characterize the similarities between objects.

It is hard to develop all-to-all comparison applications running in the distributed environment. Some researches use Message Passing Interface (MPI), Open Multi-Processing (OpenMP) or other techniques to implement parallel all-to-all comparison programs. For all these solutions, users have to consider the detail mechanisms of the network communication, data distribution and task scheduling, which bring heavy burdens on application development.

Another approach is to use computing frameworks designed for distributed computing. Providing simple programming interfaces for users to develop their applications, these frameworks hide implementation issues such as data distribution, load balancing, data locality and fault tolerance. They enable programmers to focus on the application domain without the need of considering complicated parallel computing details. Though many solutions have been provided based on the existing computing framework like Hadoop, it is inefficient because of the unmatched computing pattern, data distribution strategy and task scheduling strategy.

To tackle these challenges, this research develops a distributed computing framework for solving general all-to-all comparison problems with big data sets. A programming model for the all-to-all comparison problem is developed. By providing the powerful application programming interfaces (APIs), developers can implement different all-to-all comparison applications without the consideration of distributed system issues, which makes the application development efficiency. Beside this, user interfaces are developed to make the back-end distributed computing system transparent to users and simplify the computing framework operation.

Moreover, specific task-oriented data distribution strategies and locality-aware task scheduling strategies are designed for both homogeneous and heterogeneous distributed systems. Good data locality for all the comparison tasks, static load balancing for the system and reduced storage usage for each worker node are all considered during the design of data distribution strategies to enhance the overall computation performance and reduce data storage requirements across the network. Following data distribution strategies, both the static and dynamic task scheduling strategies are developed in a way to keep system load balancing and good data locality for all the comparison tasks without any data movement at runtime.

Table of Contents

Abstract **vii**

Nomenclature **xv**

List of Figures **xxii**

List of Tables **xxiv**

1 Introduction **1**

1.1 Research Background 1

1.2 Research Significance 3

1.3 Definition of All-to-all Comparison Problem 4

1.4 Problem Challenges 6

1.5 Problem Statement 7

1.6 Thesis Structure and Contributions 9

1.7 List of Publications 10

2 Literature Review **13**

2.1 Introduction 13

2.2 Existing Solutions for Specific All-to-all Comparison Problems 14

2.3 Existing Solutions Based on Hadoop 19

2.3.1 Hadoop Distributed Computing Framework 20

2.3.2 MapReduce Programming Model 21

2.3.3 Hadoop Distributed File System 23
2.3.4 ATAC Solutions Based on Hadoop 25
2.4 Existing Solutions Based on Other Computing Frameworks 28
2.4.1 All-Pairs 28
2.4.2 Bi-Hadoop 31
2.5 Classification of Distributed Computing Systems 33
2.6 Different Cluster Management Platforms 35
2.6.1 Mesos 35
2.6.2 Yarn 37
2.7 Summary of Literature Review 39

3 The Front-end Interfaces for Distributed Computing Framework of ATAC Problems 41

3.1 All-to-all Comparison Problems 41
3.1.1 All-to-all Comparison Problems in Bioinformatics 42
3.1.2 All-to-all Comparison Problem in Other Domains 43
3.2 Overview of the Distributed Computing Framework for ATAC Problems 44
3.3 User Interfaces Design 47
3.3.1 Data Interface 47
3.3.2 Application Interface 48
3.3.3 Result Interface 49
3.4 ATAC Programming Model 51
3.4.1 ATAC Workflow and Challenges 51
3.4.2 ATAC Programming Model Design Targets 54
3.4.3 ATAC Programming Model Design 55
3.4.4 Comparison with MapReduce Programming Model 57
3.5 ATAC Computing Framework Implementation 58
3.5.1 ATAC Computing Framework Architecture 58

3.5.2 ATAC Computing Framework Execution 60
3.6 Conclusion 63

4 Heuristic Data Distribution Strategy for ATAC Problems in Homogeneous Distributed Systems 65

4.1 Principles for Data Distribution 65
4.2 Challenges of the Data Distribution Problem 67
4.2.1 Issues of Storing All the Data to Everywhere 67
4.2.2 Issues of the Hadoop Data Strategy 69
4.3 Formulation for Data Distribution 73
4.3.1 Overall Considerations and Assumptions 73
4.3.2 Reducing the Storage Usage 74
4.3.3 Improving the Computing Performance 75
4.3.4 Optimization for Data Distribution 77
4.3.5 Theoretic Results 77
4.3.6 Special Case Analyse 79
4.4 Heuristic Data Distribution Strategy with Greedy Idea 80
4.4.1 Heuristic Rules for Data Distribution 80
4.4.2 Data Distribution Algorithm 82
4.4.3 Analysis of the Data Distribution Strategy 83
4.5 Experiments 84
4.5.1 Evaluation Criteria and Experimental Design 84
4.5.2 Storage Saving 85
4.5.3 Execution Performance 87
4.5.4 Scalability 90
4.6 Conclusion 91

5 MetaHeuristic Data Distribution Strategy for ATAC Problems in Homogeneous

Distributed Systems **93**

5.1 Problem Statement and Challenges 93

5.1.1 Task Balancing Causes Data Storage Issues 94

5.1.2 Storage Saving Causes Task Issues 94

5.2 Data Distribution Strategy with Simulated Annealing 95

5.2.1 Comparison Between Greedy and SA Idea 95

5.2.2 Annealing Module and Acceptance Probability Module 97

5.3 Static Comparison Task Scheduling Strategy 99

5.3.1 Static Scheduling for ATAC Computing Framework 99

5.3.2 Task Scheduling Strategy Design 100

5.4 Experiments 102

5.4.1 Storage Saving, Task Allocation and Data scalability 102

5.4.2 Computing Performance 103

5.4.3 Computing Scalability 104

5.5 Conclusion 106

6 Heuristic Data Distribution Strategy for Heterogeneous Distributed Systems **107**

6.1 Development of Heterogeneous systems 107

6.2 Existing solutions for Heterogeneous systems 108

6.3 Problem Statement 109

6.3.1 Feature of Heterogeneous Systems 109

6.3.2 Similar Research Problems 110

6.4 Challenge of the Data Distribution Problem 111

6.4.1 Poor Data Locality of Comparison Tasks 111

6.4.2 Invalid locality-aware Scheduling of Comparison Tasks 111

6.4.3 Inefficiency in Heterogeneous Systems 113

6.5 Data and Task Distribution Strategy 113

6.5.1 Strategy Targets 114

6.5.2 Strategy Design . . . 114
6.5.3 Strategy for Data Distribution and Task Scheduling . . . 116
6.5.4 Analysis of the Presented Strategy . . . 117
6.6 Experiments . . . 118
6.6.1 Storage Saving and Task Allocation . . . 118
6.6.2 Computing Performance . . . 119
6.6.3 Scalability . . . 120
6.7 Conclusion . . . 122

7 MetaHeuristic Data Distribution Strategy for ATAC Problems in Heterogeneous Distributed Systems 123
7.1 Problem Statement and Challenges . . . 124
7.1.1 Principles of the All-to-All Scheduling Problem . . . 124
7.1.2 Challenges of the All-to-All Scheduling Problem . . . 125
7.2 Formulation for Pre-Scheduling . . . 130
7.2.1 Overall Considerations and Assumptions . . . 130
7.2.2 Reducing Time and Storage Consumption . . . 131
7.2.3 Improving Performance for Individual Tasks . . . 132
7.2.4 Improving Overall Computational Performance . . . 133
7.2.5 Optimization for Data Pre-Scheduling . . . 133
7.3 MetaHeuristic Data Pre-Scheduling . . . 134
7.3.1 Features of the Optimization Problem . . . 134
7.3.2 Simulated Annealing . . . 135
7.3.3 Annealing Design for All-to-All Pre-Scheduling . . . 136
7.3.4 Analysis of the Data Pre-Scheduling Strategy . . . 139
7.4 Runtime Task Scheduling Design . . . 141
7.4.1 Initial Task Schedulability . . . 141
7.4.2 Runtime Dynamic Scheduling . . . 142

7.5 Experiments 145
7.5.1 Storage Saving and Data Locality 147
7.5.2 Execution Time Performance 149
7.5.3 Scalability 153
7.6 Conclusion 154

8 Conclusion **155**

8.1 Summary 155
8.2 Limitations 157
8.3 Future Work 158

A Codes for ATAC applications **159**

A.1 CVTree Application Codes 159
A.2 NMF Application Codes 165

Literature Cited **181**

Nomenclature

Abbreviations

ATAC	All-to-all Comparison
HDFS	Hadoop Distributed File System
SA	Simulated Annealing
API	Application Programming Interface
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
MSA	Multiple Sequence Alignment
BIBD	Balanced Incomplete Block Design
TODD	Task Oriented Data Distribution
MaxFF	Maximising Flexibility First
MinFF	Minimising Flexibility First
NCBI	The National Center for Biotechnology Information
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instructions Multiple Data

Symbols

ΔE	Energy difference between neighbour and current states
ΔF	Cost difference, $\Delta F = F(S) - F(S')$
Δf	The value of cost difference
$C(x, y)$	Comparison task between data items x and y

$M[i, j]$	An output similarity matrix element for all-to-all comparison problem
D_i	Data set allocated to node i
$\|D_i\|$	The number of data files in the data set D_i
$F(S)$	Fitness of solution S, $F(S) = \{\|D_1\|, \cdots, \|D_N\|\}$
F_ℓ	The flexibility for the system
f	Objective function
G_i	The set of tasks that can be executed by node i
i, j	Worker node IDs, $i, j \in \{1, 2, \cdots, N\}$
k	Iteration step in simulated annealing
L_c	The set of all worker nodes available to execute task c
$\|L_c\|$	The number of worker nodes in set L_c
M	The number of data files to be processed
N	The number of worker nodes in the distributed system
P	The total number of feasible solutions
P_i	The processing power of node i
P_r	Acceptance probability function
R_i	The completeness of node i
Q	The number of distinguishable tasks
r	The number of data replications in Hadoop (variable r)
S, S'	A feasible data allocation solution and its alternative
S_t	The Stirling number
K	The number of comparison tasks allocated to the last finished worker node
U	The set of unscheduled comparison tasks
$\|U\|$	The number of comparison tasks in set U
G_i	The set of comparison tasks that can be executed by node i
T	The set of all comparison tasks
T_i	The set of comparison tasks assigned to node i

$\lvert T_i \rvert$	The number of tasks in the task set T_i
T_{data}	Time consumption for data distribution
T_{task}	Time consumption for comparison task execution
T_{total}	Total time spent on data distribution and task execution
$T_{comparion(i)}$	The time for comparison operations for task i
$T_{accessdata(i)}$	The time for accessing the required data for task i
t	Annealing temperature

List of Figures

1.1 Details of all-to-all comparison problems 5

1.2 Organization of this thesis 8

2.1 The architecture of Hadoop cluster 20

2.2 The MapReduce programming model 22

2.3 HDFS architecture 24

2.4 Example for HDFS data placement strategy 24

2.5 Comparison between two different architectures 28

2.6 The architecture of All-pairs computing framework 29

2.7 Bi-Hadoop extension system overview 31

2.8 Bi-Hadoop user interface 32

2.9 Distributed computing system architecture taxonomy 34

2.10 The architecture of Mesos 35

2.11 An example of using Mesos 36

2.12 Different layers of using Yarn 38

2.13 The architecture of Yarn 38

3.1 Similarity matrix of all-to-all comparison problems 42

3.2 Architecture of the All-to-all computing framework 45

3.3 User interfaces for the All-to-all computing framework 47

3.4 Processing steps for solving ATAC problems 51

3.5 Modified Map tasks to support multiple inputs 53

3.6 The programming interface of our programming model 55

3.7 Comparison between two programming models 57

3.8 Overview of the ATAC programming model implementation. 59

3.9 ATAC programming model implementation 59

3.10 Overview of the execution of our computing framework 61

4.1 General work flow for solving all-to-all comparison problems in distributed computing environments 66

4.2 A possible situation for processing 9 data files in a 3 worker nodes distributed system 68

4.3 A possible situation for Hadoop's data strategy for a scenario with 6 data items and 4 worker nodes 70

4.4 Data distribution and comparison task allocation from Hadoop's data strategy for a scenario with 4 data items and 3 worker nodes 70

4.5 Data distribution and comparison task allocation from Hadoop's data strategy for a scenario with 6 data items and 8 worker nodes (The comparison tasks marked with '?' require remote access of data at runtime). 71

4.6 The growth trend of the solution space for $S_t(Q, 3)$ 73

4.7 Additional comparison tasks introduced by adding new data d to node k 81

4.8 Comparisons of the T_{total} performance between our strategy and Hadoop's strategies (lower solid-filled part of the bars: T_{data}; upper dot-filled part of the bars: T_{task}). 89

4.9 T_{task} performance from our data distribution strategy for each of the worker nodes under different M values. 90

4.10 Speed-up achieved by the data distribution strategy in this chapter. 91

5.1 A data imbalance 94

5.2 Poor data locality for comparison tasks 95

5.3 Static scheduling of the framework for All-to-All comparison problems 100

5.4 Data scalability. 103

5.5 Computation time performance. 104

5.6 Scalability of our programming model. 105

6.1 Poor data locality for comparison tasks. 112

6.2 Task scheduling with globe consideration requires $C(2, 4)$ and $C(2, 6)$ are allocated to worker nodes 3 and 4, respectively, for data locality when these two nodes are idle for more tasks. 112

6.3 The best use of computing resources requires tasks 13 to 18 are evenly distributed to worker nodes 3 and 4, which have twice as much computing power as that in the other two nodes. 113

6.4 New available comparison tasks for distributing new data d to node i. They include those that have never been allocated (dotted arrows) and those that have been allocated previously (solid arrows). 116

6.5 Comparisons of the computation time performance. 120

6.6 Demonstration of load balancing from our strategy. 121

6.7 Speed-up achieved by the data and task distribution strategy. 121

7.1 Graph representation of an all-to-all comparison problem. The vertices and edges of the graph represent data files and pairwise comparison tasks, respectively. 124

7.2 A general design for data distribution and task scheduling. 125

7.3 A possible data distribution of 6 files on 4 nodes. Solid lines, dotted lines, solid points and hollow points represent scheduled tasks, unscheduled tasks, scheduled data items and unscheduled data items, respectively. 126

7.4 A possible data distribution of 9 files on 6 nodes. 127

7.5 Load balancing in a homogeneous system (upper part), and a potential load imbalance in a heterogeneous system for the same data distribution solution (lower part), assuming Node 1 has triple the computing power of Nodes 2 and 3. (In the heterogeneous case not all tasks have been allocated yet.) 128

7.6 The growth trend of the solution space for $P(Q, 4)$, for scenarios with 4 worker nodes and Q tasks to be distributed. 130

7.7 All possible comparison tasks for each of 4 worker nodes. The row and column numbers identify the data files stored on each node. The numbers in the cells represent how many worker nodes can perform the specific comparison between two such files. 143

7.8 MaxFF (upper part) and MinFF (lower part) dynamic scheduling strategies. . . 144

7.9 Dynamic scheduling process in our distributed computing framework. 146

7.10 Total execution time T_{total} performance for all four solutions. 151

7.11 The number of comparison tasks completed on each node. Nodes 1 to 4 have twice the computing power of Nodes 5 to 8. 152

7.12 Task performance T_{task} of our static and dynamic solutions. 152

7.13 Speed-up achieved by our dynamic scheduling algorithm for the CVTree problem compared to the theoretical ideal. 154

List of Tables

3.1 Parameters of the data interface 48

3.2 Parameters of the application interface 49

3.3 Configurations of the ATAC application 49

3.4 Comparison between our programming model and MapReduce 58

4.1 Distribution of 6 data files to 4 worker nodes. 83

4.2 Storage and data locality of our approach and Hadoop with three data replications for $M = 256$ files under different numbers of nodes (N). 85

4.3 Storage and data locality of our approach and Hadoop(variable x) for $M = 256$ files under different numbers of nodes (N), where the number (x) of data replications for Hadoop has to be tuned manually for each case to achieve a similar maximum number of files on a node. 87

4.4 The CVTree problem for $N = 8$ nodes and different numbers of input data files (M). 88

5.1 SA module parameter settings (k represents the iteration step). 97

5.2 Distributing of 8 data files to 4 worker nodes. 102

5.3 Storage usage and storage savings of our work versus Hadoop for 256 files. . . 103

5.4 Experimental cases. 104

6.1 Distribution of seven data files to five worker nodes. 117

6.2 Storage usage, storage saving and task allocation. 118

7.1 Cooling parameter settings (k represents the iteration step). 136

7.2 Pre-scheduling of 7 data files to 5 worker nodes. 140

7.3 Storage and data locality of the heterogeneous approach in this chapter, our previous work on heterogeneous systems, and Hadoop(3) for $M = 256$ files and N varying from 8 to 64. Half of the nodes had twice the computing power of the other half. The number of data replications in Hadoop was set to be the default value 3. 147

7.4 Storage and data locality of our heterogeneous approach and Hadoop(variable r) for $M = 256$ files and N varying from 8 to 64. Half of the nodes had twice computing power than the other half. The number of data replications r for Hadoop was tuned manually for each case to achieve a similar maximum number of files on each node. 147

7.5 T-test at a 5% significance level for the results of our approach for the experiments shown in Table 7.3 (mean values from 10 runs for all cases). 148

7.6 Experimental scenarios for the CVTree problem with $N = 8$ and different M values. 150

Chapter 1

Introduction

This thesis addresses distributed computing of all-to-all comparison (ATAC) problems with big data sets. ATAC problems are computing problems in which each data item of a data set is pair-wise compared with all other data items in the same set. They are suitable for parallel and distributed implementation because of the independent pair-wise comparison operations. However, one major difficulty of using distributed system is the poor performance caused by massive data movement and network communications. Another major difficulty comes from the heavy burdens between the complex distributed system issues and users without rich experiences. This thesis develops a large-scale distributed computing framework for all-to-all comparison problems to overcome mentioned difficulties.

1.1 Research Background

All-to-all comparison problems in this thesis represent a type of computing problems with a unique computing pattern. In this ATAC pattern, all two different data items within the same data set need to be pair-wise compared.

Addressing problems with ATAC pattern is straightforward when the size of data is relatively small. For instance, one of the simple solutions to process ATAC problems is to load all the input data into the memory and execute pair-wise comparison sequentially [Yu et al., 2010].

However, challenges come when we try to deal with ATAC problems with big data sets. When the size of input data set and the numbers of input data files are becoming much larger, the factors such as storage space, data transmission and network bandwidth can affect the

computing performance greatly. Consider the problems with ATAC pattern in bioinformatics as example. A typical bioinformatics computing would take around 80 CPU years to BLAST [Altschul et al., 1990] the 268 Gb cow rumen metagenome data [Hess et al., 2011]. De novo assembly of the full data set by a short read assembler, such as Velvet [Zerbino and Birney, 2008], would require computers with over 1 TB RAM and take several weeks to complete.

Currently, the following three system architectures could be used to solve all-to-all comparison problems: 1) Centralized computing. 2) Distributed computing with centralized data storage and 3) Distributed computing with distributed storage.

For centralized solution, some researches use super computers to process ATAC problems. For example in the experiment developed by Catalyurek et al. [2002], a 24-processor Sun Fire 6800 system at the Ohio Supercomputer Center was chosen. However, many researcher cannot get super computing resources and the cost of using it is also very high.

Distributed computing with centralized data storage is also chosen by many researchers such as Tang and Yew [1986] and Hummel [1996] . For these solutions, though the total computing power can be scaled up by adding worker nodes in the distributed system, the data server can become a big bottleneck because of the limited storage capability and the task latency caused by waiting data transmission.

In the last decades, the distributed systems with distributed data storage architectures have been widely used due to the cost effectiveness, high availability and high scalability. Some researches such as Gunturu et al. [2009], Schatz [2009] and Chen et al. [2008] all use this architecture to solve ATAC problems. The computing framework developed in this thesis is also based on this architecture.

For the current solutions based on the distributed computing with distributed data storage, mainly two different strategies are used to distribute all the data files: 1) Distributing every input file to every worker node; and 2) Distributing each input file randomly to the system with a fixed number of duplications (The Hadoop data strategy). Storing all the data everywhere is a natural idea inherited by the centralized solutions. Thus it also has the similar problems as it is in centralized computing. For the Hadoop data strategy, due to Hadoop is not designed to support ATAC computing pattern, considerable data movement between different worker nodes during the comparison phase is required, which could degrade the performance greatly.

Hence, new data distribution strategies and related task scheduling strategies for ATAC problems are developed to overcome the drawbacks of the above two different data distribution solutions.

1.2 Research Significance

All-to-all comparison problems are found in many application domains such as bioinformatics, biometrics and data mining. Though these ATAC problems are with different backgrounds, the principle of solving ATAC problems is the same.

For biological analysis, ATAC is a key calculation stage in Multiple Sequence Alignment (MSA) and studying of phylogenetic diversity in protein families [Trelles et al., 1998]. In general, the computing of these bioinformatics problems includes the calculation of a cross-similarity matrix between each pair of sequences [Krishnajith et al., 2013, 2014]. The comparison computations become more complex when processing larger number of bioinformatic data and advanced computation techniques are required for large-scale all-to-all comparison problems [NM et al., 2001].

In biometrics, a typical problem is to identify people's physical characteristics via pairwise comparisons of a large amount of data stored in biometrics databases [Phillips et al., 2005], such as face recognition, finger geometry and palm scanning. In the face recognition area, different technologies analyse the unique shape, pattern and positioning of facial features. In the RFGC experiment [Phillips et al., 2005], the inputs to an analysis algorithm are two sets of images, which include 24,042 images totally. The output from the algorithm is a similarity matrix, which includes the similarity scores between any target-query image pairs. For these kinds of biometric problems, it is easy to turn them into problems with ATAC pattern.

ATAC pattern is also found in data mining researches. For example, in cluster analysis many clustering algorithms use a similarity matrix, which is a high dimensional square matrix containing all the pairwise dissimilarities or similarities between the objects being considered. In the music information retrieval evaluation presented by Arora et al. [2013], the samples are 3090 pieces of music and the similarity is the average of three human's fine-grained judgement of the audio similarity of a pair of samples.

Another area that uses ATAC pattern is video or audio analysis. The video comparison is a

basic analysis operation in complement of classification, extraction and structuring of videos. In the video retrieval system designed by Sav et al. [2006], 8718 shots were detected form the 50 hours of BBC rushes video footage, and two 8717 by 8717 matrices of keyframe similarities were computed by using colour, texture and shape.

It is seen from the above discussion that though ATAC problems in different areas have different purposes, use different algorithms or combine with different processing steps, in the ATAC phase they all share the same computing pattern and generate the same type of results (similarity matrix). Beside this, the size of data processed in these areas is growing dramatically, which requires efficient solutions support large-scale ATAC problems.

Therefore, it is significant to provide a general and systematic solution for ATAC problems. The solution is expected to have high performance of processing all-to-all comparison problems and also have good scalability to support big data sets. This is the focus of this thesis.

1.3 Definition of All-to-all Comparison Problem

Definition 1 (All-to-all comparison problem) *Let A, C and M denote the data set to be pair-wise compared, the comparison function on pairs in A and the output similarity matrix of A, respectively. Characterized by the Cartesian product or cross join of the data set A, the all-to-all comparison problem discussed in this thesis is mathematically stated as follows:*

$$M_{ij} = C(A_i, A_j),\ i, j = 1, 2, \cdots, |A| \tag{1.1}$$

where A_i represents the i*th item of A, M_{ij} is the element of output matrix M resulting from the comparison between A_i and A_j, and $|A|$ means the numbers of times in set A.*

The typical all-to-all comparison problems can also be expressed graphically in Figure 1.1, where each data item in a data set need to be compared with all the others. In the matrix shown in Figure 1.1, considering there is no comparison order of the data files ($C(A_i, A_j) = C(A_j, A_i)$), for this symmetric matrix, only the triangle shape of this matrix should be calculated.

In this part, our planning approach is defined by the following considerations:

1) A solution with high flexibility to extend processing power. Considering the scale of

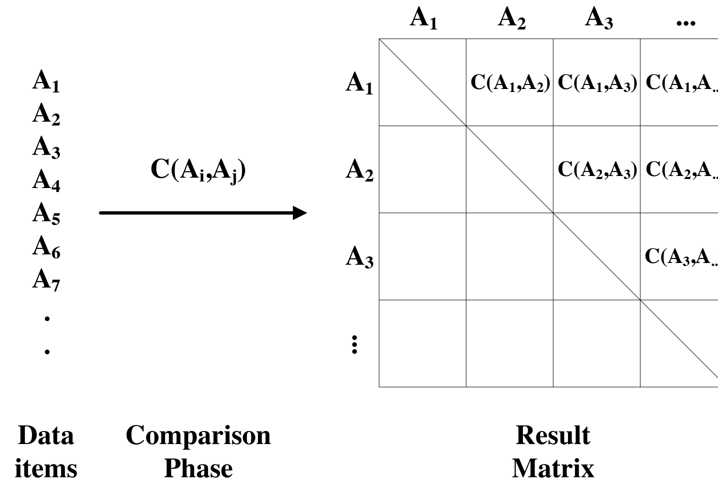


Figure 1.1: Details of all-to-all comparison problems.

ATAC problems will keep growing and the requirements for data storage and computing power would also be increasing significantly, we choose to use distributed computing systems due to the system processing power can be easily extended by adding extra worker nodes.

2) A solution can support large-scale data sets. ATAC problems are typical data-intensive computing problems and the computation scale will grow exponentially with the number of input data items growing (n by n matrix in Figure 1.1). To deal with large-scale ATAC problems, we prefer to store the data sets among the worker nodes in a distributed way, which can avoid the poor performance caused by the single centralised data server.

3) A solution can avoid batching latency. Hence, in our solution, we prefer to pre-distribute data files to different worker nodes and schedule comparison tasks based on the location of related data files. In this case, the task scheduling depends on the way we distribute and store data files. Because the data files have already been pre-distributed, the latency for scheduling comparison tasks can be greatly reduced.

Based on the consideration above, the solution we develop for solving ATAC problems is expected to have the following features: 1) All the data files are pre-distributed to the distributed computing system; and 2) The task scheduling is based on the situation of data distribution.

To develop a general solution, the key challenge is how to distribute all the data files into the distributed computing system. This will be discussed with more details in the next section.

1.4 Problem Challenges

To solve all-to-all comparison problems efficiently, the following aspects need to be improved: the performance of each comparison task, the performance of the distributed system and the cost spent on distributing data sets.

1) The performance of each comparison task. For each comparison task, if all the required data for this task are stored locally in the node that performs the task, the task will not need to access data remotely through network communications. In this case, no extra data movement is generated at runtime and the comparison task can be executed immediately.

   Let $C(x, y)$, T, T_i and D_i represent the comparison task for data x and data y, the set of all comparison tasks, the set of tasks performed by worker node i, and the data set stored in worker node i, respectively. Good data locality for all comparison tasks can be expressed as follows:

$$\forall C(x,y) \in T, \exists j \in \{1, \cdots, N\}, x \in D_j \wedge y \in D_j \wedge C(x,y) \in T_j. \tag{1.2}$$

2) The performance of the distributed system. If all the worker nodes in the distributed system can be allocated numbers of comparison tasks that is proportional to their computing power, the system can achieve load balancing and all the worker node can finish at the same time. Hence, all the computing power can be fully utilized.

   Let T_i denote the number of pairwise comparison tasks performed by worker node i. For a distributed system with N worker nodes and M data files, a total number of $M(M-1)/2$ comparison tasks need to be allocated to the work nodes. Load balancing in a homogeneous distributed system can be expressed as follows:

$$\forall T_i \in \{T_1, T_2, ..., T_N\}, T_i \leq \left\lceil \frac{M(M-1)}{2N} \right\rceil, \tag{1.3}$$

   where $\lceil \cdot \rceil$ is the ceiling function.

3) The cost spent on distributing data sets. Considering Strategy that storing all the data everywhere can also meet the above two constraints (1.2) and (1.3), in our data distribution strategy we want to also reduce the cost spent on distributing data sets. To distribute all

the data in distributed systems, the storage usage fo each worker node must be within its boundary. Beside this, the time spend on distributing data files should also be reduced. Generally, the time for data distribution is proportional to the number of data files to be distributed.

Let $|D_i|$ denote the number of files allocated to worker node i. By considering both the data distribution time and storage limitation, a data distribution strategy is expected to minimize the maximum of $|D_1|, \cdots, |D_N|$, i.e.,

$$\text{Minimize} \max\{|D_1|, |D_2|, ..., |D_N|\}. \tag{1.4}$$

Therefore, based on the above constraints (1.2, 1.3) and target (1.4), the focus of this thesis is on development of data distribution strategies for the following data distribution problem:

Definition 2 (Data distribution problem) *For a given all-to-all comparison problems with M numbers of data items and a distributed computing system with N numbers of worker nodes, distributing data items to each worker node in a way that all the data pairs, which represent all the comparison tasks can be find in at least one worker node (1.2). Moreover, all the worker node can be allocated numbers of comparison tasks based on their computing power (1.3). By reaching the above requirements, we want to minimize the maximum number of data items among all the worker nodes (1.4).*

1.5 Problem Statement

In order to address distributed computing of all-to-all comparison problems with big data sets, the following three problems need to be solved: front-end interfaces, data distribution and task scheduling. They are graphically shown in Figure 1.2.

Problem 1 Front-end Interfaces For big data problems with all-to-all comparison pattern, users need a simple way to operate the distributed computing systems. Also, developers need efficient programming interfaces that can help the development of domain-specific all-to-all comparison applications. Moreover, for the front-end interfaces, all the parallel system issues such as data distribution and job scheduling should be solved automatically by the back-end distributed computing system and hidden for the users.

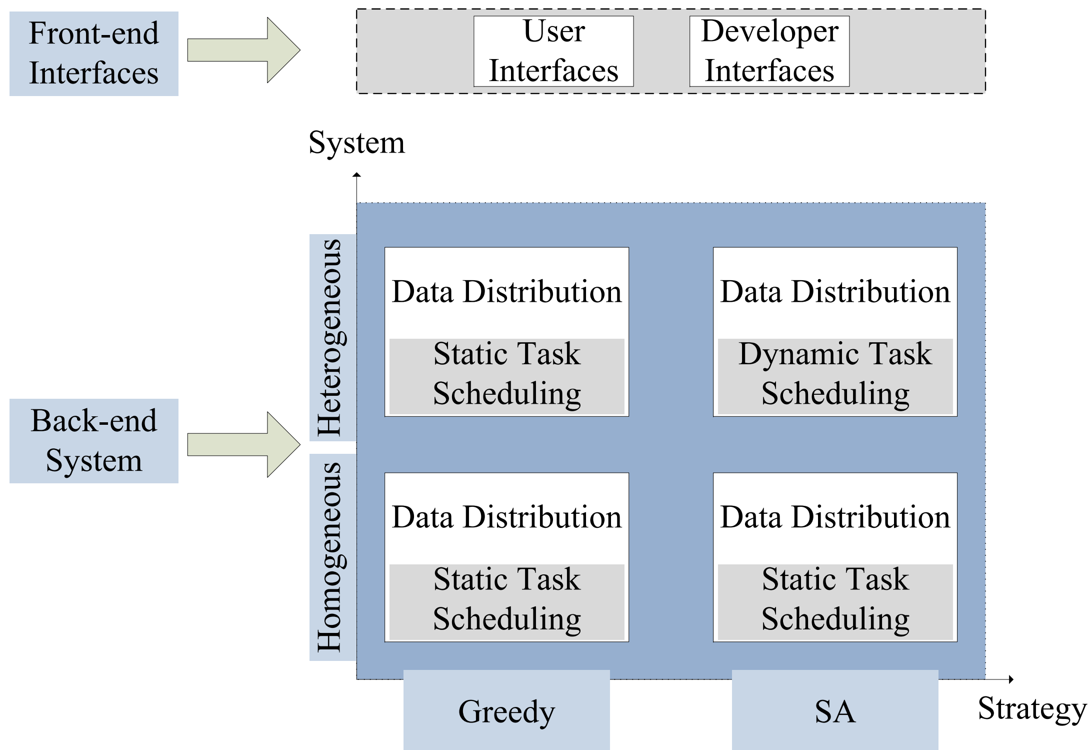


Figure 1.2: Organization of this thesis.

Problem 2 Data Distribution To distribute all the data in the distributed system by considering the constraints we mentioned above, novel data distribution strategies need be developed to support ATAC problems with large data sets. In this thesis, both the greedy and simulated annealing (SA) idea are used to solve the data distribution problem. Furthermore, the data distribution should both consider the scenarios in homogeneous distributed computing systems and heterogeneous distributed computing systems.

Problem 3 Task Scheduling After distributing all the data files by using the data distribution strategies developed in this thesis, the following static and dynamic task scheduling strategies are expected to schedule all the comparison tasks to make full use of the processing power of the distributed system.

By solving all the above problems, the solution in this thesis can achieve a high overall performance for distributed processing all-to-all comparison problems with big data sets.

1.6 Thesis Structure and Contributions

This thesis is organized around the three problems defined in the previous Section 1.5 with the following Chapters:

Chapter 2 summarises a wide range of technological approaches for solving ATAC problems. Different data distribution and task scheduling solutions are also discussed in this Chapter. Besides this, different architectures of distributed systems are discussed. In the end, as part of our implementation system, different cluster management systems are introduced.

In Chapter 3, the front-end interfaces for our distributed computing framework is designed from two different aspects. For users who operate the computing framework, the data deploying and application uploading interfaces are provided. They are defined to give users the easy way to finish all the needed operations. For developers who develop the ATAC applications, a programming model for ATAC problems is developed. It abstracts the work flow of distributed processing ATAC problems and provides simple application programming interfaces (APIs) to help develop ATAC applications.

In Chapter 4, a large-scale data distribution strategy based on greedy idea is developed for ATAC problems with big data sets in homogeneous distributed computing systems. In this strategy, the storage usage on each worker node is reduced to meet the storage limitation. The data distribution strategy can also keep all the comparison tasks have good data locality without any data movement. Moreover, static system load balancing is also considered to support static task scheduling for ATAC problems.

Similarly in Chapter 5, a large-scale data distribution strategy based on SA for homogeneous distributed systems is also developed. Specific methods are developed to make use of SA idea to solve ATAC data distribution problem. Beside this, several special cases are analysed theoretically for the data distribution problem. In the end, the results by using our work and the Hadoop data distribution strategy are compared to show the improvement of simulated annealing based solution.

Data distribution in heterogeneous systems are discussed in both Chapter 6 and Chapter 7. For heterogeneous systems, each of the worker nodes can have different processing power, by adding this consideration into data distribution, scalable data distribution strategies based on greedy and SA ideas are developed separately.

Furthermore, in Chapter 7, a dynamic task scheduling strategy is also provided to make system achieve load balancing without any data movement. System issues such as changeable computing power and different computation time for different comparison tasks are considered in this dynamic task scheduling strategy.

The thesis concludes in Chapter 8 with the summary of the high performance computing framework. The milestones and processing details for the PhD project are also shown in this chapter. In the end, further work of this project is discussed.

Three contributions are claimed as follows:

1) Both the user operation interfaces and ATAC programming model with programming interfaces are developed. ATAC programming model abstracts the workflow of distributed solving ATAC problems and these front-end interfaces are designed to help users solve ATAC problems by using distributed computing systems without considering the complex parallel issues.

2) Data distribution strategies for both the homogeneous and heterogeneous distributed systems are developed. To design the data distribution strategies fro ATAC problems, the storage usage, data locality and static load balancing have all been considered based on both greedy and SA idea.

3) Task scheduling strategies for ATAC problems are developed. Followed by the data distribution strategies, both the static and dynamic scheduling strategies are developed to allocate comparison tasks to the suitable worker nodes for the system load balancing.

1.7 List of Publications

Five papers were produced during my PhD project.

Yi-Fan Zhang, Yu-Chu Tian, Wayne Kelly and Colin Fidge, A Distributed computing Framework for All-to-All Comparison Problems. The 40th Annual Conference of the IEEE Industrial Electronics Society (IECON2014), Dallas, TX, USA, 29 Oct - 1 Nov 2014.

Yi-Fan Zhang, Yu-Chu Tian, Wayne Kelly and Colin Fidge, Distributed Computing of All-to-All Comparison Problems in Heterogeneous Systems. The 41th Annual Conference of the IEEE Industrial Electronics Society (IECON'2015), Yokohama, Japan, Nov. 9-12 2015.

Yi-Fan Zhang, Yu-Chu Tian, Wayne Kelly and Colin Fidge, Application of Simulated Annealing to Data Distribution for All-to-All Comparison Problems in Homogeneous Systems. 22nd International Conference on Neural Information Processing (ICONIP2015), Istanbul, Turkey, Nov. 9-12 2015.

Yi-Fan Zhang, Yu-Chu Tian, Wayne Kelly and Colin Fidge, Scalable and Efficient Data Distribution for Distributed Computing of Large-Scale All-to-All Comparison Problems. Future Generation Computer Systems (FGCS), 2015 (Under Review)

Yi-Fan Zhang, Yu-Chu Tian, Colin Fidge and Wayne Kelly, Data-Aware Task Scheduling for Large-Scale All-to-All Comparison Problems in Heterogeneous Systems, Journal of Parallel and Distributed Computing (JPDC), 2015 (Under Review)

Chapter 2

Literature Review

In this chapter, existing efforts for addressing all-to-all comparison problems in distributed computing systems have been discussed. Data distribution and task scheduling strategies within different approaches were summarized and analysed to show the limitations of current solutions. Beside this, the architectures of different distributed systems are outlined. In the end, different cluster management systems used for big data processing are listed.

2.1 Introduction

This thesis focuses on developing a high performance distributed computing framework for processing large-scale ATAC problems. In the front-end system, operation interfaces are designed for users to use our distributed computing framework simply. A programming model of ATAC pattern is designed and powerful programming interfaces are provided to help developers develop specific-purpose ATAC applications. Beside this, in the back-end computing system, strategies for data distribution and task scheduling are developed to make the computing system achieve high performance. For these goals, the literature review can be categorized as follows,

1. Solutions for special-purpose all-to-all comparison problems.
2. Solutions based on Hadoop computing framework.
3. Solutions based on other computing frameworks.
4. Different distributed computing platforms.

5. Different cluster management platforms.

2.2 Existing Solutions for Specific All-to-all Comparison Problems

Several approaches have been developed to address specific all-to-all comparison problems in bioinformatics. All-to-all comparisons are a key calculation stage in Multiple Sequence Alignment (MSA) [Nguyen and Pan, 2013, Ye et al., 2015, Zhu et al., 2015] and studying of phylogenetic diversity in protein families [Trelles et al., 1998]. In general, the computing of these bioinformatics problems includes the calculation of a cross-similarity matrix between each pair of sequences [Krishnajith et al., 2013, 2014]. This calculation is followed by several data grouping stages. Platforms for such bioinformatics computing include clusters, grids and clouds [Church and Goscinski, 2014].

Data intensiveness describes those applications that are I/O bound or with a need to process large volumes of data [Church and Goscinski, 2014, Gokhale et al., 2008, Ren et al., 2014]. Comparing to compute-intensive, applications of all-to-all comparison problems devote most of the processing time on I/O and movement for massive amount of data files [Howard et al., 1988]. Therefore, to improve the performance of data-intensive problems, the distribution of all the data sets must be seriously considered.

The earliest research for parallel all-to-all comparison has been carried out by Date et al. [1993]. In this attempt they focused on distributed memory systems. They proposed a farm approach to execute the all-to-all comparison in parallel. In this approach one processor (farmer) breaks the large number of work-units into sub-sets of work-units and distributes them to one of the other processor (worker). A work-unit is one pair-wise alignment and in a set of N gene sequences, $N(N-1)/2$ work-units need to be completed. The work is distributed dynamically. A work-unit is sent to a worker processor by the farmer processor, and the worker processor accepts, if it is free. The result of the alignment sent back to the farmer processor upon completion. The worker processors do not communicate with each other and all the communications are routed through the farmer processor. The farm approach has been chosen by Date et al. [1993] for parallel this problem, for two reasons.

1. The time to process a single work unit is significantly larger than the time to generate and distribute work.

2. The order in which the results are collected is not important.

By using this approach, they have archived almost linear speed gain, when the number of computers increased from 4 to 64. After 5 years Trelles et al. [1998] introduce parallelization approach for all-to-all comparison in bioinformatics. Their approach is different from Date et al. [1993]. In their approach, they try to reduce the number of pair-wise comparisons by avoiding biologically unnecessary comparisons. The method that they use to reduce the number of comparisons is related to the comparison and grouping algorithms. This kind of algorithmic related optimizations are out of the scope of this PhD research. The parallelization approach of their attempt is specific to their solution as they are not computing $N(N-1)/2$ comparisons in N number of sequences. As a result, the parallelization approach that they use cannot be applied generally for all all-to-all comparison problems.

To process all-to-all comparison problems, various distributed computing systems and run-time libraries have been used. Heitor and Guilherme [2005] proposed a methodology to parallelize a multiple sequence alignment algorithm by using a homogeneous set of computers with the Parallel Virtual Machine (PVM) library. In their work, a detailed description of the modules was provided and a special attention was paid to the execution of the multiple sequence comparison algorithm in parallel.

In 2002, Kleinjung et al. [2002] has proposed another solution for parallelizing MSA, which has an all-to-all comparison stage. In their approach they use the SIMD architecture. In their approach they have a sequence profiling stage which can be matched to pre-processing special case in this research. They use the MPICH package [Lusk et al., 1996] for parallel execution of all-to-all comparison. Their approach is to carry out multiple pair-wise comparisons in each node simultaneously. However, they have not provided detailed information on the implementation and managing memory in their approach.

Meng and Chaudhary [2010] presented a heterogeneous computing platform through a Message Passing Interface (MPI) enabled enterprise computing infrastructure for high-throughput biological sequence analysis. In order to achieve load balancing, they distributed the workload based on the hardware configuration. The whole database is split into multiple nearly-equal sized fragments; and then each of the computing nodes is assigned a number of database fragments according to its processing capacity. However, in practical computing of all-to-all comparison problems using their approach, data transmissions among the computing nodes

cannot be avoided at runtime. This drawback will be overcome in our work presented in this thesis.

Xiao et al. [2011] proposed a design and optimization of the BLAST algorithm in a GPU-CPU mixed heterogeneous computing system. Due to the specific architecture of the GPU, their implementation can achieve a six-fold speed-up for the BLAST algorithm. GPUs were also used for a parallel implementation of MAFFT for MSA analysis [Zhu et al., 2015]. In the work by Torres et al. [2012], they were configured for exact alignment of short-read genetic sequences. To accelerate the next generation long read mapping, Chen et al. [2014] made use of FPGA hardware to speed up sequence alignment. In comparison with all those hardware-dependant implementations, our work in this thesis does not rely on specific hardware.

Mendonca and de Melo [2013] proposed and evaluated a method for biological sequence comparisons. They implemented the method by adjusting workload in hybrid platforms composed of GPUs and multicores with SIMD extensions. Providing different task allocation strategies, their implementation achieved good performance benefits in executing the Smith-Waterman algorithm.

Singh et al. [2008] implemented Smith-Waterman applications by using desktop grids composed of CPUs and GPUs. In their implementation, two levels of parallel scheduling were provided: 1) a desktop grid for coarse-grained parallelization, and 2) GPUs for fine-grained parallelization. In their experiments running on a GPU-accelerated BOINC framework, over 10 times speedup were achieved in comparison with CPU-only systems.

Kedad-Sidhoum et al. [2014] proposed SWDUAL, an implementation of the Smith-Waterman algorithm on hybrid platforms consisting of multiple processors and GPUs. SWDUAL is based on a fast dual approximation scheduling algorithm. The algorithm selects the most suitable tasks to execute on the GPUs while keeping a good balance of the computational load over the platform. Different from those hardware-dependent methods, our work presented in this thesis does not rely on specific hardware such as GPU and SSE CPUs.

Zhu et al. [2011] employed the data parallelism paradigm that is suitable for handling large-scale processing to achieve a high degree of parallelism. In their research, they implemented the proposed algorithm on a homogeneous cluster using the MPI library and proposed a data partition and distribution strategy to reach the following requirements:

1. Load balance

   In their research, the completion time of sequential MSA on each processor is proportional to its load. Therefore, they use a parallel clustering scheme to partition the set of sequences into subsets based on sequence similarity. For n sequences and p processors for computation, each processors is allocated for n/p sequences

2. Minimized communication time

   To minimize communication time, each concurrent process of MSA should avoid accessing those subsets located on any of the other processors because the access of the remote data requires some form of communication. In their research, they only considered the fixed number of computers and the limitation of communication can also make the same data cannot be reused between different computers.

Catalyurek et al. [2002] described an approach for caching intermediate results for reuse in subsequent or concurrent queries for multiple sequence alignment (MSA) in Shard Memory Parallel machine. In their experiment, they used Sun supercomputing systems which have a huge memory than the distributed environment we consider.

Several approaches were also developed for load balancing in distributed computing of all-to-all comparison problems. One version of parallel all-to-all comparison of genome sequences was carried out by Hill et al. [2008]. In this attempt they have used a cluster of computers and main intention is provided on load balancing among the clusters. They have divided the triangular area in the correlation matrix which is to be calculated, into rows. Each row of comparisons has been assigned to a node of the cluster. Since, the tasks are in different size and the rows becomes smaller at the ends, they are assigning the rows dynamically to a node once a node finishes a task. In this attempt they have not considered about the scalability of the solution since if the number of nodes exceeds the number of rows in the matrix at the beginning or in the middle of the calculation, some nodes will be idle. At this moment there can still be dividable tasks available since calculation of each row also can be divided into parallel tasks, which they have not considered. In their research, the numbers of parallel tasks are not limited by the available memory, they have not addressed the similar problem considered in this PhD research.

Gunturu et al. [2009] proposed a load scheduling strategy, which depends on the length of

the sequence and the number of processors in the network. They assumed that all the processors in the network already had both sequences to be compared in their local memory. Our work in this thesis distributes data to the computing nodes with consideration of the computing tasks, thus avoiding this assumption.

An efficient grid scheduler is designed by Somasundaram et al. [2010] for parallel implementation of MSA algorithm on a computational grid. It splits a single alignment task into optimal-size subtasks and then distributes the subtasks among multiple processors. It assumes that the task running time is much higher than the communication overhead in sending data of the sub-matrix. This makes the method unsuitable for large-scale ATAC problems.

Church et al. [2011] proposed the design of a multiple sequence alignment algorithm on massively parallel, distributed memory supercomputers. In order to solve the limitations of random access memory (RAM) for dealing with large genomic populations, they proposed a method to break each sequence into small fragments that are distributed to free work nodes. Though it can improve performance, it would affect the accuracy of the final result.

Macedo et al. [2011] proposed an MPI/OpenMP master/slave parallel strategy to run DIALIGN-TX in heterogeneous multi-core clusters, with multiple allocation policies. In their research, they used three strategies: Hybrid Self Scheduling (HSS), Hybrid Fixed (HFixed) and Hybrid Weight Factoring (HWF). The results showed that the choice of an appropriate task allocation policy and a powerful master node has a great impact on the overall execution time. Self-Scheduling policy [Tang and Yew, 1986] always assigns one task to each node. When the node finishes to process a task, it requests a new one, until there are no more tasks left. This policy is able to reduce load balancing problems in a heterogeneous environment, at the expense of an increased communication cost. The Weighted Factoring (WF) policy [Hummel, 1996] assigns a weight to each node, representing its relative computing power. Task assignments are done in stages, where the number of tasks distributed in each stage is equal to half the remaining tasks and the number of tasks assigned to each processor depends on its weight.

Cui et al. [2010] proposed a novel sequence distribution strategy on the heterogeneous cluster system. In their research, they also only considered the numbers and length of sequences to determine the operation time. Besides, they did not consider the problem of resending the same sequences which would affect the performance of the whole system.

Distributing data to everywhere has its advantages and disadvantages. When the data sets

are distributed everywhere, scheduling any comparison task to any node will achieve perfect data locality, and the load balancing is straightforward to be promised. However, there are also obvious and major drawbacks. 1) The brute-force replication of data causes the worst storage usage, the longest time consumption for data transmission, and the highest cost of network communications. For example, a typical all-to-all comparison problem presented in reference [Hess et al., 2011] needs to process 268 GB cow rumen metagenome data. The worst storage usage pushes the boundary of the limitation of the available storage resources. In the experiment reported in reference [Das et al., 2013], the average time for deploying 10 GB datasets within a cluster with 14 worker nodes and 10 Mbps network takes nearly 150 minutes. The longest time consumption for data transmission deteriorates the performance of the overall execution time of the computing problem significantly. 2) Even if all the data sets can be distributed efficiently, much of the data stored in the nodes will never be used in actual caparison tasks, wasting the storage resources considerably. 3) These two drawbacks become even more evident and serious for large-scale all-to-all comparison problems with big data sets. As all-to-all comparison problems are a typical type of combinatorial problems, the complexity of processing such problems with big data sets increases exponentially with the increase of the data size.

2.3 Existing Solutions Based on Hadoop

In addition to the above mentioned specific solutions, recent attempts have been made to implement domain-specific all-to-all comparison applications [Chen et al., 2008, Schatz, 2009] by using the Hadoop framework. Enabling distributed processing of large data sets across clusters of commodity servers, Hadoop [Doug and Mike, 2005] is designed to scale up from a single server to thousands of machines. With a very high degree of fault tolerance, it can achieve high computing performance for distributed computation that well matches the MapReduce computing pattern. However, the Hadoop distributed file system (HDFS) and its data distribution strategy are very inefficient for all-to-all comparison problems due to the completely different computing pattern involved.

In this section, the Hadoop computing framework and Hadoop distributed file system (HDFS) will be discussed. Furthermore, solutions based on Hadoop will be summarized to show the limitation of this approach.

2.3.1 Hadoop Distributed Computing Framework

Hadoop [Doug and Mike, 2005] is a framework for running applications on large clusters built of commodity hardware. The Hadoop framework transparently provides applications both reliability and data motion. Hadoop implements a programming model named MapReduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. The Hadoop runtime system coupled with HDFS manages the details of parallelism and concurrency to provide ease of parallel programming with reinforced reliability.

In a Hadoop cluster, a master node controls a group of slave nodes on which the Map and Reduce functions run in parallel. The master node assigns a task to a slave node that has any empty task slot, an overview of Hadoop cluster can be expressed in Figure 2.1:

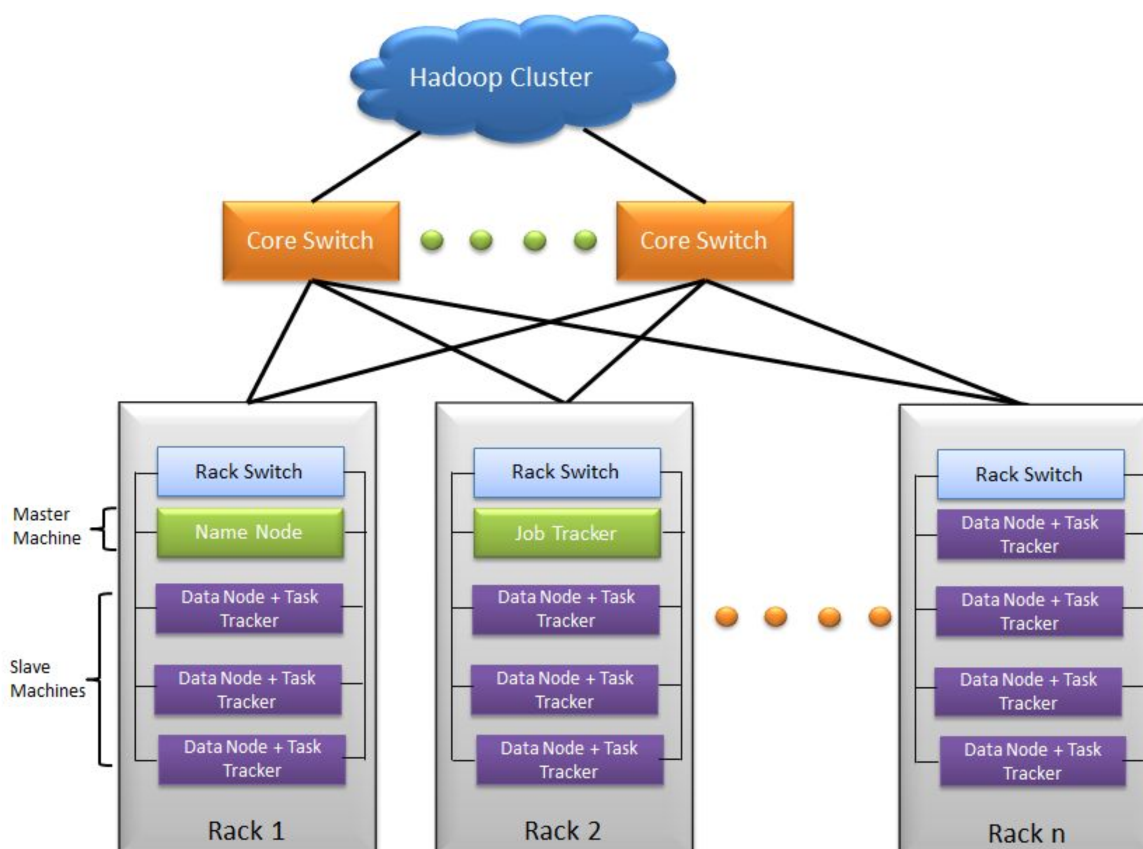


Figure 2.1: The architecture of Hadoop cluster.

Typically, computing nodes and storage nodes in a Hadoop cluster are identical from the hardware's perspective. In other words, the Hadoop's Map/Reduce framework and the Hadoop's

HDFS are, in many cases, running on a set of homogeneous nodes including both computing and storage nodes. Such a homogeneous configuration of Hadoop allows the Map/Reduce framework to effectively schedule computing tasks on an array of storage nodes where data files are residing, leading to a high aggregate bandwidth across the entire Hadoop cluster.

Hadoop framework has been widely used in solving all-to-all comparison problems in various domains. For hadoop framework, though it provides a complete solution for data distribution, task scheduling and resource management, it is designed for computing problems with MapReduce pattern. Hence, using MapReduce programming model to implement all-to-all comparison problems can cause serious performance problems, especially when processing big data set. Beside this, Hadoop is designed for homogeneous distributed systems, which makes it cannot achieve high performance when running on top of heterogeneous distributed systems.

2.3.2 MapReduce Programming Model

The MapReduce programming model was proposed by Google [Dean and Ghemawat, 2008] to support data-intensive applications running on parallel computers like commodity clusters. Two important functional programming primitives in MapReduce are Map and Reduce. The Map function is applied on application-specific input data to generate a list of intermediate (key-value) pairs. Then, the Reduce function is applied to the set of intermediate pairs with the same key. Typically, the Reduce function produces zero or more output pairs by performing a merging operation. All the output pairs are finally sorted based on their key values. Programmers only need to implement the Map and Reduce functions, because a MapReduce programming framework can facilitate some operations (e.g., grouping and sorting) on a set of (key-value) pairs.

The beauty of the MapReduce model lies in its simplicity, because the programmers just have to focus on data processing functionality rather than on parallelism details. The programmers provide high-level parallelism information, thereby allowing the Map and Reduce functions to be executed in parallel across multiple nodes.

To use MapReduce programming model, users have to implement two programming interfaces: Map and Reduce. Map takes an input pair and produces a set of intermediate key-value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function. The reduce function accepts an

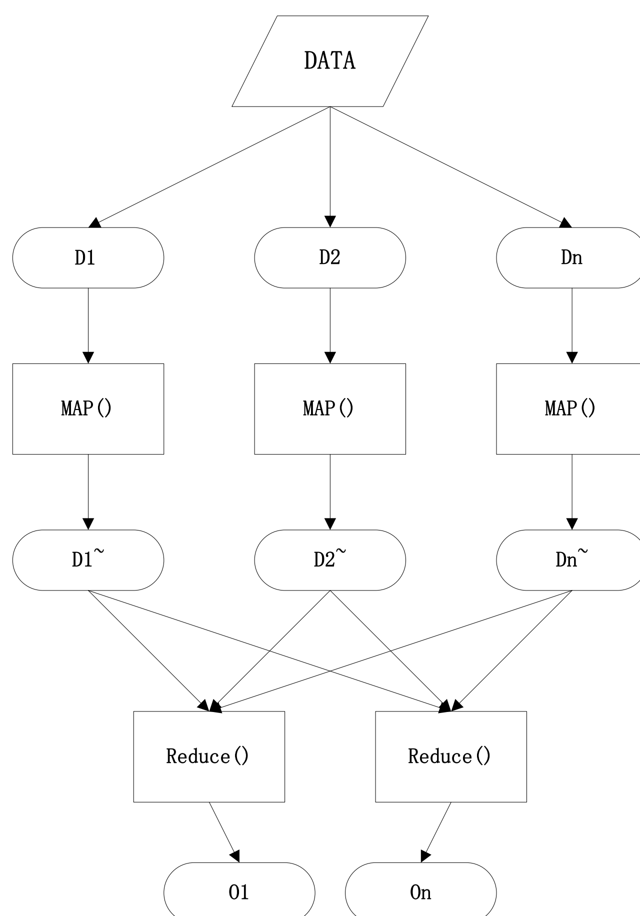


Figure 2.2: The MapReduce programming model.

intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

Many researcher chose MapReduce programming model or its extensions to solve ATAC problems. Though the procedures of processing MapReduce application and ATAC application have some similarities, there are huge differences which can significantly affect the system performance. In the following part, the issues of distributed processing ATAC problems by using MapReduce programming model will be discussed.

2.3.3 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [Borthakur, 2007] is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. The following is a subset of useful features in HDFS:

- File permissions and authentication.
- Rack awareness: to take a node's physical location into account while scheduling tasks and allocating storage.
- Safemode: an administrative mode for maintenance.
- Fsck: a utility to diagnose health of the file system, to find missing files or blocks.
- Fetchdt: a utility to fetch Delegation-token and store it in a file on the local system.
- Rebalancer: a tool to balance the cluster when the data is unevenly distributed among DataNodes.
- Upgrade and rollback: after a software upgrade, it is possible to rollback to HDFS' state before the upgrade in case of unexpected problems.
- Secondary NameNode (deprecated): performs periodic checkpoints of the namespace and helps keep the size of file containing log of HDFS modifications within certain limits at the NameNode. Replaced by Checkpoint node.
- Checkpoint node: performs periodic checkpoints of the namespace and helps minimize the size of the log stored at the NameNode containing changes to the HDFS. Replaces the role previously filled by the Secondary NameNode. NameNode allows multiple Checkpoint nodes simultaneously, as long as there are no Backup nodes registered with the system.

- Backup node: An extension to the Checkpoint node. In addition to checkpointing it also receives a stream of edits from the NameNode and maintains its own in-memory copy of the namespace, which is always in sync with the active NameNode namespace state. Only one Backup node may be registered with the NameNode at once.

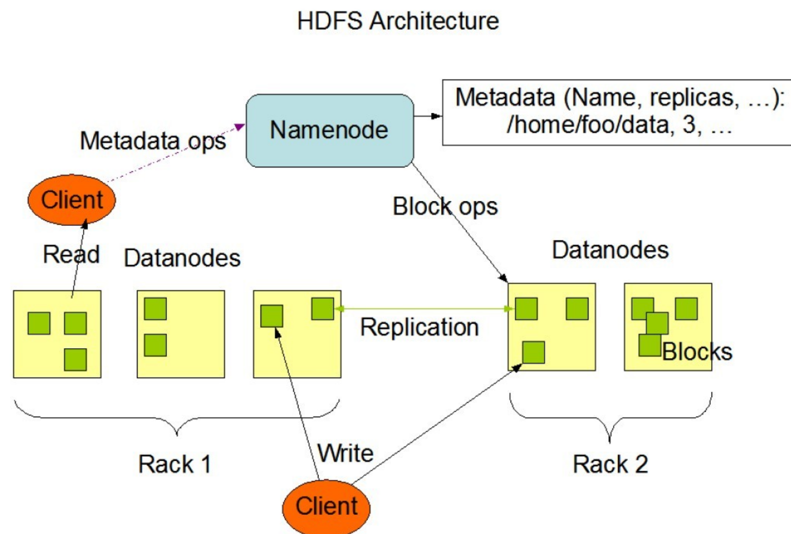


Figure 2.3: HDFS architecture.

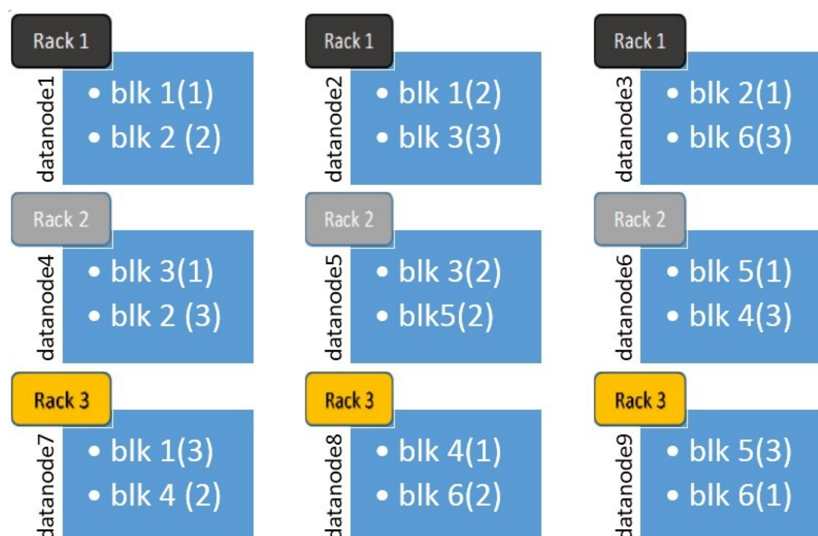


Figure 2.4: Example for HDFS data placement strategy.

The Figure 2.3 shows the system architecture of Hadoop Distributed File System (HDFS). HDFS has the following feathers:

1. Data Files stored in HDFS are divided into blocks and distributed to different worker nodes in the system.

   Each block unit is 64MB by default. With such a setting, a file can be very large since it can take advantage of all the disks in the cluster.

2. HDFS system uses the master-slave architecture.

   In particular, there is a master node (Namenode) with a number of slave nodes (Datanodes). In the cluster, Namenode maintains the file system namespace and the metadata for all the files while the Datanodes store the blocks. Periodically, every Datanode sends a heartbeat and a report of block list periodically to the Namenode such that Namenode can construct and update the blocksMap (a mapping table which maps data blocks to Datanodes). Because Namenode has the data placement information of all Datanodes, it coordinates access operations between clients and Datanodes. Finally, Datanodes are responsible for serving read and write requests.

3. HDFS replicates each block to tolerate fault.

   By default, every file has a same replication factor which equals to three. The placement of replicas follows a rackaware placement policy. Generally speaking, Namenode selects a random Datanode in the cluster to place the first replica of a block, then it asks this Datanode copies the block to a different Datanode of the same rack. The blocks third replica will be stored in a different rack. By storing two replicas in the same rack, this policy improves the write performance through reducing inter-rack write traffic. Figure 2.4 demonstrates an example of data placement. If the cluster only contains a single rack, three replicas of a block are randomly assigned to three different Datanodes.

2.3.4 ATAC Solutions Based on Hadoop

Recently, efforts have been made to process all-to-all comparison problems by using computing frameworks like Hadoop. A scalable and complete genotyping system is proposed by Marc et al. [2011] to produce genotypes from a variety of genome data. This suite of tools is built on top of Hadoop and MPI to support multiple nodes and large data sets. CloudBurst [Schatz, 2009] uses Hadoop for parallel short read-mapping, which is used in a variety of biological analyses including SNP discovery, genotyping, and personal genomics. The running time of CloudBurst

is shown to scale linearly with the number of reads mapped, and with near linear speed-up as the number of processors increases. In all these methods, all the data sets are distributed through Hadoop's data distribution strategy.

CloudBlast [Matsunaga et al., 2008] is a distributed implementation of NCBI BLAST. It integrates a number of technologies, e.g., Hadoop, virtual workspaces and ViNe together, to parallelize, deploy and manage bioinformatics applications. Comparing with MPI-based solutions such as mpiBLAST, CloudBLAST shows improvement. It also simplifies the development and management of the computing applications.

MRGIS [Chen et al., 2008] is a parallel and distributed computing platform based on MapReduce for executing geoinformatics applications. A Hadoop environment with 32 homogeneous worker nodes has been investigated for testing the efficiency of the MRGIS system.

Galaxy CloudMan [Afgan et al., 2010] is a batch queue processing system. It enables individual researchers to compose and control an arbitrarily sized compute cluster on Amazons EC2 cloud infrastructure without any informatics requirements. An entire suite of biological tools is available in CloudMan for immediate consumption. In the architecture of CloudMan, all data sets are stored in persistent storage components, and are delivered to different nodes from there. However, as a resource management system, CloudMan does not consider optimization of data distribution explicitly. In comparison, our work in this thesis investigates the optimization problem.

Beside this, researches on top of Hadoop also use Hadoop task scheduling strategy to deal with all the comparison tasks. First In, First Out (FIFO) is a simple and classic task scheduling strategy. As the default Hadoop scheduler [Dean and Ghemawat, 2008], it orders the jobs in a queue based on their arrival times, ignoring any heterogeneity in the system [Rasooli and Down, 2012]. Due to treat all the worker nodes in the system equally, Hadoop task scheduling strategy can be much inefficient when running in the heterogeneous distributed computing systems [Zaharia et al., 2008].

Recently, some researchers chose to process ATAC problems by using extended MapReduce programming models like iHadoop and iMapReduce. iHadoop [Elnikety et al., 2011] is a modified MapReduce model and optimized for iterative computations. iHadoop optimizes for iterative algorithms by modifying the dataflow and task scheduling to allow iterations to run asynchronously.

iMapReduce [Zhang et al., 2011] also provides a framework that can model iterative algorithms. It uses persistent tasks and keep all the tasks alive during the whole iterative process. iMapReduce keeps a one-to-one mapping between the map and the reduce tasks and its task scheduler assigns statically every map task and its corresponding reduce to the same worker.

Though each input data item has to be used for many comparison tasks in ATAC problems, any two different data items only have to be compared once, which makes it have fundamental differences between typical iterative algorithms [Yu and Hong, 2013]. Hence, treading ATAC problems directly as iterative computing problems cannot bring significant performance improvement. Moreover, considering these solutions are all based on MapReduce programming model, the issues of using MapReduce programming model are still exist.

The Hadoop distributed file system (HDFS) provides a strategy to distribute and store big data sets. In the HDFS data strategy, data items are randomly distributed with a fixed number of duplications among all storage nodes. While multiple copies of data items in HDFS enhance the reliability of data storage, the HDFS data strategy is inefficient for all-to-all comparison problems due to its poor data locality, unbalanced task load and big solution space for data distribution [Qiu et al., 2009].

The inefficiency of Hadoop's data distribution strategy in all-to-all comparison problems is mainly due to the following reasons: 1) Although the number of replications of data files can be manually set in HDFS, one does not know how to set this number and thus tends to use the default number of three. Once set, this number becomes a constant regardless of the number of machines and the number of data files. 2) The location of each data file is randomly determined in HDFS. This is not suitable for general all-to-all comparison problems for high performance that requires good data locality. 3) Increasing the number of replications of data files does not necessarily improve the overall computing performance much due to the poor data locality unless replicating the data files to everywhere.

Despite significant developments in processing all-to-all comparison problems, technical gaps still exist in this area. The existing solutions mainly focus on parallelizing different all-to-all comparison algorithms and providing load balancing. None of them have paid special attention to data distribution. Most previous methods have assumed that all data can be stored in each worker node, implying poor scalability for big data sets. While some of the existing methods use distributed file systems such as HDFS in Hadoop, the data strategy in use is

nonetheless inefficient when processing large-scale all-to-all comparison problems.

2.4 Existing Solutions Based on Other Computing Frameworks

As a technical tool that support processing computing problems with big data sets, computing framework has emerged recent years in different domains [Dean and Ghemawat, 2008, Hill et al., 2008, Hindman et al., 2011, Neumeyer et al., 2010, Zaharia et al., 2010]. Addressing Hadoop's weakness, improved methods have been proposed by many researchers. In this section, solutions based on different computing framework are introduced.

2.4.1 All-Pairs

All-pairs [Moretti et al., 2010] is an abstraction for data-intensive computing on campus grids. This research just focuses on the all-to-all problem and provides an abstraction for users to deal with this kind of problem.

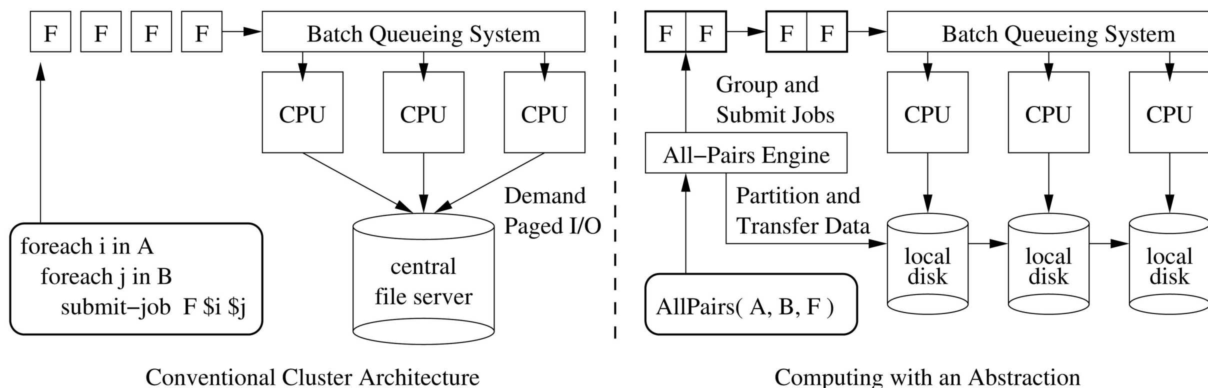


Figure 2.5: Comparison between two different architectures.

Results in Figure 2.5 compares the difference approach of distributed processing all-to-all comparison problems. When using a conventional computing cluster, the user partitions the workload into jobs, then a batch queue distributes jobs to CPUs, where they access data from a central file server on demand. When using an abstraction like All-Pairs, the user states the high-level structure of the workload, the abstraction engine partitions both the computation and the data access, transfers data to disks in the cluster, and then dispatches batch jobs to execute on the data in place.

By considering issues such as the number of compute nodes, data distribution, dispatch latency, failure probability, Resource limitations and semantics of failure, the system architecture of All-pairs framework is briefly shown in Figure 2.6.

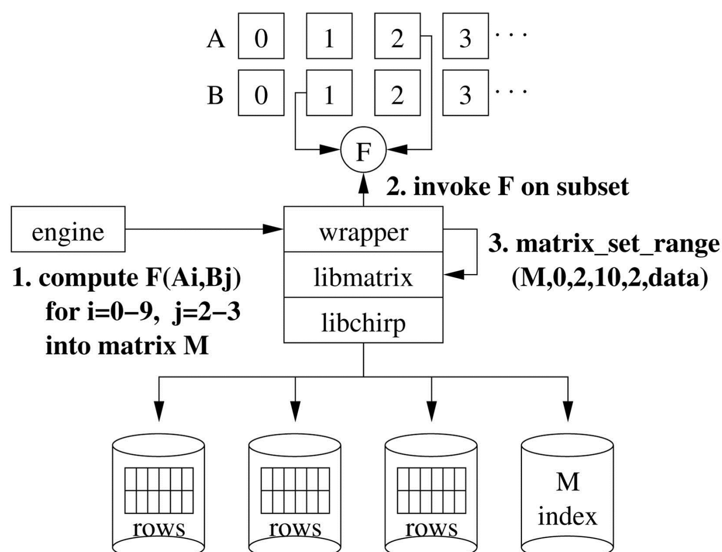


Figure 2.6: The architecture of All-pairs computing framework.

In this research, the all-pairs abstraction solves the all-to-all comparison problems in the following steps:

1. Modelling the computing system and determine the numbers of computing nodes to use.

   In order to decide how many CPUs to use and how to partition the work, this research models the performance of the system by considering the size of input data, network bandwidth and the dispatch latency of the system.

2. Distributing all the input data to every computing nodes.

   To deliver all of the data to every node, a spanning tree method is developed, which performs streaming data transfers and completes in logarithmic time.

3. Dispatching batch jobs to every computing nodes.

   After transferring the input data, the All-pairs framework constructs batch submit scripts for each of the grouped jobs and queues them in the batch system which instructions to run on computing nodes.

4. Collecting all the output results.

All the output results form All-pairs jobs goes to a distributed data structure provided by the system. The data structure is a matrix whose contents are partitioned across a cluster of reliable strong nodes maintained separately from the campus grid.

The user invokes All-Pairs as follows:

$$AllPairs \quad SetA \quad SetB \quad Function \quad Matrix,$$

where SetA and SetB are text files that list the set of files to process, Function is the function to perform each comparison, and Matrix is the name of a matrix, where the results are to be stored. Function is provided by the end user and may be an executable written in any language, provided that it has the following calling convention:

$$Function \quad SetX \quad SetY,$$

where SetX and SetY are text files that list a set of files to process, resulting in a list of results on the standard output.

In the current implementation of All-paris, the user's function is essentially a single-CPU implementation of All-Pairs. Though it has some advantages such as good usability, the benefits of distributed systems are not fully utilized and the All-pairs is running basically like controlling multiple single machine programs in the distributed environment through a centralized system.

Though this research provides the solution for solving general all-to-all comparison problems, there are still some limitations:

1. The All-pairs abstraction focuses on dealing with data that can be stored in every computing node. Therefore, the abstraction cannot deal with the all-to-all comparison problems with huge amount of data.

2. The All-pairs abstraction supports only limited numbers of computing nodes that makes the abstraction hard to be extended and it makes the users cannot make full use of all the computing resources.

3. The All-pairs abstraction is designed to run in the campus grid which has some specific

characters which makes the solution hard to run on other kinds of distributed computing systems.

4. The applications running on All-pairs system are still traditional sequential programs, which is not originally designed for distributed computing.

2.4.2 Bi-Hadoop

Bi-Hadoop [Yu and Hong, 2013] is an extension of Hadoop to better support binary-input applications. As shown in Figure 2.7, it includes three parts: 1) An easy-to-use interface for users to describe the association between a task and its inputs. 2) A task scheduling algorithm that is able to exploit data locality for binary-input applications. 3) A caching mechanism to accelerate data reads. The caching mechanism is an integral part of our extension that materializes the improved data locality exposed by our scheduling algorithm.

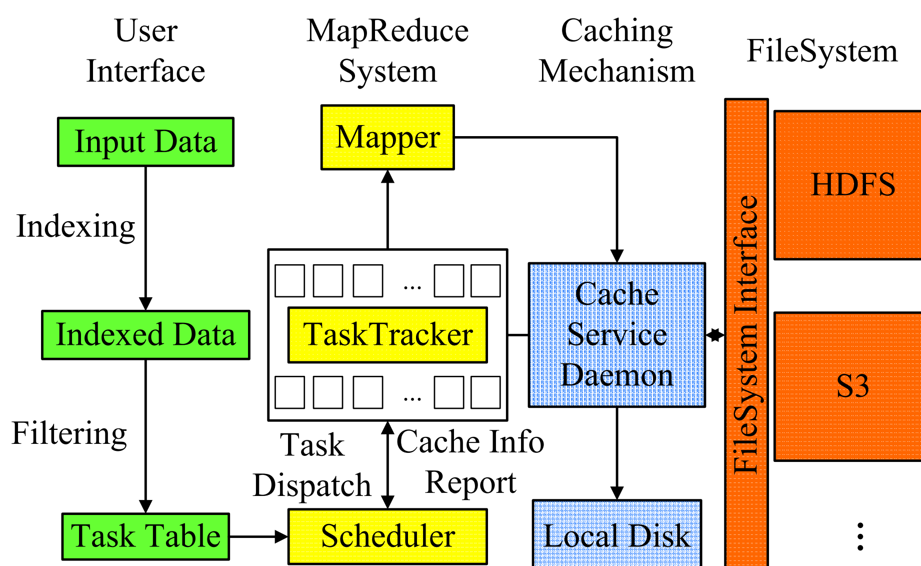


Figure 2.7: Bi-Hadoop extension system overview.

Considering Bi-Hadoop is developed based on MapReduce programing model, users also have to use programming interfaces such as Map and Reduce to implement their applications. Beside this, users only need to perform one extra piece of work: specifying which two splits form a task.

To finish this work, a user interface shown in Figure 2.8 is designed to assign IDs to splits by letting the user name splits with strings.

```
1  // The Bi-Hadoop filter interface
2  public interface BiHFilter {
3    public boolean accept(String split0Id, String
         split1Id);
4  }
5
6  // A usage example: matrix-vector multiply
7  public class MatVecMulFilter implements
       BiHFilter {
8    public boolean accept(String split0Id, String
         split1Id) {
9      if (!isAMatrix(split0Id)) return false;
10     if (!isAVector(split1Id)) return false;
11     colId = getMatrixColId(split0Id);
12     rowId = getVectorRowId(split1Id);
13     if (colId == rowId) return true;
14     return false;
15   }
16 }
```

Figure 2.8: Bi-Hadoop user interface.

Users specify the map tasks by customizing a filter class, which returns true if a pair of split IDs form a task, and false otherwise. Figure 2.8 illustrate the simple interface and a usage example of matrix vector multiplication. Users can manipulate the ID strings in a customized fashion (such as getMatrixColId()) to identify the file split and form the tasks.

For all-to-all comparison applications, the unary input oriented Hadoop system has multiple limitations:

1. Developers need to work around the unary input requirement, which makes it less natural to program the applications.

2. When a workaround method is used, the built-in locality awareness of Hadoop becomes less effective or non-effective.

3. As binary-input tasks often share their data blocks, there are many unique locality optimization opportunities in these applications that cannot be exploited by existing Hadoop.

By providing a binary-input aware task scheduler and a caching subsystem, it outperforms Hadoop by up to 3.3 times and a 48% reduction in remote data reads for binary-input applications.

The key of this research is the specific caching system designed for all-to-all comparison problems. It sits between MapReduce system and HDFS system, therefore it is user transparent. It supplies a split to the requesting map task if the split is in the cache, and will seamlessly resolve to the native HDFS data read mechanism when the requested split is missing in the cache. However, though the caching mechanism improves the computing performance, based on structure in Figure 2.7, all data still needs to be distributed through Hadoop's data strategy. Thus, these improved methods still have limitations.

2.5 Classification of Distributed Computing Systems

The computing framework and related strategies proposed in this thesis are based on the distributed computing systems. To fully understand the characteristics of distributed computing systems, the classification and development of the distributed computing systems are introduced in this section. Among different distributed systems, a special attention is paid on the distributed computing system with distributed storage, which is the target platform for this PhD thesis.

A taxonomy of different computer architectures and their memory models are shown in the figure 2.9. The broad classification of parallel computer models according to Flynn [1972], is Single Instruction Multiple Data (SIMD) and Multiple Instructions Multiple Data (MIMD).

SIMD systems were widely used early days of parallel computing, but now facing extinction. These systems consist of many numbers (even thousands) of processors and each processor has a local memory. Every processor must execute same instruction over different data, at each computing or 'clock' cycle. To bring the data to local memory, an explicit communication must pass among different processors. The complexity and often the inflexibility has restricted it be used mostly for special purpose applications [Trelles, 2011].

For MIMD systems, machines are more amendable to bioinformatics [Trelles et al., 1998]. In MIMD computers, each processor can execute asynchronously and independently from other processors, at its own speed on different data. This flexibility has lead more attention of high performance parallel computing to MIMD systems, including bioinformatics.

Shared and distributed memory structures have a clear distinction. A system is said to have 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

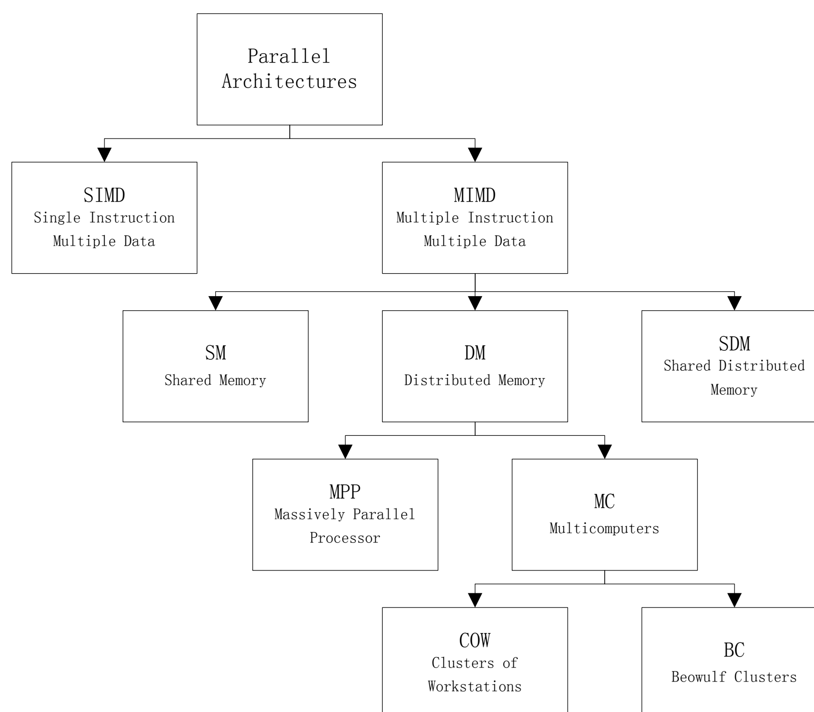


Figure 2.9: Distributed computing system architecture taxonomy.

architecture.

Shared memory systems have natural advantages on processing all-to-all comparison problems. For example in the research proposed by Catalyurek et al. [2002], they employs a read-only cache on Shared-memory system which can support multiple processors access the same data sets concurrently. However, shared memory structures also have the following limitations:

1. Bandwidth of the shared memory system does not grow as the number of processors in the machine increases, which implies poor scalability when processing all-to-all comparison problems with big data sets.

2. Specific hardware are needed to built shared memory systems, which makes high system cost.

Comparing to the shard-memory system, it is becoming increasingly cost-effective to build large scale memory distributed computing system by using currently available off-the-shelf components. Distributed memory systems have been widely used in many areas because of the high processing power, scalability and availability. In this thesis, we are choosing developing

our computing framework based on the distributed memory architecture which can make our solution support more general scenarios.

2.6 Different Cluster Management Platforms

Driven by different kinds of applications, researchers and practitioners have been developing a diverse array of cluster computing frameworks to simplify programming the cluster. It seems clear that new cluster computing frameworks will continue to emerge, and that no framework will be optimal for all applications. Therefore, a scalable and efficient system that supports a wide array of both current and future frameworks is important. Two existing resource management platforms will be introduced in this section.

2.6.1 Mesos

Apache Mesos [Hindman et al., 2011] is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine.

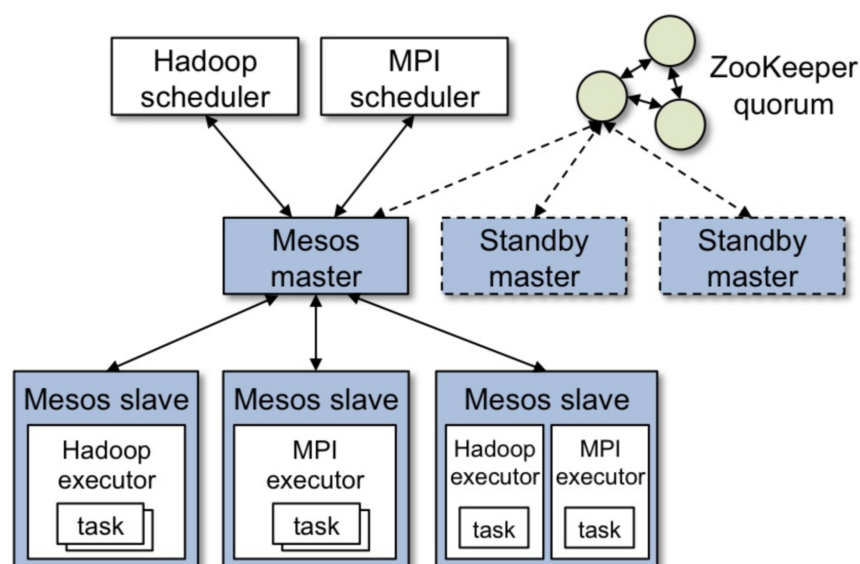


Figure 2.10: The architecture of Mesos.

Figure 2.10 shown the main components of Mesos. Mesos consists of a master daemon

that manages slave daemons running on each cluster node, and Mesos applications (also called frameworks) that run tasks on these slaves.

The master enables fine-grained sharing of resources (cpu, ram) across applications by making them resource offers. Each resource offer contains a list of . The master decides how many resources to offer to each framework according to a given organizational policy, such as fair sharing, or strict priority. To support a diverse set of policies, the master employs a modular architecture that makes it easy to add new allocation modules via a plugin mechanism.

Different computing frameworks such as Hadoop, Spark, Kafka and Elastic Search can be implemented on top of Mesos platform. To integrated these frameworks with Mesos platform, a scheduler and an executor must be implemented. The scheduler that registers with the master to be offered resources, and the executor process that is launched on slave nodes to run the frameworks tasks.

While the master determines how many resources are offered to each framework, the frameworks' schedulers select which of the offered resources to use. When a frameworks accepts offered resources, it passes to Mesos a description of the tasks it wants to run on them. In turn, Mesos launches the tasks on the corresponding slaves. An example of running computing frameworks on top of Mesos is graphically shown in Figure 2.11.

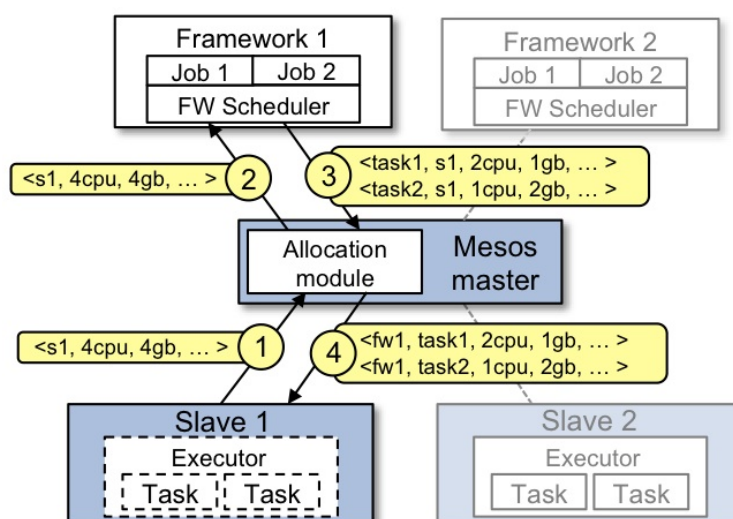


Figure 2.11: An example of using Mesos.

The system runs in the following steps:

1. Slave 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then

invokes the allocation policy module, which tells it that framework 1 should be offered all available resources.

2. The master sends a resource offer describing what is available on slave 1 to framework 1.

3. The frameworks scheduler replies to the master with information about two tasks to run on the slave, using $\langle \text{2CPUs}, 1GB\text{RAM}\rangle$ for the first task, and $\langle \text{1CPUs}, 2GB\text{RAM}\rangle$ for the second task.

4. Finally, the master sends the tasks to the slave, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2.

Mesos has the following Features:

- Fault-tolerant replicated master using ZooKeeper.
- Scalability to 10,000s of nodes using fast, event-driven C++ implementation.
- Isolation between tasks with Linux Containers.
- Multi-resource scheduling (memory and CPU aware).
- Efficient application-controlled scheduling mechanism (resource offers) that lets frameworks achieve their own placement goals (e.g. data locality).
- Java, Python and C++ APIs for developing new parallel applications.
- Web UI for viewing cluster state.

2.6.2 Yarn

Yarn is next generation implementation for MapReduce. The fundamental idea of MRv2 is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons. The idea is to have a global ResourceManager (RM) and per-application ApplicationMaster (AM). An application is either a single job in the

classical sense of Map-Reduce jobs or a DAG of jobs. By using ResourceManager and ApplicationMaster, Yarn can not only support MapReduce computing framework but also computing frameworks with other computation patterns as shown in Figure 2.12.

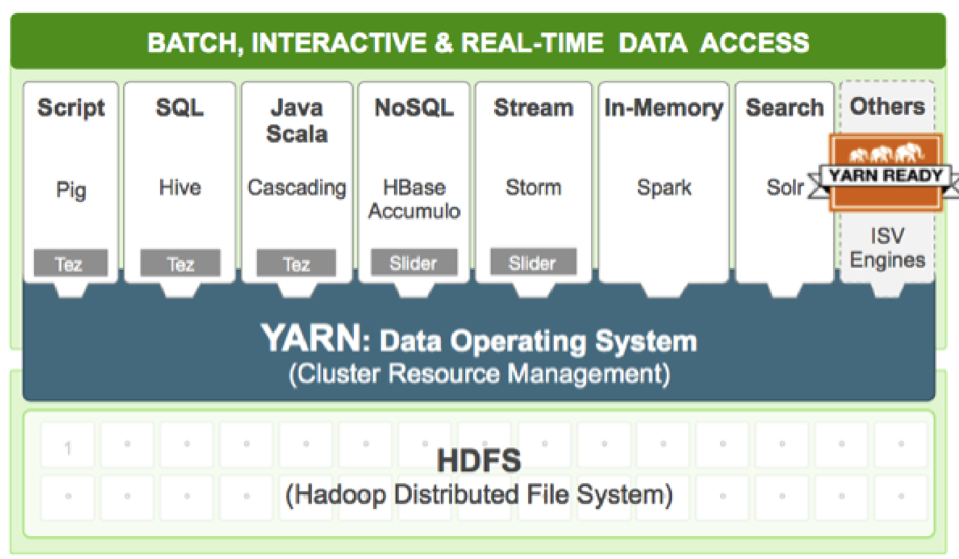


Figure 2.12: Different layers of using Yarn.

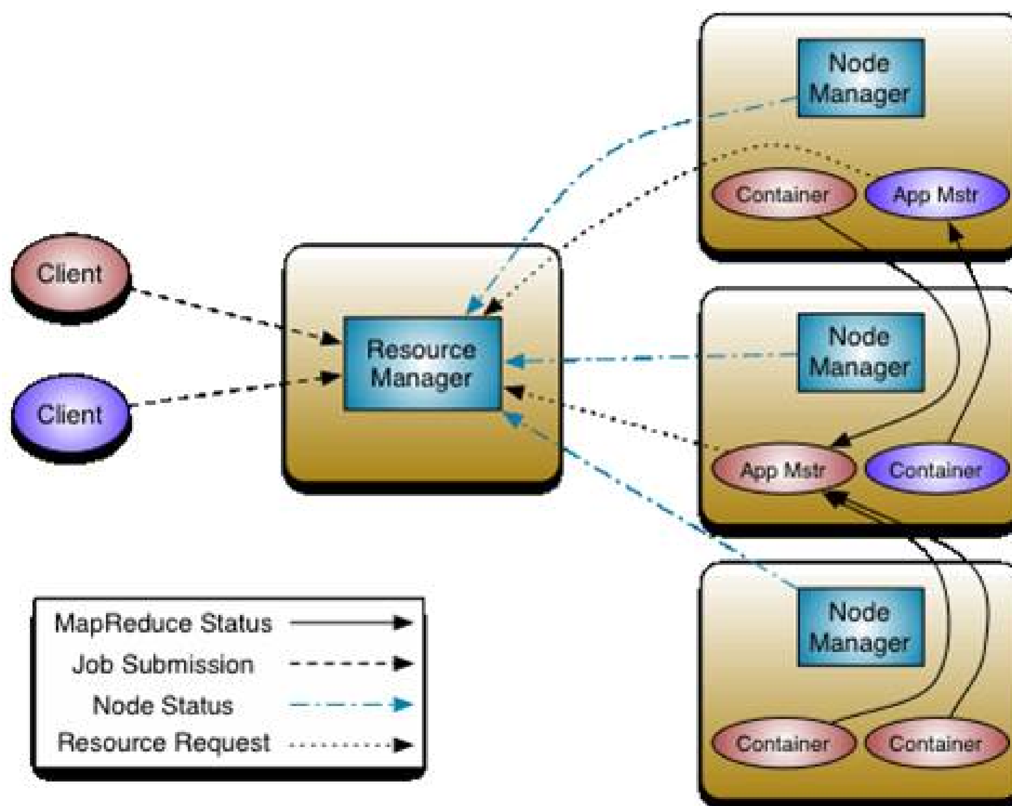


Figure 2.13: The architecture of Yarn.

The ResourceManager shown in Figure 2.13 has two main components: Scheduler and ApplicationsManager.

- The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based the resource requirements of the applications; it does so based on the abstract notion of a resource Container which incorporates elements such as memory, CPU, disk, network etc. In the first version, only memory is supported.
- The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure.

The NodeManager in each worker node is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.

The per-application ApplicationMaster in each worker node has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

2.7 Summary of Literature Review

In the literature review, we summarise the current approaches of solving all-to-all comparison problems in distributed computing systems. Though distributed computing frameworks such as Hadoop have been used, limitations are still exist in this area.

Firstly, solutions have been proposed for special-purpose all-to-all comparison problems such as the popularly used BLAST [NCBI, 1990] and ClustalW [Center, 2007], or for different types of computing architectures with GPUs [Xiao et al., 2011] or shared memory [Catalyurek et al., 2002]. However, all these solutions require that all data files be deployed to each of the nodes in the distributed system. While distributing whole data sets everywhere is conceptually

easy to implement, it causes significant time consumption and communication cost and demands a huge amount of storage space. Therefore, the data distribution strategy from these previous solutions is not scalable to the big data processing problems considered in this thesis.

Secondly, efforts based on Hadoop computing framework have been summarized. Enabling distributed processing of large data sets across clusters of commodity servers, Hadoop [Doug and Mike, 2005] is designed to scale up from a single server to thousands of machines. With a very high degree of fault tolerance, it can achieve high computing performance for distributed computation that well matches the MapReduce computing pattern. However, the Hadoop distributed file system (HDFS) and its data distribution strategy are very inefficient for all-to-all comparison problems due to the completely different computing pattern involved.

Thirdly, considering different scenarios, many researchers provide different computing frameworks to overcome the limitation of Hadoop. Though the computing performance can be improved, the existing solutions share different scenarios with our work. For our work in this thesis, we plan to design a new computing framework specific for all-to-all comparison problems, not only extend the Hadoop system or processing small scale ATAC problems.

In the following part, we introduce the different architectures of distributed platforms and show both the advantages and disadvantages of different architectures. It will help us for fully understanding the structural characteristics of distributed computing systems.

Finally, we discuss different cluster management platforms for distributed computing system, which need to be considered and integrated in our implementation system.

Overall, we have covered most of the related areas of our research and have made a good base to our research and a good understanding of the current state-of-the-art.

Chapter 3

The Front-end Interfaces for Distributed Computing Framework of ATAC Problems

---

In this chapter, after formally defining the all-to-all comparison problem, the design of the efficient and simple front-end interfaces is provided for the distributed computing framework of ATAC problems. The front-end interfaces are designed for two purposes: the operation interfaces for system users and the programming interfaces for application developers. Simple operation interfaces are developed for uses to deploy their data sets and ATAC applications and collect all the results. A programming model is designed to abstract the processing procedures of general ATAC problems. It also provides powerful application programming interfaces (APIs) to help developing ATAC applications without considering parallel system issues. After describing the design of the front-end interfaces, this chapter ends with a typical ATAC example of showing the use of the front-end interfaces.

3.1 All-to-all Comparison Problems

Let A, P, C and M denote the data set to be pairwise compared, the pre-process function on A, the comparison function on A and the output similarity matrix of A, respectively. Characterized by the Cartesian product or cross join of the data set A, the all-to-all comparison (ATAC) problem discussed in this thesis is mathematically stated as follows:

$$M_{ij} = C(P(A_i), P(A_j)),\ i, j = 1, 2, ..., |A| \tag{3.1}$$

where A_i represents the *i*th item of A, M_{ij} is the element of M resulting from the comparison between A_i and A_j, $|A|$ means the size of A.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	...
A_1		*	*	*	*	*	*	*
A_2			*	*	*	*	*	*
A_3				*	*	*	*	*
A_4					*	*	*	*
A_5						*	*	*
A_6							*	*
A_7								*
⋮								

Figure 3.1: Similarity matrix of all-to-all comparison problems.

Generally, ATAC problems have the following qualities:

1. $C(P(A_i), P(A_j)) = C(P(A_j), P(A_i))$

   There is no order between two different data items.

2. $C(P(A_i), P(A_i)) = 0$

   The result of comparing two same data items is 0.

The original motivation for studying this problem came from bioinformatics, where the elements being pairwise compared are typically genes or genomes [Hao et al., 2003, Wang, 2009]. The same pattern also applies in many other domains, e.g., comparing images, video or audio in biometrics and data mining [Phillips et al., 2005].

In the following subsections, the current researches for ATAC problems in bioinformatic and other different areas are introduced.

3.1.1 All-to-all Comparison Problems in Bioinformatics

Bioinformatics is a science to store, search and analyse biological information using computer science and technology [Xu et al., 2007] and all-to-all comparison is a key calculation stage in Multiple Sequence Alignment (MSA) and studying of phylogenetic diversity in protein families

[Trelles et al., 1998]. Normally, applications for these problems include the calculation of a cross-similarity matrix between each pair of sequences followed by several data grouping stages.

Multiple Sequence Alignments (MSA) is an essential method for protein structure and function prediction, phylogeny interference and other common tasks in sequence analysis [Edgar and Batzoglou, 2006]. The first stage for solving a MSA includes calculating a cross-similarity matrix between each pair of sequences, followed by determining the alignment topology and finally solving the alignment of sequences. Numerous methods have been proposed to align more than two sequences (MSA) which involve pair-wise all-to-all comparison [Barton, 1990, Feng and Doolittle, 1987, Higgins and Sharp, 1988, Jaap and Heringa, 1999, 2002, Notredame et al., 2000, Schuler et al., 1991, Thompson et al., 1994]. Although these methods give useful results, they are computationally intensive [Date et al., 1993]. This pair-wise comparison is a natural target for parallel as each comparison is independent from others [Trelles et al., 1998].

Beside this, alignment-free methods [Vinga and Almeida, 2003] are increasingly used for genome comparison, in particular for genome-based phylogeny reconstruction. For many popular alignment-free methods such as feature frequency profile (FFP) [Sims et al., 2009], composition vector (CV) [Chan et al., 2010, Wang, 2009], return time distribution (RTD) [Kolekar et al., 2012], they all have all-to-all comparison as a major computation step.

3.1.2 All-to-all Comparison Problem in Other Domains

Beside bioinformatics, computing problems with the all-to-all comparison pattern can also be found in domains such as Biometrics and data mining.

Biometrics is the science and technology of identifying humans from measurements of the body characteristics, such as fingerprints, eye retinas and irises, voice patterns, facial patterns. For a biometric pattern recognition system, it usually acquires biometric data from an individual, extracts a feature set from the acquired data, and compares this feature set against the template set in the database. Matching scores will be generated to quantify the similarity between the input and the template representations [Jain et al., 2004].

In the data mining area, one phase of knowledge discovery is reacting to bias or other noise within a set. In order to improve overall accuracy, researchers must determine which classifiers

work on which types of noise. To do this, they use a distribution representative of the data set as one input to the function and a type of noise (also defined as a distribution) as the other. The function returns a set of results for each classifier, allowing researchers to determine the most suitable classifier for that type of noise on that distribution of the validation set [Moretti et al., 2010].

In addition to these areas above, there are also many computing problems that are similar to all-to-all comparison pattern but for different purposes such as data clustering or filtering [Andoni and Indyk, 2008, Bayardo et al., 2007].

Based on the above introduction, it is clearly that computing problems with all-to-all comparison pattern have been widely applied in various domains and the research for processing all-to-all comparison problems by using distributed computing systems is necessary to help users solving this kind of problems efficiently.

3.2 Overview of the Distributed Computing Framework for ATAC Problems

In this section, the architecture of our distributed computing framework for all-to-all comparison problems is graphically shown in Figure 3.2. For our distributed computing framework, mainly three roles are involved: the system users, the application developers and the back-end computing system.

To simplify the utilization of our distributed computing framework, the front-end users and back-end computing systems are designed to be separated. In Figure 3.2, there are three layers designed in our distributed computing framework:

1) User Layer.

   There are two types of users in our distributed computing framework: 1) users who are focusing on solving specific ATAC problems and 2) developers who are responsible for implementing ATAC applications.

   Generally users are mainly focusing on getting and analysing the comparison results for specific purposes. Considering users usually do not have specialized knowledge

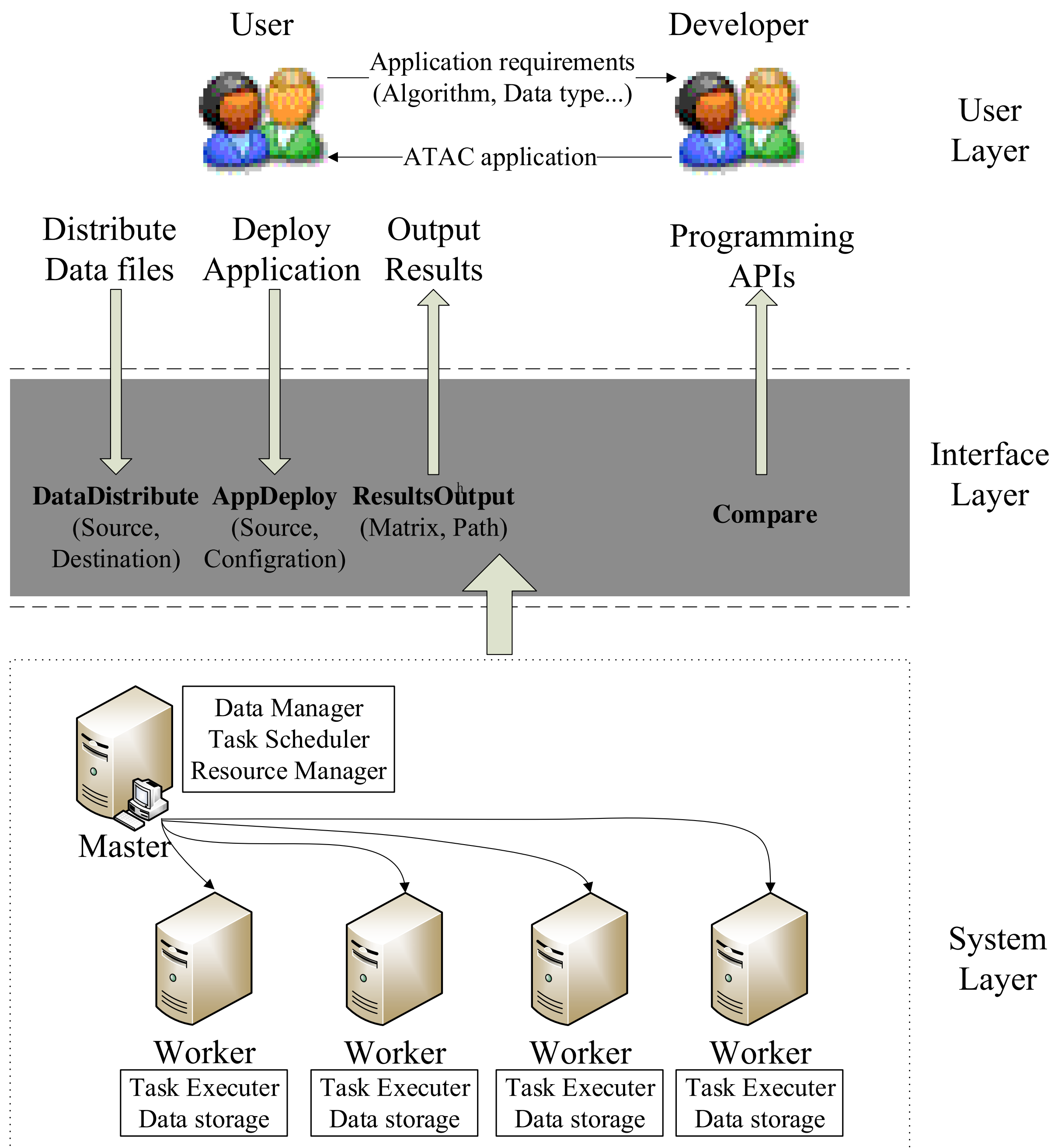


Figure 3.2: Architecture of the All-to-all computing framework.

about distributed computing and application development, freeing them from the implementation issues can greatly reduce the threshold for using our distributed computing framework.

To develop the domain-purpose ATAC applications, user should provide application requirements such as all-to-all comparison algorithms, input/output data types to developers

so the developers can implement all-to-all comparison applications based on these information. For these developers, the efficient of developing the ATAC applications in the distributed environment is closely related to the distributed system issues them have to consider.

By considering both the users and developers, two different kinds of interfaces are developed in our distributed computing framework, which will be discussed later in this chapter.

2) Interfaces Layer.

In our distributed computing framework, to minimize the affections made by the complex distributed system issues, the interface layer is designed to make the back-end computing system transparent to users. Hence, users only have to pay attention on the aspects related to their work. Complex distributed issues will be handled automatically by our computing framework.

By interacting with the simple interfaces, user and developers do not have to understand the details of the back-end computing systems. Also, the programming APIs provided to developers do not require specific parallel programming techniques, which can make developers implement all-to-all comparison applications efficiently. Moreover, the interfaces are designed for general ATAC problems so ATAC applications with different algorithms, data types or resources requirements can all be supported.

3) System Layer.

For our distributed computing framework, the data distribution, task scheduling and task execution are responsible by the back-end computing system. It is based on the distributed cluster and a master/slaves structure is used in this computing system. After users and developers providing the required data sets and ATAC application, the whole work flow will be automatically divided and distributed to different worker nodes by following the specific data distribution and task scheduling strategies developed in this thesis.

Considering our computing framework is based on general distributed computing architecture and does not have any special requirements such as shared-memory, limited number of worker nodes or specific network topology, the hardware system can straightforward to be extended to achieve higher performance, the experiments designed in the following chapters also show the high scalability for our computing framework.

In this chapter, we are focusing on describing about the design of front-end interfaces. The details of back-end computing systems will be discussed in the following chapters.

3.3 User Interfaces Design

As we mentioned in Figure 3.3, for users who aim to solve ATAC problems, the only operations needed are submitting their data and applications to the system and collecting the final comparison results. Thus, to help users finish the necessary work, our distributed computing framework provides three simple user interfaces: data interface, application interface and result interface.

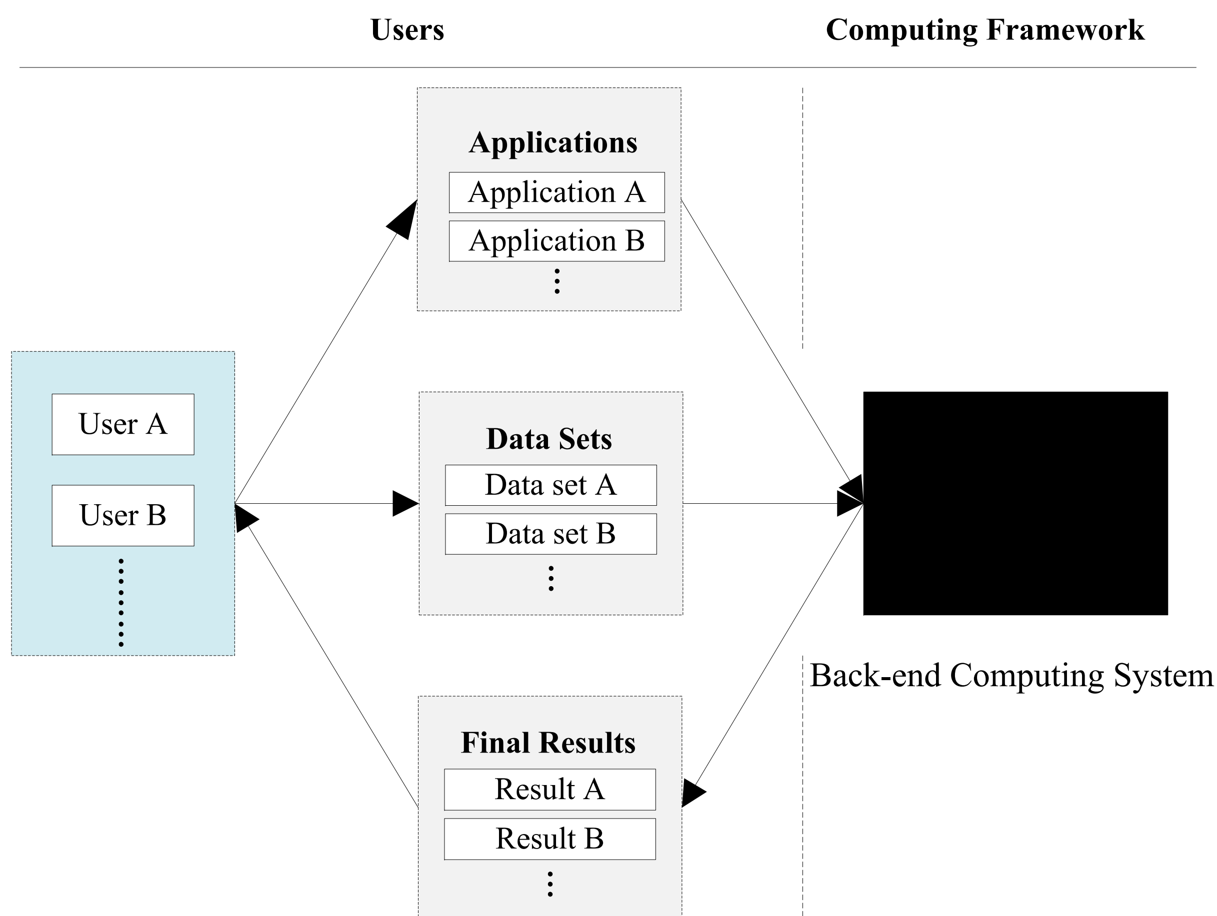


Figure 3.3: User interfaces for the All-to-all computing framework.

3.3.1 Data Interface

Before running the all-to-all comparison applications, the required data sets need to be deployed to the distributed system first. In our front-end interfaces, a data interface is designed to help

users upload their data to the distributed system.

The data interface is designed as follows:

Datadistribution (Source path, Destination path)

Table 3.1: Parameters of the data interface.

Parameters	Details
Source path	The location of the input data set
Destination path	The location to save all the distributed data

In this data interface, users only have to provide the information about the location and destination of data sets as shown in Table 3.1. Usually the input data sets is stored in the client node operated by users, where the source path can direct to a folder. For destination path, though different partitions of data sets will be stored in different worker nodes in the back-end distributed system, a global unified path is used to simplify the users' operation.

Based on the information provided by users and the information collected from distributed systems, our back-end computing system will distribute all the data files to each worker node based on our data distribution strategies.

3.3.2 Application Interface

Another information needs to be provided by users is the specific ATAC applications developed for their domain-purpose ATAC problems. Our computing framework provides an application interface for users to deploy their applications to the back-end distributed computing systems.

The application interface is designed as follows:

ApplicationDeploy (Source path , Destination path, Configuration file)

In the application interface, users need to define three parameters shown in Table 3.2.

Same as the data interface, the ATAC application is usually uploaded from the client node, where the source path can be direct to the ATAC application. A global unified destination path is used to show the location of ATAC applications.

Table 3.2: Parameters of the application interface.

Parameters	Details
Source path	The location of the application
Destination path	The location to save the deployed application
Configuration file	Information related to run the application

Table 3.3: Configurations of the ATAC application.

Items	Details
Data path	The location of data sets need to be processed
Result path	The location of all the comparison results

Configuration file is stored in the master node to provide required information for the master node. For each ATAC application, each time before starting the application, the data need to be processed should be determined and the output path of all the comparison results should also be provided. By putting all these information into the configuration file, developers do not have to compile the ATAC application each time after these value are changed.

Different from deploying data sets, in our distributed computing framework, the ATAC applications should be both deployed to the master node and worker nodes. It is because though the execution parts are finished on different worker node, the master node is responsible for starting, generating, scheduling and monitoring all the comparison tasks.

By using the application interface, users' applications can be deployed in both the master and worker nodes in the back-end computing system. Also, the configuration file can provide the system the needed information about the comparison tasks.

3.3.3 Result Interface

After finishing all the comparison tasks, all the results are collected and provided to the users for further processing. For ATAC problem shown in Chapter 2, all the results are organized as a similarity matrix. In order to make users operate all the results simply and intuitive, in our result interface a matrix data structure is developed to provide different methods for users to operate comparison results.

The matrix data structure provides the following methods for users:

1. Collect all the comparison results.

$$Matrix \qquad Getallresult()$$

   This method is provided for users to collect all the comparison results.

2. Collect the certain comparison result.

$$Result \qquad Getresult(indexx, indexy)$$

   By using this method, users can get certain comparison result by providing the value of index x and y.

3. Collect results from certain row.

$$List < Result > \qquad Getrowresult(indexx)$$

   By using this method, users can get all the comparison results from a certain row by providing the value of index x.

4. Collect results from certain column.

$$List < Result > \qquad Getcolumnresult(indexy)$$

   By using this method, users can get all the comparison results from a certain column by providing the value of index y.

By providing the matrix operation methods for users, it can improve the efficiency of operating all the comparison results based on users' purposes. Users can use these interfaces transfer the comparison results to other analysis tools for further processing.

In this section, three different interfaces are developed for users to operate our distributed computing framework. These interfaces are designed to support users deploy data sets and ATAC applications and collect all the comparison results. No distributed system issues are involved in these operation interfaces, which lowers the study cost of using our distributed computing framework. In the next section, the development of the programming model for ATAC problems and application programming interface (API) are described.

3.4 ATAC Programming Model

As an abstraction of the computation procedures, programming model is designed to handle the distributed system issues such as data distribution, task scheduling and resource management automatically and provide users simple application programming interfaces (APIs) to help implementing computing problems.

In this section, a distributed programming model for ATAC problems is developed. Followed by the design targets, our programming model integrated with the specific data distribution strategy and task scheduling strategy for ATAC problems. In the end, the comparison between our programming model and MapReduce is listed to shown the differences.

3.4.1 ATAC Workflow and Challenges

A general workflow of ATAC problem can be graphically shown as Figure 3.4 :

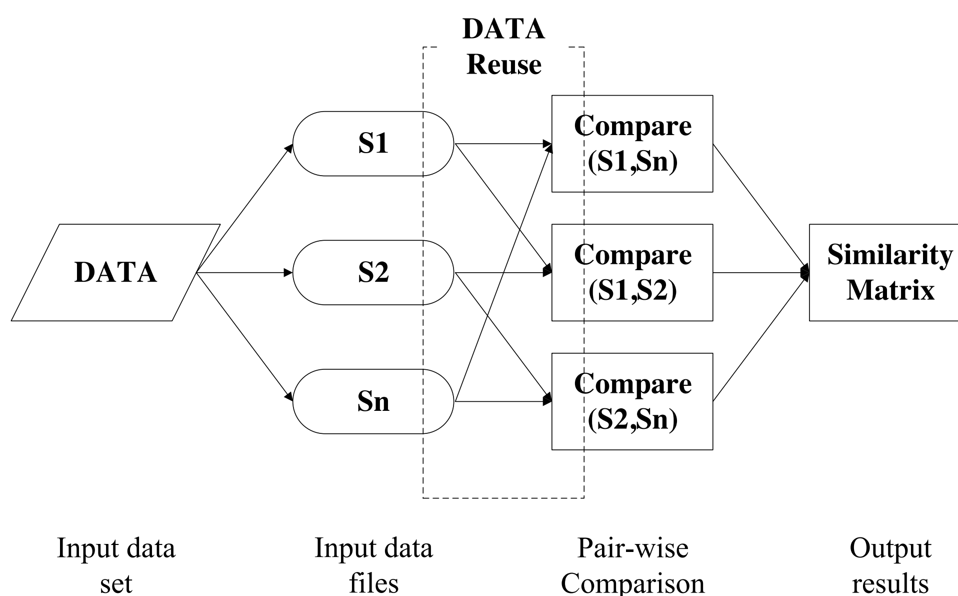


Figure 3.4: Processing steps for solving ATAC problems.

In Figure 3.4, we can see that the pair-wise comparison phase is the key step in processing ATAC problems. Moreover, each comparison operation needs to process two different data files, which means each data file has to be reused many times. In the end, all the results generated from comparison operations need to be collected and output as a similarity matrix.

Though it is straightforward to solve ATAC problems in a centralized way, challenges come

from when distributed processing ATAC problems with big data sets. In a distributed computing system, the following issues need to be considered:

1. Data reuse may cause massive data movement.

   For example in Figure 3.4, if one data file is reused by multiple comparison tasks running in different worker nodes, this data file has to be copied to all the related worker nodes. This indicates both the massive data transmission and huge storage usage caused by numbers of data replications, which can be a serious performance problem in processing large amount of data files.

2. Data location may cause inefficient task scheduling.

   As we mentioned in Chapter 2, data locality is one of the most important principles for scheduling computation tasks. Tasks gain good data locality can process input data from local disk, which greatly reduce the cost spend on remote accessing. While for ATAC problems, data pairs stored in two different worker nodes makes it hard to guarantee good data locality for the related comparison task. In this situation, scheduling the comparison task to any one of the worker nodes cannot avoid remote accessing data.

As we discussed in Chapter 2, MapReduce programming model has been chosen to solve ATAC problems in various application areas. Though the procedures of processing MapReduce application and ATAC application have some similarities, there are huge differences which can significantly affect the system performance. In this part, the challenges of distributed processing ATAC problems are summarized.

Inefficient Workflow Design

To use MapReduce programming model, ATAC applications are implemented by combining multiple Map and Reduce tasks. Though ATAC problems can be processed based on MapReduce programming model in this way, performance degradation is inevitable if we only piece together Map and Reduce tasks to form a workflow without considering the design principles of MapReduce programming model.

1. Modify Map to support binary inputs.

Figure 3.5 shows parts of a modified MapReduce workflow proposed by Chen et al. [2013]. In order to make map task support two different input data splits, FPatMiner tasks are added before Map tasks to pre-process input data splits. By using this method, two different data splits can be processed by the same Map task.

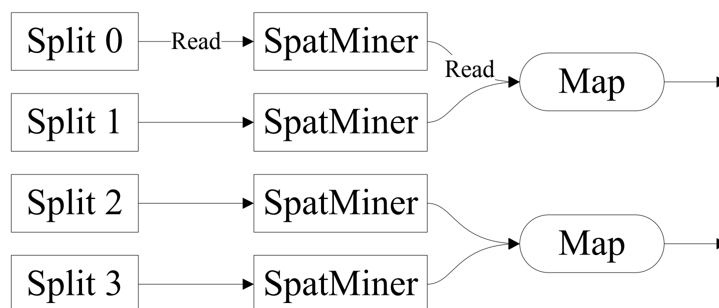


Figure 3.5: Modified Map tasks to support multiple inputs.

However, the principle for designing Map tasks in MapReduce programming model is that each Map task is prefer to be scheduled and executed in the worker node with local input data, which means a good data locality and potential high performance for all the Map tasks. In this case, it is obviously that each Map task needs to read two different data splits, which are usually located in different worker nodes.

2. Modify Reduce to take charge of heavy computations.

   Considering Reduce task can support multiple inputs, many researchers use it to process large data sets collected from different worker nodes. For example in the research made byZheng et al. [2013], scalar multiplication is finished in each Reduce task to process a complete matrix. While in MapReduce programming model, Reduce tasks are designed to sum up intermediate results from Map tasks in remote worker nodes, thus it does not have the qualities such as good data locality, locality-aware scheduling. Therefore, designing Reduce tasks for large data processing is inefficient due to the original design limitations of MapReduce programming model.

   Moreover, some researches modified Reduce tasks and Map tasks as one-to-one relations [Singhal and Guddeti, 2014] or one-to-all relations [Farzanyar and Cercone, 2013] to meet specific requirements. Though these modifications aim to implement complex workflow, extra programming work is brought for developing these enhanced functions, which sacrifices the simplicity of MapReduce programming APIs.

Inefficient Data Distribution and Task Scheduling

As a data parallel processing model, the high performance of MapReduce programming model gains not only because of the simplified programming interfaces, but also the cooperated data distribution and task scheduling strategies. The inner reason that solving ATAC problems by using MapReduce is inefficient is the unsuitable data distribution and task scheduling strategies.

In Hadoop's data distribution strategy, data items are randomly distributed with a fixed number of duplications among all storage nodes. This random data distribution strategy works well for MapReduce programming model due to the following two reasons: 1) Map Task is designed to process single input and 2) Map tasks are scheduled by locality aware strategy.

While for ATAC problems shown in Figure 3.4, due to each comparison task needs to process two different data files, randomly distributing data files cannot promise the pair-wise data co-location. Furthermore, when Map tasks have to process data files stored in two different worker node, the locality aware scheduling strategy is also invalid. Though approaches like Bi-Hadoop added data cache system and enhanced task scheduler to solve these problems, the improvement is still limited by the fundamental system architecture.

In our previous work [Zhang et al., 2014, 2015a,b,c,d], the performance degradation caused by these issues have been fully discussed. In the next section, a programming model designed for general ATAC problems is provided to solve the above challenges.

3.4.2 ATAC Programming Model Design Targets

Based on the challenges discussed in Subsection 3.4.1, our programming model for ATAC problems is designed by considering the following targets:

1. Comparison operation is designed for binary inputs.

   As we discussed in Section 3.4.1, ATAC applications are hard to be implemented by using MapReduce programming model because the multiple inputs are not supported by Map tasks. Hence, in our programming model, a comparison operation with binary inputs is provided for users to easily implement ATAC problems.

2. Good data locality for comparison operations.

Making all the comparison tasks have good data locality is important in achieving overall high performance. In our programming model, the task-oriented data distribution strategy is integrated to distribute data in a pair-wise way. Moreover, the locality aware scheduling is integrated to schedule comparison tasks based on data pairs co-location.

3.4.3 ATAC Programming Model Design

In the following parts, we will explain our programming model from three aspects: programming interface, data distribution and task scheduling.

Programming Interface

One programming interface—Compare is provided by our programming model. Figure 3.6 briefly shows the input and output of the programming interface.

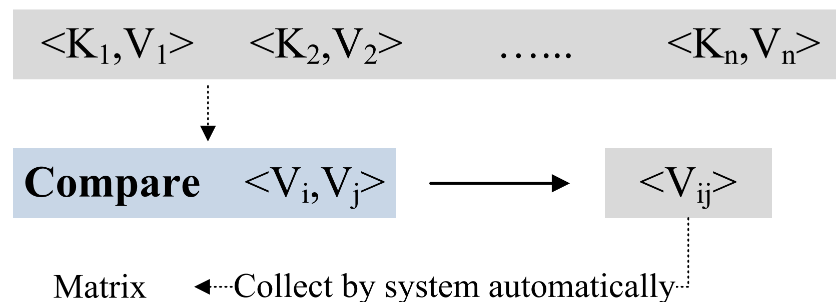


Figure 3.6: The programming interface of our programming model.

As shown in Figure 3.6, all the input are organized as key/value pairs. For instance, K_i and V_i can be the index of file i and the content of file i, respectively.

For the Compare function, it accepts two different values V_i, V_j and outputs one comparison result as V_{ij}. In our programming model, a list of comparison results will be generated from all the comparison tasks.

Then, all the comparison results generated by Compare tasks will be collected automatically by our computing framework through the Collect task. Finally, a similarity matrix is generated to store all the comparison results.

Our Java implementation gives users freedom to define the types of K_i, V_i, V_{ij} and a data structure is also developed to save comparison results as the matrix.

Data Distribution

In our programming model, each worker node only stores a subset of all the data files. As we discussed in Section 3.4.1, data distribution affects the performance of comparison tasks significantly. Our programming model integrates a task-oriented data distribution strategy (TODD), which has the following qualities:

1. Pair-wise data co-location for all the comparison tasks.

   To make all the comparison tasks have good data locality, TODD strategy promises any two different data files can be found in at least one worker node.

2. Static load balancing for the system.

   TODD strategy makes each worker node have numbers of comparison tasks proportional to its processing power. In this way, static system load balancing can be promised to make full use of computing resources.

The TODD strategy that meets the above acquirements has been developed in our previous works to support both homogeneous [Zhang et al., 2014, 2015b,d] and heterogeneous [Zhang et al., 2015a,c] systems. The details of data distribution strategy will be discussed in the following chapters.

Task Scheduling

After distributing all the data sets, comparison tasks need to be scheduled and executed. The task scheduling strategy in our programming model has the following qualities:

1. Locality aware with no data movement.

   In the data distribution phase, the data co-location for all the comparison tasks have already been considered. Hence, our locality aware scheduling strategy is designed to always allocate comparison tasks to a worker node with all the required data files.

2. System load balancing is promised.

   In our programming model, a static scheduling strategy is provided to allocate comparison tasks by accepting the task assignment suggested by our data distribution strategy Zhang

et al. [2015a]. Furthermore, considering the changing computing resources and different execution time for each comparison task, a dynamic scheduling strategy is also provided to support the more flexible scenarios Zhang et al. [2015a]. The details of task scheduling strategies will be discussed in the following chapters.

In the next subsection, we will summarise our programming model by comparing with MapReduce from different aspects.

3.4.4 Comparison with MapReduce Programming Model

One approach for distributed solving ATAC problems is to use MapReduce programming model. Though the workflow of processing MapReduce application and ATAC application looks similar (Figure 3.7), there are huge differences which can significantly affect the system performance.

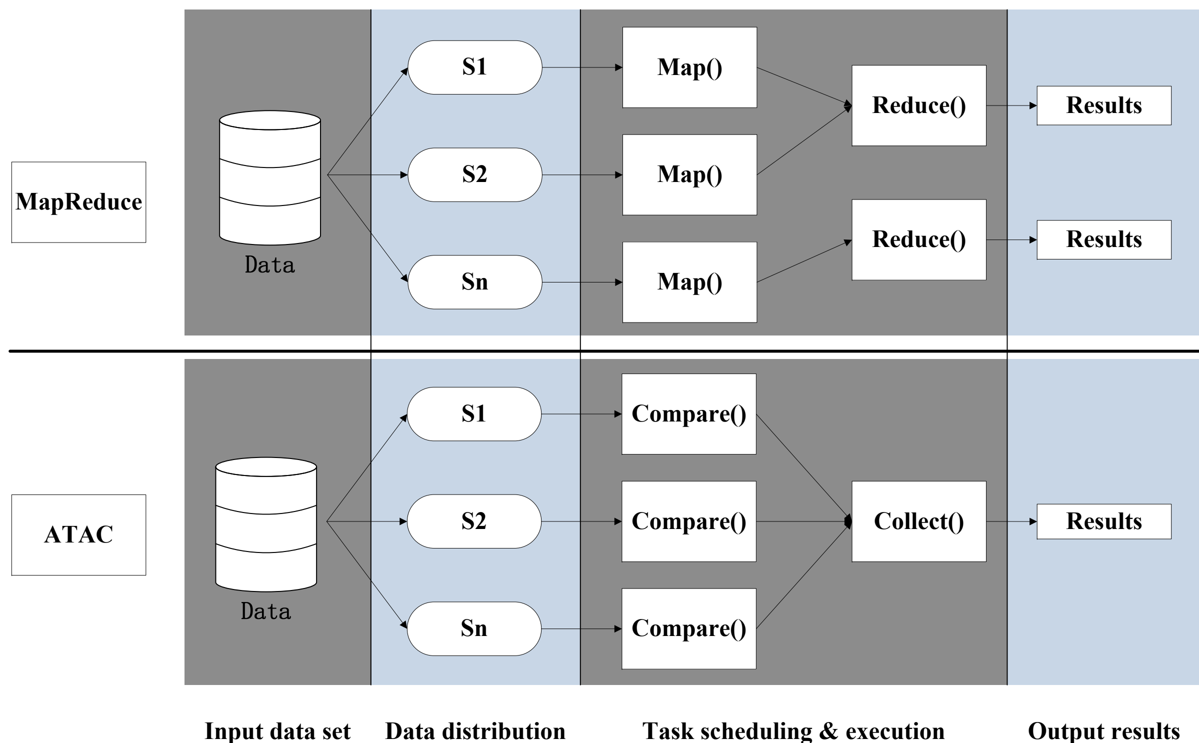


Figure 3.7: Comparison between two programming models.

Here Table 3.4 summarised the different design targets for these two programming models.

Table 3.4: Comparison between our programming model and MapReduce.

	MapReduce	ATAC Programming Model
Data distribution	Data files are randomly distributed to different worker nodes with fixed number of replications.	Data files are pair-wise and task-oriented distributed to different worker nodes.
Data locality	Map task is designed to have good data locality and it processes single input data.	Comparison task is designed to have good data locality and it processes two different input data.
Locality aware scheduling	Map task is scheduled and executed on the worker node where stores the related single input data.	Comparison task is scheduled and executed on the worker node where stores required input data pairs.
Result collect	Reduce task is designed to collect remote intermediate results and generate final output.	Collect task is designed to collect remote comparison results and generate final similarity matrix.

3.5 ATAC Computing Framework Implementation

In this section, after describing the architecture of our implementation system, we will show how to use the front-end interfaces of our distributed computing framework for solving the CVTree problem, which is a typical all-to-all comparison problem widely existed in bioinformatics.

By following the above design targets, the implementation of our programming model for ATAC problems is shown in Figure 3.8 :

3.5.1 ATAC Computing Framework Architecture

In our research, a prototype ATAC computing framework is implemented by integrating different open-source distributed systems as shown in Figure 3.9.

In Figure 3.9, there are the following four layers:

1. Application Layer. The programming interfaces described in Subsubsection 3.4.3 is provided for users to develop ATAC applications.
2. Services Layer. The key components of our computing framework is designed as services layer. By using the scheduling APIs provided by Mesos, a locality-aware task scheduler

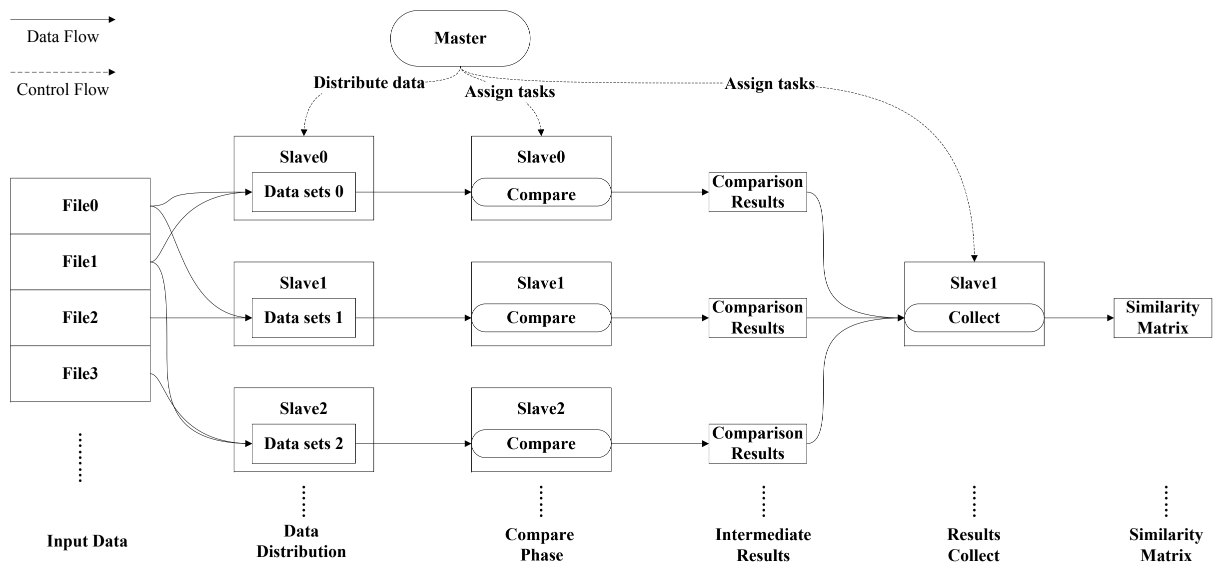


Figure 3.8: Overview of the ATAC programming model implementation.

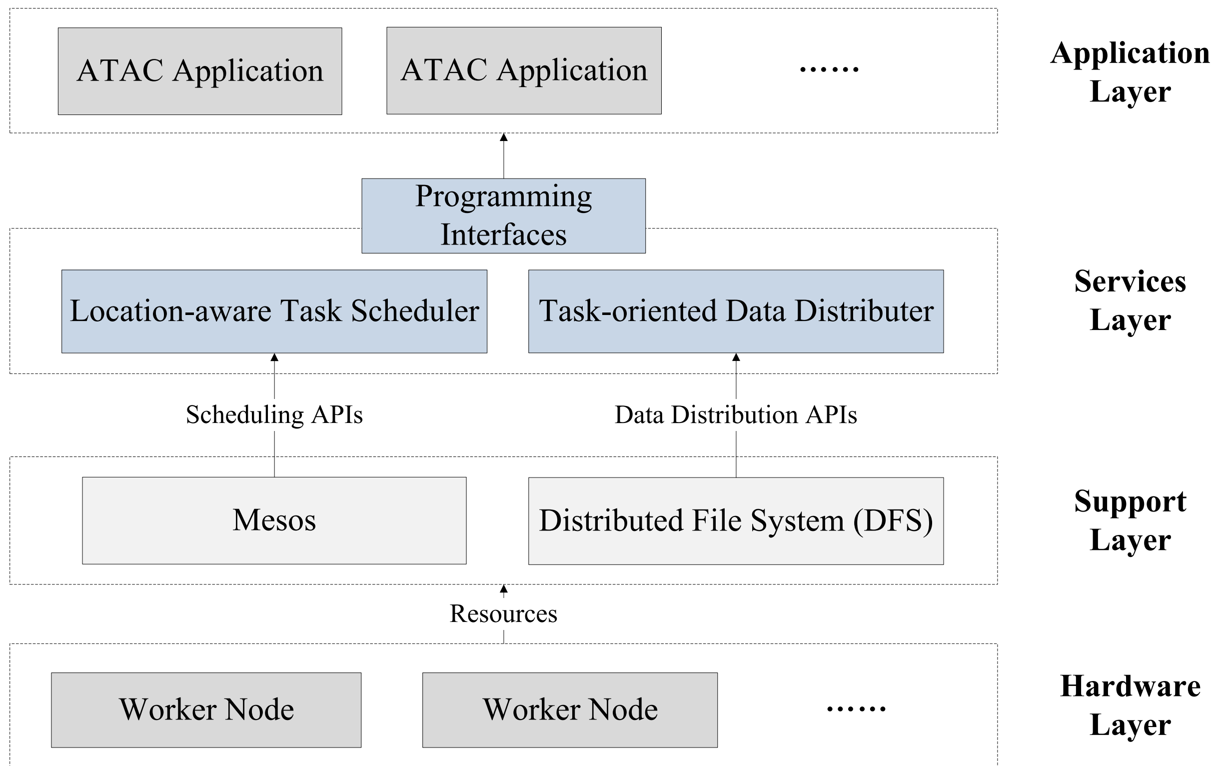


Figure 3.9: ATAC programming model implementation.

which implemented our task scheduling strategy (Subsubsection 3.4.3) is developed. A task-oriented data distributor is developed to implement our data distribution strategy (Subsubsection 3.4.3).

3. Support Layer. In our implementation, Mesos is chosen as a distributed system manager

that provides basic functions such as system resource management, network communication and task status monitoring. Beside this, distributed file systems like Hadoop distributed file system (HDFS) is chosen to provide the functions for data storage and data transmission.

4. Hardware Layer. Our computing framework is designed to run on the distributed systems with distributed storage, which is widely used because of cost effectiveness, high reliability and high scalability.

In the next subsection, a typical ATAC application is used to show the efficiency of using our programming model.

3.5.2 ATAC Computing Framework Execution

The use of our front-end interfaces can be classified into two steps: 1) developing the CVTee application by developers and 2) implementing this application on top of our back-end computing system by users.

The overview of the ATAC computing framework execution is graphically shown in Figure 3.10. In Figure 3.10, to solve an ATAC problem, users and developers only have to use the operation interfaces and programming interfaces provided by our distributed computing framework. The whole framework execution briefly includes the following steps:

1. ATAC application Development.

2. Deployment of related data sets and ATAC applications.

3. Execution of all the computing tasks in the back-end computing system.

To users, after implementing required ATAC applications and providing related data sets, all the inputs will be distributed to different nodes in the system automatically. Then the system will allocated suitable number of comparison tasks to different worker nodes based on the scheduling strategy we developed. Finally the results will return to users for the further analysis.

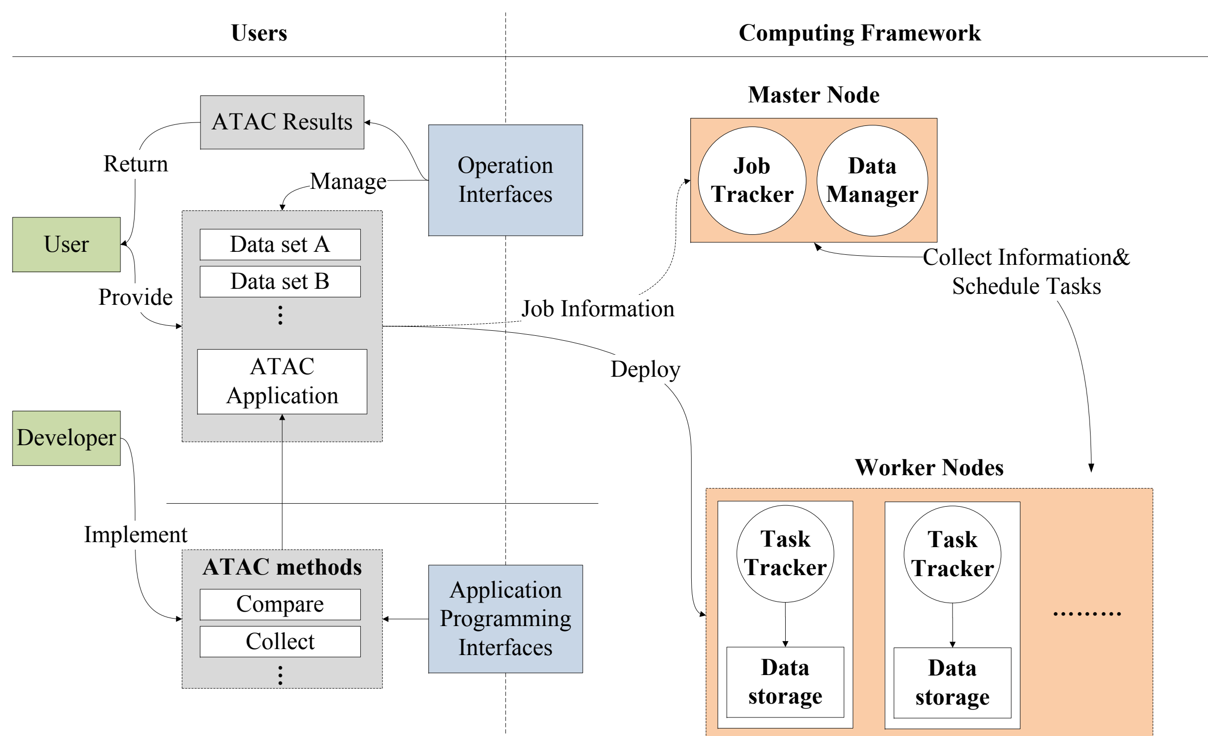


Figure 3.10: Overview of the execution of our computing framework.

Application Development

In this section, the CVTree application is developed by using the programming interfaces provided by our programming model.

To develop CVTree application, as we mentioned in Section 3.4.3, methods Compare and Collect need to be implemented by the developer. The simple codes for CVTree application is shown as follows:

```
1  public class Cvtreesystem extends
       ComputingSystem<String,String,Double>{
2      ......
3
4      public Double Compare(String content,String content){
5
6          Init();
7          InitVectors();
8          StringReader in = new StringReader(content);
9          ......
10         List<Object> intermediateData = new ArrayList<Object>();
11
12
13         counta=getcount(intermediatedataa);
14         countb=getcount(intermediatedatab);
15         ......
16        return correlation / (Math.sqrt(vector_len1) *
             Math.sqrt(vector_len2));}
17
18     public Matrix<String,Double> matrix Collect(){
```

```
19
20      Map<List<String>,Double> resultpairs = matrix.Getallresult();
21      File filename;
22      FileWriter fw = null;
23      ......
24      return resultpairs;}
25    ......}
```

In the CVTree application, we can see that developers only have to define the comparison method between two different input data. How to implement this method to all the input data files among the distributed computing systems will be automatically handles by our computing system.

Data and Application Deployment

After developing the CVTree application, to make it run in our distributed computing framework, the user needs to distribute the data sets and application to the back-end computing system by using the user interfaces developed in Section 3.3.

1. Distribute data sets.

   In this example, we plan to distribute 124 gene files to our distributed computing system. The format of the .FAA file is expressed as follows:

   $> gi|9964581|ref|NP_0 64770.1|$

   $FTKLIKLYCYNTRIDSLKGIENLIKLKEL$

   In order to distribute all the data files, users have to use the front-end data interface by providing the following informations:

   $$Datadistribution(\backslash user\backslash data, \backslash home\backslash CVTree\backslash Data)$$

2. Deploy CVTree application.

   In this step, after providing the CVTree application, users also have to provide a configuration file which includes the name of all the data files need to be processed by the CVTree application. The application interface can be used as follows:

   $$ApplicationDeploy(\backslash user\backslash app, \backslash home\backslash CVTree\backslash App, \backslash user\backslash config\backslash config)$$

After distributing all the data files and deploying the CVTree application, the user can run the CVTree application by inputting the simple commands and monitor the running details for the system graphical interface.

For this example, the final comparison results are stored like this:

Key : $[BAdV_A.faa, AtHV_3.faa]Value$: 6.855431355112651$E - 4$

Key : $[ATV.faa, BAdV_A.faa]Value$: 8.319012406344845$E - 4$

Key : $[BAdV_4.faa, BAdV_A.faa]Value$: 0.016691341041739863

The content of key stores the name of two different input data files and the value represents the comparison result between these two different data files.

In this chapter, we discussed the front-end interfaces provided by our distributed computing framework and a CVTree example was used to show the use of our front-end interfaces for solving a typical all-to-all comparison problem.

3.6 Conclusion

In this chapter, after formally defining the all-to-all comparison problems, the overview of the ATAC distributed computing framework is presented. Front-end interfaces are developed for both users and developers to solve ATAC problems. Simple operation interfaces are provided for users to use the ATAC distributed computing framework without any consideration of back-end implementation issues. Also, a distributed programming model for ATAC problems is designed. By abstracting the basic computation of ATAC computing pattern, this programming model provides powerful and efficient application programming interfaces (APIs) for developers to implement ATAC applications with different domain backgrounds. In the end, CVTree applications in bioinformatics is used to show the using of all the front-end interfaces of the ATAC distributed computing framework.

In Chapter 4, we are going to discuss about the data distribution strategy and static task scheduling strategy integrated in the back-end distributed computing systems.

Chapter 4

Heuristic Data Distribution Strategy for ATAC Problems in Homogeneous Distributed Systems

---

To solve all-to-all comparison problems with big data sets, as we mentioned in Chapter 1, distributing huge amount of data files to the distributed computing system affects the overall computing performance greatly. In this chapter, a heuristic data distribution strategy is developed for processing all-to-all comparison problems in homogeneous distributed computing systems. Beginning with the conclusion of principles for data distribution, the challenges of distributing data files for ATAC problems are discussed from different aspects. After formulating the data distribution problem, a heuristic data distribution algorithm based on greedy idea is provided. The data distribution strategy not only saves storage space and data distribution time but also achieves load balancing and good data locality for all comparison tasks of the all-to-all comparison problems. Based on the results of data distribution, a static task scheduling strategy followed with our data distribution strategy is also provided to allocate comparison tasks in a way that achieving system static load balancing. In the end, different experiments are conducted to demonstrate the effectiveness of the data distribution strategy in homogeneous distributed computing systems.

4.1 Principles for Data Distribution

A typical scenario for using distributed computing systems in all-to-all comparison problems is described as follows and is shown in Figure 4.1. In general, a data manager should manage and distribute all the data to the worker nodes first. Then, computing tasks are generated and

allocated by the job tracker to the worker nodes. Lastly, the computing tasks are executed by the task executor to process related data sets.

It is seen from the work flow in Figure 4.1 that to solve all-to-all comparison problems efficiently both the data distribution and task computation phases need to be improved.

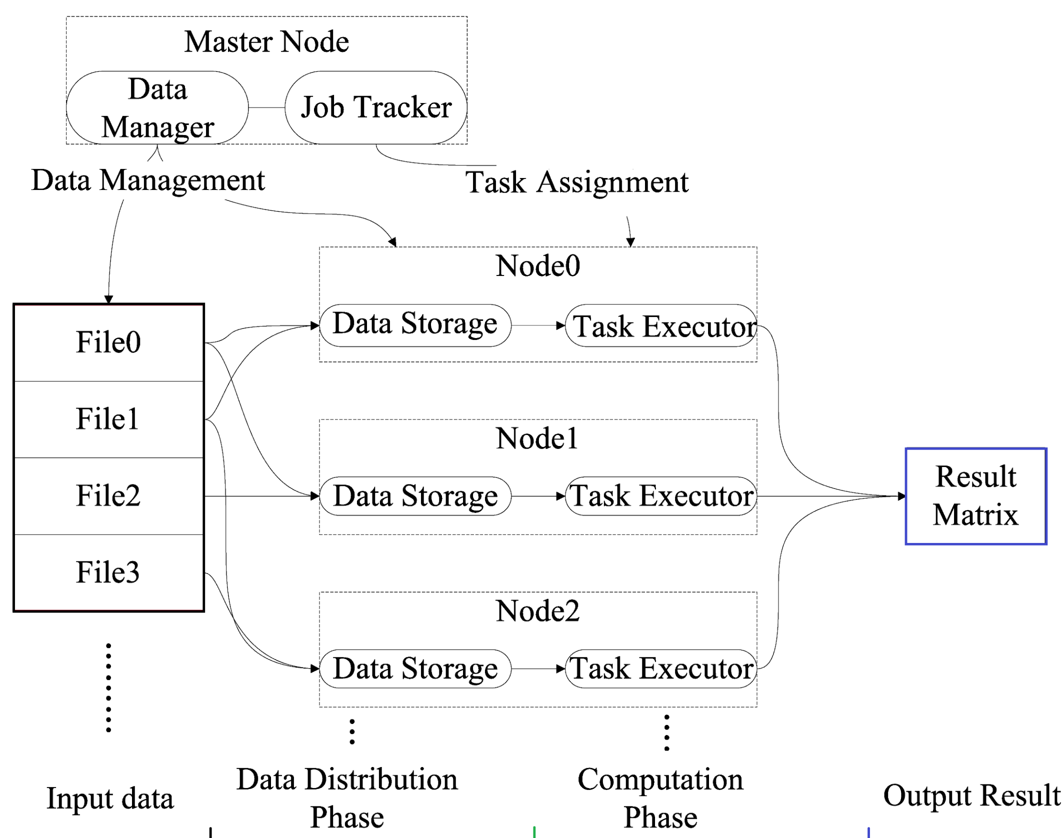


Figure 4.1: General work flow for solving all-to-all comparison problems in distributed computing environments.

Two issues affect the overall computing performance of big data processing in distributed environments. They are data locality and computing task allocation.

- Data locality is a basic principle for processing big data problems. It means that the computing operation is generally more efficient when it is allocated to a worker node near the required data. Computation tasks which need to access remote data sets can be very inefficient due to the heavy network communications and data transmission.

- The distributed computing of big data problems is more efficient when all the worker nodes are allocated suitable numbers of computing tasks which match their processing capability. In that case, all the computing power of the system can be efficiently utilized.

Therefore, a data distribution strategy for distributed computing of all-to-all comparison problems should enable:

1. Good data locality for comparison tasks.

2. Load balancing of comparison tasks through allocation of tasks with good data locality.

These are the basic principles for design of a data distribution strategy in distributed computing of all-to-all comparison problems.

4.2 Challenges of the Data Distribution Problem

As we mentioned in Chapter 2, storing all the data to everywhere and Hadoop data strategy are two widely used data distribution strategies in solving ATAC problems. In this section, the drawbacks of these two data distribution strategies will be deeply discussed.

4.2.1 Issues of Storing All the Data to Everywhere

Storing all the data files to every worker node in the system is a straightforward way in distributing all the data files. For many solutions such as [Heitor and Guilherme, 2005, Mendonca and de Melo, 2013], they all choose to make each worker node store all the required data files. Though in this way high reliability can be achieved by making each data files have the maximum number of copies and any comparison tasks can be allocated to any worker node, this data distribution strategy is not suitable for processing large amount of data files.

Huge Storage Usage

Storing all the data files to all the worker node is a natural idea inherited by the centralized solutions. Distributed systems with this data strategy is straightforward to be designed because this data strategy avoids the issues brought by partitioning data sets and makes the system only have to consider scheduling computation tasks. While for putting all the data files to everywhere, the storing usage can be too huge to afford for processing big data problems.

For example, let us consider M data files have to be processed in a distributed systems with

N worker nodes. In order to store all the data files to each worker node, totally MN data files need to be stored in the system.

Considering in the application areas as we mentioned in Chapter 2, typical all-to-all comparison problems usually need to process large amount of data files. For instance, in one experiment proposed by Moretti et al. [2010], nearly 60000 data file were pair-wise compared on a vary set of 100-200 machines. In this case, at least 6 million data files need to be stored in the system, which will cost much distribution time and huge storage usage.

Wasted Storage Usage

Moreover, storing data in this way can make most of the data files never be used. For all-to-all comparison problems with huge data sets, storing data files just to achieve high reliability can be a overnighting and waste of resources.

Considering an example with 9 data files and 3 worker nodes. Totally 36 comparison tasks need to be executed. Figure 4.2 shows a possible data distribution for these 9 data files. Distributing data files as shown in Figure 4.2 can make each worker node be allocated 12 different comparison tasks. Hence all the comparison tasks can be finished without any further data movement and the system load balancing can also be achieved.

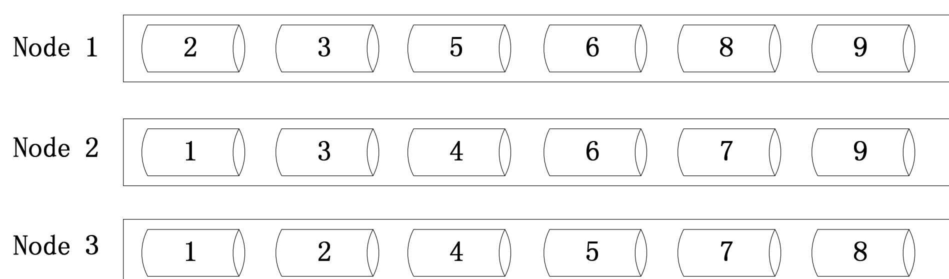


Figure 4.2: A possible situation for processing 9 data files in a 3 worker nodes distributed system.

In this simple case, for 9 data files, we only have to store 6 data files on each worker node. This means it can be 30% storage waste comparing to store 9 data files to each worker node and the result can be much more worse when the number of data files and the number of worker nodes are growing.

4.2.2 Issues of the Hadoop Data Strategy

Hadoop data strategy has been widely used in many researches for solving all-to-all comparison problems. The Hadoop distributed file system (HDFS) provides a strategy to distribute and store big data sets. In the HDFS data strategy, data items are randomly distributed with a fixed number of duplications among all storage nodes. While multiple copies of data items in HDFS enhance the reliability of data storage, the HDFS data strategy is inefficient for all-to-all comparison problems due to its poor data locality, unbalanced task load and big solution space for data distribution [Qiu et al., 2009]. In the following subsections, we will discuss the drawbacks of using Hadoop data strategy for solving ATAC problems.

Poor Data Locality in HDFS

The Hadoop's data strategy is designed for a trade-off between high data reliability and low read/write cost. From this design perspective, it does not consider the data requirements for comparison tasks that follow.

For example, consider a scenario with 6 data items and 4 worker nodes. A possible solution for the Hadoop's data strategy is shown in Figure 4.3. It is seen from Figure 4.3 that although each of the 6 data items has two copies, there is no worker node that contains all the required data for certain comparison tasks, (1,3), (1,4), (2,6), (3,5) and (4,5), indicating poor data locality for these comparison tasks. In this case, runtime data movements among the nodes through network communications cannot be avoided. These will induce runtime storage costs and also affect the overall computing performance of the all-to-all comparison problem. This problem becomes worse when the scale of the computing becomes bigger.

Unbalanced Task Load Resulting from the HDFS

To show an unbalanced task load resulting from the HDFS data strategy, consider a scenario with 4 data items and 3 worker nodes. A possible Hadoop's data distribution solution is depicted in Figure 4.4. Also shown in Figure 4.4 is a possible scheme for allocation of all 6 comparison tasks to the 3 worker nodes. With the HDFS, there is no way to ensure a balanced comparison task allocation, which requires each of the 3 worker nodes be allocated 2 comparison tasks.

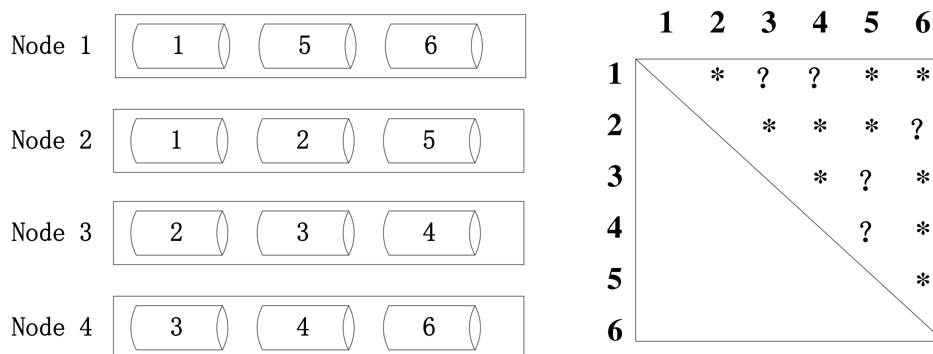


Figure 4.3: A possible situation for Hadoop's data strategy for a scenario with 6 data items and 4 worker nodes.

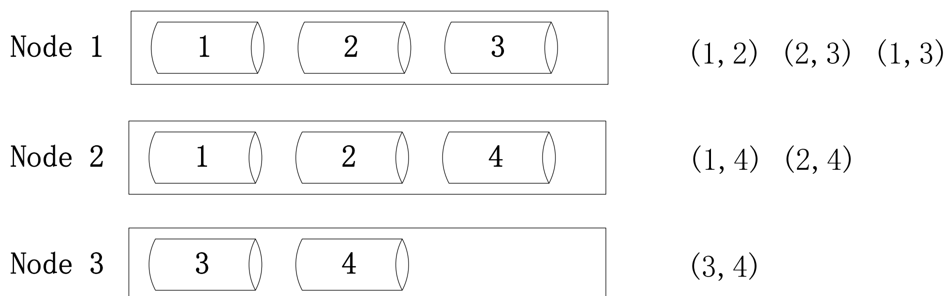


Figure 4.4: Data distribution and comparison task allocation from Hadoop's data strategy for a scenario with 4 data items and 3 worker nodes.

Inefficiency of Increasing File Replications in HDFS

In the HDFS data strategy, the number of data replications can be manually set by users. But there is no guidelines on how this number is set and thus users tend to use the default number of three. Once the number is set, it becomes a constant regardless of the number of machines to be used and the number of data files to be distributed. Moreover, the location of the data file replications is randomly determined in the sense of user's awareness. This causes poor good data locality, leading to poor performance of the distributed computing. Once might imagine that further increasing the data replications may help. However, as will be demonstrated later in the experiments, this is not the case. Unless replicating data files to everywhere, increasing the data replications in HDFS does not improve the data locality performance much but results in significant computation costs due to the increased demand in network communications for data movement at runtime.

Considering a possible HDFS solution with 6 data items and 8 worker nodes shown in Figure

4.5. It is seen from this figure that even if the number of data file replications is increased to 4, poor data locality (Section 4.2.2) and unbalanced task load (Section 4.2.2) still exist. The comparison tasks (1,2), (3,4) and (5,6) cannot be completed without remote access of data at runtime. Data files have to be moved around at runtime to complete the distributed computing of the all-to-all comparison problem.

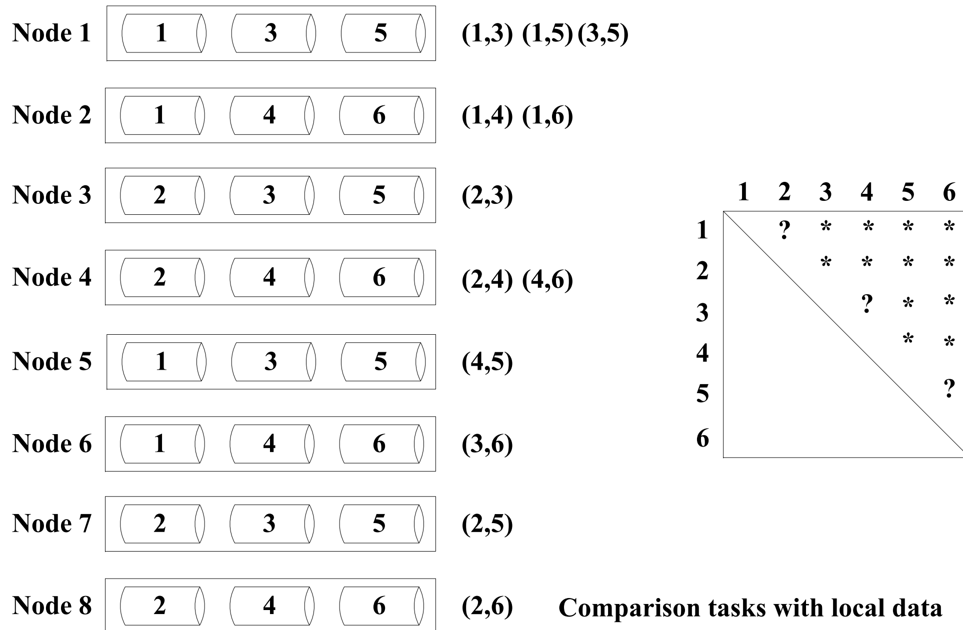


Figure 4.5: Data distribution and comparison task allocation from Hadoop's data strategy for a scenario with 6 data items and 8 worker nodes (The comparison tasks marked with '?' require remote access of data at runtime).

Big Solution Space of the HDFS Data Distribution

The problem of allocating comparison tasks to worker nodes can be treated as a classic problem in combinatorial mathematics: to place M objects into N boxes. This pairwise data distribution problem has the following characteristics:

1. All the comparison tasks are distinguishable. For all-to-all comparison problems, each of the comparison tasks is different and processes a different data pair. This characteristic should be considered when distributing related data sets.
2. All worker nodes are indistinguishable in homogeneous distributed computing systems. In this thesis, the nodes are assumed to have the same processing power and storage space, and thus can be treated as indistinguishable.

It is worth mentioning that our work on homogeneous distributed systems in this thesis can be easily extended to heterogeneous distributed systems. In order not to distract our attention from the development of a fundamental data distribution strategy in this thesis, this issue will be addressed in a subsequent publication.

Consider the data locality requirement mentioned previously. Distributing data pairs also means allocating related comparison tasks. We aim to allocate Q distinguishable comparison tasks to N indistinguishable worker nodes. From combinatorial mathematics, the total number of feasible solutions is expressed by the Stirling number $S_t(Q, N)$ [Gould, 1961]. The Stirling number of the second kind $S_t(Q, N)$ counts the number of ways to partition a set of Q distinguishable elements into N non-empty subsets. It is zero if one or both of Q or N are zero, i.e.,

$$S_t(0, 0) = 1, S_t(Q, 0) = 0, S_t(0, N) = 0.$$

For $N \geq 1$ and $N \geq 1$, we have

$$S_t(Q, N) = NS_t(Q - 1, N) + S_t(Q - 1, N - 1).$$

Let us consider some special cases for the Stirling number. A special case of $N = 2$ has been discussed in our recent conference paper [Zhang et al., 2014]. For another special case of 3 worker nodes ($N = 3$), we have

$$S_t(Q, 3) = \frac{(3^Q - 3 * 2^Q + 3)}{6}. \tag{4.1}$$

$S_t(Q, 3)$ is graphically shown in Figure 4.6. The trend depicted in Figure 4.6 indicates too many possible distribution solutions to access in practice even for a very simple case of 3 worker nodes ($N = 3$). This implies that it is generally impossible to evaluate all possible solutions to find the best answer in a reasonable period of time. Therefore, it is necessary to develop heuristic solutions for data distribution.

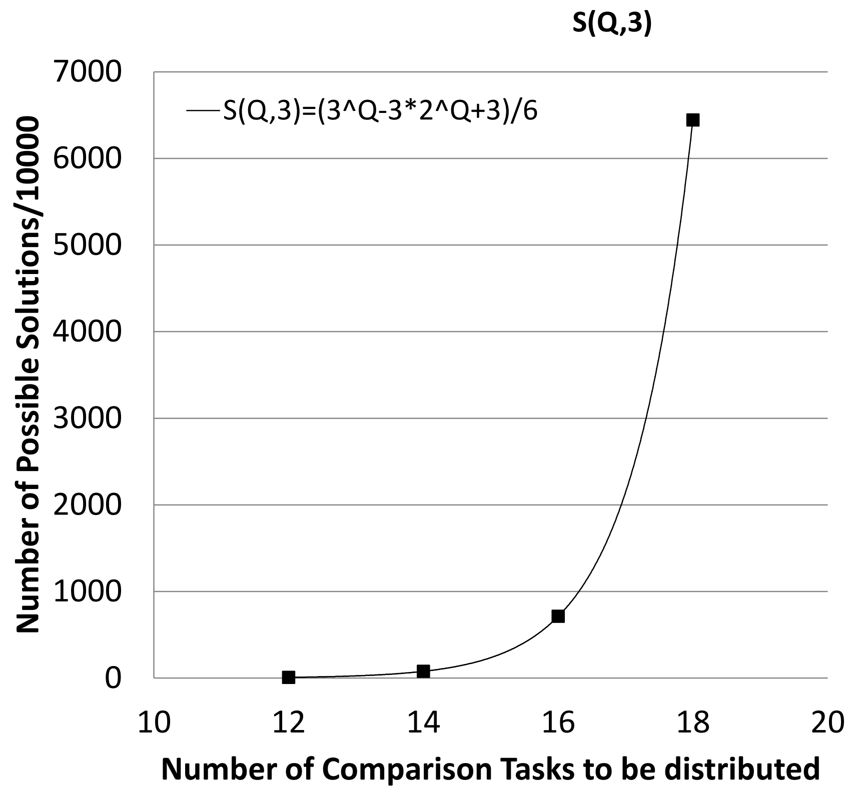


Figure 4.6: The growth trend of the solution space for $S_t(Q, 3)$.

4.3 Formulation for Data Distribution

This section develops requirements of the data distribution strategy. It begins with overall considerations and assumptions. This is followed by formulating reduction of the storage usage and improvement of the computing performance. It ends with an overall optimization problem to specify the requirements for data distribution.

4.3.1 Overall Considerations and Assumptions

To solve all-to-all comparison problems by using the work flow shown in Figure 4.1, the data distribution strategy need to be developed to guide the allocation of all the data files. In the scenario we followed, the data distribution strategy will generate the solution of data distribution first. Then, all the data files are deployed based on the solution provided.

To design the data distribution strategy, the following aspects need to be considered:

- Storage usage of the distributed system. For all-to-all comparison problems with big data

sets, distributing data sets among all worker nodes should consider not only the usage of storage space for each node within its capacity, but also keeping the total time spent on data distribution at an acceptable level.

- Performance of the comparison computation. For distributed computing, it is important to allocate comparison tasks in a way to make full use of all available computing power in the system. Also, good data locality for all comparison tasks can help improve the computing performance greatly. Hence, the data distribution strategy should be designed to allocate data items in a way to improve the performance of comparison computation.

To design a data distribution strategy for homogeneous distributed computing systems, the following assumptions are made in this chapter:

1. All the worker nodes in the distributed system have the same processing power and storage capability.
2. All the data items have the same size.
3. All the comparison tasks have the same execution time.

In this chapter, we are focusing on providing a data distribution strategy designed for homogeneous distributed computing systems and the further consideration of heterogeneous distributed computing systems will be discussed in the following chapters.

Although these assumptions may not necessarily be realistic, they are made for easy understanding of the simplified design of a fundamental data distribution strategy in this chapter. They can be relaxed for more realistic development, while retaining the fundamentals of the data distribution strategy.

In the following, we analyse the requirements for storage usage of the distributed system and performance of the comparison computation, which we mentioned previously.

4.3.2 Reducing the Storage Usage

Many factors contribute to the time spent on data distribution. Given the network bandwidth, network topology and the size of the data items, from the assumptions above, the time for data

distribution is proportional to the number of data items to be distributed. The time for data distribution, T_{data}, can be expressed as:

$$T_{data} \propto \sum_{i=1}^{N}(|D_i|). \tag{4.2}$$

Beside this, as mentioned previously, the storage usage for each of the worker nodes must be within its limitation. Given our assumptions, this can be achieved if all data sets are evenly distributed.

Hence, in the storage usage aspect, the data distribution strategy is designed to meet two purpose:

1. The total number of data files stored in the distributed system need to be reduced.

2. The number of data files stored on each worker node should be reduced.

Now, we consider both data distribution time and storage limitation. Let $|D_i|$ denote the number of files allocated to worker node i. A data distribution strategy is expected to minimize the maximum of $|D_1|, \cdots, |D_N|$, i.e.,

$$\text{Minimize} \max\{|D_1|, |D_2|, ..., |D_N|\}. \tag{4.3}$$

Choosing to minimize the maximum number of data files in the worker nodes has the following benefits: 1) this target makes all the worker nodes have the similar number of data files. In the ideal case, the differences of the number of data files among the nodes are at most one, meaning a balance data storage usage for the distributed systems. 2) Considering the number of comparison tasks can be executed is proportional to the number of data files stored in the worker node, this target also makes all the worker nodes have the similar number of executable comparison tasks.

4.3.3 Improving the Computing Performance

In distributed computing of an all-to-all comparison problem, the overall computation time of the computing tasks executed is determined by the last finished worker node. To complete each

of the comparison tasks, the corresponding worker node has to access and process the required data items.

Let K, $T_{comparion(i)}$ and $T_{accessdata(i)}$ represent the number of comparison tasks allocated to the last finished worker node, the time for comparison operations for task i and the time for accessing the required data for task i, respectively. The total elapsed computation time of the all-to-all comparison problem is then expressed as:

$$T_{task} = \sum_{i=1}^{K} \left(T_{comparison(i)} + T_{accessdata(i)} \right). \tag{4.4}$$

As it has been assumed that all the comparison tasks have the same comparison time C, the computation time T_{task} can be simplified to:

$$T_{task} = CK + \sum_{i=1}^{K} T_{accessdata(i)}. \tag{4.5}$$

From Equation (4.5), our new data distribution strategy minimizes the computation time T_{task} by meeting two constraints: load balancing for comparison tasks on the worker nodes, and good data locality for all pairwise comparison tasks.

For load balancing, the maximum number K of the comparison tasks allocated to the last finished worker nodes can be minimized. Let T_i denote the number of pairwise comparison tasks performed by worker node i. For a distributed system with N worker nodes and M data files, a total number of $M(M-1)/2$ comparison tasks need to be allocated to the work nodes. Minimizing the value of k can be expressed as follows:

$$\forall T_i \in \{T_1, T_2, ..., T_N\}, T_i \leq \left\lceil \frac{M(M-1)}{2N} \right\rceil, \tag{4.6}$$

where $\lceil \cdot \rceil$ is the ceiling function.

Good data locality can also be mathematically formulated. If all required data for a comparison task are stored locally in the node that performs the task, the task will not need to access data remotely through network communications. Good data locality implies a minimized value of $T_{accessdata(i)}$ with its lowest possible value of 0. Let $C(x,y)$, T, T_i and D_i represent the comparison task for data x and data y, the set of all comparison tasks, the set of tasks performed by worker node i, and the data set stored in worker node i, respectively. Good data locality for

all comparison tasks can be expressed as follows:

$$\begin{aligned} &\forall C(x,y) \in T, \exists i \in \{1, \cdots, N\}, \\ &x \in D_i \wedge y \in D_i \wedge C(x,y) \in T_i. \end{aligned} \tag{4.7}$$

In other words, there must be at least one node i capable of performing each comparison $C(x,y)$ with local data only.

4.3.4 Optimization for Data Distribution

Considering both storage usage and computing performance as discussed above, a data distribution strategy is expected to meet the target in Equation (4.3) and also satisfy the constraints in Equations (4.6) and (4.7). When the target in Equation (4.3) is achieved, the storage usage for all work nodes can be reduced greatly, thus reducing the time spent on distributing all data sets (T_{data}). Meeting the constraints in Equations (4.6) and (4.7) means that the overall comparison time T_{task} can be minimized. As a result, the following total time elapsed for data distribution and task execution is effectively reduced:

$$T_{total} = T_{data} + T_{task}. \tag{4.8}$$

Therefore, the data distribution problem can be expressed as constrained optimization problem:

$$\begin{aligned} &\text{Minimize} \max\{|D_1|, |D_2|, ..., |D_N|\} \\ &\text{s.t. Equations (4.6) and (4.7)} are satisfied. \end{aligned} \tag{4.9}$$

As discussed previously, the large number of combinations of data and related comparison tasks makes the above mentioned optimization problem for data distribution difficult to solve in practical applications.

4.3.5 Theoretic Results

Theoretic analysis is conducted to derive a lower bound $\underline{d_{max}}$ for $\max\{|D_1|, \cdots, |D_N|\}$ as well as an insight to the system reliability items of data availability. These theoretic results are

summarized in two theorems.

Theorem 1 *For the constrained optimization problem defined in Equation (4.9) for distributing* M *data files to* N *computing nodes, a solution* $\max\{|D_1|, \cdots, |D_N|\}$ *has a lower bound* $\underline{d_{max}}$ *as:*

$$\max\{|D_1|, |D_2|, ..., |D_N|\} \geq \frac{1}{2}(1 + \frac{\sqrt{4M^2 - 4M + N}}{\sqrt{N}}) \triangleq \underline{d_{max}} \tag{4.10}$$

In the following part, two methods based on combinatorial mathematics and graph theory are used to proof Theorem 1.

Proof For M data files, the total number of comparison tasks in an all-to-all comparison problem is $M(M-1)/2$. Consider the following extreme scenario: if each worker node is allocated no more than $\lceil M(M-1)/(2N) \rceil$ comparison tasks (Equation (4.6)), at least how many data items are needed to complete all comparisons. As each comparison task needs two different data items as characterized in Equation (4.7), we have the following relationship:

$$\binom{\max\{|D_1|, |D_2|, ..., |D_N|\}}{2} = \left\lceil \frac{M(M-1)}{2N} \right\rceil. \tag{4.11}$$

Solving the equation gives the result in Equation (4.10). This completes the proof. ∎

Proof For the complete graph $G = (V, E)$ with M vertices, totally there are $M(M-1)/2$ edges. Considering an extreme situation, the weight of all the edge in every sub-graph is set to 1, which means each edge $e \in E$ is covered by one and only one sub-graph. Therefore, the maximum order of the induced sub-graphs should be the theoretical minimum value.

In this situation, the subgraph with the maximum order is a complete graph with $\lceil M(M-1)/(2N) \rceil$ edges. If we let $v = \max_{i=1,..N} |V_i|$ represent the number of vertices in this sub-graph, U=$\lceil M(M-1)/(2N) \rceil$, based on the relationship of vertices and edges,

$$\binom{v}{2} = U \tag{4.12}$$

After solving the equation, we can get that,

$$\max_{i=1,..N} |V_i| \geq \frac{1}{2}(1 + \frac{\sqrt{4M^2 - 4M + N}}{\sqrt{N}}) \tag{4.13}$$

which is the same as the results in Equation 4.10. This completes the proof. ∎

If a data file has only one copy in a distributed system, it will become inaccessible from anywhere if the node where the data file is stored fails. In this case, the distributed computing system for the all-to-all comparison problem is unreliable. Therefore, it is an essential reliability requirement to have at least two data copies stored at different nodes for each data file. The following theorem shows that this system reliability requirement is guaranteed by the data distribution approach presented in this chapter.

Theorem 2 *For the constrained optimization problem defined in Equation (4.9) for distributing M data files to N computing nodes, the distribution strategy presented in this chapter gives a solution that promises at least two data copies stored in different nodes for each data file.*

Proof If a data item stored in one worker node is not duplicated on another node, in order to meet the constraint in Equation (4.7), all the other data items have to be stored in the same worker node. This implies that the worker node stores all data items. Furthermore, if one node store all data items, based on Equation (4.3), all the other nodes should store all data items, which means all the data is distributed to everywhere for no optimization. Hence, for any optimization results generated, at least two copies stored in different nodes are guaranteed for each data file. This completes the proof. ∎

4.3.6 Special Case Analyse

In this subsection, we start to analyse the solution of this data distribution problem from some special cases. We use $Data(M, N)$ to represent the data distribution problem with M data items and N worker nodes.

To analysis the following cases, let $D = \{d_i | i = 1, 2, ...M\}$ represent all the data needs to be distributed, $\max$ presents the maximum number of data stored among all the nodes.

Data(M,2)

$$\max\{|D_1|, |D_2|\} = M \tag{4.14}$$

Proof Considering that if worker node 1 does not store data item d_i, all the comparison tasks related to d_i must be executed by worker node 2, which means worker node 2 should store data item d_i and all the other data items. Hence, worker node 2 must store all the data items. ∎

Data(M,3)

$$\max\{|D_1|, |D_2|, |D_3|\} \geq \left\lceil \frac{2}{3}M \right\rceil \tag{4.15}$$

Proof Assume all the worker nodes store less than $\lceil \frac{2}{3}M \rceil$ numbers of data items. Considering any two of them, for example worker node 1 and 2. The number of same data items for these two node should be $|D_1 \cap D_2|$. It is obviously that $|D_1 \cap D_2| < \lceil \frac{1}{3}M \rceil$. In this situation, the third worker node must execute all the comparison tasks between data items in $D - D_1$ and $D - D_2$. We can see that $|(D - D_1) \cup (D - D_2)| > \lceil \frac{2}{3}M \rceil$, which is not meet our assumption. Therefore, all the three worker nodes should store at least $\lceil \frac{2}{3}M \rceil$ numbers of data items. ∎

Data(M,M(M-1)/2)

$$\max\{|D_1|, |D_2|, ..., |D_N|\} = 2 \tag{4.16}$$

Proof When the number of worker nodes just equals to the number of comparison tasks, it is clearly that each worker nodes should just be allocated two different data items. ∎

In the next section, a heuristic algorithm will be developed to solve general cases for the data distribution problem.

4.4 Heuristic Data Distribution Strategy with Greedy Idea

This section starts with discussions on heuristic rules for data distribution. Then, our data distribution algorithm is presented with detailed steps. Our data distribution strategy is further analysed through an example.

4.4.1 Heuristic Rules for Data Distribution

A way to derive a feasible solution to the distribution problem formulated in Equation (4.9) is to meet the constraints in Equations (4.6) and (4.7). It is seen from the constraint in Equation (4.7) that if we can determine the location for a specific comparison task $C(x, y)$, the location of the required data x and y can also be determined. Thus, we will allocate all comparison tasks to the worker nodes in a way to meet the constraints in Equations (4.6) and (4.7). From this task

allocation, a feasible solution to the data distribution problem is also obtained.

Consider the scenario shown in Figure 4.7. The right column of the comparison matrix in the figure shows additional comparison tasks that can be allocated to a specific node k when a new data file d is distributed to a node with p data files already stored. Therefore, if we can allocate as many comparison tasks as possible to node k for each new data file d, the total number of data files needing to be distributed can be minimized.

	1	2	3	4	...	...	p	d
1		*	*	*	*	*	*	*
2			*	*	*	*	*	*
3				*	*	*	*	*
4					*	*	*	*
⋮						*	*	*
⋮							*	*
p								*
d								

Figure 4.7: Additional comparison tasks introduced by adding new data d to node k.

The additional comparison tasks introduced by adding new data d to node k include those that have never been allocated before (marked by circles in Figure 4.7) and those that have already been allocated (marked by triangles in Figure 4.7). The related rules for distributing data are developed as follows:

Rule 1: For those comparison tasks that have never been allocated before, a data distribution strategy can be designed to allocate as many of these tasks as possible to node k by following Constraint (4.6).

Rule 2: For those comparison tasks that have already been allocated, the data distribution strategy can be designed to re-allocate each of these tasks by following Constraint (4.6). For instance, if a comparison task t has already been allocated to node q, the strategy compares the numbers of allocated comparison tasks between node k and node q. If node k has fewer comparison tasks, task t is re-allocated to node k.

From these heuristic rules, an algorithm with detailed steps can be developed for actual data distribution.

4.4.2 Data Distribution Algorithm

Our heuristic and task-driven data distribution algorithm is described in detail in the following steps:

1. Find all unallocated pairwise comparison tasks.
2. Find all data files needed for these unallocated pairwise comparison tasks. Put these data files in set I, which is initially empty.
3. From the set I, find the data file that is needed by the greatest number of the unallocated comparison tasks. Let d denote this data file.
4. Choose a set of storage nodes that
   - do not have the data file d;
   - have been allocated the least number of pairwise comparison tasks; and
   - have stored the least number of data files.

   Let C denote this set of storage nodes.
5. Check Rule 1 in Subsection 4.4.1 for all nodes in set C. If none of the nodes meet the constraint in Equation (4.6), remove this data file d from set I and go back to Step 3.
6. Find a node k in set C such that the node is empty or can be allocated the largest number of new comparison tasks that are introduced by adding the data file d and have not been allocated before. Distribute data file d to this node k.
7. For comparison tasks that are introduced by adding data file d in Step 6 and have already been allocated to other nodes before, use Rule 2 in Subsection 4.4.1 to re-allocate these tasks.
8. Repeat Steps 1 to 7 until all pairwise comparison tasks are allocated to the worker nodes.

This heuristic data distribution algorithm contributes to the minimization problem in Equation (4.9). Allocating all comparison tasks with as few data items as possible reduces the total number of data items to be distributed. Evenly distributing data among all worker nodes helps meet the storage limitation for each node. If all nodes have the same or a similar number

of comparison tasks, the requirement for load balancing in Equation (4.6) as a constraint in Optimization (4.9) can be easily satisfied.

4.4.3 Analysis of the Data Distribution Strategy

To analyze the data distribution strategy, consider an example with 6 data files $\{0, 1, 2, 3, 4, 5\}$ and 4 worker nodes $\{A, B, C, D\}$. Our data distribution algorithm presented above gives the solution shown in Table 4.1.

Table 4.1: Distribution of 6 data files to 4 worker nodes.

Node	Distributed data files	Allocated comparison tasks
A	0, 2, 3, 4	(0, 3) (0, 4) (2, 4) (3, 4)
B	0, 1, 4, 5	(0, 1) (1, 4) (1, 5) (4, 5)
C	0, 2, 3, 5	(0, 2) (0, 5) (2, 5) (3, 5)
D	1, 2, 3	(1, 2) (1, 3) (2, 3)

Driven by comparison task allocation, the data distribution algorithm not only allocates data files to the worker nodes, but also provide comparison task assignments corresponding to the data file allocation. As a result, each of the worker node is allocated data files and comparison tasks.

For data allocation, each of the worker nodes stores part of the whole data set. In comparison with the scenario in which the whole data set is stored everywhere, our data distribution algorithm reduces the size of data stored in each node greatly. In the specific example considered here, each of the worker nodes stores no more than 4 data items.

For comparison tasks assignment, each of the worker nodes is assigned a similar and thus a balanced number of comparison tasks. This fulfils the requirement for load balancing among all worker nodes in the distributed system. In this specific example considered here, the total 15 comparison tasks generated from 6 data files need to be executed by 4 worker nodes. Three nodes get 4 tasks and the remaining node gets 3 tasks, indicating a well-balanced task allocation.

Allocation of data files and comparison tasks at the same time also ensures good data locality for the comparison tasks. This is clearly shown in Table 4.1 for the specific example considered here. No runtime data movements are required from one node to another through network communications during the execution of the comparison tasks. This eliminates any runtime

network communications cost, which is inevitable in a Hadoop based system.

4.5 Experiments

This section presents experiments that demonstrate the effectiveness of our data distribution strategy. It includes both simulation studies and experiments in a real distributed computing system. The section begins with development of evaluation criteria. Then, it evaluates the behaviour and performance of our data distribution strategy against these criteria.

4.5.1 Evaluation Criteria and Experimental Design

Three criteria were used to evaluate our data distribution strategy: storage usage, task execution performance and scalability.

Storage Saving is one of the objectives of our data distribution strategy. It is measured in this chapter by a group of experiments with different numbers of storage nodes. The storage saving behaviour of our data distribution strategy is also compared with that of Hadoop's data distribution strategy.

Data Locality reflects the quality of data distribution and is an important indicator of the computing performance. As mentioned in Subsection 4.1 and Equation (4.9), an all-to-all comparison task with good data locality means it can access all the required data pairs locally. Considering comparison tasks are allocated based on the data distribution, the number of comparison tasks with good data locality can be measured after the data distribution. To show the different levels of data locality between our data strategy and the Hadoop one, different numbers of worker nodes and Hadoop data files replications are tested in the experiments.

Execution Performance, characterized by the total execution time for processing an all-to-all comparison problem, is the ultimate goal we aim to improve subject to the storage usage and other constraints. As discussed in Section 4.3, the total execution time (T_{total}) includes the time for data distribution (T_{data}) and the time for processing comparison tasks (T_{task}). All these time metrics were measured from computing of all-to-all problems on real distributed systems. Comparisons were made on these performance metrics for our data distribution strategy and Hadoop's.

Scalability is significant for large-scale distributed computing of all-to-all comparison problems with big data sets. Various scenarios were investigated with a change in the number of processors on the worker nodes in the same experimental environment. This demonstrates the scalability of our distributed computing framework. As all data and related comparison tasks are allocated using our data distribution strategy, this also illustrates the scalability of our data distribution strategy.

4.5.2 Storage Saving

Consider a scenario with 256 data files and a set of storage nodes with the number of nodes ranging between 1 and 64. From the optimization problem in Equation (4.9) for data distribution, the maximum value of the numbers of data items distributed to the worker nodes is used to characterize the storage usage from our data distribution strategy.

The first group of experiments compares our data distribution strategy with Hadoop's one with the number of data duplications in Hadoop being set to be the default value of 3. The experimental results are tabulated in Table 4.2, which shows storage usage, storage saving and data locality for both our and Hadoop's data distribution strategies. The storage saving is calculated against the storage space required when distributing all data files to every node as many existing approaches do.

Table 4.2: Storage and data locality of our approach and Hadoop with three data replications for $M = 256$ files under different numbers of nodes (N).

	N	4	8	16	32	64
$\max\{\lvert D_1, \cdots, \lvert D_N\rvert\}$:						
	$\underline{d_{max}}$ (Thm.1)	129	91	65	46	33
	This paper	192	152	117	85	59
	Hadoop(3)	192	96	48	24	12
Storage saving (%):	This paper	25	41	54	67	77
	Hadoop(3)	25	63	81	91	95
Data locality (%):	This paper	100	100	100	100	100
	Hadoop(3)	56	48	28	14	7

In comparison with the the data distribution strategy to distribute all data to all nodes, it is seen from Table 4.2 that both our data distribution strategy and the Hadoop one have significant storage savings for large-scale all-to-all comparison problems while the Hadoop one saves even

more space. This implies less data distribution time, especially when the number of storage nodes becomes very big. For example, for a distributed system with 64 worker nodes, the storage saving reaches as much as over three quarters (77%) from our data distribution strategy and even as high as 95% from the Hadoop one.

Though with a lower storage saving in storage space, our data distribution strategy achieves 100% data locality for all computing tasks. This is clearly shown in Table 4.2. In comparison, the higher storage savings from Hadoop's data distribution are achieved with a significant sacrifice of data locality. For example, for a distributed system with 64 data nodes, the data locality from Hadoop's data distribution is as low as 7% compared to 100% from our data distribution strategy. Good data locality is particularly important for large-scale all-to-all comparison problems. It will reduce the data movements among worker nodes at runtime through network communications for comparison task execution. Thus, it will benefit the overall computing performance of the all-to-all comparison problem.

One may argue that manually increasing the number of data replications may solve the data locality problem in Hadoop's data distribution. However, there are no guidelines on how this number is set for an all-to-all comparison problem in a given distributed environment. Also, once set, this number becomes constant in the deployed environment, leading to inflexibility for a range of other all-to-all comparison problems. Furthermore, even if this number can be manually tuned every time, it does not fundamentally solve the data locality problem. In comparison, our data distribution strategy automatically determines the number of data replications with 100% data locality. To support these claims, the second group of experiments are conducted, in which we manually tune the number of data replications for Hadoop's data distribution strategy to the value that gives a similar maximum number of data files on a node to that of our distribution strategy. Therefore, a data file is replicated 6, 9, 12 and 15 times for a distributed system with 8, 16, 32 and 64 data nodes, respectively, as shown in Table 4.3. With these manually settings, the experimental results in Table 4.3 show that our data strategy behaves with better storage saving performance than the Hadoop one. With more data replications than our data stretagy, the Hadoop one still demonstrates very poor data locality. For example, for 64 data nodes, Hadoop's data distribution only achieves 26% data locality compared to 100% from our data locality.

Table 4.3: Storage and data locality of our approach and Hadoop(variable x) for $M = 256$ files under different numbers of nodes (N), where the number (x) of data replications for Hadoop has to be tuned manually for each case to achieve a similar maximum number of files on a node.

	N	4	8	16	32	64
	Setting of x in Hadoop(x)	3	6	9	12	15
$\max\{\lvert D_1, \cdots, \lvert D_N\rvert\}$:						
	$\underline{d_{max}}$ (Thm.1)	129	91	65	46	33
	This paper	192	152	117	85	59
	Hadoop(x)	192	192	144	96	60
Storage saving (%):	This paper	25	41	54	67	77
	Hadoop(x)	25	25	44	64	77
Data locality (%):	This paper	100	100	100	100	100
	Hadoop(x)	56	52	38	20	26

4.5.3 Execution Performance

Total execution time (T_{total}) is used to measure the execution performance of processing all-to-all comparison problems. As discussed previously, the total execution time includes the time for data distribution and the time for comparison computations. All these time metrics were evaluated in our experiments, and were compared with those from Hadoop-based data distribution and distributed computing.

The settings for our experiments were as follows:

- The distributed computing system. A homogeneous Linux cluster was built with 9 servers, which all run 64-bit Redhat Enterprise Linux. Among the five servers, one node acts as the master node and the remaining eight are worker nodes. All 9 worker nodes use Intel(R) Xeon E5-2609 and 64GB memory.

- Experimental application. As a typical all-to-all comparison problem in bioinformatics [Hao et al., 2003], the CVTree problem was chosen for our experiments. The computation of the CVTree problem has been recently investigated for single computer platforms [Krishnajith et al., 2013, 2014]. It is further studied in this chapter in a distributed computing environment. The problem was re-programmed in our experiments by using the Application Programming Interfaces (APIs) provided by our distributed computing framework presented in Chapter 3. For comparison, a sequential version of the CVTree

program was also developed for our experiments and the details of the CVTree program was also shown in Chapter 3.

- Experimental data.

  A set of dsDNA files from the National Center for Biotechnology Information (NCBI) [NCBI, 1988] is chosen as the input data. The size for each data file is around 150MB and over 20GB data in total is used in the experiments.

- Experimental cases.

  In this experiment, to show that arbitrarily increasing the number of data replications is inefficient in achieving high computing performance for all-to-all comparison problems (discussed in Subsection 4.5.2), the Hadoop data strategy with different data replication numbers are used to compare with our approach. Table 4.4 clearly shows that when each data has 4 copies for Hadoop's data distribution (Hadoop(4)), the Hadoop data strategy distributes more data files to each worker node that our approach.

Table 4.4: The CVTree problem for $N = 8$ nodes and different numbers of input data files (M).

M	$\max\{\lvert D_1\rvert, \cdots, \lvert D_8\rvert\}$			
	$\underline{d_{max}}$	This chapter	Hadoop(3)	Hadoop(4)
93	34	53	35	59
109	39	63	41	69
124	45	71	47	78

Let us consider the execution time performance of distributed computing and data distribution. For a given all-to-all comparison problem with 3 different sizes of data sets, both the data distribution time (T_{data}) and computation time (T_{task}) were measured, respectively. Adding up these two time measurements gives the total execution time for the all-to-all comparison problem.

For comparison of the execution time performance between our approach presented and Hadoop-based approaches, it was clear that the Hadoop MapReduce computing framework is not suitable for supporting the computation pattern of all-to-all comparisons problems directly. Therefore, the distributed computing framework presented in Chapter 3 for all-to-all comparison problems was integrated with the Hadoop's data distribution strategy in our experiments when the execution time performance was evaluated for Hadoop-based distributed computing.

Figure 4.8 shows the total execution time T_{total} for three different data distribution strategies: ours, Hadoop(3) and Hadoop(4), under different M values. For each of the bar plot which shows T_{total}, the upper part represents T_{data}, and the upper part shows T_{task}.

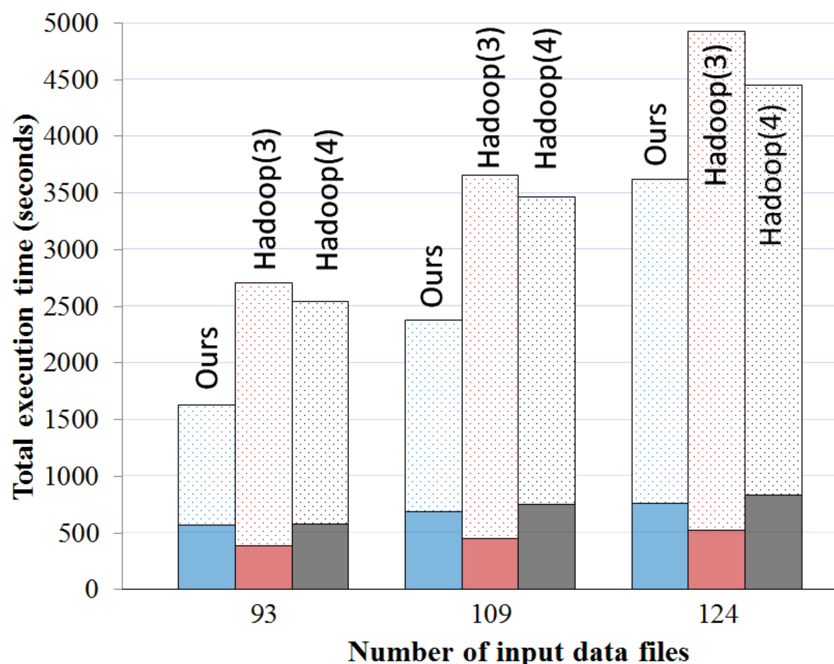


Figure 4.8: Comparisons of the T_{total} performance between our strategy and Hadoop's strategies (lower solid-filled part of the bars: T_{data}; upper dot-filled part of the bars: T_{task}).

It is clearly seen from Figure 4.8 that our data distribution strategy achieves much better performance than the Hadoop one. It also confirms that for Hadoop's data distribution strategy, simply increasing the number of data replications from 3 to 4, which generates more data files on a node than our approach, does not improve much the total execution time performance. this is due to the poor data locality from the Hadoop data distribution and thus requires movement of a huge amount of data at runtime.

To show the good load balancing from our data distribution strategy, Figure 4.9 depicts T_{task} performance measurements for each of the eight worker nodes under different M values. It is seen from the figure that for the same M value,T_{task} for each of the worker nodes is very similar and well within the load balancing requirement from Equation (4.6). The balanced tasks on each node all use local data without the need for data movements among the nodes through network communications.

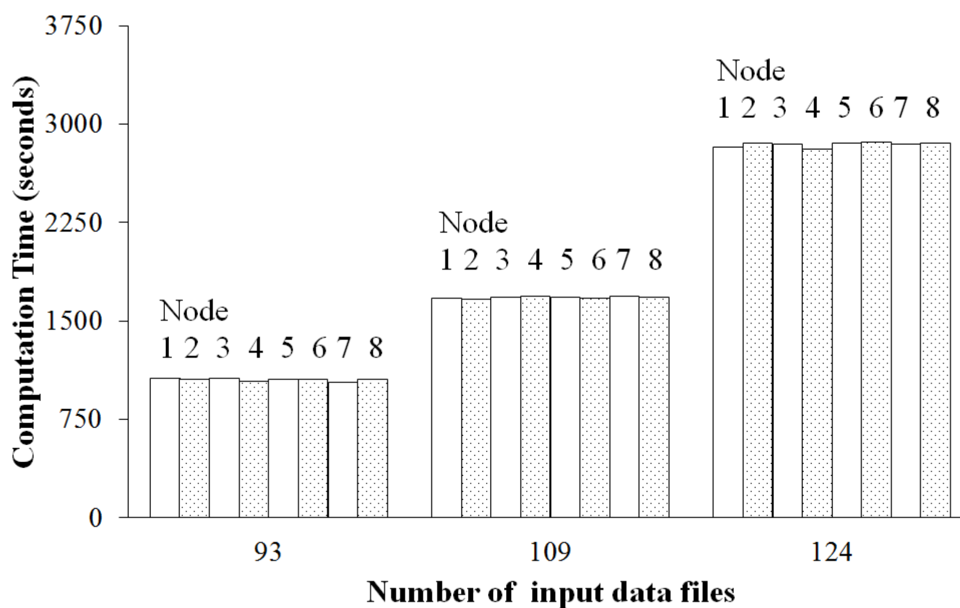


Figure 4.9: T_{task} performance from our data distribution strategy for each of the worker nodes under different M values.

4.5.4 Scalability

Scalability characterizes the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth. To support processing all-to-all comparison problems with big data sets, scalability is an important ability for our data distribution strategy. It is evaluated in the following experiments by using the speed-up metric.

Let $time(n, x)$ denote the time required by an n-processor system to execute a program to solve a problem of size x. Then, $time(1, x)$ is the time required by a sequential version of the program. Speed-up is measured as:

$$Speedup(n, x) = \frac{time(1, x)}{time(n, x)}. \tag{4.17}$$

In general, if communication overhead, load imbalance and extra computation are not considered [Li et al., 1999], a system can achieve a linear speed-up with the increase of the number of processors. Shown in Figure 4.10 for up to eight worker nodes (plus a manager node) in our lab, this linear speed-up dotted line can be considered as an ideal speed-up.

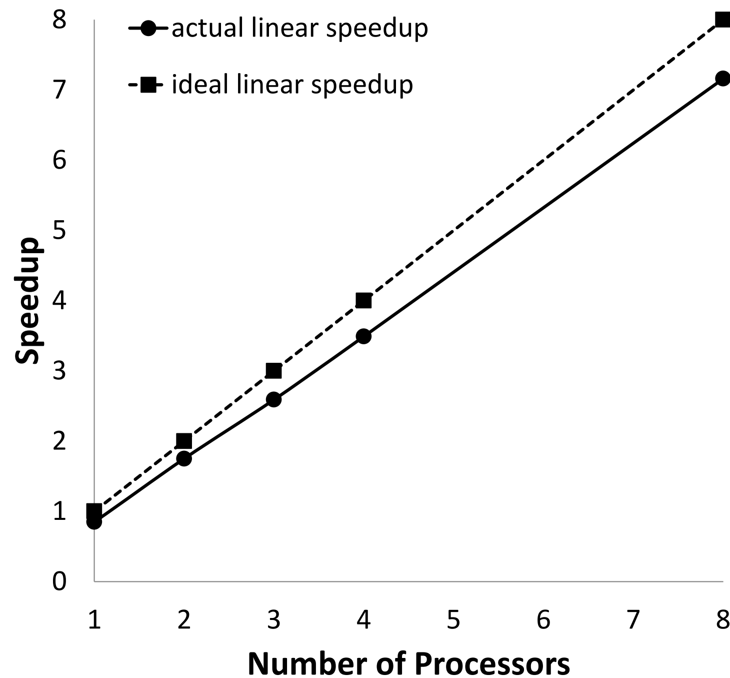


Figure 4.10: Speed-up achieved by the data distribution strategy in this chapter.

Also shown in Figure 4.10 is the actual speed-up achieved from our data distribution strategy. It shows that with the increase of the number of processors, our data distribution strategy behaves with a linear speed-up. This implies good scalability of the overall distributed computation. It is worth mentioning that although all-to-all comparison problems incur inevitable costs in network communications, extra memory demand and disk accesses, the data distribution strategy presented in this chapter can achieve about 89.5% of the performance capacity of the ideal linear speed-up. The is measured by $7.16/8 = 89.5\%$ from the results shown in Figure 4.10.

4.6 Conclusion

In this chapter, to address distributed computation of large-scale all-to-all comparison problems with big data, a scalable and efficient data distribution strategy has been presented. Driven by comparison task allocation, it is designed to minimize and balance storage usage in the distributed worker nodes while still maintaining load balancing and good data locality for all comparison tasks in homogeneous distributed systems. Followed by the data distribution strategy, a static task scheduling strategy is also developed for allocating comparison tasks with good data locality and system load balancing. Different Experiments are used to show the high

performance and scalability of our strategies in solving ATAC problems.

In the next chapter, a metaheuristic data distribution strategy for homogeneous distributed computing systems will be developed to solve the same data distribution problems by using a different method.

Chapter 5

MetaHeuristic Data Distribution Strategy for ATAC Problems in Homogeneous Distributed Systems

---

In the previous chapter, in order to solve the data distribution for ATAC problems, a data distribution strategy with greedy idea has been provided. Though the experiments show the high computing performance for the heuristic solution, the greedy idea still has some fundamental limitations in solving optimization problems. Hence, in this chapter, a metaheuristic data distribution strategy is developed for solving the same data distribution problem in homogeneous distributed computing systems. Considering from different aspects, the challenges of data distribution are further discussed. After designing the metaheuristic data distribution strategy based on Simulated Annealing (SA), experiments show the performance improvement by both comparing Hadoop's data distribution strategy and our previous heuristic data distribution strategy. The scalability of our data distribution strategy is also shown in the end.

5.1 Problem Statement and Challenges

An ATAC problem is a specific Cartesian product of a data set. Let A, A_i, $C(A_i, A_j)$, $M[i, j]$ represent the set of input data items, a single data item in set A, the comparison operation between data items A_i and A_j, and an output similarity matrix element, respectively. The ATAC

problem is to calculate

$$M[i,j] = C(A_i, A_j) \quad \text{for} \quad i,j = 1,2,\ldots,|A| \ . \tag{5.1}$$

For distributed processing of ATAC problems, both data set A and all comparison tasks $C(A_i, A_j)$ need to be distributed to different worker nodes. While different strategies have been developed previously, performance issues still exist.

5.1.1 Task Balancing Causes Data Storage Issues

Comparison tasks are usually allocated by rows or columns [Moretti et al., 2010, Pedersen et al., 2015]. Though load balancing is considered, unoptimized data distribution causes severe data imbalances and high storage usage. Consider an example of 6 data items and 3 nodes. The workload is divided by the rows. The result in Figure 5.1 shows that although each of the three nodes has 5 different comparison tasks, the data files are stored in inefficiently. Node 1 has to store copies of all the data items, but this should be avoided when for data-intensive computing. Moreover, three worker nodes have 6, 5 and 4 data items, respectively, implying a system data imbalance.

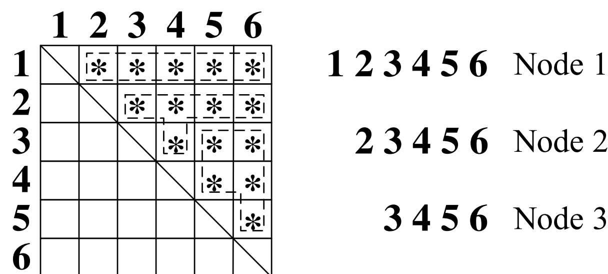


Figure 5.1: A data imbalance.

5.1.2 Storage Saving Causes Task Issues

When Hadoop-based solutions have been used to solve ATAC problems, each data item is randomly distributed to the worker nodes with a fixed number of replications. Although this achieves high data reliability due to replication, poor performance is inevitable due to the lack of consideration of comparison task allocation. Take 6 data items and 4 worker nodes for example. In Figure 5.2, each data item has three copies in three different nodes for reliability.

Each node stores 50% of the data. However, 9 comparison tasks do not have local data, requiring massive data movement at runtime to complete the comparisons and consequently poor overall performance.

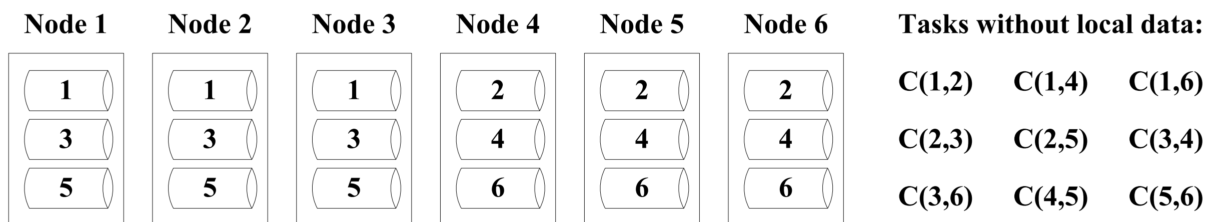


Figure 5.2: Poor data locality for comparison tasks.

5.2 Data Distribution Strategy with Simulated Annealing

This section presents our metaheuristic data distribution strategy for distributed computing of ATAC problems in homogeneous systems. For efficient derivation of data distribution and task scheduling, a simulated annealing (SA) algorithm is developed with specific methods for generating and selecting solutions.

Considering the challenges involved in solving ATAC problems, a data distribution strategy is presented below to meet the following requirements:

1. The system has good static load balancing (Load balancing);
2. All comparison tasks have the data they need locally (Data locality); and
3. The maximum number of data among all nodes is minimized (Storage saving).

5.2.1 Comparison Between Greedy and SA Idea

In this subsection, we will discuss two techniques for solving optimization problems: greedy idea and simulated annealing.

Greedy Idea

A greedy algorithm is an algorithm that constructs an object X one step at a time, at each step choosing the locally best option. Generally greedy algorithms have the following advantages:

1. Greedy algorithms are often easier to describe and code up than other algorithms.
2. Greedy algorithms can often be implemented more efficiently than other algorithms.

However, greedy idea always gets stuck in a local maxima because downward moves are not allowed.

Simulated Annealing

Comparing to greedy idea, simulated annealing [Kirkpatrick et al., 1983] (SA) is a probabilistic optimization technique derived from the physical process of crystallization. It allows downward steps in order to escape from a local maxima. Annealing emulates the concept in metallurgy; where metals are heated to very high temperature and then gradually cooled so its structure is frozen at a minimum energy configuration.

The SA algorithm starts from a randomly chosen initial solution, generates a series of Markov chains through the decreasing of the control parameter (i.e. temperature). In these Markov chains, a new solution is chosen by making a small random perturbation of the solution, and, if the new solution is better, then it is kept, but if it is worse, it is kept with some probability related to the current temperature and the difference between the new solution and the previous solution. Each Markov chain is associated with a given temperature, and it can be described as a thermal equilibrium procedure, or the inner loop. According to the iteration of solutions, an optimal one was found [Wu et al., 2014].

While many efforts have been made to both theoretical and experimental studies of these ideas, it is hard to say which method is preferable to others because the effectiveness of the algorithms always have a strong relationship with the specific optimization problems. Hence, in this chapter, we plan to developed an algorithm based on SA ideas to solve the data distribution and task scheduling problems for ATAC computations.

To use an SA approach, we must determine:

1. The Annealing module and Acceptance Probability module.
2. An initial solution, the neighbourhood selection method and the fitness equation.

5.2.2 Annealing Module and Acceptance Probability Module

The setting of the SA module has significant effects on the final result [Wu et al., 2014]. As one of the fastest decreasing temperature methods, we use Cauchy scheduling [Keikha, 2011]. Parameters used for the example in Section 5.4 are shown in Table 5.1.

Table 5.1: SA module parameter settings (k represents the iteration step).

Item	Setting
Temperature decreasing function	$t_k = T_0/k$
Starting temperature	1.0
Ending temperature	10^{-5}
Inner loop iteration threshold	100
Acceptance probability function	$P(\Delta E) = \exp(-\Delta E/t)$

Initial Solution

For ATAC problems with M data items, $M(M-1)/2$ comparison tasks must be allocated. Hence, for a homogeneous system with N nodes, an initial solution can be generated by randomly and evenly allocating all comparison tasks and related data items. Let M, N, D_i, T_i and U represent the number of data files to be processed, the number of nodes in the system, the set of data files stored on node i, the comparison task set allocated to node i and the set of tasks that have not yet been scheduled, respectively. An initial solution is generated as follows:

1. Keep picking up comparison tasks from set U and allocating them to each node $i \in \{1, 2, \ldots, N\}$ until all have been allocated, i.e., $|T_i| = \left\lceil \frac{M(M-1)}{2N} \right\rceil$ or $U = \emptyset$; and
2. Based on each task set T_i, distribute all related data files to data set D_i.

The solution $S = \{(T_1, D_1), (T_2, D_2), ..., (T_N, D_N)\}$ is then a feasible solution, which meets both our initial requirements.

Neighbourhood Selection Method

Following the design of the initial solution, a new neighbourhood solution S' can be generated from a solution S from the following steps:

1. Randomly choose two nodes i and j;
2. Randomly pick up two comparison tasks $t_k \in T_i$ and $t_l \in T_j$ and swap them; and
3. Update related data files in data set D_i and D_j.

Considering that all the nodes in a homogeneous system can be treated as indistinguishable, this method promises that each new solution has the capability to solve the ATAC problem and all possible solutions can be generated theoretically.

Fitness Equation

Considering the requirements mentioned at the beginning of this section, the fitness equation $F(S)$ for a solution S is defined as the set of the number of data files allocated to each of the worker nodes:

$$F(S) = \{|D_1|, |D_2|, \ldots, |D_N|\} . \tag{5.2}$$

The difference ΔF between two different solutions S and S' is calculated as follows. Firstly, the elements in both $F(S)$ and $F(S')$ are sorted in descending order. Then, ΔF is obtained as:

$$\Delta F = F(S) - F(S') = \{(|D_1| - |D'_1|), \ldots, (|D_N| - |D'_N|)\} . \tag{5.3}$$

Finally, the value of the cost change Δf is defined as:

$$\Delta f = \begin{cases} \text{the 1st non-zero element value in Eqn. (5.3)}, & \text{if one exists} \\ 0, & \text{otherwise.} \end{cases} \tag{5.4}$$

This method promises that solution S with less maximum values in $F(S)$ always be accepted as required by SA. Moreover, unlike only comparing the maximum values in $F(S)$ and $F(S')$, this method utilizes much more information from other elements. Hence, the SA algorithm has a higher efficiency in searching for better solutions.

Data Distribution Algorithm

By integrating all the above designs, our data distribution algorithm using SA is presented as Algorithm 1.

Algorithm 1 Data Distribution Algorithm

Initialisation:

1: Randomly generate initial solution S using the method in **Initial Solution**;
2: Set parameters based on Table 5.1;
3: Set the current temperature t to be the starting temperature.

Distribution:

4: **while** The current temperature t is higher than the ending temperature **do**
5:     **while** The iteration step is below the inner loop iteration threshold **do**
6:         Generate a new solution S' from S (using the **Neighbourhood Selection Method**);
7:         Calculate the change of fitness, Δf, from Equation (5.4)
8:           (The fitness method for F, ΔF and Δf are developed in **Fitness Equation**);
9:         **if** $\exp(-\Delta f/t) > random[0,1)$ **then**
10:             Accept the new Solution: $S \leftarrow S'$
11:         **end if**
12:         Increment the iteration step by 1;
13:     **end while**
14:     Lower the current temperature t based on the function in Table 5.1;
15: **end while**
16: Return final solution S.

5.3 Static Comparison Task Scheduling Strategy

After distributing data files, all the comparison tasks need to be scheduled to different worker nodes. In our data distribution strategy designed in Section 5.2, by considering the system load balancing during deploying data files, comparison tasks need to be executed by each worker node have also be generated simultaneously. Hence, a static task scheduling strategy is developed followed by our data distribution strategy.

5.3.1 Static Scheduling for ATAC Computing Framework

In our ATAC computing framework designed in Chapter 3, a static comparison task scheduling function is provided. In static task scheduling, the work load is distributed depending upon the performance of each worker node at the beginning of execution. A task is always executed on the node to which it is allocated. Though static load balancing methods are nonpreemptive, it can minimize the communication delays in the system.

In Figure 5.3, after distributing all the data files by using our data distribution strategy designed in this chapter, the master node can generate related comparison tasks for all the worker node. Thus, for each work node in the system, all the comparison tasks have to be executed are determined before the computation begin.

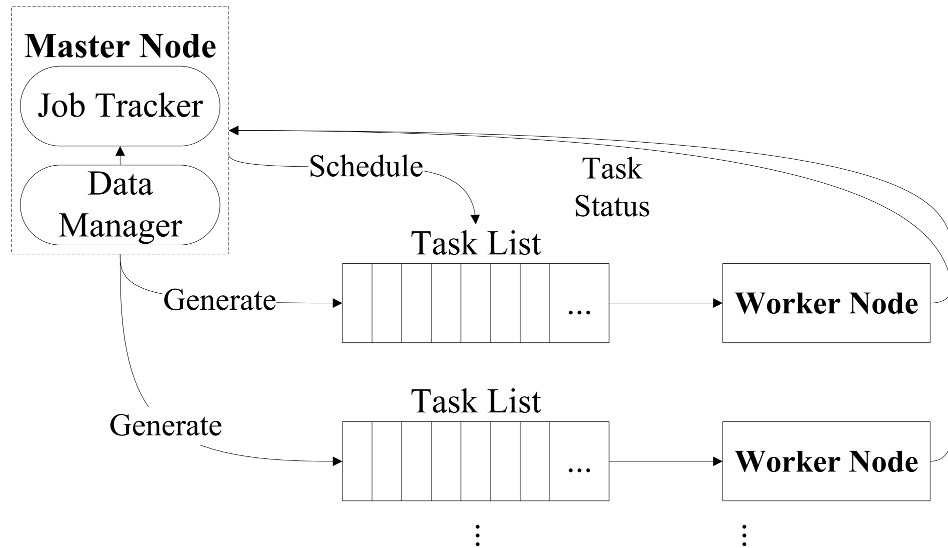


Figure 5.3: Static scheduling of the framework for All-to-All comparison problems.

In our static scheduling, each worker node execute their own comparison tasks without considering the runtime statues of other worker nodes, which can minimize the network communication among the network. During the computation, the status of each comparison task is upgraded by the worker node and managed by the master node until all the comparison tasks are finished.

Due to all the task lists are generated based on the data distribution, data movement can be avoid during computation and the number of comparison tasks on each list is determined by following the load balancing constraint in our data distribution strategy.

5.3.2 Task Scheduling Strategy Design

In this subsection, the static task scheduling strategy generated with our data distribution strategy is formally described. For all-to-all comparison problems, considering all the comparison tasks are independent and there is no running priority for general ATAC problems, the static task scheduling strategy in this chapter is focusing on achieve system load balancing for homogeneous distributed computing systems.

Due to the static scheduling strategy is developed based on our data distribution strategy, the static scheduling strategy is designed to have good performance for the following assumptions:

1. All the comparison tasks have the same execution time.

2. The processing power for each worker node is not changing during computation.

The static task scheduling strategy is designed in algorithm 2. In the algorithm, each worker node in the system keeps executing the comparison tasks that have been allocated to it when it has available computing resources.

Algorithm 2 Static Task Scheduling

Initial:

1: Task set T_i composed of all comparison tasks allocated to worker node i;

Static Scheduling:

2: **for** Each worker node i **do**
3:     **while** There are unscheduled tasks in task set T_i and
4:             enough available computing resources on node i **do**
5:         Pick an unscheduled comparison task from T_i;
6:         Assign this task to node i;
7:         Mark this comparison task as scheduled;
8:         Update the available computing resources on this node i;
9:     **end while**
10: **end for**

In Algorithm 2, scheduling is conducted for each of the worker nodes (Line 2). While there are still unallocated tasks in the task set (Line 3) and the node has sufficient computing resources to accommodate more tasks (Line 4), keep allocating tasks from the unallocated task set one after another to the node (Lines 5 to 8). Since the tasks have been pre-allocated to the nodes, the only scheduling decision made here is to choose the order in which to perform the comparisons on each node.

Moreover, the static scheduling is also inherently considered in our data distribution strategy described in Chapter 4. Hence, the algorithm described in Algorithm 2 can be also used in Chapter 4.

By implementing the data distribution strategy and static task scheduling strategy, all-to-all comparison problems can be solved by using our computing framework designed in Chapter 3.

This static scheduling algorithm works well provided we have good data locality and (static) load balancing. It requires that all information acquired about the nodes, data files and comparison tasks is accurate. It also requires that the distributed system does not have major changes in its environment and thus does not exhibit much uncertainty. If any of these conditions is not met, runtime dynamic scheduling may be required to compensate, which will be discussed in Chapter 7.

In the next section, different experiments are designed to show the performance of solving all-to-all comparison problems in homogeneous distributed systems.

5.4 Experiments

We conducted experiments to evaluate the following aspects of our algorithm: storage savings, task allocations, data scalability and computing performance.

5.4.1 Storage Saving, Task Allocation and Data scalability

An example with 4 nodes and 8 data items is used to show the effectiveness of our data distribution strategy. The results are summarized below in Table 5.2:

Table 5.2: Distributing of 8 data files to 4 worker nodes.

Node	Distributed data files	Allocated comparison tasks
A	0,2,3,6,7	(0,2) (0,3) (0,6) (0,7) (2,3) (2,6) (2,7)
B	1,3,5,6,7	(1,3) (1,5) (1,6) (1,7) (3,7) (5,7) (6,7)
C	0,1,2,4,5	(0,1) (0,4) (0,5) (1,2) (1,4) (2,4) (2,5)
D	3,4,5,6,7	(3,4) (3,5) (3,6) (4,5) (4,6) (4,7) (5.6)

It can be seen from these results that data balancing and static load balancing are achieved. Each worker node only stores 5 data items. Moreover, each node is allocated 7 comparison tasks all with good data locality.

As the numbers of data items and nodes increases, Table 5.3 shows our strategy still achieves good results in storage saving, load balancing and data locality, compared with Hadoop's strategy (using 3 copies of each data item). Each node has an equal number of comparison tasks with good data locality in our solution. Although Hadoop's strategy uses less storage space overall, runtime performance issues are inevitable due to the poor data locality for comparison tasks. Figure 5.4 shows that our approach still has good data scalability, and is far better than the brute-force ATAC solution of copying all data items onto every node.

Table 5.3: Storage usage and storage savings of our work versus Hadoop for 256 files.

No. of nodes	Max. # of files on a node		Storage space saving		# of tasks on each node
	This work	Hadoop	This work	Hadoop	This work
8	150	96	41%	63%	4080
16	116	48	55%	81%	2040
32	83	24	68%	91%	1020
64	56	12	78%	95%	510

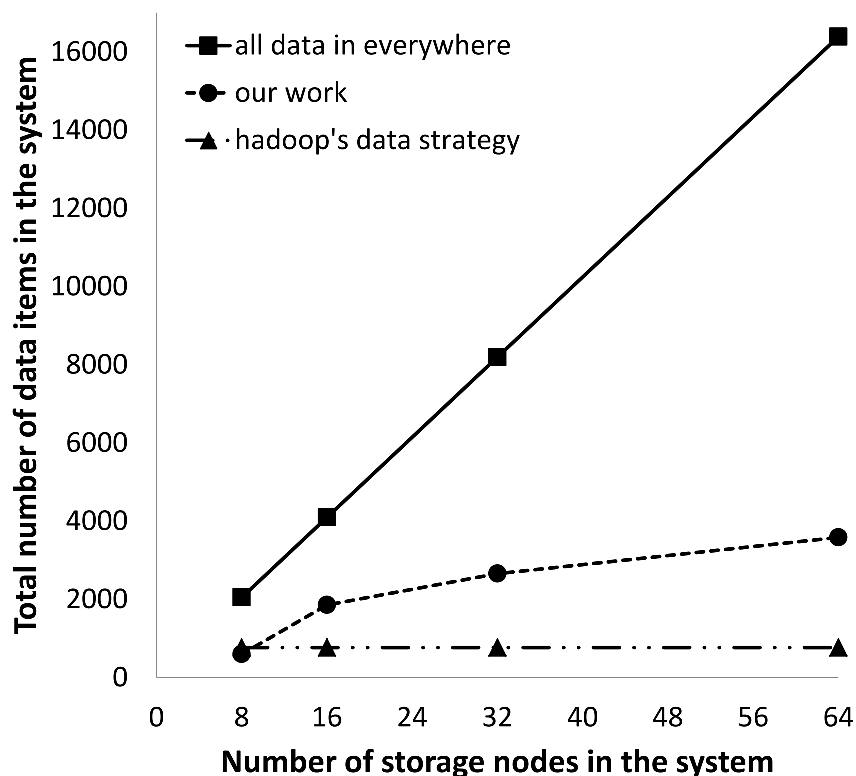


Figure 5.4: Data scalability.

5.4.2 Computing Performance

We also conducted experiments with a bioinformatics ATAC application. The experimental environment was set as:

- A homogeneous cluster with 5 machines, all running Redhat Linux. One acts as the master node, and all the nodes have one core and 64 GB of RAM.
- Sequential and distributed versions of the CVTree application, which is a typical ATAC problem in bioinformatics [Hao et al., 2003].
- DsDNA data files from the National Center for Biotechnology Information (NCBI).

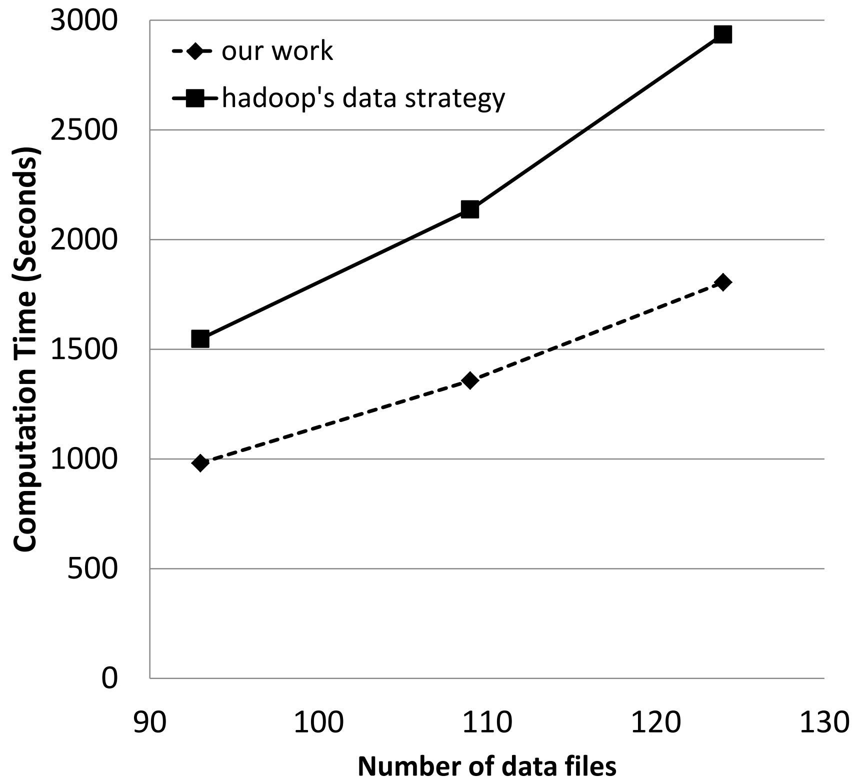


Figure 5.5: Computation time performance.

Figure 5.5 shows the different computation times between our data distribution strategy and Hadoop's strategy. By considering the three requirements summarized in Section 5.2, our data distribution strategy achieves much higher computing performance than Hadoop's strategy. As we discussed in Section 5.1, this is because Hadoop's strategy needs to move numerous data items between nodes during the computation, due to poor data locality.

5.4.3 Computing Scalability

In order to evaluate the scalability of our data distribution strategy, two different data sets from the UCI machine learning repository are chosen to be processed by using our solution with various number of worker nodes. The details of data sets are described in Table 5.4.

Table 5.4: Experimental cases.

ATAC application	Data sets	
	PubMed abstracts	synthetic data
NMF application	9.5G	62G

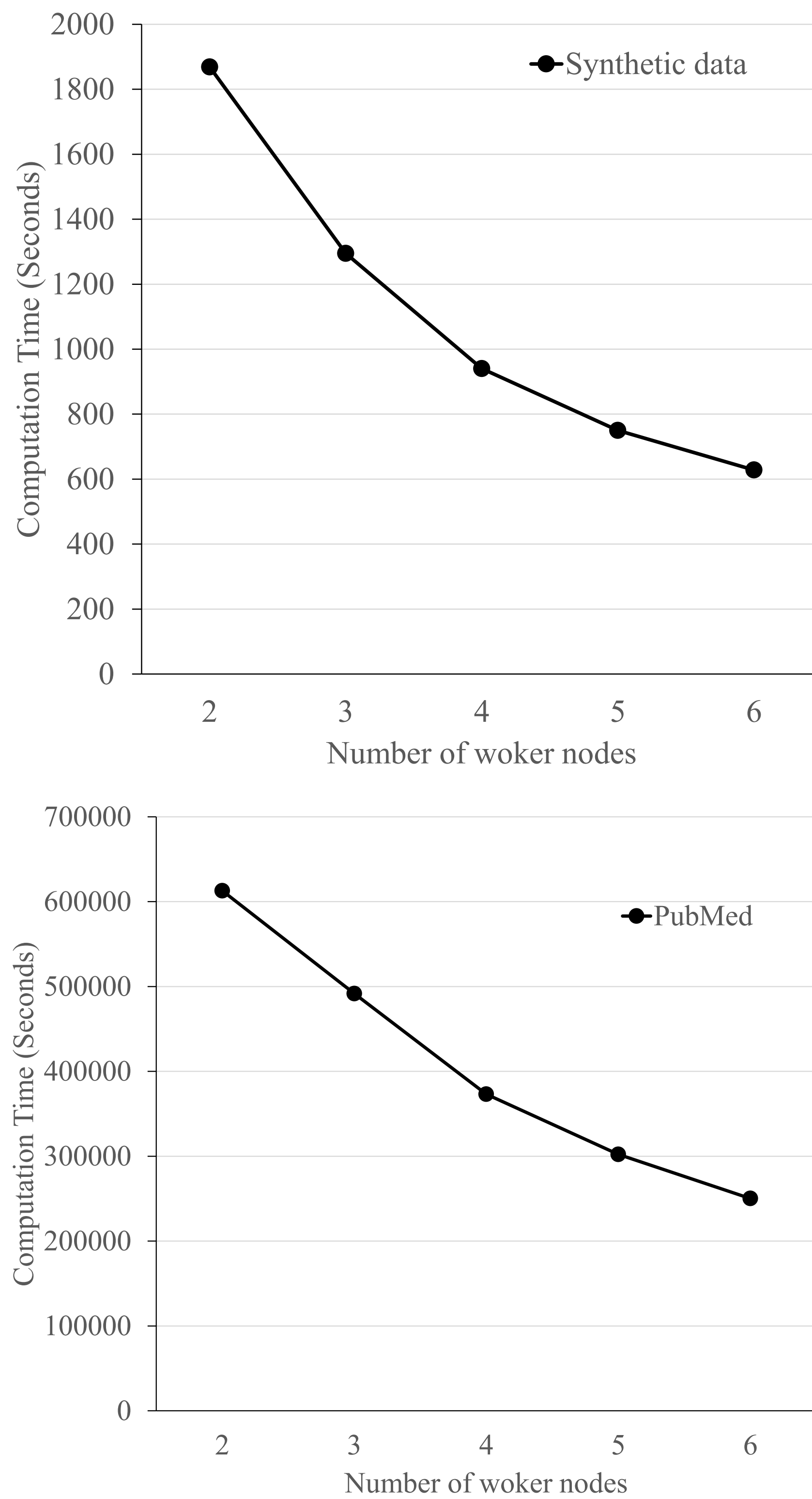


Figure 5.6: Scalability of our programming model.

The two parts of the Figure 5.6 illustrate the computation time caused by processing two different data sets PubMed abstracts and synthetic data. As is shown in these two figures, with the number of worker nodes increasing, the time spend on computation decrease gradually, which indicates a good scalability for our programming model to support large scale ATAC problems.

5.5 Conclusion

A scalable and efficient data distribution strategy using simulated annealing has been presented for distributed computing of all-to-all comparison problems in homogeneous distributed systems. It is designed to use as little storage space as possible while still achieving system load balancing and good data locality. Experiments have shown that although our approach uses more overall storage than Hadoop's, we achieve greatly reduced computation times.

Chapter 6

Heuristic Data Distribution Strategy for Heterogeneous Distributed Systems

---

Recently, distributed computing systems with heterogeneous configurations have been widely used in solving all-to-all comparison problems. For these systems, different worker nodes may have different processing power and storage capability, which makes it hard to make full use of all the computing resources. In this chapter, a data distribution strategy for all-to-all comparison problems is developed to support heterogeneous distributed systems. With the same problem backgrounds, the new challenges caused by heterogeneous distributed systems are summarized and discussed. By formulating the targets and constraints designed for heterogeneous systems, the data distribution algorithm for heterogeneous systems are developed. Also, the static task scheduling strategy is provided to achieve system load balancing for heterogeneous systems. Experiments have been designed to show the performance of the data distribution and static task scheduling strategies in the end.

6.1 Development of Heterogeneous systems

In the previous Chapters 4 and 5, the data distribution strategies for solving all-to-all comparison problems in homogeneous distributed systems have been well developed. For homogeneous distributed systems, the unified hardware configurations make us can treat all the worker nodes as the same, which some kind simplifies the data distribution problems.

However, over the last decade, heterogeneous distributed systems have been emerging as

popular computing platforms for solving all-to-all comparison problems. Heterogeneous distributed systems are usually composed of diverse sets of resources with different capabilities and interconnected with networks to meet the requirements of widely varying applications Bharathi and Kumaresan [2012]. For these systems, due to different worker nodes have different processing power and storage capabilities, using the data distribution strategy for homogeneous systems cannot achieve high performance. Hence, in this chapter, the data distribution strategy designed for heterogeneous distributed systems is developed.

Comparing to homogeneous distributed systems, the heterogeneous system is more widely used because of the following reasons:

1. Hardware updating is very frequently, especially to meet the requirements of big data computing problems. Considering hardware cost, resource utilization and system scale, mixing worker nodes with different capabilities is a popular choice for many research and commercial organizations.

2. To make full use of the computing power provided by the distributed systems, usually multiple applications are running on top of the system at the same time. The shared using of system can also cause the available computing resources become different for different worker nodes.

Therefore, providing data distribution strategy that supports heterogeneous distributed systems is a must for meeting the growing requirements from various application areas.

6.2 Existing solutions for Heterogeneous systems

Currently, though many researches have been proposed based on heterogeneous distributed systems, they all have the following limitations:

1. Original designed for homogeneous distributed systems.

   As we mentioned in Chapter 2, most existing methods for distributed computing of ATAC problems are inherently assumed for homogeneous systems. Though they can also be used in heterogeneous distributed systems, the overall performance can be greatly degraded.

2. Lack the consideration of data distribution.

   Some researchers schedule work loads based on the processing power of each worker node in the system. While load balancing for these solutions can be achieved for heterogeneous distributed systems, the lack consideration of data distribution will cause massive unnecessary data deployment. Moreover, massive data transmission at runtime can be a serious performance problem, especially when the data need to be processed is growing big.

To address those challenges, advanced strategies are required for distributed computing of ATAC problems in heterogeneous distributed systems.

6.3 Problem Statement

An ATAC problem is a specific Cartesian product of a data set. Let A, A_i, $C(A_i, A_j)$, $M[i, j]$ represent the set of input data items, single data item in set A, the comparison operation between data items A_i and A_j, and output similarity matrix element, respectively. The ATAC problem is formulated as:

$$M[i, j] = C(A_i, A_j) \qquad i, j = 1, 2, ..., |A| \tag{6.1}$$

6.3.1 Feature of Heterogeneous Systems

As we mentioned in Section 6.1, many solutions designed for homogeneous distributed systems cannot achieve high performance when they are running on top of heterogeneous distributed systems. To distribute data files and all the comparison tasks to the heterogeneous systems, there are the following differences need to be considered:

1. Different processing power.

   For heterogeneous distributed systems, different processing power for each worker node makes the system load balancing should be allocated by considering the processing power for each worker node.

2. Different storage capability.

Usually, worker nodes in the heterogeneous system have different storage capability. Data distribution in the system should also consider the different capability for each worker node.

In the following Subsection, the challenges of distributing data files for ATAC problems in heterogeneous systems are discussed.

6.3.2 Similar Research Problems

(D,c)-coloring

There are some research problems in graph theory that are similar to our data distribution problem, such as graph covering, graph partitioning, graph coloring. Though neither of them have the exactly same constraints and objective functions, they are still very useful to help us understand the complex of the data distribution problem.

> A $(D, c) - coloring$ of the complete graph K^n is a coloring of the edges with c colors such that all monochromatic connected sub-graphs have at most D vertices.

Many research focuses on the $(D, c) - coloringProblem$. The solution bound and special cases with small value of D and c have been discussed in a mathematical way but the methods to get the solution have not been discussed.

Balanced Incomplete Block Design

A BIBD is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks (for $k, r, \lambda > 0$). The construction of BIBDs was initially attacked in the area of experiment design; however, nowadays BIBD can be applied to a variety of fields such as cryptography and coding theory.

BIBD generation is a NP-hard problem [Corneil and Mathon, 1978] and it makes complete methods be inherently limited by the size of the problem instances. Though different solutions have been provided, there are still a number of open instances that have not been solved yet [Rueda et al., 2009].

6.4 Challenge of the Data Distribution Problem

In this section, we will discuss about the challenges of solving the data distribution problem from different aspects. Some challenging situations of solving the data distribution problem are summarised and analysed.

Hadoop-based solutions are widely used for distributed computing of ATAC problems, they are inefficient for the ATAC pattern. However, with the focus on the MapReduce pattern, Hadoop is inefficient for the ATAC pattern of the ATAC problems. Challenges of executing ATAC problems by using Hadoop are summarised in the following three aspects:

6.4.1 Poor Data Locality of Comparison Tasks

Good data locality is one of the principles in processing data intensive problems. It is often better to move the computation close to where the data is located [Khan et al., 2014].

For Hadoop, good data locality for Map tasks are almost promised due to two reasons: 1) Hadoop assumes that a Map task is expected to process a single data split; and 2) In Hadoop's data distribution, each data with a fixed number of replications is randomly distributed into different worker nodes. However, for ATAC problems, each comparison task needs to process two different data items. Therefore, Hadoop's data distribution becomes inefficient.

Let us consider an example of 4 data items and 6 worker nodes. If each data stored in HDFS has 3 copies (default setting), a possible data distribution is depicted in Figure 6.1. It is seen from Figure 6.1 that poor data locality exists for four comparison tasks $C(1, 2), C(2, 3), C(1, 4)$ and $C(3, 4)$. To execute these four tasks, data items must be transferred at runtime among the nodes. For large-scale ATAC problems, moving massive data around at runtime causes significant degradation of the overall computing performance.

6.4.2 Invalid locality-aware Scheduling of Comparison Tasks

Data locality is important in processing a large volume of data sets. In Hadoop, data locality information is used to schedule Map tasks. Map tasks are executed in the worker nodes with required data. However, the data locality for Hadoop's Map tasks is lost when dealing with ATAC problems. For example, again for the scenario shown in Figure 6.1, it is a problem for

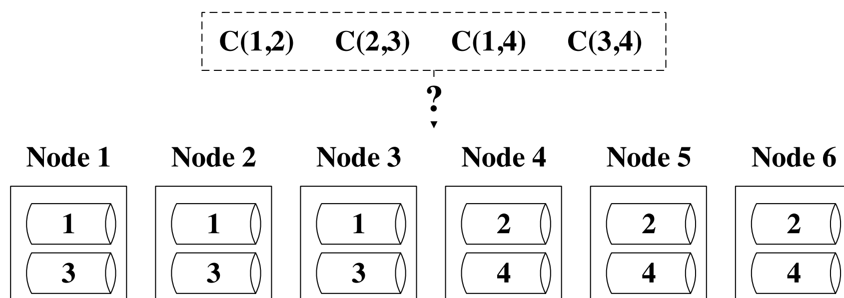


Figure 6.1: Poor data locality for comparison tasks.

Hadoop's task scheduler to decide in which worker nodes the tasks $C(1,2), C(2,3), C(1,4)$ and $C(3,4)$ are executed because each of the nodes stores only half of the required data items for each of these comparisons.

Another example is shown in Figure 6.2. Assume that two comparison tasks $C(2,4)$ and $C(2,6)$ are ready to dispatch for execution and only work nodes 3 and 4 are idle at the moment. If $C(2,4)$ and $C(2,6)$ are scheduled to worker nodes 3 and 4, respectively, both tasks have local data to complete the execution. However, if these two tasks are swapped between worker nodes 3 and 4, task $C(2,6)$ will have no local data for execution on node 3. Hadoop's task scheduling does not guarantee that the later occasion does not happen.

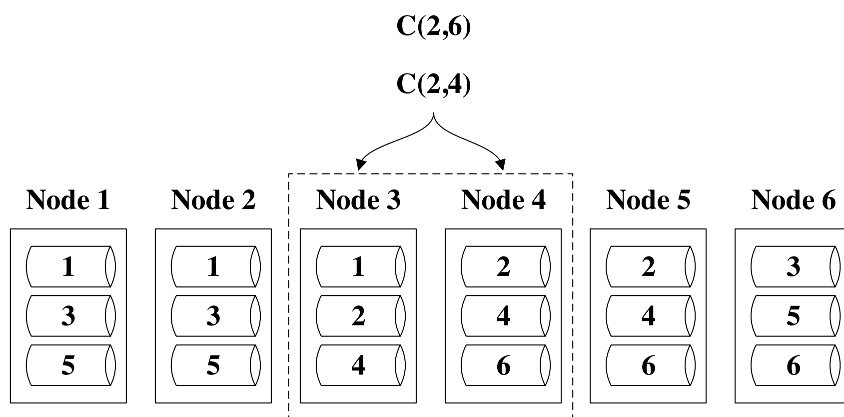


Figure 6.2: Task scheduling with globe consideration requires $C(2,4)$ and $C(2,6)$ are allocated to worker nodes 3 and 4, respectively, for data locality when these two nodes are idle for more tasks.

Hadoop is not effective in the above two examples because Hadoop is not designed for ATAC computing pattern. It has a lack of the capability to make a global decision to schedule comparison tasks with consideration of all available idle nodes in ATAC computing. Therefore, task scheduling in Hadoop is not data locality aware in general for ATAC problems.

6.4.3 Inefficiency in Heterogeneous Systems

Hadoop inherently assumes all the machines have the same processing capabilities. Though this simplifies the system modeling and implementation, it leads to poor use of the computing resources of a heterogeneous system.

For instance, in a cluster with four worker nodes, two nodes have twice as much computing power as that in other nodes. If there are 18 comparison tasks waiting for execution, each of the four nodes are allocated three tasks first. Then, the remaining six tasks should be evenly distributed to nodes 3 and 4 for the best use of the computing resources. This is clearly shown in Figure 6.3. However, Hadoop does not have such a mechanism to dispatch tasks with full consideration of computing resources in heterogeneous systems.

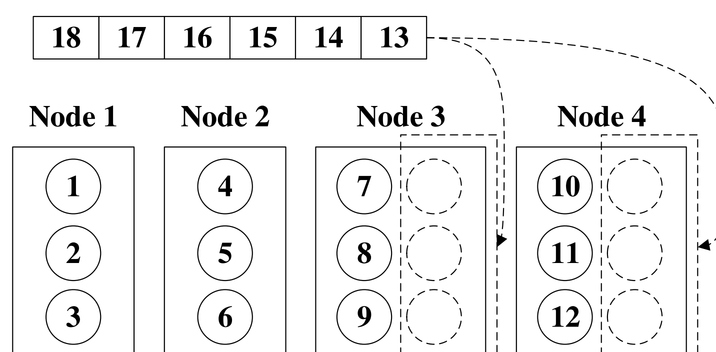


Figure 6.3: The best use of computing resources requires tasks 13 to 18 are evenly distributed to worker nodes 3 and 4, which have twice as much computing power as that in the other two nodes.

The above three challenges indicate the ineffectiveness of Hadoop's strategy in solving ATAC problems in heterogeneous environments. The following section develops a new data and scheduling strategy to address these challenges.

6.5 Data and Task Distribution Strategy

In this section, an abstraction is developed for the requirements of data distribution and task distribution from the challenges discussed in Section 6.4. Then, a heuristic strategy is presented for data distribution and task scheduling for distributed computing of ATAC problems in heterogeneous systems. Examples will also be given for further analysis of the strategy.

6.5.1 Strategy Targets

To address the challenges discussed in Section 6.4, a data and task scheduling strategy will be designed to meet the following requirements:

1. Data are distributed in a way to enable all comparison tasks to have good data locality (Challenge 6.4.1).

2. Comparison tasks are always scheduled with good data locality (Challenge 6.4.2).

3. Each of the worker node is assigned comparison task load proportional to its computing power (Challenge 6.4.3).

4. For storage saving, data replications are as fewer as possible, and the maximum number of data items among all worker nodes is minimized.

Distributing data items to worker nodes in a heterogeneous system can be treated as a complex combinatorial mathematics problem. The optimization of a combinatorial mathematics problem is difficult, even in a homogeneous system as we have investigated previously [Zhang et al., 2014]. Therefore, a heuristic algorithm will be developed to address this problem in the following part.

6.5.2 Strategy Design

For the data and task scheduling problem, a reasonable idea is to pre-consider task allocation during data distribution. In this way, the task scheduling will become data-aware. Consequently, runtime data re-arrangement can be avoided.

In actual heuristic task scheduling, the computing tasks are assigned to the work nodes one after another. Therefore, choosing which node to assign a task most appropriately is critical. This chapter uses a concept of node completeness to quantify the degree of the appropriateness for a node to have the next task assignment. Then, a full process is designed to assign all tasks to the worker nodes.

Completeness of a Node

Let $|T_i|$ denote the total number of comparison tasks that should be allocated to node i. From the load balancing requirement 3) listed above, $|T_i|$ should be proportional to the computing power of node i. As an ATAC problem has a finite number of comparison tasks, $|T_i|$ can be determined before data distribution is conducted. Now, let $|A_i|$ denote the number of comparison tasks that have already be assigned to node i. The following ratio R_i is defined to describe the completeness of node i:

$$R_i = |A_i| \, / \, |T_i| \, , \; 0 \leq R_i \leq 1 \tag{6.2}$$

When $R_i = 0$, no tasks have been assigned to node i. If node i has been fully assigned tasks in comparison with all other nodes in terms of their computing power, $R_i = 1$.

With the definition of node completeness, the next task should be assigned to a node with the smallest completeness value among all nodes. Using R_i instead of $|A_i|$ in determination of a node for assignment of the next task will help avoid load imbalance among the nodes in a heterogeneous system.

Rules for Data and Task Distribution

Figure 6.4 shows the distribution of a new data d to node i that has already been assigned p data files $(1, 2, \cdots, p)$. The connecting arrows represent new comparison tasks available at node i due to the distribution of the new data. There are two types of comparison tasks: those that have never been allocated (dotted arrows), and those that have already been allocated previously (solid arrows). We have developed rules for both types of tasks:

Rule 1. For the comparison tasks that have never been allocated, allocate as many tasks as possible to node i by meeting the load balancing requirement 3).

Rule 2. For the comparison tasks that have already been allocated previously, re-allocate each of them by meeting the load balancing requirement 3). During data distribution, for a comparison task $C(x, y)$, there could be more than one worker nodes with both data items x and y. In this case, compare the completeness of all these worker nodes and re-allocate the comparison task $C(x, y)$ to the node with the lowest completeness value defined in Eq. (6.2).

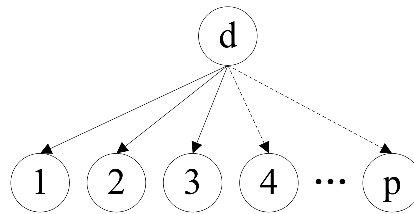


Figure 6.4: New available comparison tasks for distributing new data d to node i. They include those that have never been allocated (dotted arrows) and those that have been allocated previously (solid arrows).

For an ATAC problem with M data items, the total number of comparison tasks is $M(M-1)/2$. All these tasks are allocated by meeting the load balancing requirement 3). If the task allocation is further constrained by meeting the data locality requirements 2) and 3) discussed previously, a solution to the problem of data distribution and task scheduling is obtained to fulfil all of our design targets. Furthermore, for each new distributed data file d to node i, by allocating as many newly introduced tasks as possible to node i and re-locating those tasks that have already been allocated before, the total number of data files need to be distributed can be minimized. This tends to give a good solution to meet the requirement 4) discussed before.

6.5.3 Strategy for Data Distribution and Task Scheduling

From the discussions above, a heuristic strategy is formally given in the following algorithm for data distribution and task scheduling of ATAC problems in heterogeneous systems:

1. Calculate the number of comparison tasks that should be allocated to each worker node (Requirement 3).
2. Find all unallocated comparison tasks.
3. Find all data items needed for these unallocated tasks; Put them in set I, which is initially empty.
4. From the set I, find the data item which is needed by the greatest number of unallocated comparison tasks. Let d denote this data file.
5. Choose a set of nodes (denoted by C) that
    - do not have the data file d,

- have the lowest completeness computed from Eq. (6.2),
- have stored the least number of data files.

6. Check **Rule 1** in Section 6.5.2 for all nodes in set C. If none of the nodes meet the requirement 3), remove this data file d from set I and go back to Step 4.
7. Find a node i in set C such that the node is empty or can be allocated the largest number of new comparison tasks that are introduced by adding the data file d and have not been allocated before. Distribute data file d to this node i.
8. For comparison tasks that are introduced by adding data file d in Step 7 and have already been allocated to other nodes before, use **Rule 2** in Section 6.5.2 to re-allocate these tasks.
9. Repeating Steps 2 to 8 until all comparison tasks have been allocated.

By following the above steps, allocating all comparison tasks with as less number of data items as possible will help compress the maximum number of data among all worker nodes. Also, allocating comparison tasks according to the computing power of each node leads to good load balancing.

6.5.4 Analysis of the Presented Strategy

Consider an example with seven data files numbered $\{0, 1, 2, 3, 4, 5, 6\}$ and five nodes labelled $\{A, B, C, D, E\}$. Assume that worker nodes A and B have twice as much computing power as that in the other nodes. A solution obtained from our strategy is shown in Table 6.1.

Table 6.1: Distribution of seven data files to five worker nodes.

Node	Distributed data files	Allocated comparison tasks
A	1,2,3,5,6	(1,2) (1,5) (1,6) (2,3) (2,6) (5,6)
B	0,1,3,4,6	(0,6) (1,3) (1,4) (3,4) (3,6) (4,6)
C	0,3,5	(0,3) (0,5) (3,5)
D	0,1,2,4	(0,1) (0,2) (0,4)
E	2,4,5	(2,4) (2,5) (4,5)

It is seen from Table 6.1 that the task scheduling solution obtained from our strategy allocates twice as many tasks to each of nodes A and B as those to each of the other nodes. This

is consistence with the computing power of the worker nodes. Also, all allocated tasks to each node have good data locality. Therefore, good load balancing can be achieved during actual task execution. This implies that the three challenges discussed in Section 6.4 have been well addressed.

6.6 Experiments

In this section, experiments are conducted to evaluate our strategy for data distribution and task scheduling strategy from three aspects: 1) storage saving and task allocation; 2) computing performance; and 3) scalability.

6.6.1 Storage Saving and Task Allocation

An ATAC problem with a data set with 256 data files is considered in a heterogeneous system. The heterogeneous system is composed of a number of nodes with the number changing between 2 and 64. Half of the nodes (Type B) have twice as much computing power as that of the other nodes (Type A). For Hadoop's strategy, the number of replication is set to the default value 3.

With the number of nodes growing, Table 6.2 shows at least how much storage space our strategy and Hadoop's data strategy can save in comparison with the method to distribute the whole date set to every node. The allocation of comparison tasks by using our strategy is also presented in Table 6.2.

Table 6.2: Storage usage, storage saving and task allocation.

No. of	Max. no. of files on a node		Storage space saving (%)		No. of tasks on each node (Ours)	
nodes	Our	Hadoop	Ours	Hadoop	Type A	Type B
4	225	192	12	25	5440	10880
8	207	96	19	63	2720	5440
16	163	48	36	81	1360	2720
32	113	24	56	91	680	1360
64	79	12	69	95	340	680

It is seen from Table 6.2 that Hadoop saves the storage space the most. But this is achieved with significant sacrifice of data locality [Zhang et al., 2014]. Our strategy also saves much

storage space, especially when the mount of nodes becomes great. Although our strategy does not save as much storage space as the Hadoop strategy, all allocated comparison tasks from our strategy have 100% data locality. This improves the overall computing performance of the ATAC problem significantly.

Results in Table 6.2 also show a static task scheduling obtained from our strategy. Worker nodes of B Type with more processing power are allocated more comparison tasks. This is not implicitly considered in Hadoop's strategy.

6.6.2 Computing Performance

Both computing performance and scalability are important in processing big data problems in heterogeneous systems. Experiments are designed with a real ATAC application to evaluate our data and task distribution strategy.

A heterogeneous cluster with 5 machines is built for experiments. All the machines run 64-bit Redhat Enterprise Linux. One node acts as the master node, the other four are worker nodes. Two of the worker nodes have dual cores and 64 GB RAM, while the other two worker nodes have a single core and 32 GB RAM.

We have implemented a CVTree application, which is a typical ATAC problem in bioinformatics [Hao et al., 2003]. In the CVTree problem, a set of dsDNA viruses files from the National Center for Biotechnology Information [NCBI, 1988] are chosen as the input data. A sequential version of the CVTree computing is also developed as the basis for scalability evaluation.

Figure 6.5 shows comparisons of the computation time performance between our strategy and Hadoop's one. By considering the three requirements summarized in Section 6.5.1, our data and task distribution strategy achieves much higher computing performance than Hadoop's strategy. This is because our strategy guarantees good data locality and load balancing. However, Hadoop's strategy causes runtime movement of a large amount of data during computation. Also, it does not make full use of the computing resources in the heterogeneous environment, leading to poor load balancing. Therefore, the computation time performance of the ATAC problem by using Hadoop's strategy becomes much poorer.

The good load balancing of our strategy is confirmed by the experimental results shown in Figure 6.6. This results from the fact that the worker nodes are allocated the number of

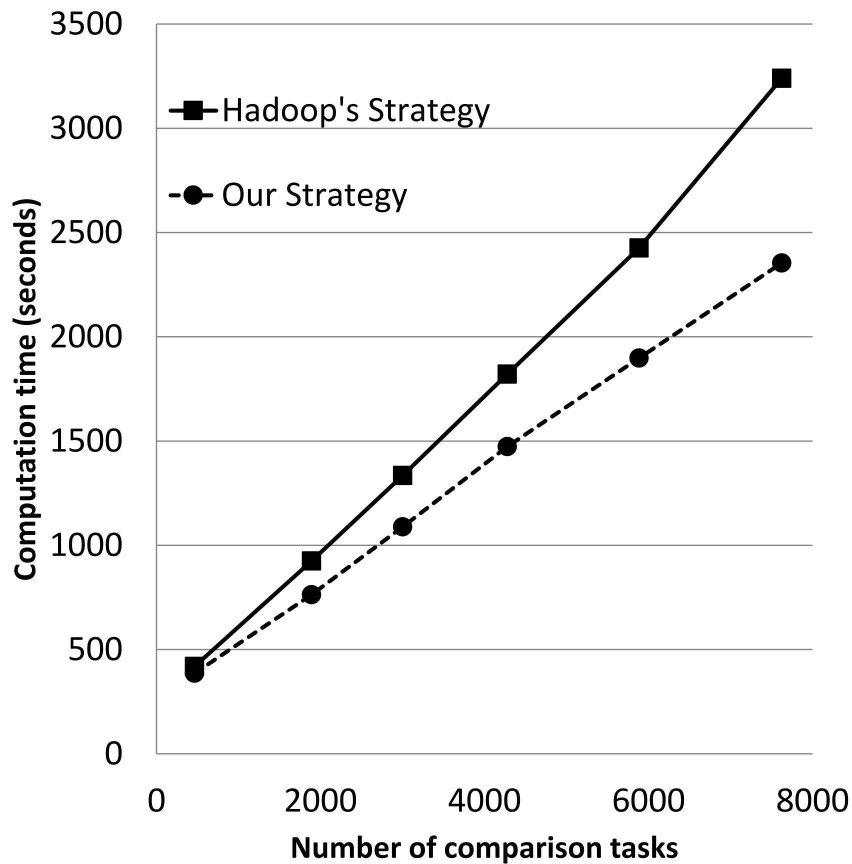


Figure 6.5: Comparisons of the computation time performance.

comparison tasks proportional to their respective computing power. Therefore, the computing resources of the heterogeneous system are fully utilized.

6.6.3 Scalability

Scalability is the capability of a system to have a linearly increased performance when the system ensemble size is scaled up. It is widely used to predict the performance of distributed systems at a large system size based on their performance at small size [Sun et al., 2005]. In general, a system exhibits a linear speedup as the number of processors increases if the communication overhead, load imbalance and extra computation are not considered [Li et al., 1999]. In Figure 6.7, the 1:1 linear speedup in dot line can be considered as an ideal speedup.

Results from our experiments are depicted in Figure 6.7. It is seen from Figure 6.7 that that with the increase of the number of processors, our data and task distribution strategy achieves a linear speedup. This implies that though ATAC problems incur inevitable costs in network communications, memory and disk I/O, good scalability is still achieved on the

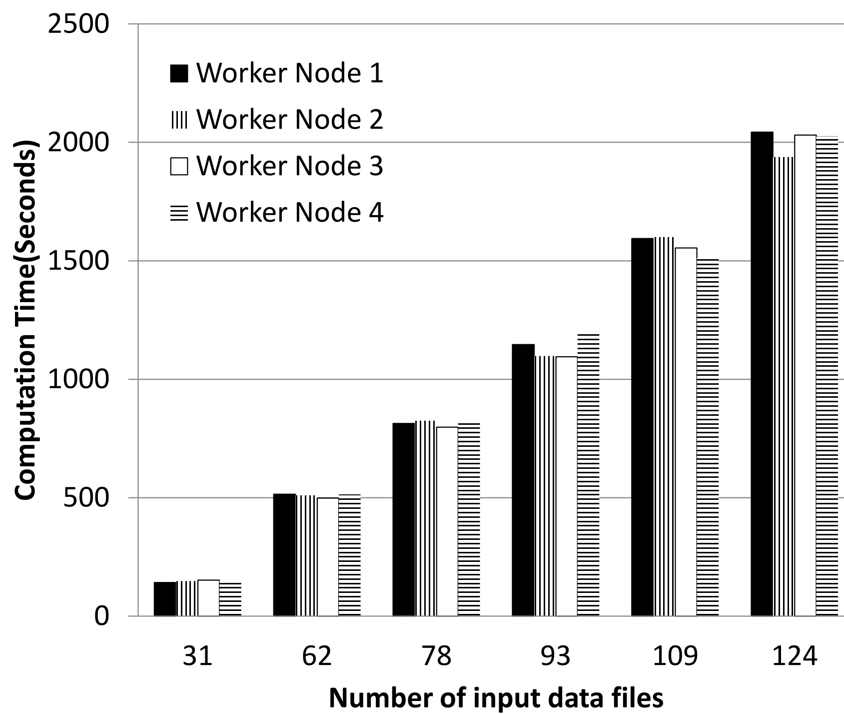


Figure 6.6: Demonstration of load balancing from our strategy.

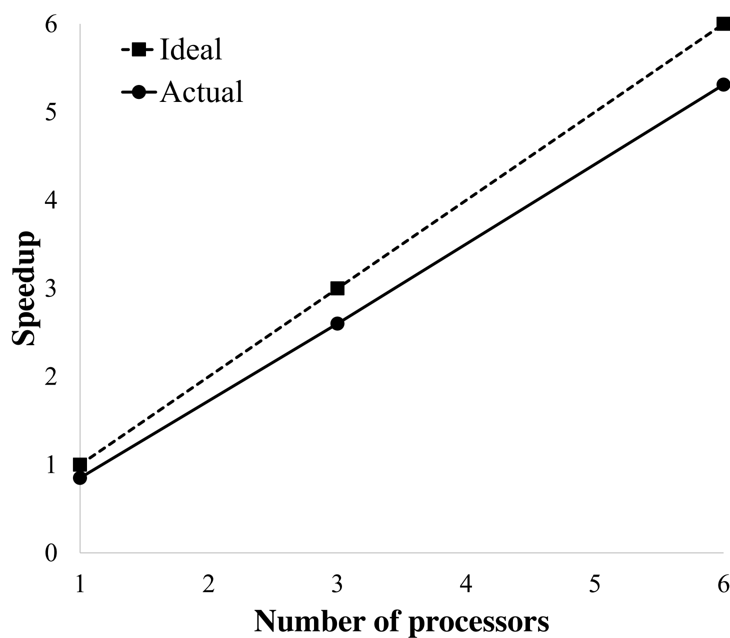


Figure 6.7: Speed-up achieved by the data and task distribution strategy.

overall distributed computing. Our strategy presented in this chapter achieves about 88.5% of the performance capacity of the ideal 1:1 linear speedup. The is measured by $5.31/6 = 88.5\%$ at 6 cores from the results shown in Figure 6.7.

6.7 Conclusion

A new data distribution strategy has been presented which solves all-to-all comparison problems by using distributed computing systems. It is designed to minimize and balance the usage of storage space while still keep load balancing and good data locality for all the comparison tasks. Besides, to allocate comparison tasks that fully utilize the computing performance in heterogeneous distributed systems, both the high performance static and dynamic scheduling strategies are developed. Experiments have been conducted to show the good results of the data distribution strategy and the computing framework for all-to-all comparison problems.

Chapter 7

MetaHeuristic Data Distribution Strategy for ATAC Problems in Heterogeneous Distributed Systems

---

Solving large-scale all-to-all comparison problems using distributed computing environments is increasingly significant for various applications. Previous work in Chapter 6 provides a heuristic data distribution strategy for processing all-to-all comparison problems in heterogeneous systems. Static task scheduling is also considered in that strategy. In Chapter 5, a metaheuristic data distribution strategy is also developed for processing all-to-all comparison problems in homogeneous systems. Considering metaheuristic algorithm shows a great performance improvement in solving all-to-all comparison problem, in this chapter, we present a data-aware task scheduling approach for solving all-to-all comparison problems in heterogeneous distributed systems. Our approach formulates the requirements for data distribution and comparison task scheduling simultaneously as a constrained optimization problem. Then, metaheuristic data pre-scheduling and dynamic task scheduling strategies are developed along with an algorithmic implementation. This approach provides perfect data locality for all comparison tasks, avoiding the need to rearrange data at run time, achieves load balancing among the heterogeneous computing nodes, to enhance the overall computation time, and reduces data storage requirements across the network. The effectiveness of our approach is demonstrated through experimental studies.

7.1 Problem Statement and Challenges

An all-to-all comparison calculation pairwise compares data items for a whole data set. An example is shown in Figure 7.1 as a graph where vertices represent data files to be compared and edges represent each necessary comparison task between two data files. As it can be seen from the figure, for an all-to-all comparison problem with M data files, the total number of comparison tasks is $M(M-1)/2$. We denote the number of vertices and edges in such a graph as the pair $(M, M(M-1)/2)$.

7.1.1 Principles of the All-to-All Scheduling Problem

As noted above, existing all-to-all comparison approaches have considered data deployment and task scheduling in two separate phases. This means that data distribution is conducted without considering the requirements of pairwise comparisons, and task scheduling is carried out without considering the data distribution results. This leads to poor data locality and a consequent need for remote data access to complete comparison tasks. As a result, a large number of data files need to be relocated at runtime, degrading overall execution time performance.

Efficient scheduling of comparison tasks for all-to-all comparison problems with big data sets requires us to consider the available computational resources and to avoid runtime data movement. From this perspective, two basic principles are essential for scheduling all-to-all comparison tasks with big data sets:

1. Data distribution must be task-oriented; and
2. Comparison tasks must be scheduled to avoid remote data access.

Following these two principles, a general design of data distribution and task scheduling

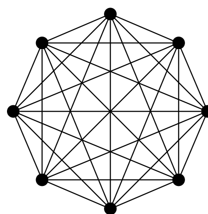


Figure 7.1: Graph representation of an all-to-all comparison problem. The vertices and edges of the graph represent data files and pairwise comparison tasks, respectively.

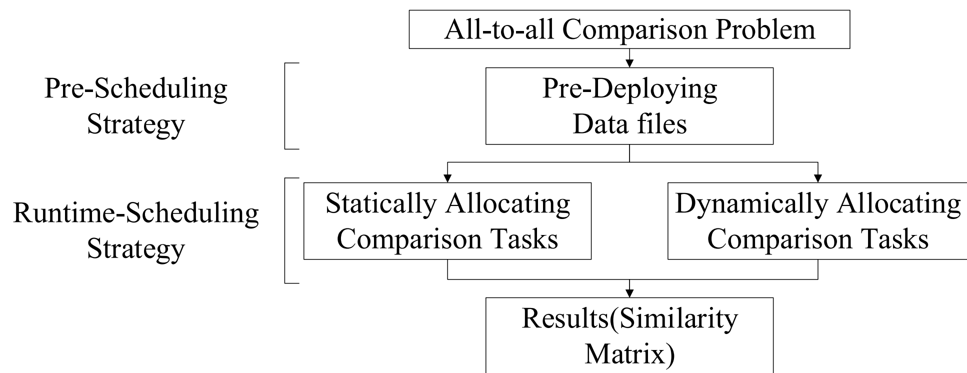


Figure 7.2: A general design for data distribution and task scheduling.

is depicted in Figure 7.2. It consists of two main stages: pre-scheduling, and runtime static and dynamic scheduling. (These major stages are followed by a trivial results-gathering step to produce the final matrix of pairwise comparison outcomes.) For each of the two major stages, we present below theoretical development and detailed design of our solution.

7.1.2 Challenges of the All-to-All Scheduling Problem

When each worker node stores only part of the data set, a well-designed data distribution strategy is needed to meet the above two principles, especially when the size of the data set becomes large. However, the Hadoop framework, widely used in distributed computing with big data sets, deviates from these two principles when used for all-to-all comparisons. With a fixed number of data replications, it allocates data files randomly to distributed worker nodes. Experiments reported by Qiu et al. [2009] have demonstrated the inefficiency of the Hadoop framework for all-to-all comparison problems:

1. The computation iteratively switches between 'map' and 'communication' activities; and
2. Hadoop's simple scheduling model may not produce optimal scheduling of comparison tasks.

These properties result in poor data locality and unbalanced task loads when using a framework like Hadoop for all-to-all data processing.

We can summarize the characteristics of the all-to-all scheduling problem as the following four challenges.

Distributing Data Evenly Does not Mean Load Balancing

In order to achieve system load balancing, an obvious approach is to evenly distribute data files to worker nodes. However, for all-to-all comparison problems, such a data distribution method does not promise that the data pairs needed for comparisons are available locally on the same worker node. For example, given a comparison problem with 6 data items the total workload can be expressed as a graph with the number of vertices and edges equal to $(6, 15)$, as shown in Figure 7.1. Assuming we have 4 worker nodes, a possible task allocation using an even data distribution strategy is shown in Figure 7.3. There are 2 copies of each of the 6 data items and each of the 4 worker nodes stores 3 data items. However, load balancing is not achieved as the nodes have different numbers of comparison tasks: Node 1 and Node 3 have 3 tasks, but Nodes 2 and 4 only have 2 tasks. Even worse, there are 5 comparison tasks that cannot even execute due to the lack of data locality. For instance, no node has copies of both data items 3 and 6.

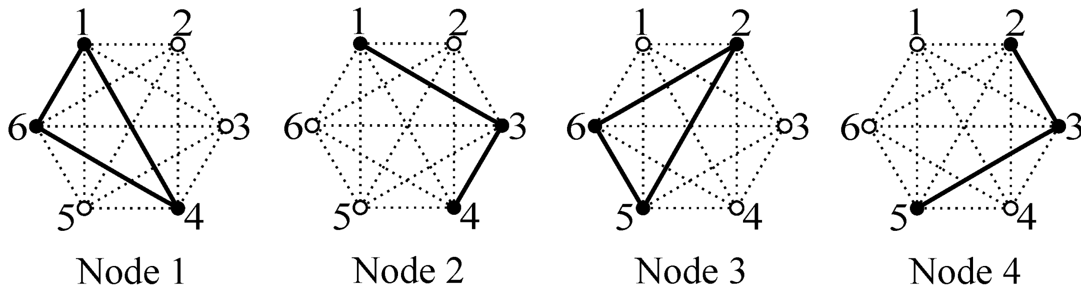


Figure 7.3: A possible data distribution of 6 files on 4 nodes. Solid lines, dotted lines, solid points and hollow points represent scheduled tasks, unscheduled tasks, scheduled data items and unscheduled data items, respectively.

Task Load Balancing May Cause Data Imbalances

Another way of attempting to achieve load balancing is to schedule similar workloads to each of the worker nodes. While this promises to put a similar number of comparison tasks on each node, the separation of task allocation from data distribution can cause a severe data imbalance, resulting in large amounts of wasted storage, particularly when processing big data sets. Consider a scenario with 9 data items, which means that 36 comparisons are required. Figure 7.4 shows how these 36 comparison tasks could be distributed across 6 worker nodes. For load balancing, each worker node is allocated 6 tasks. It can be seen that to achieve the

necessary data locality, it is sufficient for Node 1 to store only 4 data items. However, Node 6 has to store 8 data items, double the number of Node 1, to avoid the need for remote data accesses.

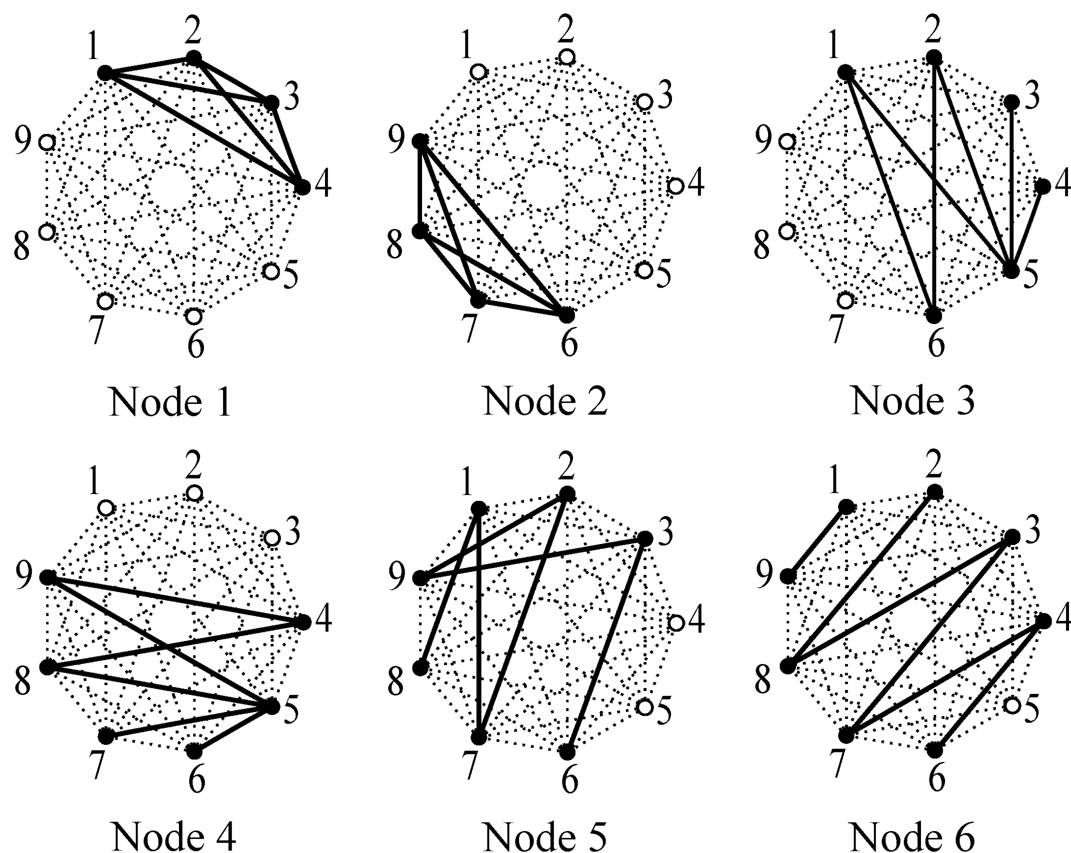


Figure 7.4: A possible data distribution of 9 files on 6 nodes.

Heterogeneous Systems are Harder to Schedule than Homogeneous Ones

To make full use of the computing resources in heterogeneous distributed systems, the worker nodes should be allocated a task workload proportional to their respective processing power. However, many existing solutions, e.g., Hadoop, are designed with an implicit assumption of a homogeneous environment. For instance, consider a scenario with 6 data items, thereby requiring 15 comparisons, and 3 worker nodes, as shown in Figure 7.5. A possible data task scheduling solution for a homogeneous environment, with balanced data distribution, is shown in the upper part of Figure 7.5. In this case, each of the 3 worker nodes is allocated 5 comparison tasks to ensure good data locality, thus balancing data distribution and the number of comparison tasks. However, the same solution may cause a significant load imbalance in a

heterogeneous system in which the worker nodes have different computing power. For instance, if Node 1 has triple the processing power of either Node 2 or 3, load balancing can only be achieved when Node 1 is assigned 9 tasks and each of the other two nodes is allocated 3 tasks. Unfortunately, with the data distribution strategy suitable for a homogeneous system, there is no way to allocate enough tasks to Node 1 to prevent it finishing its computation long before the other two nodes. An incomplete attempt at allocating the tasks in this scenario is shown in the lower part of Figure 7.5. Not all of the 15 comparisons have been scheduled, but adding any further tasks to node 1 will unbalance the data distribution.

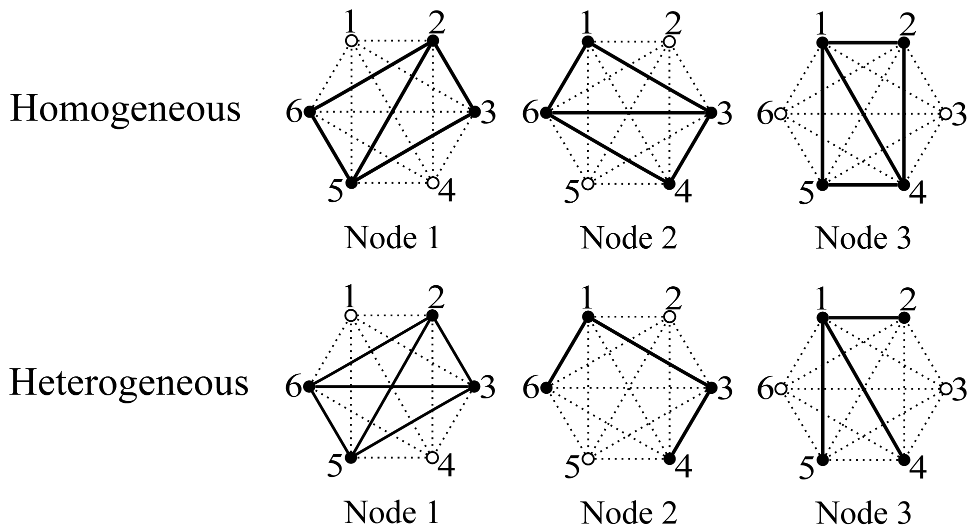


Figure 7.5: Load balancing in a homogeneous system (upper part), and a potential load imbalance in a heterogeneous system for the same data distribution solution (lower part), assuming Node 1 has triple the computing power of Nodes 2 and 3. (In the heterogeneous case not all tasks have been allocated yet.)

The Large Solution Space for Task Scheduling

The problem of scheduling comparison tasks and distributing related data to worker nodes can be treated as a classic problem in combinatorial mathematics: to place M objects into N boxes under certain constraints. The scheduling problem investigated in this chapter has the following characteristics:

1. All comparison tasks are distinguishable. For all-to-all comparison problems, each of the comparison tasks is different and processes a different data pair.

2. All worker nodes are distinguishable in a heterogeneous distributed environment, and

generally have different computing power and storage capacities.

These characteristics must be considered when designing task scheduling and data distribution strategies.

Consider a scheduling problem that allocates Q distinguishable comparison tasks to N distinguishable worker nodes. Each worker node should be allocated at least one comparison task. From combinatorial mathematics, the total number of feasible solutions $P(Q, N)$ is expressed based on the Stirling number $S_t(Q, N)$ [Gould, 1961] as:

$$P(Q, N) = N! S_t(Q, N). \tag{7.1}$$

The Stirling number $S_t(Q, N)$ counts the number of ways to partition a set of Q distinguishable elements into N non-empty subsets:

$$S_t(Q, N) = \frac{1}{N!} \sum_{i=0}^{N} (-1)^i \binom{N}{i} (N - i)^Q, \tag{7.2}$$

where $\binom{N}{i}$ is a binomial coefficient.

Let us consider a special case of $N = 4$. From Equations (7.1) and (7.2), the number of all possible distribution solutions $P(Q, 4)$ is:

$$P(Q, 4) = 3 * 2^{Q+1} - 4 * 3^Q + 4^Q - 4\ . \tag{7.3}$$

$P(Q, 4)$ is shown graphically in Figure 7.6. It can be observed from the figure that the number of possible solutions increases exponentially as the number of tasks increases. For 10 tasks and 4 worker nodes, $P(10, 4)$ is over 800,000! In real-life applications, this means that any non-trivial scenario has hundreds of thousands of possible solutions that would need to be evaluated in order to find an optimal one [Zhang et al., 2014].

This growth trend, for even the trivial case of 4 workers, implies that it is generally impossible to evaluate all possible solutions to find the best answer in a reasonable period of time. Thus, developing heuristic solutions is the only viable approach for all-to-all scheduling problems in heterogeneous environments. This is discussed further below after the problem is formalized in Section 7.2.

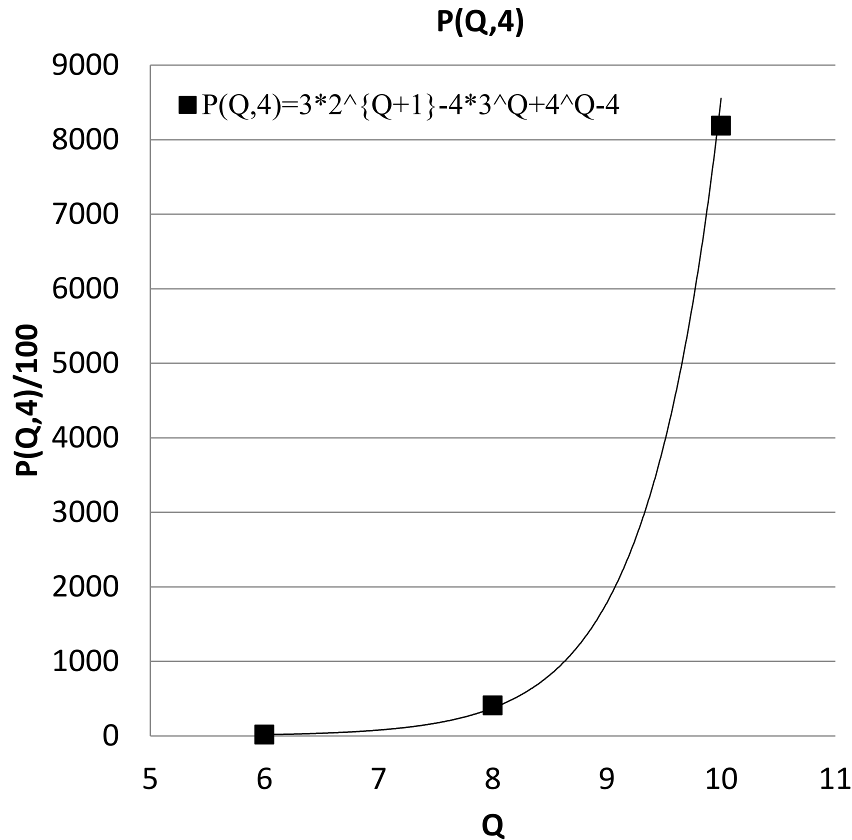


Figure 7.6: The growth trend of the solution space for $P(Q, 4)$, for scenarios with 4 worker nodes and Q tasks to be distributed.

7.2 Formulation for Pre-Scheduling

As discussed in Section 7.1.1, to solve the scheduling of all-to-all comparison problems in heterogeneous distributed systems, data distribution should be task-oriented. Since the location of the distributed data files has a direct impact on task scheduling, here we formalize the requirements for pre-scheduling of data distribution and comparison tasks. Formulating the overall objective and constraints from these requirements, pre-scheduling is defined as a constrained optimization problem.

7.2.1 Overall Considerations and Assumptions

From the two principles developed in Section 7.1.1, the following three aspects must be considered for task-oriented data distribution:

- Time and storage consumption for distributing data. For large-scale all-to-all comparison

problems, distributing data sets among all worker nodes should consider not only the storage used on each node, which must be within its storage capacity, but also the time consumed in distributing the data.

- Execution performance of a single comparison task. Executing a comparison task requires access to and processing of the related pair of data files. Remotely accessing data will delay a task's execution. Hence, data pre-scheduling must co-locate pairs of data files to be compared on a given node, i.e., it must maximize 'data locality' for all comparison tasks.

- Overall execution time performance of the all-to-all comparison problem. In a heterogeneous system, all worker nodes, each with different computing power, perform their own comparison tasks concurrently. The overall comparison problem is completed only when all worker nodes have completed their respective tasks. Therefore, making full use of the computing power for each of the worker nodes and allocating data and tasks to the worker nodes with a minimum load imbalance are critical to improving the overall performance of the computing problem.

These three aspects are formalized in Sections 7.2.2 to 7.2.5.

To design a pre-scheduling algorithm for data distribution and task allocation, two assumptions are made herein:

A1. All data items have the same size; and

A2. All comparison tasks have the same execution time.

Although these assumptions are not necessarily realistic, they are made for easy understanding of the pre-scheduling problem's initial development. In Section 7.4.2, dynamic scheduling is used to allow us to relax these assumptions to support more general and realistic scenarios.

7.2.2 Reducing Time and Storage Consumption

Though time consumption for data distribution is affected by many factors, it is largely proportional to the total size of the data sets to be distributed for a given system configuration. Let $|D_i|$ denote the number of files to be allocated to worker node i and N represent the number of

worker nodes in the system. From Assumption A1 in Section 7.2.1, that all data files have the same size, the time consumption T_{data} for distribution of all data files to N worker nodes can expressed as:

$$T_{data} \propto \sum_{i=1}^{N}(|D_i|) \ . \tag{7.4}$$

For storage usage, each of the worker nodes must be assigned data files within its storage capacity. When the storage usage on every worker node is reduced, the total storage usage in the distributed system can also be reduced.

Considering the goal of reducing the data distribution time and data storage usage, pre-scheduling of data distribution aims to minimize the amount of data stored on any node 1 to N, i.e., the constraint is to:

$$\text{Minimize} \max\{|D_1|, |D_2|, ..., |D_N|\} \ . \tag{7.5}$$

7.2.3 Improving Performance for Individual Tasks

For all-to-all comparison problems, each of the comparison tasks running on the corresponding worker node has to access and process the required data items. The time spent on data processing is problem-specific. It depends on what the specific comparisons are, what algorithms are used and how the algorithm is implemented. The time spent on data access can be minimized to its lowest possible value when all required data items are made available locally.

The requirement for good data locality is formulated as follows. For a worker node i, let D_i and T_i respectively denote the data and task sets allocated to that node. Also, let $C(x, y)$ represent the comparison task for data items x and y. T denotes the set of all comparison tasks of the all-to-all comparison problem and N is the number of worker nodes. Perfect data locality for all comparison tasks is then defined as a situation where each task is allocated to a node containing both data items it needs, i.e.:

$$\begin{aligned} &\forall C(x,y) \in T, \exists i \in \{1, \cdots, N\}, \\ &\quad x \in D_i \wedge y \in D_i \wedge C(x,y) \in T_i \ . \end{aligned} \tag{7.6}$$

7.2.4 Improving Overall Computational Performance

The best possible overall performance of an all-to-all comparison problem in a heterogeneous distributed environment is achieved if all worker nodes are allocated a workload proportional to their respective processing capabilities. This is load balancing requirement means that all worker nodes will complete their respective set of comparison tasks at around the same time.

The requirement for load balancing is formalized as follows. Let $|T_i|$ be the number of pairwise comparison tasks performed by worker node i, and P_i be the processing power of node i. For an all-to-all comparison problem with M data items pairwise compared on N worker nodes, the total number of comparison tasks is $M(M-1)/2$. Load balancing requires the number of comparison tasks allocated to each node to be proportional to its processing power. This is achieved if the tasks are divided as evenly as possible among the nodes, taking into account each node's relative processing power, i.e., for each node i we have

$$|T_i| \leq \left\lceil \frac{P_i}{\sum_{i=1}^{N} P_i} \cdot \frac{M(M-1)}{2} \right\rceil \tag{7.7}$$

where $\lceil \cdot \rceil$ is the ceiling function.

7.2.5 Optimization for Data Pre-Scheduling

With the three requirements described above, the data pre-scheduling problem can be formalized. For the overall system, we aims to achieve the objective in Equation (7.5) while meeting the constraints in Equations (7.6) and (7.7). Minimizing the objective in Equation (7.5) implies saving storage space and data distribution time, while meeting the constraints in Equations (7.6) and (7.7) means improving the computing performance of both individual comparison tasks and also the whole set of tasks.

Therefore, the data pre-scheduling requirement is formalized as the following constrained optimization problem:

$$\begin{aligned} &\text{Minimize} \max\{|D_1|, |D_2|, ..., |D_N|\} \\ &\text{while satisfying Equations (7.6) and (7.7).} \end{aligned} \tag{7.8}$$

Importantly, similar optimization problems have been tackled previously. Examples include

block design [van der Linden et al., 2004] and graph coving [Thite, 2008]. In these optimization problems, Balanced Incomplete Block Design is an NP-hard problem [Corneil and Mathon, 1978] and the graph coving problem is NP-complete [Thite, 2008].

Similar to these known NP problems, our scheduling problem formalized in Equation (7.8) is a typical combinational optimization problem. It faces the challenge of a large number of combinations of data items and related comparison tasks. Finding the best solution from this large solution space is difficult, particularly for heterogeneous distributed systems. In the next section, a metaheuristic strategy is developed to solve this challenge.

7.3 MetaHeuristic Data Pre-Scheduling

In this section, we present a metaheuristic data pre-scheduling strategy based on the simulated annealing (SA) process. To improve the SA's performance, specific methods are developed for generation and selection of solutions. The pre-scheduling strategy is then implemented as an algorithm and its properties are analyzed through an example.

7.3.1 Features of the Optimization Problem

The pre-scheduling optimization problem formulated in Equation (7.8) has the following characteristics:

1. It is not difficult to find a feasible solution that meets both constraints but which does not necessarily optimize the objective in Equation (7.8). For a given all-to-all comparison problem as shown in Figure 7.1, the total number of comparison tasks (edges) is finite and fixed. Any distribution of these tasks and related data files to worker nodes by following these two constraints gives a feasible solution for pre-scheduling.

2. The quality of each solution can be evaluated quantitatively, by checking the two constraints to validate the feasibility of the solution, and calculating and comparing the objectives of two candidate solutions.

3. New solutions can be generated by changing the existing solutions. There are many ways to do so, but a simple strategy is to swap tasks between two worker nodes.

From these features, solving the pre-scheduling optimization problem can begin with a randomly-generated initial solution. Then, we keep generating new solutions and checking their feasibility and their improvement on the objective, retaining improved solutions and discarding others. This is achieved below through metaheuristics and simulated annealing.

7.3.2 Simulated Annealing

Simulated annealing [Kirkpatrick et al., 1983] has been widely used in solving combinatorial optimization problems. Its name derives from annealing in metallurgy. The process imitates the behaviour of physical systems which melt a substance and lower its temperature slowly until it reaches the freezing point.

Let S denote the solution space, i.e., the set of all possible solutions. The objective function is denoted by f, which is defined on members of S. The aim of simulated annealing is to find a solution $S_i \in S$ that minimizes f over S. SA makes use of an iterative improvement procedure determined by neighbourhood generation. Firstly, an initial state is generated. Then, SA generates neighbourhood solutions at each temperature, which is gradually lowered, until a stopping criterion becomes true. The SA procedure is controlled by a group of parameters, which are called a cooling schedule. Theoretically, SA is guaranteed to converge to the global optimum.

While the features of our pre-scheduling optimization problem justify the suitability of SA for solving the problem, the following issues need to be addressed:

- Determine the Annealing and Acceptance Probability modules. For an SA module, a suitable set of parameters of the cooling schedule is important for the efficiency and accuracy of the algorithm. The parameters to be set include the starting temperature, ending temperature, temperature reduction function and termination criterion. In addition, an acceptance probability function must also be chosen for SA to accept an undesirable intermediate state which may lead to a better global solution. In a local optimization algorithm, a new state is accepted when it optimizes the cost function, however, SA can accept an undesirable state based on the value of the chosen acceptance probability function.

- Determine the neighbourhood selection method and fitness equation. As a local searching

technique, within each temperature, SA generates a new neighbourhood solution randomly from the current solution. For each solution, a fitness equation is used to determine the quality of the solution. Both the neighbourhood selection method and fitness function need to be designed to be specific for our pre-scheduling problem.

7.3.3 Annealing Design for All-to-All Pre-Scheduling

In this section we address the issues identified in Section 7.3.2 for our pre-scheduling problem.

Determination of the Annealing Module

For a SA strategy with relatively high starting temperature and slow temperature decreasing, if the ending temperature is low enough, the classic SA strategy can theoretically converge to global optimum but the long searching time also means very poor efficiency for the computation, which can be a serious problem when dealing with a huge solution space. Also, a SA strategy with lower starting temperature and decreases temperature rapidly can reduce the computation time greatly. However, the accuracy of the final solution is not convincing enough because of the insufficient local searching for the neighbourhood solutions within each temperature.

To balance the accuracy of the final solution and the performance of the SA process, here we choose geometric cooling, which is one of the most widely used SA schedules [Cohn and Fielding, 1999] and determine the SA algorithm's parameters specifically for our pre-scheduling problem. At each temperature, a certain number of iterations are carried out to search for more solutions. Table 7.1 shows a choice of parameter settings, which are used in Section 7.3.4 below.

Table 7.1: Cooling parameter settings (k represents the iteration step).

Item	Setting
Temperature decreasing function	$t_{k+1} = 0.99t_k$
Starting temperature	1.0
Ending temperature	10^{-5}
Inner loop iteration threshold	100

Acceptance Probability Function

The second SA module to be designed is the acceptance probability function. Here we choose the Boltzmann function [Keikha, 2011, Pepper et al., 2002], which is defined as

$$P_r(\Delta E) = \exp(-\Delta E/t) \tag{7.9}$$

where $P_r(\cdot)$ is the acceptance probability function, ΔE is the energy difference between the neighbouring state and the current state, and t is the temperature.

The chance of accepting an energy-increasing move, i.e., a solution with poorer quality, decreases with the temperature. This enables SA to escape shallow local minima early on and explore deeper minima more fully as the SA process progresses.

Initial Solution

For the pre-scheduling problem in Equation (7.8), a feasible solution can be derived by following the constraint in Equation (7.7) to allocate comparison tasks and also by following the constraint in Equation (7.6) to distribute data. We use the same notations explained before: M for the number of data files to be processed, N for the number of worker nodes, D_i for the data files stored on node i, and T_i for the task set allocated to node i. In addition, let U denote the set of tasks that have not yet been scheduled. An initial solution can be randomly generated as follows:

1. For each worker node $i \in \{1, 2, \cdots, N\}$, randomly pick comparison tasks from set U and assign them to T_i until $|T_i|$ meets Equation (7.7) or $|U| = 0$. Thus a scheduling of all comparison tasks is obtained: $T = \{T_1, T_2, \cdots, T_N\}$.

2. For each task set $T_i \in T$, schedule all required data files to node i. This forms data set D_i.

Once this is done the set of task and data allocations to the worker nodes, $S = \{(T_1, D_1), (T_2, D_2)$, ..., $(T_N, D_N)\}$, is a feasible solution, which meets both constraints in Equation (7.8).

Such a randomly-generated initial solution S is unlikely to be optimal in general, however, so the SA process improves it step by step until a solution is obtained with acceptable quality.

Neighbourhood Selection Method

The solution neighbourhood defines the way to get from the current solution to another. Our basic idea for generating a new neighbouring solution is to move a comparison task from one node to another or to swap comparison tasks between two nodes. The generated neighbourhoods should cover the whole solution space. Therefore, the new neighbourhood solution S' is generated from current solution S by using the following neighbourhood selection method:

1. Randomly pick two different worker nodes i and j from all worker nodes $1, \cdots, N$.

2. Randomly swap two comparison tasks between Nodes i and j, then update set T_i and T_j.

3. Re-schedule all required data files to node i and j based on sets T_i and T_j.

Using this method, our SA algorithm will have enough randomness to reach an arbitrary point in the neighbouring solution space.

Fitness Equation

A fitness equation is used to determine the quality of candidate solutions. Our solutions are expressed as $S = \{(T_1, D_1), (T_2, D_2), \cdots, (T_N, D_N)\}$ as defined in Section 7.3.3. From the solution's structure, the fitness equation for S can be defined as the set of numbers of data files allocated to each of the worker nodes. Let $F(S)$ denote the fitness of solution S:

$$F(S) = \{|D_1|, |D_2|, \cdots, |D_N|\} . \tag{7.10}$$

The cost difference between two alternative solutions S and S' is calculated as follows. Firstly, the data set sizes in both $F(S)$ and $F(S')$ are sorted in descending order. Then, the cost difference ΔF is defined as:

$$\begin{aligned} \Delta F &= F(S) - F(S') \\ &= \{(|D_1| - |D'_1|), \cdots, (|D_N| - |D'_N|)\} . \end{aligned} \tag{7.11}$$

Finally, the value of the cost difference Δf is defined as:

$$\Delta f = \begin{cases} \text{the } 1^{\text{st}} \text{ non-zero size in } \Delta F, & \text{if any} \\ 0, & \text{otherwise.} \end{cases} \tag{7.12}$$

In this way solutions with smaller differences (Equation (7.11)) are always accepted. Also, in comparison with other methods that only compare the maximum values in $F(s)$ and $F(S')$, our evaluation method makes full use of the information of all elements in $F(S)$ and $F(S')$. The resulting SA algorithm thus has a higher efficiency in moving from one solution to better ones.

Data Pre-Scheduling Algorithm

Algorithm 3 summarizes all the features presented in Sections 7.3.3 to 7.3.3. Our solution uses SA as the underlying optimization technique with consideration of the specific requirements of all-to-all comparison problems. In the algorithm, the temperature decreases from its initial value (Line 3) to the stopping value (Line 4) by following the temperature decreasing function (Line 16). At each temperature, the algorithm generates a number of new solutions (Lines 5 and 7). For each new solution, the algorithm calculates its fitness and the change in the fitness level (Line 9), and then decides whether to accept or discard this new solution (Lines 11 to 13). At the end of the algorithm, the final solution is returned (Line 18).

7.3.4 Analysis of the Data Pre-Scheduling Strategy

To analyze our data pre-scheduling strategy, consider a scenario with 7 data files (numbered from 0 to 6) and 5 heterogeneous worker nodes (labelled from A to E). Assume worker nodes A and B are twice as powerful as the other three. By applying Algorithm 3 to this scenario, a pre-scheduling solution is obtained as shown in Table 7.2.

Table 7.2 shows that with task-oriented data distribution, our pre-scheduling algorithm not only suggests data allocations to the worker nodes but also proposes task assignments corresponding to the data allocations. For data allocation, each worker node stores part of the whole data set. In this example, each worker node is allocated 4 data files. For task assignments, worker nodes A and B are each assigned 6 tasks, which is double the number of tasks given to nodes C, D and E. This is in accordance with the assumed computing power of the workers,

Algorithm 3 Data Pre-Scheduling Algorithm

Initial:

1: Randomly generate initial solution S using the method in Section 7.3.3;
2: Set parameters based on Table 7.1;
3: Set the current temperature t to be the starting temperature.

Pre-Scheduling:

4: **while** The current temperature t is higher than the ending temperature **do**
5:     **while** The iteration step is below the inner loop iteration threshold **do**
6:             (The loop threshold is defined in Table 7.1)
7:         Generate a new solution S' from S: $S' \leftarrow new(S)$
8:             (The method new is described in Section 7.3.3);
9:         Calculate the change of Fitness, Δf, from Equation (7.12)
10:             (Fitness methods for F, ΔF and Δf are given in Section 7.3.3);
11:         **if** $\exp(-\Delta f/t) > random[0, 1)$ **then**
12:             Accept the new Solution: $S \leftarrow S'$
13:         **end if**
14:         Increment the iteration step by 1;
15:     **end while**
16:     Lower the current temperature t based on the function in Table 7.1;
17: **end while**
18: Return final solution S.

Table 7.2: Pre-scheduling of 7 data files to 5 worker nodes.

Node	Distributed data files	Allocated comparison tasks
A	0, 1, 2, 3	(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
B	2, 4, 5, 6	(2, 4) (2, 5) (2, 6) (4, 5) (4, 6) (5, 6)
C	0, 3, 4, 6	(0, 4) (0, 6) (3, 6)
D	0, 1, 3, 5	(0, 5) (1, 5) (3, 5)
E	1, 3, 4, 6	(1, 4) (1, 6) (3, 4)

and thus suggests a good load balance in this heterogeneous environment.

Moreover, in our previous work on heterogeneous systems [Zhang et al., 2015c], we showed that solving the same example using a greedy strategy distributed 5 data files to two of the worker nodes. Thus even in a small case like this our new data pre-scheduling strategy presented here results in better storage outcomes. A further comparison with the heterogeneous case appears in Section 7.5.

Task-oriented data distribution and task assignment as per Table 7.2 also achieves perfect data locality for all comparison tasks. Executing the comparison tasks based on this static task assignment avoids runtime data movement completely, which would be, however, inevitable in a Hadoop-based system.

7.4 Runtime Task Scheduling Design

As per Figure 7.2, two scheduling mechanisms can be used to schedule comparison tasks, static and dynamic. In static task scheduling, the tasks are distributed to worker nodes based on fixed information about the computing power of the worker nodes. Once allocated to a specific worker node, a task will always execute on that node. Static scheduling allows for a simplified system design and minimized runtime communication costs.

In the previous chapters, we have already developed the static task scheduling strategy for all-to-all comparison problem, which can also be used in this chapter.

In practice, however, the properties of computing nodes can change dynamically, especially in situations where the worker nodes are shared with other system users. In comparison with static scheduling, dynamic task scheduling makes decisions about task assignments at run time, allowing the computation to adapt to changes in the computing environment, such as the processing power on a particular node being preempted by other system users. In this section we define the dynamic scheduling of large-scale all-to-all comparison problems.

7.4.1 Initial Task Schedulability

Given the pre-scheduling approach developed in Section 7.3 and the basic principles presented in Section 7.1.1, the runtime task scheduling problem can be defined as the need to find a worker node with all required data items for each comparison task.

Using the notations previously defined, let $C(x, y)$ represent the comparison task between data items x and y and D_i is the data set stored on worker node i. Then L_c is the set of all worker nodes available to execute task $C(x, y)$:

$$L_c = \{i \mid x \in D_i \wedge y \in D_i\} \ . \tag{7.13}$$

From Equation (7.6), for each comparison task $C(x, y)$, node set L_c is guaranteed to be non-empty after distributing all data items by using our pre-scheduling strategy, thereby ensuring the schedulability of all comparison tasks.

7.4.2 Runtime Dynamic Scheduling

Our design for runtime dynamic task scheduling also follows the data pre-scheduling approach from Section 7.3. To give an insight into the dynamic scheduling strategy, consider the following simple example. Assume an all-to-all comparison problem with 10 data items (numbered from 0 to 9) to be pairwise compared on 4 worker nodes. Figure 7.7 shows a feasible data distribution solution. Nodes 1 to 4 are allocated data items $(0, 1, 4, 5, 6, 8, 9)$, $(0, 1, 2, 4, 5, 7, 9)$, $(1, 2, 3, 6, 7, 8, 9)$ and $(0, 2, 3, 4, 5, 6, 7)$, respectively.

Comparison tasks for each worker node are also shown in Figure 7.7, in which the number in each cell represents how many worker nodes can perform that particular comparison task. For example, comparison task $(0, 4)$ between data items 0 and 4, can be executed on Nodes 1, 2 or 4, so in each of the matrices corresponding to these nodes the number 3 appears in the cell for this comparison. With such a range of options, dynamic task scheduling then needs to decide to which node each specific comparison task is allocated.

Flexibility for Dynamic Scheduling

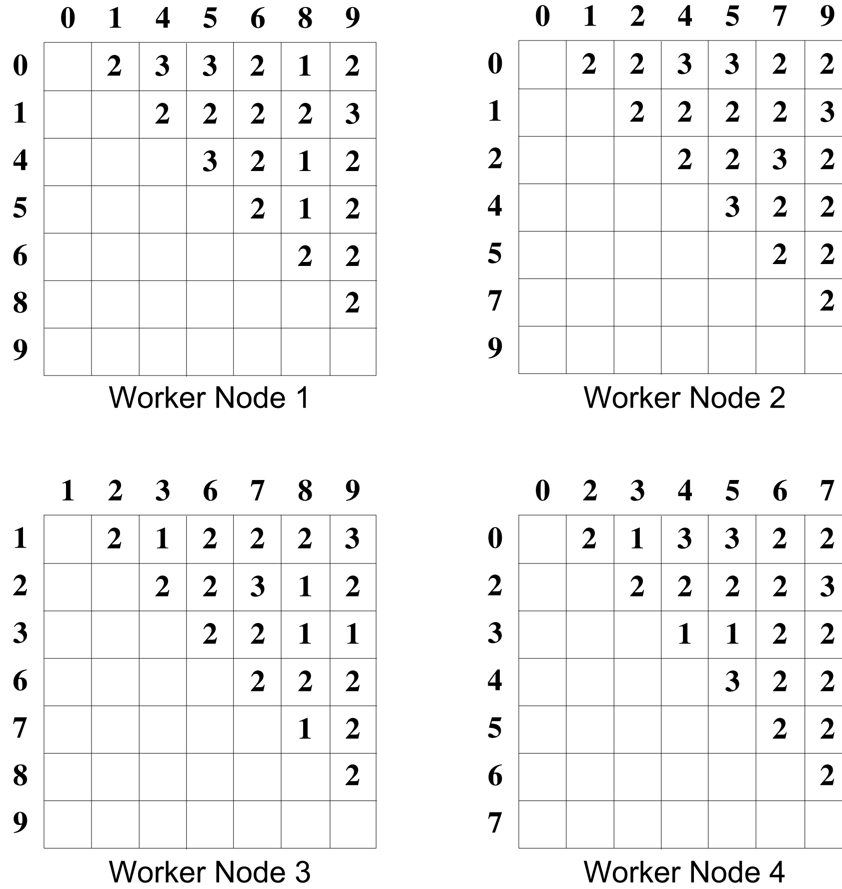
As shown in Figure 7.7, the location of all the data items are fixed by our pre-schedule, so there is only a limited number of worker nodes that can execute each comparison task.

Let $|L(c)|$ denote the number of worker nodes that can execute comparison task c with data locality. Set U represents all comparison tasks that have not yet been scheduled in the system. We quantify the 'flexibility' of the schedule as:

$$F_\ell = \sum_{c \in U} |L(c)| \ . \tag{7.14}$$

The highest flexibility occurs when any comparison task can be executed on any worker node. In this extreme situation, the system is able to easily resolve any load imbalance at runtime. In contrast, if each of the comparison tasks can only execute on one worker node, the system has no flexibility to make any runtime adjustment for load balancing.

For all-to-all comparison problems, the initial system flexibility is determined by our data pre-scheduling before the computation begins. Thus, runtime dynamic task scheduling focuses on allocating comparison tasks that have not yet started to execute. The number of these tasks



Worker Node 1

	0	1	4	5	6	8	9
0		2	3	3	2	1	2
1			2	2	2	2	3
4				3	2	1	2
5					2	1	2
6						2	2
8							2
9							

Worker Node 2

	0	1	2	4	5	7	9
0		2	2	3	3	2	2
1			2	2	2	2	3
2				2	2	3	2
4					3	2	2
5						2	2
7							2
9							

Worker Node 3

	1	2	3	6	7	8	9
1		2	1	2	2	2	3
2			2	2	3	1	2
3				2	2	1	1
6					2	2	2
7						1	2
8							2
9							

Worker Node 4

	0	2	3	4	5	6	7
0		2	1	3	3	2	2
2			2	2	2	2	3
3				1	1	2	2
4					3	2	2
5						2	2
6							2
7							

Figure 7.7: All possible comparison tasks for each of 4 worker nodes. The row and column numbers identify the data files stored on each node. The numbers in the cells represent how many worker nodes can perform the specific comparison between two such files.

decreases as the computation progresses.

Let set G_i represent those comparison tasks that can be executed by worker node i and set U be all comparison tasks that have not yet been scheduled or have not started to execute in the system. When a comparison task c is scheduled to worker node i, the resulting change in the system's flexibility is denoted by ΔF_ℓ. We have:

$$\Delta F_\ell = |L(c)| \,, \text{ where } c \in U, c \in G_i \,. \tag{7.15}$$

How to manipulate ΔF_ℓ for dynamic task scheduling determines the scheduling priorities. Two typical examples are to maximize ΔF_ℓ and or to minimize ΔF_ℓ.

Maximising Flexibility First (MaxFF). As shown in the upper part of Figure 7.8, the comparison tasks with the most suitable worker nodes are scheduled in this strategy (maximum ΔF_ℓ).

At the beginning, all worker nodes can get enough workload. However, as the computation progresses, the system's imbalance cannot be resolved. Though some worker nodes have idle computing resources (e.g., Node 2), the remaining comparison tasks cannot be scheduled to them because they cannot execute on those nodes.

Minimising Flexibility First (MinFF). The lower part of Figure 7.8 shows that the system can start with allocating and executing the comparison tasks with the least number of suitable worker nodes (minimum ΔF_ℓ). As the computation proceeds, the remaining comparison tasks have the flexibility to be assigned to more suitable worker nodes. This enables the system to have enough flexibility to deal with load imbalances at runtime.

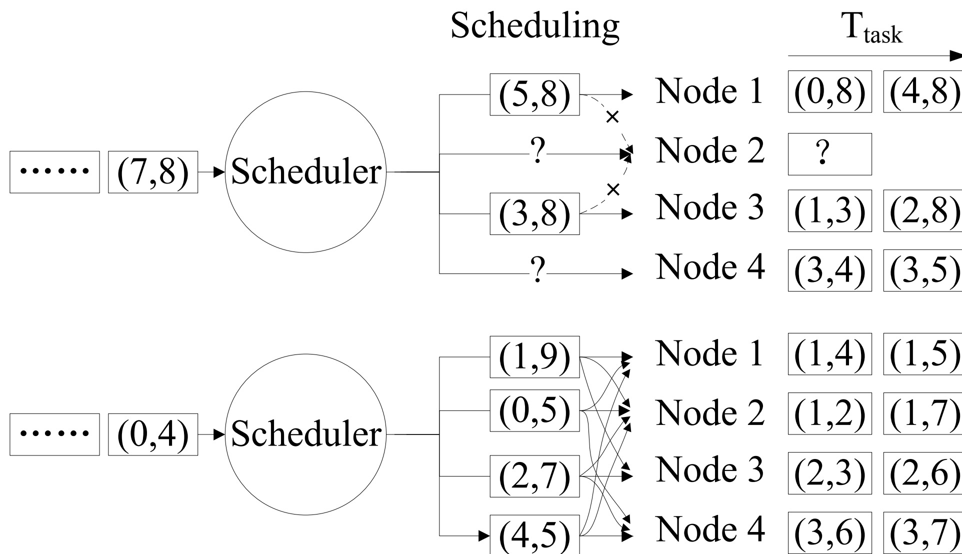


Figure 7.8: MaxFF (upper part) and MinFF (lower part) dynamic scheduling strategies.

Dynamic Scheduling Design

The aim of dynamic scheduling is to minimize the change in the system's flexibility after scheduling each comparison task while avoiding any remote data accesses. This is expressed as:

$$\text{Minimize } \Delta F_\ell \,. \tag{7.16}$$

Using previously defined notations, let G_i represent the comparison tasks that can execute on worker node i, set U be all comparison tasks that have not been scheduled or have not start to execute, and $|L(c)|$ be the number of worker nodes that can perform comparison task c. The

objective in Equation (7.16) can be achieved by choosing a task c using the following rule:

$$|L(c)| = \min\{|L(i)| \mid i \in U \wedge i \in G_i\}. \tag{7.17}$$

Accordingly, our dynamic scheduling strategy is designed in Algorithm 4.

Algorithm 4 Dynamic Task Scheduling

Initial:

1: Set U composed of all the unscheduled comparison tasks;
2: Set G_i composed of all comparison tasks that can be executed by each node i;
3: $|L(c)|$ for each comparison task c in set U.

Dynamic Scheduling:

4: **while** There are unscheduled tasks in set U and
5: a worker node i asks for a comparison task **do**
6: **while** There are unscheduled tasks in set U and
7: enough available computing resources on node i **do**
8: Pick a comparison task c from U, satisfying Equation (7.17);
9: Assign this task c to node i;
10: Mark this comparison task c as scheduled and update set U;
11: Update the available computing resources on this node i;
12: **end while**
13: **end while**

Figure 7.9 shows the dynamic scheduling mechanism provided by our distributed computing framework in Chapter 3. In our distributed computing framework, when a worker node in the system asks for comparison tasks, the master node will determine and schedule suitable comparison tasks based on Algorithm 4.

The effectiveness of this dynamic scheduling strategy is demonstrated below through experimental studies.

7.5 Experiments

In this section we evaluate the performance of our data-aware task scheduling approach for large-scale all-to-all comparison problems in heterogeneous distributed systems. The performance evaluation was conducted from four main perspectives: storage saving, data locality, execution time and scalability.

Storage Saving. Storage savings were measured in our experiments by using a group of experiments with different numbers of storage nodes in heterogeneous environments. The

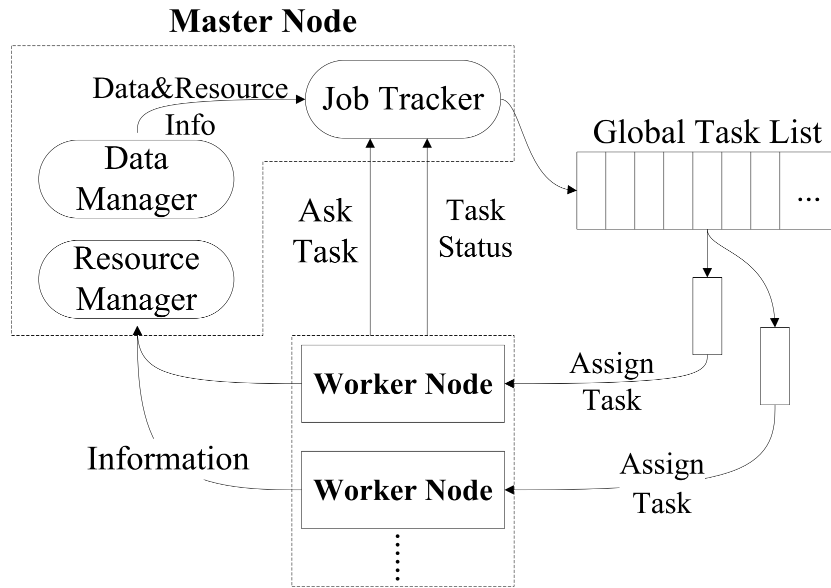


Figure 7.9: Dynamic scheduling process in our distributed computing framework.

results from our approach were also compared with those from Hadoop's data distribution strategy.

Data locality. Our task-oriented data pre-scheduling aims to achieve good data locality. By meeting the constraint in Equation (7.6), all data files are distributed to the worker nodes to allow local accessibility of data pairs for comparisons. The data locality performance of our pre-scheduling approach was compared with that of the widely used Hadoop data distribution strategy.

Execution Time Performance. Execution time performance was measured for distributing data and pairwise comparing data items, respectively. The time spent on comparison tasks was evaluated for individual tasks, individual worker nodes, and all worker nodes as a whole. Considering both time for data distribution and time for task execution, the total execution time of the all-to-all comparison problem could be evaluated. The results were compared with those from the Hadoop framework.

Scalability. Scalability is one of the main issues that have to be addressed in large-scale distributed computing. Experiments at different scales were carried out to evaluate the scalability of our data-aware task scheduling approach. Changes in the problem's scale included the numbers of input files and worker nodes.

Table 7.3: Storage and data locality of the heterogeneous approach in this chapter, our previous work on heterogeneous systems, and Hadoop(3) for $M = 256$ files and N varying from 8 to 64. Half of the nodes had twice the computing power of the other half. The number of data replications in Hadoop was set to be the default value 3.

	N	8	16	32	64
$\max\{\vert D_1, \cdots, \vert D_N\vert\}$:					
	This chapter	162	128	94	60
	Previous work	207	163	113	79
	Hadoop(3)	96	48	24	12
Storage saving (%):	This chapter	37	50	63	77
	Previous work	19	36	56	69
	Hadoop(3)	63	81	91	95
Data locality (%):	This chapter	100	100	100	100
	Previous work	100	100	100	100
	Hadoop(3)	48	28	14	7

Table 7.4: Storage and data locality of our heterogeneous approach and Hadoop(variable r) for $M = 256$ files and N varying from 8 to 64. Half of the nodes had twice computing power than the other half. The number of data replications r for Hadoop was tuned manually for each case to achieve a similar maximum number of files on each node.

	N	8	16	32	64
	Setting of x in Hadoop(x)	6	9	12	15
$\max\{\vert D_1, \cdots, \vert D_N\vert\}$:					
	This chapter	162	128	94	60
	Hadoop(r)	192	144	96	60
Storage saving (%):	This chapter	37	50	63	77
	Hadoop(r)	25	44	63	77
Data locality (%):	This chapter	100	100	100	100
	Hadoop(r)	52	38	26	20

7.5.1 Storage Saving and Data Locality

In our experiments, an all-to-all comparison problem with 256 data files was investigated. The distributed system was composed of multiple worker nodes with the number of nodes varying from 8 up to 64. These nodes were grouped into two halves: each of the two halves had the same computing power, but one half had twice more computing power than the other. For comparisons with Hadoop's framework, the number of data replications was set to be the default value 3 unless otherwise specified.

With the increase in the number of worker nodes, Table 7.3 shows the storage savings as a percentage for the data pre-scheduling from this paper, our previous work on heterogeneous systems [Zhang et al., 2014] and Hadoop's data distribution strategy. It can be seen from Table 7.3 that all three strategies save much storage space (Comparing to put all data to all nodes), implying a lower time cost on data distribution. This is important especially for big data problems with a large number of worker nodes. For instance, for a distributed system with 64 worker nodes, the storage saving reaches as high as over three quarters (77%) for our data pre-scheduling strategy. It is even as high as 95% for Hadoop.

However, Hadoop is designed without consideration of data locality for comparison tasks. Therefore, a large number of comparison tasks have to be executed with remote data access. This causes runtime data re-arrangement and significant computing performance degradation. For example, it can be seen from Table 7.3 that only 7% of comparison tasks have data locality in a system with 64 nodes, implying that 93% of the comparison tasks rely on remote data access to execute. By comparison, for our data pre-scheduling strategy, which meets the constraint in Equation (7.6), all comparison tasks have perfect data locality for any number of worker nodes.

Moreover, compared to our previous work on heterogeneous systems [Zhang et al., 2014], the approach in this chapter achieves better results. For instance, $(79 - 60)/79 = 24\%$ more storage savings can be made in a system with 64 nodes, which means much less storage usage and predictable performance improvement.

Table 7.5: T-test at a 5% significance level for the results of our approach for the experiments shown in Table 7.3 (mean values from 10 runs for all cases).

N	$\overline{\max\{\lvert D_1\rvert, \cdots, \lvert D_N\rvert\}}$	Pass	Ci	Variance
8	162	Yes	[159.2392 162.9608]	6.766667
16	128	Yes	[127.9879 129.6121]	1.288889
32	94	Yes	[93.87054 95.52946]	1.344444
64	60	Yes	[59.76032 61.43968]	1.377778

Beside this, quantitative t-test results using the R language show that for each of the four cases, the null hypothesis that the results come from a normal distribution with the mean value shown in Table 7.5 cannot be rejected at the significance level of 5%. The parameter Ci in Table 7.5 indicates the region into which 95% of the results will fall. It further demonstrates that 95% of the results in each case fall into a small region around the mean value, implying

that consistent results can be obtained by using our approach.

One might argue that storing more data copies in the distributed system using the Hadoop framework would solve the problem of the lack of data locality. However, the experimental results in Table 7.4 do not support this argument. Instead, they show the inefficiency of arbitrarily increasing and distributing data copies to the worker nodes. For Hadoop's data strategy, the number of data replications was manually tuned to achieve a similar maximum number of data files at each node when compared to our data distribution strategy. For a system with 64 worker nodes, even with a complicated tuning, Hadoop achieved only as low as 20% data locality. Even then the effort of manually tuning Hadoop is prohibitive and the flexibility of using the Hadoop framework is lost.

7.5.2 Execution Time Performance

To evaluate the computation's performance, experiments were designed with the following settings:

(1) Distributed computing system. A heterogeneous Linux cluster was built with 9 servers, which all run 64-bit Redhat Enterprise Linux. One node acted as the master, and the other eight were workers. Among the eight worker nodes, four had dual cores and 64 GB RAM memory, and the other four had a single core and 32 GB RAM memory.

(2) Domain application. As a typical all-to-all comparison problem, the bioinformatics CVTree problem [Hao et al., 2003] was chosen for our experiments. The computation of the CVTree problem has been recently investigated in single computer platforms [Krishnajith et al., 2013, 2014]. We re-programmed the problem in our experiments for distributed computing. For comparison, a sequential version of the CVTree program was also developed for our experiments.

(3) Experimental data. A set of dsDNA files from the National Center for Biotechnology Information (NCBI) [NCBI, 1988] was chosen as the input data for the CVTree problem. The size of each data file was around 150 MB, and over 20 GB of data in total were used in the experiments.

(4) Experimental scenarios. Tables 7.3 and 7.4 compare the storage savings and data locality

Table 7.6: Experimental scenarios for the CVTree problem with $N = 8$ and different M values.

M	$\max\{\lvert D_1\rvert, \cdots, \lvert D_8\rvert\}$		
	This chapter	Hadoop(3)	Hadoop(6)
78	53	30	59
93	68	35	70
109	79	41	82
124	90	47	93

between our data distribution strategy and Hadoop's under different settings for general all-to-all comparison problems. For the specific CVTree problem, Table 7.6 further compares our data distribution strategy, Hadoop(3) and Hadoop(6). Hadoop(r) means the Hadoop data distribution strategy with r data replications. It can be seen from Table 7.6 that Hadoop(6) distributes more data files to each worker node than our approach.

(5) To demonstrate our data distribution and task scheduling strategies, four different solutions were designed, which were quantitatively compared through the CVTree problem:

1. The Hadoop(3) solution using Hadoop's data distribution (3 data replications) and task scheduling;
2. The Hadoop(6) solution;
3. Our solution using our data pre-scheduling approach and static task scheduling; and
4. Our solution using our data pre-scheduling approach and dynamic task scheduling.

Figure 7.10 compares the total execution time performance T_{total} from the four solutions to the CVTree problem. In Figure 7.10, each bar chart is composed of two parts: the lower part represents the time T_{data} spent on data distribution, while the upper part represents the time T_{task} spent on comparison task execution. The height of the bar represents the total execution time T_{total} of the CVTree problem for the specific solution: $T_{total} = T_{data} + T_{task}$. The first observation from Figure 7.10 is that both our static and dynamic solutions outperform the two Hadoop solutions significantly in terms of overall performance T_{total}. The second observation is that increasing the number of data replications from 3 to 6 in Hadoop improves T_{total} but at a cost of increased storage demands. However, the improved performance is still far behind our static and dynamic solutions. This is mainly due to our solutions' perfect data locality, which

Hadoop's solutions do not have. The third observation from Figure 7.10 is that our dynamic solution behaves with better T_{total} performance than the static one, as expected.

The dynamic scheduling strategy proposed in Subsection 7.4.2 achieves better performance than the static scheduling strategy shown in Section 7.3 mainly because different comparison tasks may cost different execution time in practical, which makes the theoretical static load balancing hard to be promised. Figure 7.11 points out that to make all the worker nodes finish at the similar time, some worker nodes may need more or less number of comparison tasks than the static allocation, which is considered in our dynamic scheduling strategy.

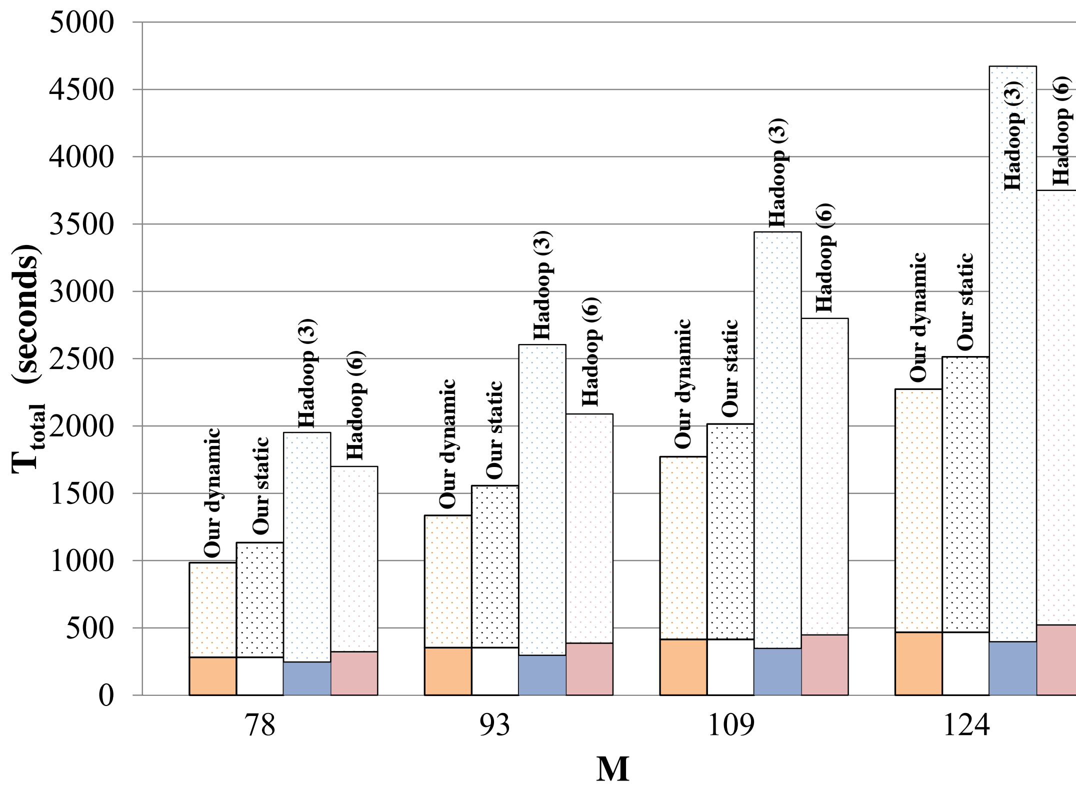


Figure 7.10: Total execution time T_{total} performance for all four solutions.

Figure 7.11 shows the numbers of comparison tasks allocated to each worker node from all four solutions. As half of the 8 nodes (numbered 5 to 8) have twice the processing power of the other half (numbered 1 to 4), both our static and dynamic solutions allocate roughly twice as many comparisons to the four high-performance nodes as to the other four nodes. The static schedule allocates comparison tasks purely based on prior knowledge of the system resources, while the dynamic solution uses real-time information of the system's state. In comparison with our solutions, both Hadoop(3) and Hadoop(6) solutions do not differentiate high-performance and low-performance nodes directly in task scheduling. As a result, the numbers of tasks

Figure 7.11: The number of comparison tasks completed on each node. Nodes 1 to 4 have twice the computing power of Nodes 5 to 8.

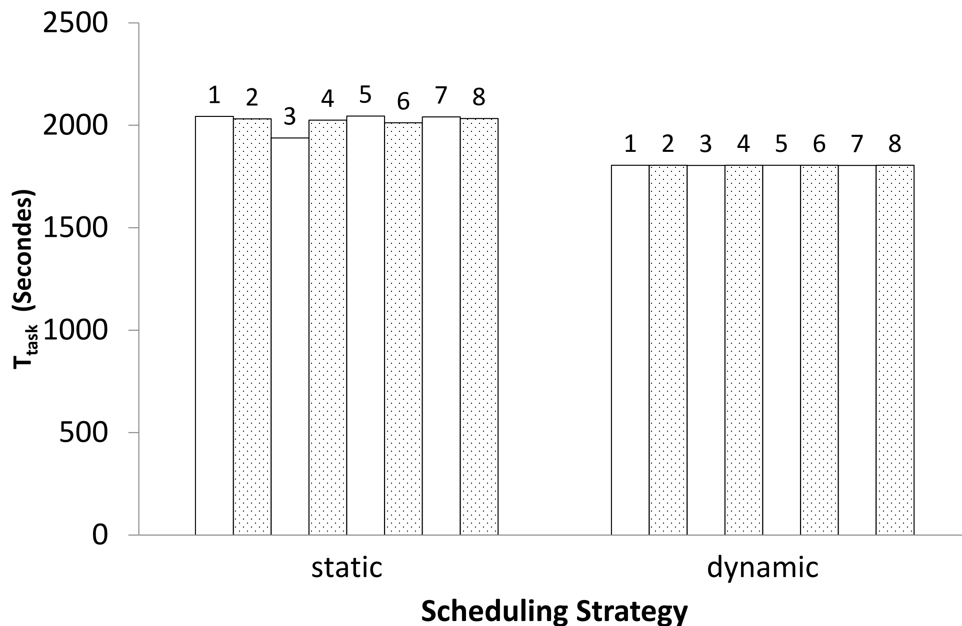


Figure 7.12: Task performance T_{task} of our static and dynamic solutions.

allocated tasks by Hadoop is quite different from those from our solutions, leading to poor load balancing and the need for remote data access. This is clearly shown in Figure 7.11.

Comparisons of the task execution performance between our static and dynamic solutions are given in Figure 7.12. It can be seen from this figure that for either the static or dynamic

solution, all nodes complete their respective comparison tasks at a similar time with small deviations, implying good load balancing. But the deviations in task execution time are smaller in the dynamic solution than in the static solution. This means that the dynamic solution balances the load better than the static one. It can also be seen from Figure 7.12 that the dynamic solution completes all comparison tasks around 10% faster than the static solution. This mainly results from the fact that the dynamic solution makes use of real-time information of the system for better task scheduling and execution, while the static one does not.

7.5.3 Scalability

Scalability characterizes the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [Bondi, 2000]. Scalability is a critical requirement to support distributed computing of large-scale all-to-all comparison problems with big data sets. It was evaluated in our experiments by using the speed-up metric.

Let $time(n, M)$ denote the time required by an n-processor system to execute a program to solve a problem of size M. Then, $time(1, M)$ is the time required by a sequential version of the program. Speed-up is measured as [Mark, 1990]:

$$Speedup(n, M) = \frac{time(1, M)}{time(n, M)} . \tag{7.18}$$

The dotted line in Figure 7.13 shows the ideal 1:1 linear speed-up which could be achieved if communication overheads, load imbalances and extra computation effort were not considered [Li et al., 1999]. In practice, of course, no solution can achieve this. For the CVTree problem investigated in our experiments, the actual speed-up achieved by our data pre-scheduling and dynamic task scheduling is also depicted in Figure 7.13. It increases linearly with the increase in the number of processors available for the computation of the problem, indicating good scalability of our scheduling approach in distributed computing of all-to-all comparison problems.

It is also worth mentioning that despite the inevitable overheads due to network communications, extra memory demands and disk accesses, our heterogeneous scheduling approach achieved about 93% of the ideal 1:1 linear speed-up performance. The was measured at 12

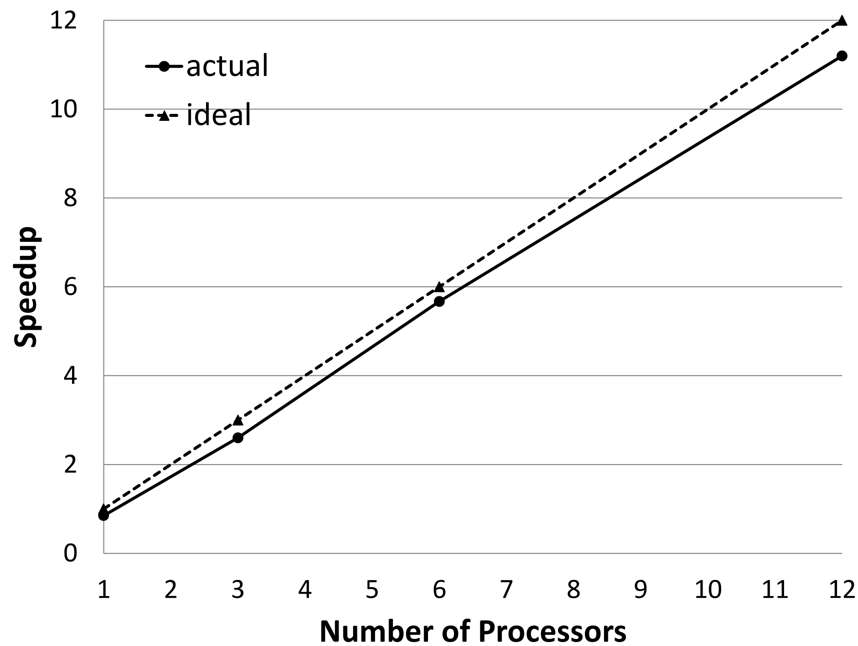


Figure 7.13: Speed-up achieved by our dynamic scheduling algorithm for the CVTree problem compared to the theoretical ideal.

cores as $11.2/12 = 93\%$ from the results shown in Figure 7.13.

7.6 Conclusion

We have presented a data-aware task scheduling approach for distributed computing of large-scale all-to-all comparison problems with big data sets in heterogeneous environments. It was developed from a formalization of the requirements for storage saving, data locality and load balancing as a constrained optimization problem. To solve this optimization problem, metaheuristic scheduling was used for task-oriented pre-scheduling of data distribution. Then, static and runtime dynamic scheduling strategies were developed for allocation of comparison tasks to worker nodes. Experiments have shown that our data-aware task scheduling approach achieves advantages of storage savings, perfect data locality for comparison tasks, improved total execution time performance, and good scalability.

Chapter 8

Conclusion

8.1 Summary

All-to-all comparison is a type of computing problem with a unique pairwise computing pattern. It involves comparing two different data items from a data set for all possible pairs of data items. All-to-all comparison problems are widely found in various application domains such as bioinformatics, biometrics and data mining.

The performance of distributed processing ATAC problems largely depends on the data distribution, task decomposition, and task scheduling strategies. For existing solutions, significantly degraded performance is inevitable due to inappropriate data distribution, poor data locality for the computing tasks, and unbalanced computational loads among the distributed worker nodes. Moreover, to develop ATAC applications running in distributed systems, distributed system issues such as data management, network communication and load balancing have to be considered, which bring heavy burdens to developers without experts experiences in distributed systems.

Hence, in this thesis, a high performance and scalable distributed computing framework for all-to-all comparison problems with big data sets is developed. By improving the performance of data distribution and task execution, this thesis has proposed the following distinctive contributions:

1. Simple and powerful front-end interfaces. For framework users, operation interfaces are developed for uploading applications, data sets and collecting comparison results. All

the distributed system implementation issues are hidden from users, which makes the distributed computing framework can be easily used by users without any distributed system experiences.

Besides this, a distributed programming model for all-to-all comparison problems has been designed. It separates the front-end interfaces and back-end implementation systems and provides powerful application programming interfaces (APIs) for developers. By using four APIs developed in this thesis, developers only have to focus on implementing specific all-to-all comparison algorithms.

2. High performance and scalable data distributed strategies. In this thesis, the task oriented data distribution strategies both for homogeneous and heterogeneous distributed systems are developed. By considering data locality, load balancing and storage usage, heuristic strategies with greedy idea and metaheuristic strategies with simulated annealing are developed. Experiments show great performance improvement and high scalability for our data distributed strategies.

3. Static and dynamic task scheduling strategies. In this thesis, followed by our data distribution strategies, both the static and dynamic task scheduling strategies are developed to achieve system load balancing. In the static scheduling part, all the comparison tasks are scheduled based on the pre-allocation during data deployment. Though static load balancing methods are nonpreemptive, it can minimize the communication delays in the system.

   Beside this, dynamic task scheduling strategy is also developed to adaptive the resource changing during computation. In the dynamic scheduling design, all the comparison tasks are scheduled by considering the flexibility of the distributed systems. In this way, the change in the system's flexibility after scheduling each comparison task can be minimised and remote data accessing can be avoided. Experiments in Chapter 7 show high performance is achieved by using our scheduling strategies.

Comparing to other researches for solving all-to-all comparison problems, the solution in this thesis has the following advantages:

1. Designed for general all-to-all comparison problems. Unlike other specific solutions, our distributed computing framework provides a high performance solution for general

all-to-all comparison computing problems. For computing problems with all-to-all comparison pattern, applications can be implemented by using our application programming interfaces (APIs) and automatically executed based on the back-end computing systems.

2. Designed for general distributed systems. The strategies for data distribution and task scheduling in this thesis are developed without any specific requirements on system architectures, which make them support general distributed computing systems. Comparing to many strategies, our solution also consider the heterogeneous features of the distributed systems, which make it be able to achieve high performance in heterogeneous environment.

3. Data distribution with the consideration of task execution. Distributing and processing data files are the two key phases in solving all-to-all comparison problems. In this thesis, the data distribution and task execution are considered together, which makes the deployment of all the data files can not only save storage space but also improve the following computation performance. Beside this, scheduling comparison tasks based on the result of data distribution can also make full use of the good qualities generated by using our data distribution strategy.

8.2 Limitations

The thesis has some limitations that have not been investigated. These limitations are beyond the scope of this research work.

1. System Reliability is not fully considered in this work. Reliability is an important quality for large scale distributed computing systems. Both the hardware and software failure can happen during the computation phase. While in this thesis, we mainly focus on improving the performance of solving all-to-all comparison problems.

2. Incremental data distribution is not considered in this work. Our research is based on the scenario that the all-to-all comparison problem needs to be solved is given first. Thus, our work in this thesis mainly focuses on improving the overall performance of processing all-to-all comparison problems in distributed systems. However, our current work is easily to be extended to support adding extra data items to the existing data distribution solutions..

8.3 Future Work

There are several interesting directions that this study could be extended to follow in the future.

Firstly, for some scenarios, users may want to add more data files into the distributed computing systems and combine with the exiting distributed data files. The data distribution strategies developed in this thesis can be straightforward extended to support this requirement.

Secondly, as a data intensive and computing intensive problems, many all-to-all comparison problems require frequent memory operations and huge memory space. Hence, one way to accelerate the computation is to optimize the in-memory operations. The current strategies for data distribution and task scheduling are file and hard disk based. Hence, the memory-based strategies for all-to-all comparison problems can be further considered in the future work.

Appendix A

Codes for ATAC applications

A.1 CVTree Application Codes

To develop CVTree application, as we mentioned in Chapter 3, Compare method needs to be implemented by the developer. The simple codes for CVTree application is shown as follows:

```
public class Cvtreesystem extends
   ComputingSystem<String,String,List<Object>,Double>{
  /*
   * Read all the files in folder
      hdfs://192.168.11.188:9000/root/cvtree/input1 and generate a
      Map saving all the data;
   * the keys are file name, the values are the whole file content
   * @see ComputingSystem#Reader()
   */

  public HashMap<String,String> Reader(){
    String path="/home/yifan/Public/part/";
      HashMap<String,String> pairsforpreprocess = new
         HashMap<String,String>();
      File inputfiles = new File(path);
      String[] filename = inputfiles.list();
      File[] files= inputfiles.listFiles();

      for(int i=0;i<files.length;i++){
         StringBuffer wholeContent = new StringBuffer();
       try{
           FileReader freader = new FileReader(files[i]);
           BufferedReader buffreader = new BufferedReader(freader);
         while((buffreader.ready())){
              wholeContent.append((char)buffreader.read());
           }
         freader.close();
           buffreader.close();
         }
           catch(Exception e){
              e.printStackTrace();
           }
         pairsforpreprocess.put(filename[i],wholeContent.toString());
      }
      return pairsforpreprocess;
```

```
32     }
33
34
35     /*
36      * the following methods are belonging to the Preprocess part,
          including the Preprocess method and other methods
37      * other methods are used to do the preprocess;
38      */
39
40    void Init(){
41      M2 = 1;
42      for (int i=0; i<LEN-2; i++) // M2 = AA_NUMBER ^ (LEN-2);
43        M2 *= AA_NUMBER;
44      M1 = M2 * AA_NUMBER;   // M1 = AA_NUMBER ^ (LEN-1);
45      M = M1 *AA_NUMBER;    // M = AA_NUMBER ^ (LEN);
46    }
47
48     void InitVectors(){
49      vector = new int[M];
50      second = new int[M1];
51      one_l= new int[AA_NUMBER];
52      total = 0;
53      total_l = 0;
54      complement = 0;
55    }
56
57     void init_buffer(char[] buffer){
58      complement++;
59      indexs = 0;
60      for (int i=0; i<LEN-1; i++){
61        int enc = encode(buffer[i]);
62        one_l[enc]++;
63        total_l++;
64        indexs = indexs * AA_NUMBER + enc;
65      }
66      second[indexs]++;
67    }
68
69     void cont_buffer(char ch){
70      int enc = encode(ch);
71      one_l[enc]++;
72      total_l++;
73      int index = indexs * AA_NUMBER + enc;
74      vector[index]++;
75      total++;
76      indexs = (indexs % M2) * AA_NUMBER + enc;
77      second[indexs]++;
78    }
79
80     int encode(char ch){
81      return code[ch-'A'];
82     }
83
84     /*
85      * Preprocess method, generate a list including all the values,
          values can be different type;
86      * @see ComputingSystem#Preprocess(java.lang.Object)
87      */
88     public List<Object> Preprocess(String content){
89      Init();
90        InitVectors();
91        StringReader in = new StringReader(content);
92
93      try{
```

```
94        int ch=0;
95
96       while ((ch = in.read())!=-1){
97         if (ch == '>'){
98           while ((in.read()) != '\n'); // skip rest of line
99           char[] buffer = new char[LEN-1];
100          in.read(buffer,0,LEN-1);
101          init_buffer(buffer);
102        }
103        else if (ch != '\r' && ch != '\n')
104          cont_buffer((char)ch);
105      }
106      in.close();
107      }
108      catch(IOException e){
109        e.printStackTrace();
110      }
111
112       int total_plus_complement = total + complement;
113      double total_div_2 = total * 0.5;
114      int i_mod_aa_number = 0;
115      int i_div_aa_number = 0;
116      int i_mod_M1 = 0;
117      int i_div_M1 = 0;
118
119      double[] one_l_div_total = new double[AA_NUMBER];
120      for (int i=0; i<AA_NUMBER; i++)
121        one_l_div_total[i] = (double)one_l[i] / total_l;
122
123      double[] second_div_total = new double[M1];
124      for (int i=0; i<M1; i++)
125        second_div_total[i] = (double)second[i] /
            total_plus_complement;
126
127      one_l=null;
128      second = null;
129      count = 0;
130      double[] t = new double[M];
131
132      for(int i=0; i<M; i++){
133        double p1 = second_div_total[i_div_aa_number];
134        double p2 = one_l_div_total[i_mod_aa_number];
135        double p3 = second_div_total[i_mod_M1];
136        double p4 = one_l_div_total[i_div_M1];
137        double stochastic = (p1 * p2 + p3 * p4) * total_div_2;
138
139        if (i_mod_aa_number == AA_NUMBER-1){
140          i_mod_aa_number = 0;
141          i_div_aa_number++;
142        }
143        else
144          i_mod_aa_number++;
145
146        if (i_mod_M1 == M1-1){
147          i_mod_M1 = 0;
148          i_div_M1++;
149        }
150        else
151          i_mod_M1++;
152
153        if (stochastic > EPSILON) {
154          t[i] = (vector[i] - stochastic) / stochastic;
155          count++;
156        }
```

```
157         else
158            t[i] = 0;
159      }
160
161      one_l_div_total=null;
162      second_div_total=null;
163      vector=null;
164
165      tv = new double[count];
166      ti = new int[count];
167
168      int pos = 0;
169      for (int i=0; i<M; i++){
170         if (t[i] != 0){
171            tv[pos] = t[i];
172            ti[pos] = i;
173            pos++;
174         }
175      }
176
177      t=null;
178
179        List<Object> intermediateData = new ArrayList<Object>();
180
181        intermediateData.add(count);
182        for(int m=0;m<count;m++)
183           intermediateData.add(tv[m]);
184        for(int n=0;n<count;n++)
185           intermediateData.add(ti[n]);
186
187        tv=null;
188        ti=null;
189
190        return intermediateData;
191   }
192
193
194     /*
195      * the following methods are belonging to the Compare part;
196      * first few methods define how to deal the List and get separate
           values;
197      */
198
199     int getcount(List<Object> list){
200        int thecount;
201        thecount = (int)list.get(0);
202        return thecount;
203     }
204
205     double[] gettv(List<Object> list){
206        int thecount;
207        thecount = getcount(list);
208        double[] tv = new double[thecount];
209
210        for(int i=0;i<thecount;i++)
211           tv[i]=(double)list.get(i+1);
212        return tv;
213     }
214
215     int[] getti(List<Object> list){
216        int thecount;
217        thecount = getcount(list);
218        int[] ti = new int[thecount];
219
```

```
220         for(int i=0;i<thecount;i++)
221            ti[i]=(int)list.get(i+thecount+1);
222         return ti;
223      }
224
225      /*
226       * Compare method, compare two different intermediate value and
             get the result
227       * @see ComputingSystem#Compare(java.lang.Object,
             java.lang.Object)
228       */
229      public Double Compare(List<Object> intermediatedataa,List<Object>
            intermediatedatab){
230         counta=getcount(intermediatedataa);
231         countb=getcount(intermediatedatab);
232         tva=gettv(intermediatedataa);
233         tvb=gettv(intermediatedatab);
234         tia=getti(intermediatedataa);
235         tib=getti(intermediatedatab);
236
237         double correlation = 0;
238        double vector_len1=0;
239        double vector_len2=0;
240        int p1 = 0;
241        int p2 = 0;
242
243     while (p1 < counta && p2 < countb){
244       int n1 = tia[p1];
245       int n2 = tib[p2];
246       if (n1 < n2){
247             double t1 = tva[p1];
248            vector_len1 += (t1 * t1);
249            p1++;
250          }
251       else if (n2 < n1){
252         double t2 = tvb[p2];
253         p2++;
254         vector_len2 += (t2 * t2);
255       }
256       else{
257         double t1 = tva[p1++];
258         double t2 = tvb[p2++];
259         vector_len1 += (t1 * t1);
260         vector_len2 += (t2 * t2);
261         correlation += t1 * t2;
262       }
263     }
264     while (p1 < counta){
265       double t1 = tva[p1++];
266       vector_len1 += (t1 * t1);
267     }
268     while (p2 < countb){
269       double t2 = tvb[p2++];
270       vector_len2 += (t2 * t2);
271     }
272
273     tva=null;
274      tvb=null;
275      tia=null;
276      tib=null;
277
278     return correlation / (Math.sqrt(vector_len1) *
           Math.sqrt(vector_len2));
279      }
```

```
280
281     /*
282      * Users use Writer function to operate the final results
283      * here just print all the index and results;
284      * @see ComputingSystem#Writer(Matrix)
285      */
286
287     public void Writer(Matrix<String,Double> matrix,String Taskvalue){
288
289       Map<List<String>,Double> resultpairs = matrix.Getallresult();
290       Iterator<Entry<List<String>, Double>> resultiterator =
            resultpairs.entrySet().iterator();
291
292       String outputpath = "";
293       File filename;
294       FileWriter fw = null;
295
296       try {
297         outputpath = "/home/yifan/result/outputfile"+Taskvalue+".txt";
298         filename = new File(outputpath);
299         filename.createNewFile();
300
301         fw = new FileWriter(outputpath);
302
303        while(resultiterator.hasNext()){
304           Map.Entry<List<String>, Double> entry =
                resultiterator.next();
305           fw.write("Key:"+entry.getKey().toString()+
306           "Value:"+entry.getValue().toString()+"\n");
307        }
308         fw.close();
309         } catch (IOException e) {
310            // TODO Auto-generated catch block
311            e.printStackTrace();
312          }
313       resultpairs=null;
314     }
315
316     int number_bacteria;
317     char[][] bacteria_name;
318     int M, M1, M2;
319     static int code[] = new int[] {0, 2, 1, 2, 3, 4, 5, 6, 7, -1, 8,
          9, 10, 11, -1, 12, 13, 14, 15, 16, 1, 17, 18, 5, 19, 3};
320     private static final int LEN=6;
321     private static final int AA_NUMBER=20;
322     private static final double EPSILON=1e-010;
323     int counta;
324     double[] tva;
325     int[] tia;
326     int countb;
327     double[] tvb;
328     int[] tib;
329     int[] vector;
330     int[] second;
331     int[] one_l;
332     int indexs;
333     int total;
334     int total_l;
335     int complement;
336     int count;
337     double[] tv;
338     int[] ti;
339     List<Object> intermediateData;
340 }
```

A.2 NMF Application Codes

Another ATAC application we used in Chapter 3 is NMF, which is a data mining program. The simple codes for NMF application is shown as follows:

```
public class Datamining extends
    ComputingSystem<String,ArrayList<String>,ArrayList<String>,Double>{

  public static List<String> alllines;
  public static double[][] A = new double[2000][2000];
  static boolean initial=false;

  HashMap<String,ArrayList<String>> Reader() throws IOException{

    if(initial==false){
      try {
        Datamining.getAllLine();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

    HashMap<String,String> map = new HashMap<String,String>();
    HashMap<String,ArrayList<String>> finalmap = new
        HashMap<String,ArrayList<String>>();

    //call getAllLines to initialize alllines
    String curline = "";
    if(!alllines.isEmpty()){
      curline = alllines.get(0);
      alllines.remove(0);
      System.out.println(curline);
    }else{
      System.out.println("All A are finished!!");
    }

    String contentA = "";//is not used for sparse format
    String contentW = "";
    String contentH = "";
    int itr = 1;

    //inside while{}
    String[] tmp = curline.split("-");//line A index

    //construct matrix A
    //read content A
    String filea = dataminingpath2 + curline;

    File file = new File(filea);
    if (file.exists()) {
      BufferedReader brA = new BufferedReader(new FileReader(file));
      String lineA;
      while ((lineA = brA.readLine()) != null) {
        String[] str = lineA.split(",");
        int row = Integer.parseInt(str[0]);
        int column = Integer.parseInt(str[1]);
        double entry = Double.parseDouble(str[2]);
        int highrow = row/50;//50 is row block number
        int highcolumn = column/10;//10 is column block number
        int startrow = highrow*50;
        int startcolumn = highcolumn*10;
```

```
56          A[row-startrow][column-startcolumn] = entry;
57        }
58        brA.close();
59      }else{
60        System.out.println("File " + filea + " does not exist");
61      }
62
63      //read content W
64      String filew = dataminingpath3 + (itr-1) + "/w" + tmp[1];
65      file = new File(filew);
66      if (file.exists()) {
67        BufferedReader brW = new BufferedReader(new FileReader(file));
68        String lineW;
69        while ((lineW = brW.readLine()) != null) {
70          contentW += lineW + ";"; //; is the line break mark and
              within one line, the element is separated by " " or Tab
71        }
72        brW.close();
73      }else{
74        System.out.println("File " + filew + " does not exist");
75      }
76
77      //read content H
78      String fileh = dataminingpath4 + (itr-1) + "/h" + tmp[0];
79      file = new File(fileh);
80      if (file.exists()) {
81        BufferedReader brH = new BufferedReader(new FileReader(file));
82        String lineH;
83        while ((lineH = brH.readLine()) != null) {
84          contentH += lineH + ";"; //; is the line break mark and
              within one line, the element is separated by " " or Tab
85        }
86        brH.close();
87      }else{
88        System.out.println("File " + fileh + " does not exist");
89      }
90
91      //concatenate content of A_ij, W_i and H_j for the value of
          HashMap<String,String> map
92      //to be noted, there is only one set of <k,v> in this map
93      String value = contentW + "/+" + contentH; //contentA + "+" +
94      map.put(curline, value);
95
96      ArrayList<String> firstline = new ArrayList<String>();
97      ArrayList<String> secondline = new ArrayList<String>();
98      firstline.add(value);
99      secondline.add(curline);
100     finalmap.put(curline,firstline);
101      finalmap.put(curline+"1",secondline);
102
103     return finalmap;
104   }
105
106   public ArrayList<String> Preprocess(ArrayList<String> map){
107     return map;
108   }
109
110   public Double Compare(ArrayList<String> inputlines,
         ArrayList<String> inputfileIndex){//input is one line of map,
         the content of A_ij, W_i and H_j
111
112     ArrayList<String> lines = new ArrayList<String>();
113     ArrayList<String> fileIndex = new ArrayList<String>();
114
```

```
115      for(int i=0;i<inputlines.size();i++)
116        lines.add(inputlines.get(i));
117
118      if(lines.get(0).contains("/+")){
119        for(int i=0;i<inputfileIndex.size();i++)
120          fileIndex.add(inputfileIndex.get(i));
121      } else{
122        lines = new ArrayList<String>();
123        for(int i=0;i<inputfileIndex.size();i++)
124          lines.add(inputfileIndex.get(i));
125
126        for(int i=0;i<inputlines.size();i++)
127          fileIndex.add(inputlines.get(i));
128      }
129
130      //let X be m by n, W m by k, H n by k
131      String v = lines.get(0);
132
133      String[] str = v.split("/+");//sparse format only contains two:
            str[0] W, str[1] H, dense format has three: str[0] A, str[1]
            W, str[0] H
134
135      //construct matrix W
136      double[][] W = new double[2000][10];//10 is the value of k
137      String[] wline = str[0].split(";");//; line break
138      for(int i = 0; i < W.length; i++){
139        String[] temp = wline[i].split(",");//, element break
140        for(int j = 0; j < W[0].length; j++){
141          W[i][j] = new BigDecimal(temp[j]).doubleValue();
142        }
143      }
144      //get W'
145      double[][] WT = transpose(W);
146      //construct matrix H
147      double[][] H = new double[2000][10];//10 is the value of k
148      wline = str[1].split(";");//; line break
149      for(int i = 0; i < H.length; i++){
150        String[] temp = wline[i].split(",");//, element break
151        for(int j = 0; j < H[0].length; j++){
152          H[i][j] = Double.parseDouble(temp[j]);
153        }
154      }
155      //get H'
156      double[][] HT = transpose(H);
157
158      //define intermediate matrices
159      double[][] S = new double[10][10];// k by k
160      double[][] X = new double[10][2000];//k by n
161      double[][] Y = new double[10][2000];//k by n
162      double[][] HTnew = new double[10][2000];//n by k
163
164      //calculate S=w'w
165      S = multiply(WT,W);
166
167      //calculate X=w'A
168      X = multiply(WT,A);
169
170      //calculate Y=SHT
171      Y = multiply(S,HT);
172
173      //caluculate HT = HT*(X/Y) entry-wise
174      for(int i = 0; i < HT.length; i++){
175        for(int j = 0; j < HT[0].length; j++){
176          if(Y[i][j] == (double) 0){
```

```
177             HTnew[i][j] = (double) 0;
178           }else{
179             HTnew[i][j] = HT[i][j]*(X[i][j]/Y[i][j]);
180           }
181         }
182       }
183 
184       double[][] Hnew = transpose(HTnew);
185 
186       //define intermediate matrices
187       double[][] P = new double[10][10];// k by k
188       double[][] U = new double[10][2000];//k by n
189       double[][] V = new double[10][2000];//k by n
190       double[][] Wnew = new double[2000][10];//n by k
191 
192       //calculate P=hh'
193       P = multiply(HTnew,H);//use HTnew instead of HT
194 
195       //calculate U=Ah'
196       U = multiply(A,Hnew);
197 
198       //calculate V=wP
199       V = multiply(W,P);
200 
201       //caluculate W = W*(U/V) entry-wise
202       for(int i = 0; i < W.length; i++){
203         for(int j = 0; j < W[0].length; j++){
204           if(V[i][j] == (double) 0){
205             Wnew[i][j] = (double) 0;
206           }else{
207             Wnew[i][j] = W[i][j]*(U[i][j]/V[i][j]);
208           }
209         }
210       }
211 
212 
213       ////////////////////write Hnew and Wnew into
             Disk////////////////////////
214       //need the value of itr to write into folder W/itr/ or H/itr/
215       //also need the key of map, curline, to write as the W_i and H_j
216       String[] index = fileIndex.get(0).split("-");//index[0] is i
             and index[1] is j
217       ////////////////////////
218 
219       return 123.00;
220      }
221 
222     public static void cacheWH(int itr) throws IOException{
223 
224       String[] w = new String[50];
225       String[] h = new String[10];
226       File file;
227       File folder = new File(dataminingpath3 + itr);
228       File[] listOfFiles = folder.listFiles();
229 
230       //read content W
231        for (int i = 0; i < listOfFiles.length; i++) {
232           if (listOfFiles[i].isFile()) {
233            String filew = listOfFiles[i].getName();
234            file = new File(filew);
235            if (file.exists()) {
236              BufferedReader brW = new BufferedReader(new
                    FileReader(file));
237              String lineW;String contentW = "";
```

```
238            while ((lineW = brW.readLine()) != null) {
239              System.out.println(lineW);
240              contentW += lineW;
241            }
242            brW.close();
243            w[i] = contentW;//content of w_i
244          }else{
245            System.out.println("File " + filew + " does not exist");
246          }
247        } else if (listOfFiles[i].isDirectory()) {
248          System.out.println("Directory " +
               listOfFiles[i].getName());
249        }
250      }
251
252      //read content H
253      folder = new File(dataminingpath4 + itr);
254      for (int i = 0; i < listOfFiles.length; i++) {
255        if (listOfFiles[i].isFile()) {
256          String fileh = listOfFiles[i].getName();
257          file = new File(fileh);
258          if (file.exists()) {
259            BufferedReader brH = new BufferedReader(new
                FileReader(file));
260            String lineH;String contentH = "";
261            while ((lineH = brH.readLine()) != null) {
262              contentH += lineH;
263            }
264            brH.close();
265            h[i] = contentH;//content of h_i
266          }else{
267            System.out.println("File " + fileh + " does not exist");
268          }
269        } else if (listOfFiles[i].isDirectory()) {
270          System.out.println("Directory " +
               listOfFiles[i].getName());
271        }
272      }
273
274      //at the end we get content of all W and H block matrices in
           two array
275      //we can set two global array for this and update them at the
           end of each iteration
276    }
277
278    public static void getAllLine() throws IOException{
279
280      List<String> lines =
           Files.readAllLines(Paths.get(dataminingpath1),
           StandardCharsets.UTF_8);
281      alllines = lines;
282      initial=true;
283    }
284
285    static Properties p;
286    static String dataminingpath1;
287    static String dataminingpath2;
288    static String dataminingpath3;
289    static String dataminingpath4;
290
291    //matrix operation functions
292
293     // return C = A * B
294     public static double[][] multiply(double[][] A, double[][] B) {
```

```
295         int mA = A.length;
296         int nA = A[0].length;
297         int mB = B.length;
298         int nB = B[0].length;
299         if (nA != mB) throw new RuntimeException("Illegal matrix
               dimensions.");
300         double[][] C = new double[mA][nB];
301         for (int i = 0; i < mA; i++)
302            for (int j = 0; j < nB; j++)
303               for (int k = 0; k < nA; k++)
304                  C[i][j] += (A[i][k] * B[k][j]);
305         return C;
306      }
307      // return C = A^T
308      public static double[][] transpose(double[][] A) {
309         int m = A.length;
310         int n = A[0].length;
311         double[][] C = new double[n][m];
312         for (int i = 0; i < m; i++)
313            for (int j = 0; j < n; j++)
314               C[j][i] = A[i][j];
315         return C;
316      }
317   }
```

Literature Cited

Afgan, E., Baker, D., Coraor, N., Chapman, B., Nekrutenko, A., and Taylor, J. (2010). Galaxy cloudman: delivering cloud compute clusters. *BMC Bioinformatics*, 11(12):4–10.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Andoni, A. and Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122.

Arora, R., Gupta, M. R., Kapila, A., and Fazel, M. (2013). Similarity-based clustering by left-stochastic matrix factorization. *Journal of Machine Learning Research*, (14):1715–1746.

Barton, G. (1990). Protein multiple sequence alignment and flexible pattern matching. *Methods in enzymology*, 183:403–428.

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up all pairs similarity search. In *Proceedings of the 16 International World Wide Web Conference (WWW2007)*, pages 131–140, Banff, Alberta, Canada. ACM.

Bharathi, S. and Kumaresan (2012). Performance evaluation of SDS algorithm with fault tolerance for distributed system. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 1(2):50–56.

Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. *Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203.

Borthakur, D. (2007). The Hadoop Distributed File System: Architecture and Design. *The Apache Software Foundation*.

Catalyurek, U., Stahlberg, E., Ferreira, R., Kurc, T., and Saltz, J. (2002). Improving performance of multiple sequence alignment analysis in multi-client environments. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS02)*, page 0183b, Washington, DC, USA. IEEE Computer Society.

Center, K. U. B. (2007). Multiple sequence alignment by CLUSTALW. http://www.genome.jp/tools/clustalw/. accessed: 3 July 2014.

Chan, R. H., Wang, R. W., and Yeung, H. M. (2010). Composition vector method for phylogenetics-a review. In *Proceedings of the 9th International Symposium on Operations Research and Its Applications (ISORA10)*, page 13. Citeseer.

Chen, H., Lin, T. Y., Zhang, Z., and Zhong, J. (2013). Parallel mining frequent patterns over big transactional data in extended mapreduce. In *IEEE International Conference on Granular Computing (GrC)*, pages 43 – 48, Beijing. IEEE.

Chen, P., Wang, C., Li, X., and Zhou, X. (2014). Accelerating the Next Generation Long Read Mapping with the FPGA-Based System. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(5):840–852.

Chen, Q., Wang, L., and Shang, Z. (2008). MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS. In *IEEE Fourth International Conference on eScience (eScience '08)*, pages 646 – 651, Indianapolis, IN.

Church, P. C. and Goscinski, A. (2014). A Survey of Cloud-Based Service Computing Solutions for Mammalian Genomics. *IEEE Transactions on Service Computing*, 7(4):726–740.

Church, P. C., Goscinski, A., Holt, K., Inouye, M., Ghoting, A., Makarychev, K., and Reumann, M. (2011). Design of multiple sequence alignment algorithms on parallel, distributed memory supercomputers. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 924–927, Boston, Massachusetts USA.

Cohn, H. and Fielding, M. (1999). Simulated annealing: Searching for an optimal temperature schedule. *SIAM Journal on Optimization*, 9(3):779 – 802.

Corneil, D. and Mathon, R. (1978). Algorithmic Techniques for the Generation and Analysis of Strongly Regular Graphs and other Combinatorial Configurations. *Annals of Discrete Mathematics*, 2:1–32.

Cui, X., Zhong, C., and Lu, X. (2010). Parallel local alignment algorithm for multiple sequences on heterogeneous cluster systems. In *Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithm and Programming (PAAP)*, pages 316–320.

Das, A., Lumezanu, C., Zhang, Y., Singh, V., Jiang, G., and Yu, C. (2013). Transparent and flexible network management for big data processing in the cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA. USENIX.

Date, S., Kulkarni, R., Kulkarni, B., Kulkarni-Kale, U., and Kolaskar, A. (1993). Multiple alignment of sequences on parallel computers. *Computer applications in the biosciences (CABIOS)*, 9(4):397–402.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Doug, C. and Mike, C. (2005). Hadoop. http://hadoop.apache.org. accessed: 3 July 2014.

Edgar, R. and Batzoglou, S. (2006). Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373.

Elnikety, E., Elsayed, T., and Ramadan, H. (2011). iHadoop: Asynchronous Iterations for MapReduce. In *Third IEEE International Conference on Coud Computing Technology and Science*, pages 81 – 90, Athens. IEEE.

Farzanyar, Z. and Cercone, N. (2013). Accelerating Frequent Itemsets Mining on the Cloud: A MapReduce -Based Approach. In *IEEE 13th International Conference on Data Mining Workshops*, pages 592 – 598, Dallas, TX. IEEE.

Feng, D. and Doolittle, R. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360.

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

Gokhale, M., Cohen, J., Yoo, A., Miller, W., Jacob, A., Ulmer, C., and Pearce, R. (2008). Hardware technologies for high-performance data-intensive computing. *Computer*, 41(4):60–68.

Gould, H. W. (1961). The q -stirling numbers of first and second kinds. *Duke Mathematical Journal*, 28(2):281–289.

Gunturu, S., Li, X., and Yang, L. (2009). Load scheduling strategies for parallel DNA sequencing applications. In *11th IEEE International Conference on High Performance Computing and Communications (HPCC'09)*, pages 124–131, Seoul. IEEE.

Hao, B., Qi, J., and Wang, B. (2003). Prokaryotic phylogeny based on complete genomes without sequence alignment. *Modern Physics Letters*, 2(4):14–15.

Heitor, S. and Guilherme, L. (2005). A distributed approach for a multiple sequence alignment algorithm using a parallel virtual machine. In *Proceedings of the 25rd Engineering in Medicine and Biology Annual Conference*, pages 2843–2846, Shanghai, China.

Hess, M., Sczyrba, A., Egan, R., and Kim, T.-W. (2011). Metagenomic discovery of biomass-degrading genes and genomes from cow rumen. *Science*, 331(6016):463–467.

Higgins, D. and Sharp, P. (1988). Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244.

Hill, J., Hambley, M., Forster, T., Mewissen, M., Sloan, T. M., Scharinger, F., Trew, A., and Ghazal, P. (2008). SPRINT: A new parallel framework for R. *BMC Bioinformatics*, 9(1):558–559.

Hindman, B., Konwinski, A., and Zaharia, M. (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation*, volume 11, page 22.

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81.

Hummel, S. (1996). Load sharing in heterogeneous systems via weighted factoring. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 318–328.

Jaap, D. and Heringa, J. (1999). Two stragegies for sequence comparison: profile-preprocessed and secondary structure-induced multiple alignment. *Computers & Chemistry*, 23(34):341–364.

Jaap, D. and Heringa, J. (2002). Local weighting schemes for protein multiple sequence alignment. *Computers & Chemistry*, 26(5):459–477.

Jain, A. K., Ross, A., and Prabhakar, S. (2004). An Introduction to Biometric Recognition. *IEEE Transactions on circuits and systems for video technoloty*, 14(1):4–20.

Kedad-Sidhoum, S., Mendonca, F. M., Monna, F., Mounié, G., and Trystram, D. (2014). Fast biological sequence comparison on hybrid platforms. In *43rd International Conference on Parallel Processing (ICPP)*, pages 501 – 509, Minneapolis. IEEE.

Keikha, M. (2011). Improved simulated annealing using momentum terms. In *2011 Second International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*, pages 44 – 48, Kuala Lumpur. IEEE.

Khan, M., Liu, Y., and Li, M. (2014). Data Locality in Hadoop Cluster Systems. In *11th International Conference on Fuzzy Systems and Knowledge Discovery*, pages 720 – 724, Xiamen. IEEE.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680.

Kleinjung, J., Douglas, N., and Heringa, J. (2002). Parallelized multiple alignment. *Bioinformatics*, 18(9):1270–1271.

Kolekar, P., Kale, M., and Kulkarni-Kale, U. (2012). Alignment-free distance measure based on return time distribution for sequence analysis: Applications to clustering, molecular phylogeny and subtyping. *Molecular Phylogenetics and Evolution*, 65(2):510–522.

Krishnajith, A. P. D., Kelly, W., Hayward, R., and Tian, Y.-C. (2013). Managing memory and reducing I/O cost for correlation matrix calculation in bioinformatics. In *Proceedings of the IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, pages 36–43, Singapore. IEEE.

Krishnajith, A. P. D., Kelly, W., and Tian, Y.-C. (2014). Optimizing I/O cost and managing memory for composition vector method based correlation matrix calculation in bioinformatics. *Current Bioinformatics*, 9(3):234–245.

Li, K., Pan, Y., Shen, H., and Zhang, S. (1999). A study of average-case speedup and scalability of parallel computations on static networks. *Mathematical and Computer Modelling*, 29(9):83–94.

Lusk, K., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standand. *Parallel Computing*, 22:789–828.

Macedo, E., Melo, A., Pfitscher, G., and Boukerche, A. (2011). Hybrid MPI/OpenMP strategy for biological multiple sequence alignment with DIALIGN-TX in heterogeneous mluticore clusters. In *Proceedings of the IEEE International Paraeel and Distributed Workshops and PhD Forum (IPDPSW)*, pages 418–425, Shanghai. IEEE.

Marc, E. C., Matthew, W. P., Scott, M., and Lynette, H. (2011). Nephele: genotyping via complete composition vectors and MapReduce. *Source Code for Biology and Medicine*, 6:13.

Mark, D. (1990). What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21.

Matsunaga, A., Tsugawa, M., and Fortes, J. (2008). CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In *IEEE 4th International Conference on eScience*, pages 222 – 229, Indianapolis, IN.

Mendonca, F. M. and de Melo, A. C. M. A. (2013). Biological Sequence Comparison on Hybrid Platforms with Dynamic Workload Adjustment. In *27th Int Symp on Parallel & Distributed Processing Workshops and PhD Forum*, pages 501 – 509, Cambridge, MA. IEEE.

Meng, X. and Chaudhary, V. (2010). A High-Performance Heterogeneous Computing Platform for Biological Sequence Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1267–1280.

Moretti, C., Bui, H., Hollingsworth, K., Rich, B., Flynn, P., and Thain, D. (2010). All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems*, 21:33–46.

NCBI (1988). National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/. accessed: 3 July 2014.

NCBI (1990). BLAST: Basic local allignment search tool. http://blast.ncbi.nlm.nih.gov/Blast.cgi. accessed: 3 July 2014.

Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed Stream Computing Platform. In *IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170 – 177, Sydney, NSW, Australia. IEEE.

Nguyen, K. D. and Pan, Y. (2013). A Knowledge-Based Multiple-Sequence Alignment Algorithm. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(4):884–896.

NM, L., D, G., and M., G. (2001). What is bioinformatics? a proposed definition and overview of the field. *Methods Inf Med*, 40(4):346–358.

Notredame, C., Higgins, D., and Heringa, J. (2000). T-coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–217.

Pedersen, E., Raknes, I. A., Ernstsen, M., and Bongo, L. A. (2015). Integrating data-intensive computing systems with biological data analysis frameworks. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 733 – 740, Turku. IEEE.

Pepper, J., Golden, B., and Wasil, E. (2002). Solving the traveling sales man problem with annealing-based heuristics: A computational study. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 32(1):72 – 77.

Phillips, P., Flynn, P., Scruggs, T., Bowyer, K., Chang, J., Hoffman, K., Marques, J., Min, J., and Worek, W. (2005). Overview of the face recognition grand challenge. In *Computer Society Conference on Computer Vision and Pattern Recognition (CVPR05)*, volume 1, pages 947 – 954, San Diego, CA, USA. IEEE.

Qiu, X., Ekanayake, J., Beason, S., Gunarathne, T., and Fox, G. (2009). Cloud Technologies for Bioinformatics Applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS '09)*, Portland, Oregon, USA.

Rasooli, A. and Down, D. G. (2012). A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems. In *SC Companion: High Performance Computing, Networking Storage and Analysis*, Salt Lake City, USA.

Ren, Z., Wan, J., Shi, W., Xu, X., and Zhou, M. (2014). Workload analysis, implications, and optimization on a production Hadoop cluster: A case study on Taobao. *IEEE Transactions on Service Computing*, 7(2):307–321.

Rueda, D. R., Cotta, C., and Fernndez, A. J. (2009). Finding balanced incomplete block designs with metaheuristics. In *Proceedings of the 9th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP '09)*, pages 156 – 167, Tbingen, Germany. Springer Berlin Heidelberg.

Sav, S., Jones, G. J. F., Lee, H., OConnor, N. E., and Smeaton, A. F. (2006). Interactive experiments in object-based retrieval. In *International Conference on Image and Video Retrieval (CIVR 2006)*, pages 1–10, Tempe, AZ, USA.

Schatz, M. C. (2009). Cloudburst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369.

Schuler, G., Altschul, S., and Lipman, D. (1991). A workbench for multiple alignment construction and analysis. *Proteins: Structure, Function and Bioinformatics*, 9(3):180–190.

Sims, G. E., Jun, S.-R., Wu, G. A., and Kim, S.-H. (2009). Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions. *National Academy of Sciences of the United States of America*, 106(8):2677–2682.

Singh, A., Chen, C., Liu, W., Mitchell, W., and Schmidt, B. (2008). A Hybrid Computational Grid Architecture for Comparative Genomics. *IEEE Transactions on Information Technology in Biomedicine*, 12(2):218 – 225.

Singhal, H. and Guddeti, R. (2014). Modified mapreduce framework for enhancing performance of graph based algorithms by fast convergence in distributed environment. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1240 – 1245, New Delhi. IEEE.

Somasundaram, K., Karthikeyan, S., Nayagam, M. G., and RadhaKrishnan, S. (2010). Efficient resource scheduler for parallel implementation of MSA algorithm on computational grid. In *International Conference on Recent Trends in Information, Telecommunication and Computing (ITC)*, pages 365 – 368, Kochi, Kerala. IEEE.

Sun, X.-H., Chen, Y., and Wu, M. (2005). Scalability of Heterogeneous Computing. In *International Conference on Parallel Processing*, pages 557 – 564, Oslo, Norway. IEEE.

Tang, P. and Yew, P. (1986). Processor self-scheduling for multiple nested parallel loops. In *proceedings of International conference on Parallel Processing (ICPP)*, pages 528–535.

Thite, S. (2008). On covering a graph optimally with induced subgraphs. *arXiv preprint cs/0604013*.

Thompson, J., Higgins, D., and Gibson, T. (1994). Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680.

Torres, J. S., Espert, I. B., Dominguez, A. T., Garcia, V. H., Castello, I. M., Gimenez, J. T., and Blazquez, J. D. (2012). Using GPUs for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(4):1245–1256.

Trelles, O. (2011). On the parallelisation of bioinformatics applications. *Brief Bioinform*, 2(2):181–194.

Trelles, O., Andrade, M., Valencia, A., Zapata, E., and Carazo, J. (1998). Computational space reduction and parallelization of a new clustering approach for large groups of sequences. *Bioinformatics*, 14(5):439–451.

van der Linden, W. J., Veldkamp, B. P., and Carlson, J. E. (2004). Optimizing balanced incomplete block designs for educational assessments. *Applied Psychological Measurement*, 28(5):317–331.

Vinga, S. and Almeida, J. (2003). Alignment-free sequence comparison-a review. *Bioinformatics*, 19(4):513–523.

Wang, W. (2009). *Composition Vector Methods for Phylogeny*. PhD thesis, The Chinese University of Hong Kong.

Wu, W., Li, L., and Yao, X. (2014). Improved simulated annealing algorithm for task allocation in real-time distributed systems. In *IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pages 50 – 54, Guilin. IEEE.

Xiao, S., Lin, H., and Feng, W. (2011). Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *IEEE International Parallel & Distributed Processing Symposium*, pages 1212–1222.

Xu, G., Lu, F., Yu, H., and Xu, Z. (2007). A Distributed Parallel Computing Environment for Bioinformatics Problems. In *proceedings of the 6th International Conference on Grid and Cooperative Computing (GCC 2007)*, pages 593–599.

Ye, Y., Cheung, D. W.-L., Wang, Y., Yiu, S.-M., Zhang, Q., Lam, T.-W., and Ting, H.-F. (2015). Glprobs: Aligning multiple sequences adaptively. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(1):67–78.

Yu, X. and Hong, B. (2013). Bi-Hadoop: Extending Hadoop To Improve Support For Binary-Input Applications. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 245 – 252, Delft. IE.

Yu, Z.-G., Chu, K. H., Li, C. P., Anh, V., Zhou, L.-Q., and Wang, R. W. (2010). Whole-proteome phylogeny of large dsdna viruses and parvoviruses through a composition vector method related to dynamical language model. *BMC Evolutionary Biology*, 10(192).

Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing withworking sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. (2008). Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*, number 14, pages 29–42, San Diego, CA, USA. USENIX Association.

Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829.

Zhang, Y., Gao, Q., Gao, L., and Wang, C. (2011). A Distributed Computing Framework for Iterative Computation. In *Proceedings of the 1st International Workshop on Data Intensive Computing in the Clouds*.

Zhang, Y.-F., Tian, Y.-C., Fidge, C., and Kelly, W. (2014). A Distributed Computing Framework for All-to-All Comparison Problems. In *The 40th Annual Conference of the IEEE Industrial Electronics Society (IECON'2014)*, Dallas, TX, USA. IEEE.

Zhang, Y.-F., Tian, Y.-C., Fidge, C., and Kelly, W. (2015a). Data-Aware Task Scheduling for Large-Scale All-to-All Comparison Problems in Heterogeneous Systems. *IEEE Transactions on Computers. Manuscript submitted for publication.*

Zhang, Y.-F., Tian, Y.-C., Kelly, W., and Fidge, C. (2015b). Application of Simulated Annealing to Data Distribution for All-to-All Comparison Problems in Homogeneous Systems. In *22nd International Conference on Neural Information Processing (ICONIP2015)*, Istanbul, Turkey. IEEE.

Zhang, Y.-F., Tian, Y.-C., Kelly, W., and Fidge, C. (2015c). Distributed Computing of All-to-All Comparison Problems in Heterogeneous Systems. In *The 41th Annual Conference of the IEEE Industrial Electronics Society (IECON'2015)*, Yokohama, Japan. IEEE.

Zhang, Y.-F., Tian, Y.-C., Kelly, W., and Fidge, C. (2015d). Scalable and Efficient Data Distribution for Distributed Computing of Large-Scale All-to-All Comparison Problems. *IEEE Transactions on Services Computing. Manuscript submitted for publication.*

Zheng, J.-H., Zhang, L.-J., Zhu, R., Ning, K., and Liu, D. (2013). Parallel Matrix Multiplication Algorithm Based on Vector Linear Combination Using MapReduce. In *IEEE 9th World Congress on Services (SERVICES)*, pages 193 – 200, Santa Clara, CA. IEEE.

Zhu, X., Li, K., and Li, R. (2011). A Data Parallel Strategy for Aligning Multiple Biological Sequences on Homogeneous Multiprocessor Platform. In *Proceedings of the 6th Annual ChinaGrid Conferene (ChinaGrid)*, pages 188–195.

Zhu, X., Li, K., Salah, A., Shi, L., and Li, K. (2015). Parallel implementation of MAFFT on CUDA-enabled graphics hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(1):205–218.