8-2013

# DEVELOPMENT OF MAP/REDUCE BASED MICROARRAY ANALYSIS TOOLS

Guangyu Yang
*Clemson University*, guangyy@clemson.edu

# DEVELOPMENT OF MAP/REDUCE BASED MICROARRAY ANALYSIS TOOLS

---

A Thesis
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
School of Computing

---

by
Guangyu yang
August 2013

---

Accepted by:
Dr. Feng Luo, Committee Chair
Dr. Pradip K Srimani
Dr. Mark Smotherman

## ABSTRACT

High density oligonucleotide array (microarray) from the Affymetrix GeneChip® system has been widely used for the measurements of gene expressions. Currently, public data repositories, such as Gene Expression Omnibus (GEO) of the National Center for Biotechnology Information (NCBI), have accumulated very large amount of microarray data. For example, there are 84389 human and 9654 Arabidopsis microarray experiments in GEO database. Efficiently integrative analysis large amount of microarray data will provide more knowledge about the biological systems. Traditional microarray analysis tools all implemented sequential algorithms and can only be run on single processor. They are not able to handle very large microarray data sets with thousands of experiments. It is necessary to develop new microarray analysis tools using parallel framework. In this thesis, I implemented microarray quality assessment, background correction, normalization and summarization algorithms using the Map/Reduce framework. The Map/Reduce framework, first introduced by Google in 2004, offers a promising paradigm to develop scalable parallel applications for large-scale data. Evaluation of our new implementation on large microarray data of rice and Arabidopsis showed that they have good speedups. For example, running rice microarray data using our implementations of MAS5.0 algorithms on 20 computer nodes totally 320 processors has a 28 times speedup over using previous C++ implementation on single processor. Our new microarray tools will make it possible to utilize the valuable experiments in the public repositories.

ACKNOWLEDGMENTS

Foremost, I’d like to express my sincere gratitude to my advisor, Dr. Feng Luo, for his support me through the many trials and tribulations of finishing this degree, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Dr. Luo has played a large and unique part in helping me reach this point in my journey.

I would also like to gratefully thank Dr. Pradip K Srimani and Dr. Mark Smotherman for being on my committee, reviewing my thesis and their comments and suggestions.

I would like to thank my girlfriend – Yujing Xie for accompanying me during the harsh times, giving me warmth. Without her, this thesis might never have been finished. Thanks for being aside with me where I looked like I was sticked to the computer and seemed like I was completely ignoring you, which I wasn’t.

Finally, I would like to thank my parents Chao Yang and Xiuqiong Chen for supporting my education throughout my life. I would not have achieved this without their constant love and support.

TABLE OF CONTENTS

Table of Contents (Continued)

Table of Contents (Continued)

Page

## LIST OF TABLES

List of Tables (Continued)

List of Tables (Continued)

Table Page

## LIST OF FIGURES

ACRONYM TABLE

| | |
|---|---|
| QA | Quality Control |
| MAS 5.0 | Affymetrix MicroArray Suite 5.0 |
| RMA | Robust Multi-array Analysis |
| NUSE | Normalized Unscaled Standard Error |
| RLE | Relative Log Expression |
| SE | Standard Error |
| PM | Perfect Match |
| MM | Miss-Match |
| Orange FS | Orange File System |
| HPC | High Performance Computing |
| PBS | Portable Batch System |
| Sfs | Scale Factor |
| Avbg | Average Background |
| Pps | Present Percent |
| RNAdeg | RNA Degradation |
| GC | Garbage Collection |

# Chapter 1

# INTRODUCTION

Recent years, biologists have produced massive amount of microarray data using the Affymetrix GeneChip® platform. For example, the size of microarray datasets for human genomes from thousands of experiments has reached the terabyte scale. How to deal with these large data sets and find useful biological information inside them remains a challenge for the bioinformatics research. Current Affymetrix microarray analysis tools are all designed for the single machine and cannot process the large data sets with sufficient performance.

Hadoop is an implement of Map/Reduce programming model, which is proposed by Google supported by many large companies and communities. Hadoop provides a high performance parallel file system HDFS that is powerful in capability, commonality and scalability. Hadoop is an ideal framework for processing very large datasets as well as parallel programming due to its reliability, fault-tolerant, and well support from communities.

In this thesis, we implemented a set of Map/Reduce based microarray analysis tools using Hadoop framework for analyzing large Affymetrix GeneChip microarray datasets. We implemented two widely used algorithms: Affymetrix MicroArray Suite (MAS 5.0) and Robust Multi-array Analysis (RMA) algorithms

The remainder of this thesis is organized as follows. Chapter 2 provides background and terminology information of microarray technique and parallel model.

Chapter 3 deeply analyzes the RMA, MAS 5.0 preprocess algorithm and designs Map/Reduce implementations to parallelize these algorithm. Chapter 4 compares the methods and discusses the results of the comparison. Chapter 5 offers tuning, optimization and deployment for the Map/Reduce microarray tools. Finally Chapter 7 offers conclusions and future works.

# Chapter 2

# BACKGROUND

## 2.1 Microarray Technology

Microarray is a popular technique to measure genome-wide gene expressions (Alizadeh, et al., 2000). Microarray is a glass surface with numerous fragments of samples, called *probes*. A labeled sample contains the unknown quantities of molecules, called *target.* Under the right chemical conditions, single stranded fragments of target will complement pair with the probes, this reaction is called *hybridization.* In this way thousands of messenger RNA fragments in a target sample can be measured by the microarrays.

Microarray technique obtains the RNA sample with following steps: (1) Isolating the RNA sample. (2) Labeling the RNA sample by a reverse transcription procedure with fluorescent markers. (3) Purifying the labeled RNA sample. (4) Hybridizing the RNA sample. (5) Scanning the fluorescently labeled sample at each spot and emitting as a characteristic wavelength. (6) Capturing the wavelength in a photomultiplier tube.

### 2.1.1 Affymetrix GeneChip® Technology

There are two major microarray technologies: the cDNA arrays developed at Stanford University (DeRisi et al., 1996), (Brown et al., 1999) and the high-density oligonucleotide array system, also known as Affymetrix GeneChip®, produced by

Affymetrix (Lockhart et al., 1996). In this thesis, we focused on data generated by Affymetrix GeneChip®.

In Affymetrix GeneChip, the expression of each gene is typically measured by a set of 11±20 pairs of probes. There are two types of probes: perfect match (PM) probe and mismatch (MM) probe. Probes group into pair with each PM probe pairing with a MM probe. The PM probe is a 25 base oligonucleotide being designed to hybridize with messenger RNA from the intended gene. The MM probe is used to measure non-specific binding by changing the middle (13th) base to the complementary of the corresponding position in the PM set. MM probes are intended to help measure the background and stray signals.

The analyses of microarray data need two types of files: the .cdf file and the .cel file. The *.cdf* (Chip Description File) file includes the layout information of expression, genotyping, customSeq, copy number and/or tag probe sets in Affymetrix GeneChip microarray. All probe set in the *.cdf files* have unique names.

The *.cel* files store the intensity information of individual probes on the probe array. Each of the *.cel* file includes an intensity value (perfect match intensity and mismatch intensity), standard deviation of the intensity, the number of pixels used to calculate the intensity value, a flag to indicate an outlier computed by the algorithm and a user defined flag marking the feature should be excluded from future analysis.

## 2.2 Affymetrix microarray quality control

The quality of microarray data from public repositories usually varies greatly from different experiments. To maintain data integrity, we need to filter out low quality data.

Quality assessment is the stage to identify and remove low quality microarray data to keep data homogeneity. In this thesis, we used six quality assessment metrics from three famous methods:  Affymetrix Micro Array Suite (MAS 5.0), Robust Multi-array Analysis (RMA) and R affy package. Among them, three quality metrics: scaling factor (sfs), average background (avBg) and percentage of present calls (pps) are from MAS 5.0; two metrics: Normalized Unscaled Standard Errors (NUSE), and Relative Log Expression (RLE) are from RMA; and one quality metric, the RNA degradation (RNAdeg) is from R affy package.  Figure 2-1 illustrates the workflow for microarray quality assessment.

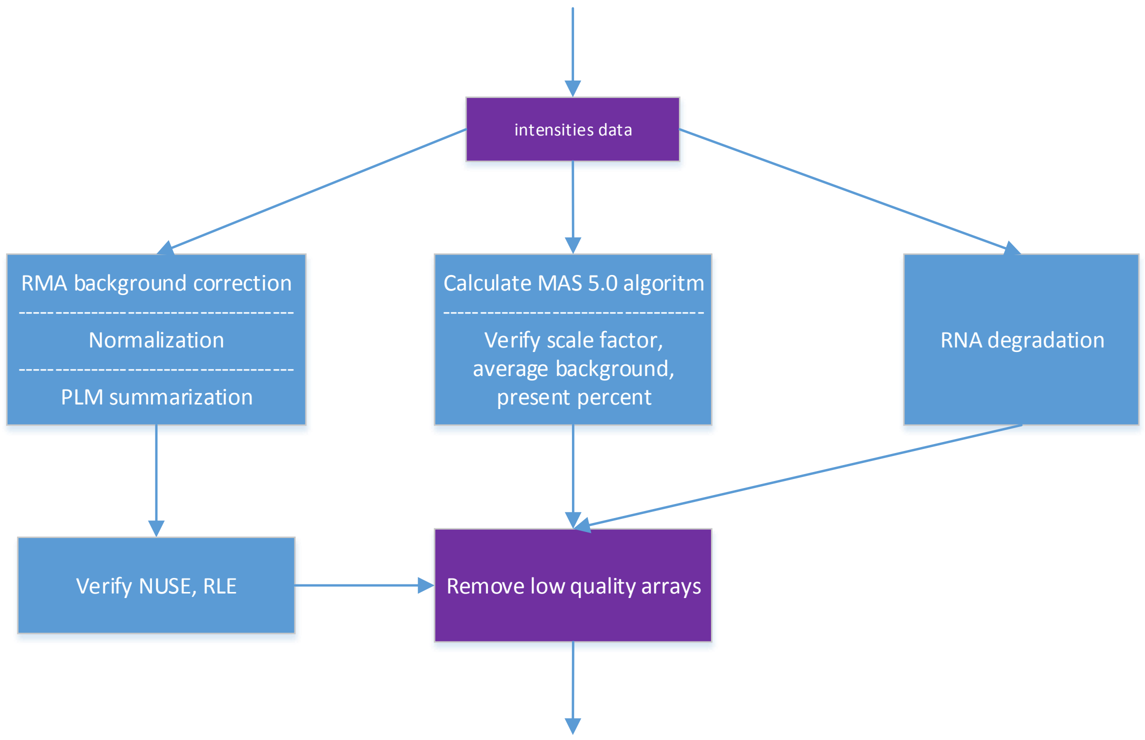
intensities data
RMA background correction
Normalization
PLM summarization
Calculate MAS 5.0 algoritm
Verify scale factor, average background, present percent
RNA degradation
Verify NUSE, RLE
Remove low quality arrays

Figure 2-1. Quality control workflow

### 2.2.1 Average Background

In MAS 5.0 algorithm, the background intensity is based on the mis-match probes values for each array. Since average, minimum and maximum background intensity of arrays should be comparable, an array with a significantly higher (or lower) background value indicates low quality. There are many reasons that an array has significantly different average backgrounds, for example, abnormal hybridization or too many cDNA in samples. MAS 5.0 algorithm computes average background as the 2nd percentile of the feature intensities in a given region of the array. Typically Affymetrix recommends average background values in a good quality array should between 20 and 100.

### 2.2.2 Scale Factors

In most of normalization methods, there is an assumption that the expressions of most genes are unchanged for high-throughput expression arrays. Namely, it says that the trimmed mean intensity for each array should be constant. If arrays are comparable, the average signal intensities should be similar and not be affected by the proportion of up- and down-regulated genes. Affymetrix MAS 5.0 algorithm scales the intensity for every array so that each array has the same mean. *Scale factor* represents the amount of scaling applied to the array. Low quality arrays have significant higher or lower scale factors. This may due to different issues occurred during RNA extraction, labeling, scanning or array manufacture.

### 2.2.3 Percent Present

MAS 5.0 algorithm generates present, marginal, absent calls for each probe pair of a probeset on an array based on the difference between PM and MM values. When the PM values of a probeset are not significantly above the values of MM probes, the probeset will be flagged to marginal or absent. The *percent present call* is defined as the percentage of probesets called present on an array. Differences in array processing pipelines, variations in the amount of starting material would lead to low present calls. So, we considered array with low percent present calls as poor quality array.

### 2.2.4 Normalized Unscaled Standard Error (NUSE)

The NUSE and RLE metrics from RMA are based on probe-level model (PLM) summarization. The PLM summarization uses an M-estimator robust regression expression model to measures the expressions.

For a given gene $j$ and a given array $i$, the NUSE is defined as ratio between its expression standard error and the median standard error of all genes:

$$NUSE(\hat{\theta}_{ij}) = \frac{SE(\hat{\theta}_{ij})}{median_i[SE(\hat{\theta}_{ij})]} \qquad (2\text{-}1)$$

We can use NUSE to assess array quality, since NUSE addresses the variability between genes. The NUSE values should be standardized at the probeset level across the arrays. If an array have SE higher than the median SE, this array will be considered to be low quality. Generally, the median NUSE of an array larger than 1.05 or the array have a large IQR indicates low quality.

### 2.2.5 Relative Log Expression (RLE)

The Relative Log Expression (RLE) is defined as the difference between the expressions of a probese and the median expressions for the same probeset across all arrays. The assumption behind RLE is that the median expressions for most probe sets are not changed across the arrays. For gene $i$ on array $j$,

$$RLE(\theta_{ij}) = \theta_{ij} - median_j(\theta_{ij}) \tag{2-2}$$

RLE value not near zero means that the number of up-regulated genes does not approximately equal the number of down-regulated genes. And a large RLE IQR reveals that most genes are differentially expressed. The RLE of a high quality array should be around 0 on a log scale.

### 2.2.6 RNA Degradation

The RNA can be degraded from its 5’ end. Thus, the intensities of probes at the $3'$ end of a probeset are higher than those at the $5'$ end. The RNA Degradation algorithm in R affy package uses a t-distribution linear model to identify the degradation of the RNA. High slope of RNA degradation line of an array indicates degradation, in other words, the poor quality.

### 2.2.7 Cut-off of Quality Control Metrics

We chose the cut-off for each quality control metric according to Lukk, Margus, et al. (2010) and Bolstad, et al. (2003). The final cut-off chosen for filtering the low quality array were: $-2<sfs<2$, $20<avBg<100$, $pps>35$, $NUSE<1.025$, $-0.15<RLE<0.15$, $RNAdeg<4.5$.

## 2.3 Microarray Preprocessing Methods

### 2.3.1 Introduction

Microarray preprocessing methods include three main steps (Figure 2-2): background correction, normalization and summarization. Several preprocessing approaches have been proposed. Two widely used methods are Affymetrix Micro Array Suite (MAS 5.0) and Robust Multi-array Analysis (RMA).

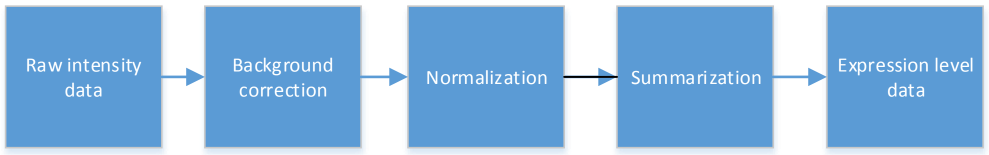

Figure 2-2. Preprocess work flow

**Background correction**

The raw intensity obtained from array usually includes the background intensities. Since even there is no RNA in the sample, the laser scanner can still detect low level of fluorescence on the array. The background correction step tries to remove the background noise from the raw intensity.

**Normalization**

Normalization detects and corrects systematic differences between arrays by removing the global effects, so that data from different arrays can be directly compared. Studies showed that the normalizing procedure has a marked impact on the final expression measures (Bolstad et al., 2002). After normalization, biological differences can be more easily detected.

**Summarization**

Summarization is the step to obtain expression measure, which represents the amount of the corresponding mRNA in original sample for each gene by summarizing probe intensities from each probeset on each array.

### 2.3.2 MAS 5.0 Preprocess.

Affymetrix Micro Array Suite 5.0 (MAS 5.0) is a set of models developed by Affymetrix company for image processing, signal quantification, background correction, preprocessing, scaling, and normalization of Affymetrix arrays.

**MAS 5.0 Background Correction**

MAS 5.0 background correction compute the background for each probeset by taking a robust average of the log ratios of PM to MM for each probe pair. MAS 5.0 background correction method provides a Tukey’s biweight estimate to adjust PM intensities for each gene (Hubbell et al., 2002).

**Scale Normalization**

Consider a matrix $X$ with $I$ rows and $J$ columns. Let $x_{ij}$ denote the entry in row $i$ and column $j$, let

$$m_j = median\left(x_{1j},\ x_{2j},\ ...,\ x_{Ij}\right). \tag{2-3}$$

$$MAD_j = median\left(\left|x_{1j} - m_j\right|, \left|x_{2j} - m_j\right|, ..., \left| x_{Ij} - m_j \right|\right). \tag{2-4}$$

Here, $\mathrm{MAD}_j$ represents median absolute deviation.

MAS 5.0 Scale normalization is to calculate the constant value of a column by multiplying all the entries in the $jth$ column by $C / MAD_j$, where $C = (\prod_{j=1}^{J} MAD_j)^{1/J}$

### 2.3.3 Robust Multi-array Analysis

Robust Multi-array Analysis (RMA) is written by Bolstad and is motivated by a log scale linear additive model (Rafael. A. et al., 2003). RMA preprocess method contains three steps: background correction, quantile normalization, and median polish summarization.

**RMA Background Correction**

RMA background correction method estimates a common mean background from perfect match on each array using a convolution model and then subtracts this background from perfect match to generate the corrected perfect match.

**Quantile Normalization**

In quantile normalization step, the perfect match intensities will be averaged, and the individual perfect match intensity will be replaced by the average.

Quantile normalization algorithm has five steps:

1. Build a matrix to store all the perfect match intensities of all the arrays that each column records the intensities belong to an array, each row represents the intensities across all arrays identify by the same probe.
2. Store the rank value of the intensities in each column from ascending order and set aside to use in step 6.
3. Sort each column in ascending order.
4. Calculate the mean value for each row.
5. Sort the mean values in ascending order and record the rank.
6. Replace the intensities with the mean value which have the same rank.

**Summarization**

Each gene is represented on the Affymetrix microarray by one or more probe sets. Median polish summarization step combines the probe-level intensities into one value representing the expression level of a gene using the robust median polish approach.

RMA median polish summarization include following steps:

1. For each probeset, build a matrix to store perfect match intensities for of all arrays that each column records the intensities belong to an array, each row represents the intensities across all arrays identify by the same probe.
2. Compute the median value for each row, and record the value as the row grand effect. Then, the intensities are subtracted with the row grand effect of this row.

3. Compute the median value for each column, and record the value as the column grand effect. Then, the intensities subtract with the column grand effect of this column.
4. Repeat steps 1 ~ 4 until no changes occur with the row or column medians and we got a residual matrix.
5. Original matrix subtracts with the residual matrix and then calculates the column mean for each column. The mean values are the expression of the probesets for the array.

## 2.4 Map/Reduce

Nowadays, researchers are facing increasing ultra large scale data sets. Recent developments in open source software based on MapReduce programming model, for example the Apache Hadoop project and associated software, provide a foundation for scaling analyses of  terabyte even petabyte scale on large clusters of commodity hardware in a reliable, fault tolerant manner (J. Dean et al., 2004). This software also provides a simple programming environment that makes it easy for programmers to design a parallel program that can efficiently perform a data-intensive computation.

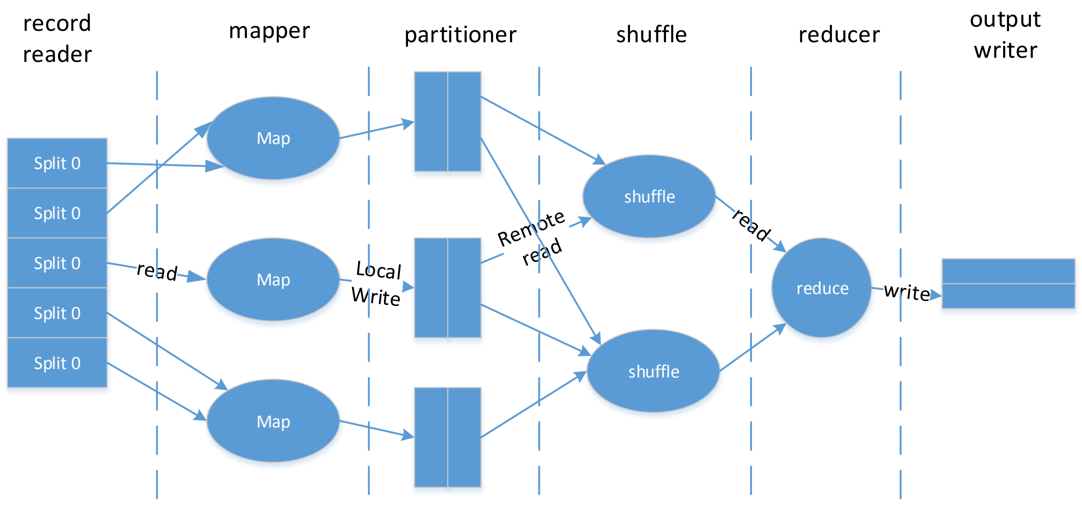

Figure 2-3. The illustration of MapReduce framework

The main idea of Map/Reduce programming model is splitting a large problem into sub-parts, computing partial solutions on sub-parts independently, and then assembling the partial solutions into the final solution (Figure 2-1). Standard MapReduce programming model includes seven major parts (Donald et al., 2012):

**Record reader**

The record reader parses an input data into records, which are data with default chunk size (typically 64 MB). Then, the record reader passes the data to the mapper in the form of $< key, value >$ pairs. The key contains the positional information and the value is the corresponding chunk of record data.

**Mapper**

The mapper runs the code provided by user on each $< key, value >$ pair to generate new intermediate $< key, value >$ pairs. We should carefully decide the key and value which will affect the MapReduce job accomplishment. The key is what the data will be grouped on and the value is the information being analyzed in the reducer.

$$Map(key,\ value) -> List(key',\ value')$$

**Combiner**

The combiner can assemble data of mappers on the same local node. It uses a user-provided method to aggregate values of the same intermediate key. Combiner can significantly reduce the amount of data that will be moved over the network when there are many intermediate pairs generated by mappers on computer node.

$$Combine(key',\ \mathrm{List}(values')) -> List(key'',\ value'')$$

**Partitioner**

The partitioner parses the intermediate $< key,\ value >$ pairs from the mapper (or combiner if available) into shards, and pass one shard to each reducer. The partitioner randomly distributes the keys equally over the reducers and sends the keys with the same value produced by mappers to the same reducer. The partitioner stores the data to the local file system, which will be retrieved by respective reducer later. The default behavior of the partitioner can be changed by the programmer.

**Shuffle and sort**

The shuffle and sort step is the first step of reduce task. The shuffles pull all the output files written by the partitioners to the local machine where the reducer is running. Then, the individual data pieces are sorted by key and form one larger data list. The sorted data will be easily iterated in the reduce task. This shuffle and sort handled by the framework automatically and cannot be changed by user. Developers can only control the way to sort and group key using a custom Comparator object through the configuration parameter provided by framework.

**Reducer**

The reducer applies the user-provided function to the grouped data once per key. The input of the function is the key and an iterator (i.e. Iterable calss) over the values of that key. Various functions can be implemented in reducer, such as aggregating, filtering, and combining. After the reducer executes the function, it generates zero or more $< key,\ value >$ pairs and sends to output writer.

$$\text{Reduce (key', List(values'))} \to \text{List(key'', value'')}$$

**Output writer**

The output writer receives the $< key,\ value >$ pairs from the reducer and formats it by separating the key and value with a tab and separating records with a newline. Then, the output writer writes it out to HDFS. The developers can define their own richer output format.

The parallelism of the MapReduce framework comes from the fact that each map or reduce operation can be executed on a separate processor independently of the others. Thus, the user simply defines the function $\mu$ as mapper function and function $\rho$ as reducer function, and the system automatically routes data to available processors.

## 2.5 Hadoop

Apache Hadoop is widely used open source software that implements the MapReduce parallel programming framework. Hadoop provides a simple programming interface that makes it easy for developers to efficiently design parallel programs for data intensive computations. Hadoop can be installed on large clusters (with thousands of nodes) and process vast amounts of data (as much as terabyte or even petabyte datasets) in parallel. No hardware modification is needed other than possible changes to meet minimum recommended RAM, disk space, etc. The initial version of Hadoop was created in 2004 by Doug Cutting inspired by Google's three famous MapReduce papers. Hadoop became a top-level Apache Software Foundation project in January 2008. There have been many academic and commercial contributors, such as Yahoo (Yahoo 2011.), Facebook (J. S. Sarma. 2011), Intel, Microsoft and etc., and a broad and rapidly growing user community.

The current Apache Hadoop platform is composed of three key functional components: the Hadoop Distributed File System (HDFS), Hadoop MapReduce and Hadoop Scheduler. The HDFS is a distributed file system and provides fault-tolerant access to large data. The Scheduler provides run-time tasks, such as scheduling, load balancing, failure recovery, inter-machine communication, and distributed partitioning of data. The Hadoop MapReduce supports the execution of Map/Reduce applications. It also consists of a number of utility projects such as Apache Hive, HBase and Zookeeper.

Each Hadoop MapReduce system includes of a single *master* node with one *JobTracker* and many *slave* nodes with several *TaskTrackers* (Figure 2-4), one TaskTracker per slave node. The master node schedules the job on the slave nodes, monitor them and re-execute the failed tasks. The slave nodes execute the jobs assigned by the master node.

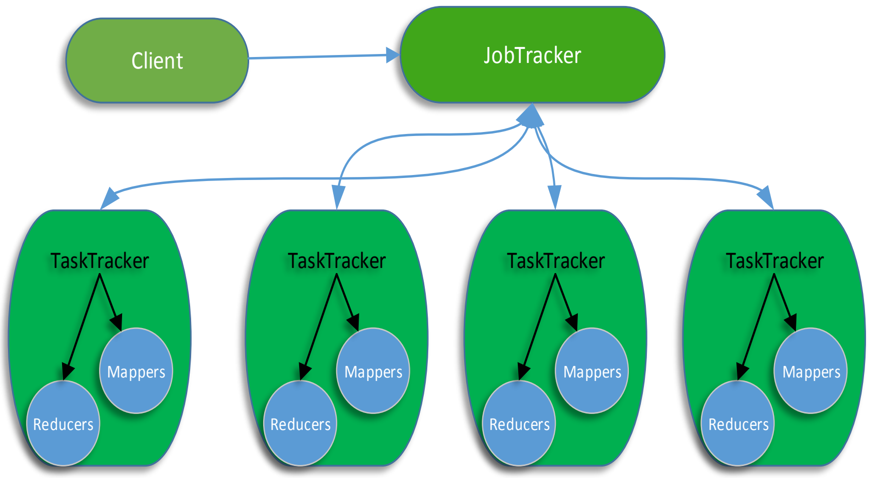

Figure 2-4. JobTracker and TaskTracker interaction in Hadoop

The HDFS allows parallel accessing the data across the nodes of the cluster using the MapReduce paradigm. For portability across a variety of platforms, HDFS is written in Java and only requires commodity hardware. In Hadoop, the compute nodes and the storage nodes are the same (Figure 2-4), namely, the MapReduce framework and the HDFS are running on the same set of nodes. Thus, the computation jobs can be effectively executed on the nodes where data is already presented.

There are three types of daemons in a standard HDFS cluster (Figure 2-3). The *namenode* stores file system metadata, stores file to block map, and provides a global picture of the file system. The *secondary namenode* performs internal namenode transaction log check pointing. Many *datanodes* store block data (file contents).

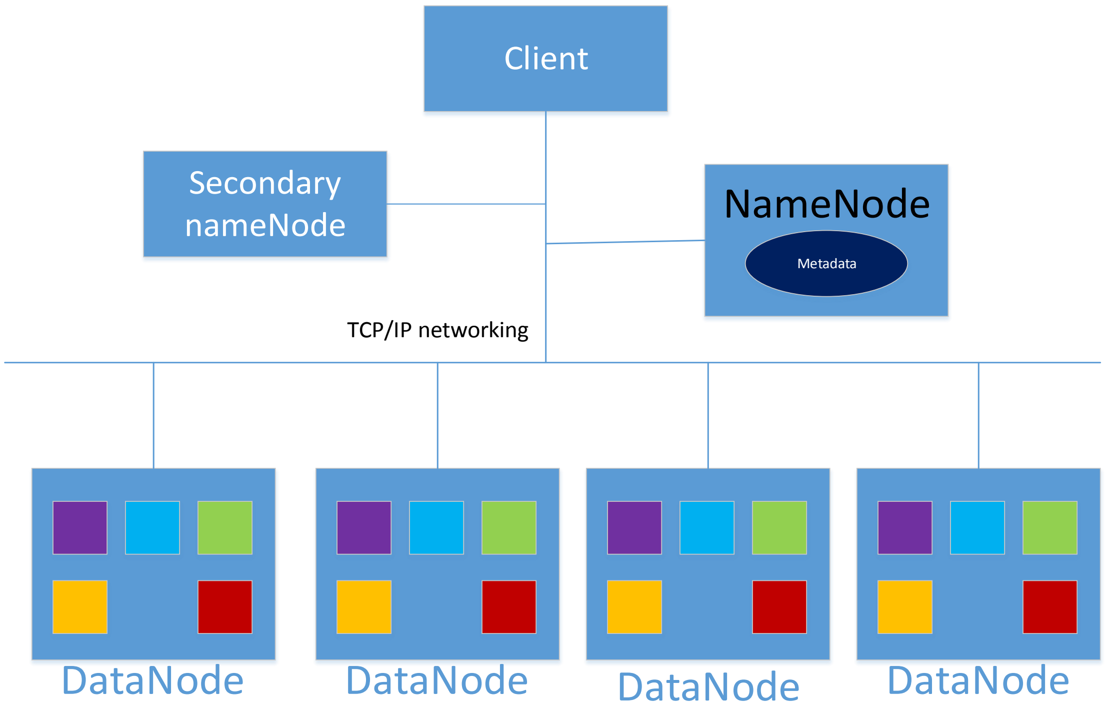

Figure 2-5. Architecture overview of HDFS

Advantages of Hadoop framework:

***Suitable to process a very large dataset.*** The Map/Reduce framework is designed to address data-intensive tasks with the emergence of Big Data.

***Multiple programming language API***. People can use codes written in other languages, such as Python, C, bsh, perl through Hadoop Streaming, which is a utility of Hadoop that allows users to create and run jobs with any executables as the mapper and/or the reducer. People can also use Hadoop pipes, a software development tool to implement MapReduce applications that connects programs written in C and C++ with a variety of high-level programming languages.

***Data locality***. As the data is collocated with the computing nodes in Hadoop, it can schedule Map tasks close to the data on the same node or the same rack.

***Fault-tolerant, shared-nothing architecture***. (M. Stonebraker, 1986) Tasks are independent in Map/Reduce framework except the output of mappers feeding into reducers under Hadoop control. Hadoop can detect node failures automatically and restart the task on other healthy nodes.

***Reliability***. In Hadoop, data is stored in HDFS and replicated across multiple nodes.

## 2.6 Parallel Computing Challenges

**MapReduce**

The MapReduce framework does not provide a general solution to big data. It provides clear boundaries for what you can and cannot do, making the number of options you have to consider fewer than those you may be used to. We have to fit our problems into the MapReduce framework, which might be challenging.

**Hadoop**

There is a challenge to load the data into and out of the HFDS files system as the HDFS cannot be directly mounted onto the existing operating system. We can only use I/O operating packages providing by Hadoop to manipulate the HDFS, all the Java original I/O functions become invalid.

Tuning Hadoop to achieve good performance is also a challenge. There are a large set of configuration parameters in Hadoop and many of them have an impact on performance. We need to familiar with the internal working of the Hadoop framework to optimally tuning these configuration parameters.

**Palmetto cluster**

The co-existing of Hadoop framework with HPC resource management systems is a challenge. Both systems have their own job submissions and management. Hadoop uses a shared-nothing style architecture, whereas most HPC resources including Palmetto cluster employ a shared-disk setup. Palmetto's Orange FS "newscratch" has compatibility issues with Java. We cannot directly operate data from newscratch parallel distribution by using Java I/O API. Also, the "local_scratch" mount on each node does not have enough space to store all the data. Furthermore, Palmetto cluster only allow commonly users submitting jobs running no more than 72 hours.

# Chapter 3

# DESIGN OF MAP/REDUCE BASED ANALYSIS TOOLS

## 3.1 Overview

Our development has four stages (Figure 3-1). In first stage, we implemented a sequential version of quality control and preprocessing algorithms using Java. We tested and optimized the sequential program to make sure that they are correct and efficiency. In the second stage, we analyzed the sequential program and found the potential steps that can be parallelized with Map/Reduce model. In the third stage, we implemented the Map/Reduce based parallel program, and deployed, tested it. In the fourth stage, we optimized the program. In this chapter, we discussed how to parallelize the microarray quality control and preprocessing methods with Map/Reduce framework.

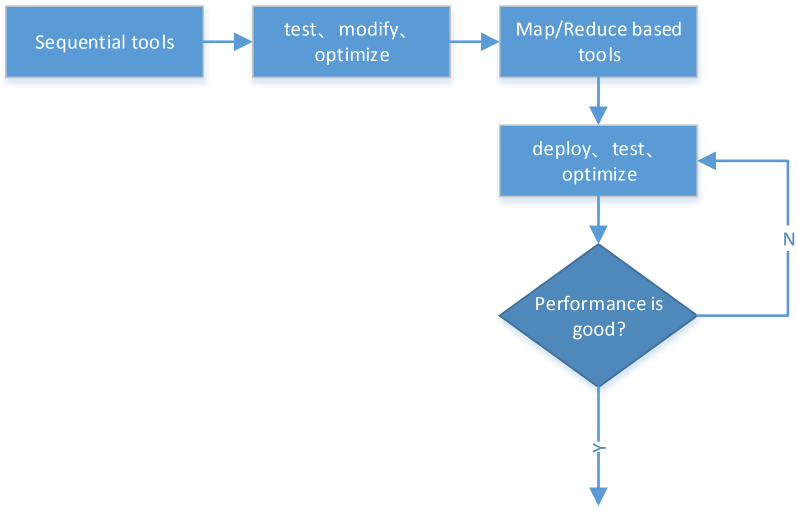

Figure 3-1. Software design work flow

Our Map/Reduce program contains three components: The first component reads all the intensities data from *.cel* file into HDFS and reads the *.cdf* file, extract the probe position information to HDFS. The second component does the MAS 5.0 and RMA preprocess calculations, including background correction, normalization and summarization. The third component performs quality control calculations of six parameters.

## 3.2 Read Array Information into HDFS

To perform quality control and preprocess, we first need to extract perfect match intensities and mis-match intensities for all arrays from each *.cel* file into HDFS. We used Affymetrix fusion Software Developers Kit (SDK), which is a Java package to parse Affymetrix GeneChip® microarray files to extract position information of each probe from *.cdf* file and extract perfect match and mis-match intensities from each *.cel* file. We used the probe position information to associate the intensities with corresponding probes.

The two main problems we faced in this stage are, (1) *.cel* files is stored in the “newscatch” orange file system (orange FS) and we can only use Java to read, copy and move data from this file system. However, directly reading data from or writing data to the orange FS lead to some unknown errors. Then, we first copied the data to the “local_scratch” local file system, and read and processed it. After we finished, we deleted the data from the local file system. (2) Fusion SDK cannot parse data stored in HDFS, we had to put data in the local file system and call fusion SDK API to extract the information and store them to the HDFS for later usage.

For the getting intensities step (Table 3-1), we used the mapper to read a file with the name of .cel files and send to reducer. The reducer first read the .cel file from orange FS to local file system. Then, it parsed the .cel file and stored intensities into HDFS. For getting probe positions (Table 3-2), we used a map-only to read the position information from .cdf file and stored in HDFS. Table 3-1, 2 lists the pseudo-code of Map/Reduce implementation for these two steps.

Table 3-1. Pseudo-code of map and reduce functions for getting intensities

```
Mapper:
    map (Long offset, String celName)
        emit(offset, celName);

Reducer:
    reduce (Long offset, List<String> celNames)
        for celName in celNames
            CDF cdf = new CDF ( get_cdf_data ( cdf_filename ))
            ChipSet chipset = new ChipSet ( cdf )
            File local = new File ( "/local_scratch/"+ celName )
            Copy_file ( new File (celPath + celName), local )
            Chip chip = new Chip ( cdf, get_cel_data( local ))
            chipset.add_chip ( chip )
        delete_file_or_directory( local )
        for chip in chipset
            String [] intensities = get_intensities_from_chip ( chip )
            emit ( celName, intensities )
```

Table 3-2. Pseudo-code of map functions for getting probe positions

```
Mapper:
   map (Long offset, String celName)
        CDF cdf = new CDF ( get_cdf_data ( cdf_filename ))
        File local = new File ( "./"+ celName )
        Copy_file ( new File (celPath + celName), local )
        Chip chip = new Chip ( cdf, get_cel_data( local ))
        delete_file_or_directory( local )
        String [] positions = get_position_from_chip ( chip, cdf )
        for probeset in chip
            emit ( probesetName, position)
```

## 3.3 Map/Reduce Implementation for MAS 5.0 Methods

Since the MAS 5.0 performs the quality control and preprocessing algorithms independently for each array, it is easy to parallelize the MAS 5.0 algorithms. We first re-implemented the MAS 5.0 algorithms in Java. We then created a map-only job to call the MAS 5.0 algorithm functions individually to process the intensities data for each array. The mapper wrote the name of the array to HDFS if it is low quality, or wrote the background corrected and normalized intensity values of the array to HDFS, otherwise. Table 3-3 lists the pseudo code of Map/Reduce implementation for MAS 5.0 methods.

Table 3-3. Pseudo-code of map function for MAS 5.0

```
Mapper:
    map ( String arrayName, String [] intensities )
        do_mas5_algorithm ( intensities )
        scaleFactor = get_scale_factor ()
        if output low quality array
            if scaleFactor < -2 || scaleFactor > 2
                emit ( arrayName, “sfs” )
            if averageBackground < 20 || scaleFactor > 100
                emit ( arrayName, “avbg” )
            if percentPresent < 35
                emit ( arrayName, “pps” )
        if output background corrected and normalized intensities
                emit (arrayName, intensities)
```

## 3.4 Map/Reduce Implementation for RMA Methods

### 3.4.1 Implementation of RMA Quality Control Methods

**PLM summarization Map/Reduce implementation**

We designed a Map/Reduce based job for PLM summarization. The mappers in this job read the intensities and position information from the files stored in HDFS. Then the mappers use the position information to determine which intensities belong to a given probeset. Mappers emitted probeset name and the intensities belong to this probeset as key-value pairs. Reducers received the key-value pairs and built a matrix to store all the intensities for each probe set. Each row of the matrix contains the intensities from the same array; each column contains the intensities identified by the same probe. After that, for each probeset, reducer called the PLM summarization function (our own Java implementation) to calculate the expression value and standard errors. Finally, the reducer wrote the expressions and standard errors to the HDFS. Table 3-4 lists the pseudo code of Map/Reduce implementation of PLM summarization.

Table 3-4. Pseudo-code of map and reduce functions for PLM summarization

```
Mapper:
    map (String arrayName, String [] PMintensities)
        String [] positions = get_position_from_file ( positionFile )
        Probesets = get_probeset_info_from_positions ( positions )
        for probeset in probesets
        String intensities
        find_intensities_belong_to_the_probeset (PMintensities, probeset )
            emit ( probesetName, intensities )

Reducer:
    reduce ( String probesetName, List<String> intensities )
        for intensitiesInTheSamechip in intensities
            z.addRow (intensitiesInTheSamechip )
        PLM_summarization ( z, expressions, standardErrors )
    emit ( probesetName, expressions )
    emit ( probesetName, standardErrors )
   .
```

**NUSE and RLE Map/Reduce Implementation**

We used one Map/Reduce job to calculate NUSE and RLE. This job contains two sub-jobs, one for computing NUSE metric, the other for calculating RLE metric. The reason we implemented these two algorithms together is that the work flow of these two algorithms are similar. We can reduce the codes for these two algorithms. Beside, running these two algorithms together can reduce the total running time.

In NUSE sub-job, the mapper read the standard errors (SEs) of probesets from HDFS and calculated the median SE. The mapper then computed NUSE values and emitted array name with NUSE value as output key-value pair. Reducers collected all NUSE values of an array, found the median of NUSE and calculated the IQR. Finally reducer wrote the array name to the HDFS if the array is a low quality array. Table 3-5 lists the pseudo code of Map/Reduce implementation of calculating NUSE.

Table 3-5. Pseudo-code of map and reduce functions for calculating NUSE

```
Mapper:
    map ( String probesetName, String []standardErrors )
        double median = calculate_median (standardErrors)
        if median = 0
            median = 1
        if median != -1
            for standardError in standardErrors
                standardError = standardError / median
        else
            for standardError in standardErrors
                standardError = NaN
        for standardError in standardErrors
            emit (arrayName, standardError)
Reducer:
    reduce ( String arrayName, List<String> standardErrors)
        String [] buffer = new String [standardErrors.length]
        for ith standardError in standardErrors
            if standardError = NaN
                buffer[i] = Double.positive_infinitive
            else
                buffer[i] = standardError
        median = get_median ( buffer )
        double [] IQR = do_quartiles ( buffer without positive infinitive value)
        if median > 1.025
        emit (arrayName, “NUSE” )
```

The RLE sub-job is almost the same as NUSE sub-job. First, each mapper read all the expression values from HDFS. Next, the mapper calculated the median expression and computed the RLE for each probeset. Third, mapper emitted array name and RLE as output key-value pair. Forth, each reducer collected all the REL values belong to an array, found out the median value among REL and calculated the IQR. At last, reducer wrote the array name to the HDFS if it's a low quality array. Table 3-6 lists the pseudo code of Map/Reduce implementation of calculating RLE.

Table 3-6. Pseudo-code of map and reduce functions for calculating RLE

```
Mapper:
    map ( String probesetName, String [] expressions )
        double median = calculate_median ( expressions )
        for expression in expressions
            expression = expression - median
            emit (arrayName, expression )

Reducer:
    reduce ( String arrayName, List<String> expressions )
        String [] buffer = new String [expressions.length]
        for ith expression in expressions
            buffer[i] = expression
        median = get_median ( buffer )
        double [] IQR = do_quartiles ( buffer without positive infinitive value)
        if median > 0.15 || median < -0.15
        emit (arrayName, "RLE" )
```

### 3.4.2 Implementation of RMA Preprocessing Methods

#### RMA background Correction Map/Reduce Implementation

The RMA adjusts background for each array individually. So, we used a mapper only job to perform background correction. The mapper read the array perfect match intensities from HDFS and corrected the background noise, then wrote the background corrected intensities to the HDFS for later usage. Table 3-7 lists the pseudo code of Map/Reduce implementation for RMA background correction.

Table 3-7. Pseudo-code of mapper for RMA background correction

```
Mapper:
    map (String arrayName, String []PMintensities)
            String [] backgroundCorrectedIntensities
            = background_correction ( PMintensities )
            emit (celName, backgroundCorrectedIntensities)
```

**RMA quantile normalization Map/Reduce Implementation**

We divide the quantile normalization job into three sub jobs: *"calculate mean"*, "*merge mean files"*, and *"do quantile normalize"*. The workflow to perform quantile normalization using those three sub tasks are shown in Figure 4-2. We implemented the Map/Reduce based algorithm for each sub task separately.

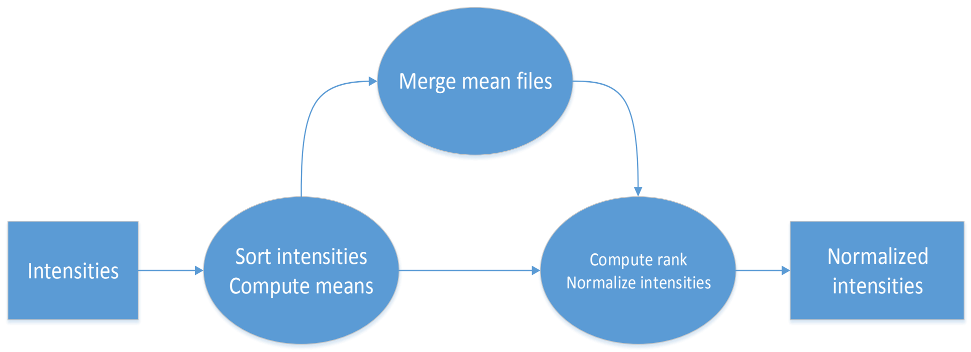

Figure 3-2: Map/Reduce normalization work flow

In "*calculate mean"* step, each mapper read the perfect match intensities from HDFS and sort these intensities. Then, the mapper emitted the rank $i$ and the $ith$ largest intensity as key-value pair to the reducer. Each reducer received the rank $i$ as key, the $ith$ largest intensities from all arrays as value. Reducer calculated the mean values of the $ith$

largest intensities and wrote it to HDFS. Table 3-8 lists the pseudo code of Map/Reduce implementation for calculating mean.

Table 3-8. Pseudo-code of map and reduce functions for mean calculation

```
Mapper:
   map (String arrayName, String [] PMintensities)
          sort ( PMintensities )
          for index in range 0 … size of PMintensities
               intensity = PMintensities[index]
               emit (index, intensity)

Reducer:
     reduce ( int index, List<String> intensities )
          mean = calculate_mean ( intensities )
     emit ( index, mean )
```

Since each reducer generated on mean file in "*calculate mean*" step, there are multi-files store the results in HDFS. We need an extra map/reduce job to merge all the files into one file. The mappers read each file from HDFS and emitted the key-value pairs to the reducer. Here, we specified one reducer to receive all the key-value pairs and write these key-value pairs to a file in HDFS. Table 3-9 lists the pseudo code for merging mean files.

Table 3-9. Pseudo-code of map and reduce functions for merging mean files

```
Mapper:
   map (int index, String mean)
          emit (index, mean)

Reducer:
     reduce ( int index, String mean )
          emit (index, mean)
```

The final step is to compute the rank for each mean value and replace the intensities having the same rank with the mean value. We created a map-only job for this step. The mapper read the mean values from the file in HDFS, and created a structure called *Item,* who contains two fields, *data* and *rank*. The *data* stores the intensity and *rank* contains the original rank of this intensity. For each intensity value, the mapper created *Item* and stored it to an *Item* list. Then, the mapper sorted the mean values and computed the rank for each mean values. Based on the rank, the mapper replaced the intensity with the corresponding mean value. Finally, the mapper wrote the normalized intensities to the HDFS. Table 3-10 lists the pseudo code for computing rank and normalizing intensities.

Table 3-10. Pseudo-code of map function for computing rank and normalizing intensities

```
Mapper:
    map (String arrayName, String [] intensities)
        String [] means = read_means_from_file ( mean_file )
        for intensity in intensities
                Item item = new Item ()
                item.data = intensity
                item.rank = index in intensities array
                itemList.add ( item )
        sort ( itemList )
        int [] ranks = rank_order ( means )
        for ith item in itemList
                if ranks[i]-$\lfloor ranks[i] \rfloor$>0.4
                        item.data = $0.5 \times mean[\lfloor ranks[\text{i}] \rfloor - 1] - mean[\lfloor ranks[\text{i}] \rfloor]$
                else
                        item.data = $0.5 \times mean[\lfloor ranks[\text{i}] \rfloor - 1]$
        // create new normalizedIntensities array according to itemList
                emit ( arrayName, normalizedIntensities )
```

**Median polish summarization Map/Reduce implementation**

The median polish summarization used the median polish method to perform the summarization of gene expressions. The map/reduce job for median polish summarization is similar to the job for PLM summarization. The mapper read the intensities and position information from the files stored in HDFS. Then the mappers use the position information to find the intensities for each probeset. Mappers emitted probeset name and its intensities as key-value pairs. Reducers received the key-value pairs and built a matrix to store all the intensities for each probe set. Then, the reducer called the median polish summarization function (our own Java implementation) to calculate the expression value for each probeset. Finally, the reducer wrote the expressions to the HDFS. Table 3-11 lists the pseudo code for median polish summarization.

Table 3-11. Pseudo-code of map and reduce functions for Median polish summarization

```
Mapper:
    map (String arrayName, String [] PMintensities)
        String [] positions = get_position_from_file ( positionFile )
        Probesets = get_probeset_info_from_positions ( positions )
        for probeset in probesets
        String intensities
      = find_intensities_belong_to_the_probeset (PMintensities, probeset )
            emit ( probesetName, intensities )

Reducer:
    reduce ( String probesetName, List<String> intensities )
        for intensitiesInTheSamechip in intensities
            z.addRow (intensitiesInTheSamechip )
        median_polish_summarization ( z, expressions)
    emit ( probesetName, expressions )
```

## 3.5 Map/Reduce Implementation for RNA Degradation Method

The RNA degradation was also calculated for each array individually. We designed a map-only job to calculate the RNA degradation. In this job, each mapper call RNA degradation function (our own Java implementation) to compute the RNA degradation metric for an array and output the array name if the array is low quality. Table 3-12 lists the pseudo code of Map/Reduce implementation for RNA degradation.

Table 3-12. Pseudo-code of map function for RNA degradation

**Mapper**:
```
    map ( String arrayName, String [] PMintensities )
              String [] positions = get_position_from_file ( positionFile )
              Probesets = get_probeset_info_from_positions ( positions )
              for probeset in probesets
              String intensities
            = find_intensities_belong_to_the_probeset (PMintensities, probeset )
            String [] loggedIntensities = log_2 ( intensities )
            loggedIntensitiesList.addRow (loggedIntensities )

    double [] means
        = get_mean_according_to_intensities_list ( loggedIntensitiesList )
    double [] standardDeviations
    = get standard_deviation_according_to_intensities_list ( loggedIntensitiesList )
       firstMean = get_first_element ( means )
       for ith mean in means
              mean = mean – firstMean / (standardDeviations[ i ] / $\sqrt{N}$ )
       double slope = linear_regression ( means )

       if slope > 4.5
              emit ( arrayName, "RNA degradation" )
```

# Chapter 4

# EXPERIMENTAL RESULTS

## 4.1 Introduction

We applied our tools to the microarray data of Arabidopsis and rice, the two model plants with complete genome sequences.

All CEL files download from the Gene Expression Omnibus (GEO) website http://www.ncbi.nlm.nih.gov/geo/. GEO is a public repository that stores microarray and other forms of high-throughput functional genomic data. The data in GEO is free to public.

## 4.2 Sequential Tools for micro array data analysis

We compared our Map/Reduce based microarray analysis tools to sequential tools, MAS 5.0 tools (apt-mas5) from Affymetrix power tools, RMA preprocess tool and RMA quality control tool from RMAexpress. These tools are widely used by bioinformatics communities.

**Affymetrix Power Tools**

The Affymetrix Power Tools (APT) is a set of cross-operating system command line programs developed by Affymetrix using C/C++ language for processing and analyzing data from any Affymetrix GeneChip® array. The APT is obtained from the main APT website, http://www.affymetrix.com/support/developer/powertools/index.affx.

**RMAexpress**

RMAExpress is a program written in C/C++ language for Windows (and Linux) to calculate gene expression values for Affymetrix Genechip® data using the Robust Multichip Average (RMA) expression summary. RMAExpress is available through http://rmaexpress.bmbolstad.com/.

## 4.3 Hadoop Deployment and Execution

Palmetto cluster uses TORQUE as its standard batch processing systems. In this thesis, we used a Portable Batch System (PBS) wrapper script based on MyHadoop (Krishnan et al., 2011) script to provide Hadoop instances on traditional supercomputing resources. The Hadoop system requests resources via TORQUE and Hadoop environment is configured based on the set of resources TORQUE provided (Figure 4-1).

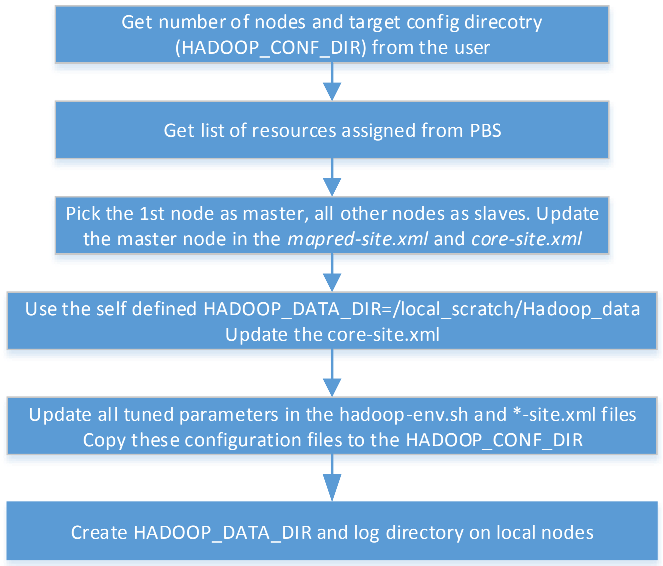
Get number of nodes and target config direcotry (HADOOP_CONF_DIR) from the user
Get list of resources assigned from PBS
Pick the 1st node as master, all other nodes as slaves. Update the master node in the *mapred-site.xml* and *core-site.xml*
Use the self defined HADOOP_DATA_DIR=/local_scratch/Hadoop_data Update the core-site.xml
Update all tuned parameters in the hadoop-env.sh and *-site.xml files Copy these configuration files to the HADOOP_CONF_DIR
Create HADOOP_DATA_DIR and log directory on local nodes

Figure 4-1. Hadoop deployment work flow

### 4.3.1 Deploying and configuring Hadoop:

To deploy Hadoop on the palmetto cluster, we first created a PBS command *qsub* with configure parameters to request number of nodes we want (select= ), size of memory per node (mem=), number of cores per node (ncpu=), the kind of network we want to use (interconnect=) and how many time we what to run the Map/Reduce job (walltime=). Then, we generated a PBS script to record the number of resources (nodes) we required and configured the site specific parameters using the PBS wrapper configuration scripts together with the tuned Hadoop configuration files to generate new Hadoop configuration directory (HADOOP_CONF_DIR)

```
$MY_HADOOP_HOME/bin/pbs-configure.sh -n $nodes -c $HADOOP_CONF_DIR
$HADOOP_HOME/bin/hadoop --config $HADOOP_CONF_DIR namenode -format
$HADOOP_HOME/bin/start-all.sh
```

These scripts created and formatted HDFS and started the Hadoop daemons automatically. After configuration, we uploaded input data into the directory (i.e., input folder) of HDFS using Hadoop command,

```
$HADOOP_HOME/bin/hadoop --config $HADOOP_CONF_DIR fs -put input input
```

The Hadoop files that we need to configure include:

*Masters*: This file set a node with machine name or ip address as master node.

*Slaves*: This file specifies the nodes with machine name or ip address as the slave nodes on the cluster.

*hadoop-env.sh*: This script contains some environment variable settings used by Hadoop , such as the location of the logs, the maximum heap size, and JVM parameters for garbage collection and heap management.

*mapred-site.xml*: This is the MapReduce site configuration file that includes important parameters, such as the number of parallel copies reducer use to download mappers output results, the host and port for the JobTracker, the JAVA_OPTS for the child JVMs of the mappers and reducers  and the maximum number of map and reduce tasks .

*core-site.xml*: This is the core site configuration file that consists of the location of the HDFS (HADOOP_DATA_DIR) on every node, and the URI for the HDFS server .size of the read/write buffers and in-memory file system to merge map outputs, the memory limit used for sorting data. .

*hdfs-site.xml*: This is the HDFS site configuration file that includes parameters for configuring the distributed file system, for example, the number of replications, the number of DataNode handlers and the HDFS block size.

### 4.3.2 Running jobs on Hadoop

By far, all setup steps had been done, and then we can start running our program in configured Hadoop environment (Figure 4-2).

```
$HADOOP_HOME/bin/hadoop jar mapReduceApplication.jar
```

Once the Hadoop jobs were finished, the results can be downloads back from HDFS.

```
$HADOOP_HOME/bin/hadoop --config $HADOOP_CONF_DIR fs -get output output
```

Downloading output files back to shared file system is important because the output files are stored in HDFS distributed across the compute nodes and PBS in Palmetto Cluster will clear the local file system on the required nodes after the PBS job is finished. Thus, all results must be saved before the resources are re-allocated. Finally, we shut down all Hadoop daemons and exited PBS.

```
$HADOOP_HOME/bin/stop-all.sh
$MY_HADOOP_HOME/bin/pbs-cleanup.sh -n $nodes
```

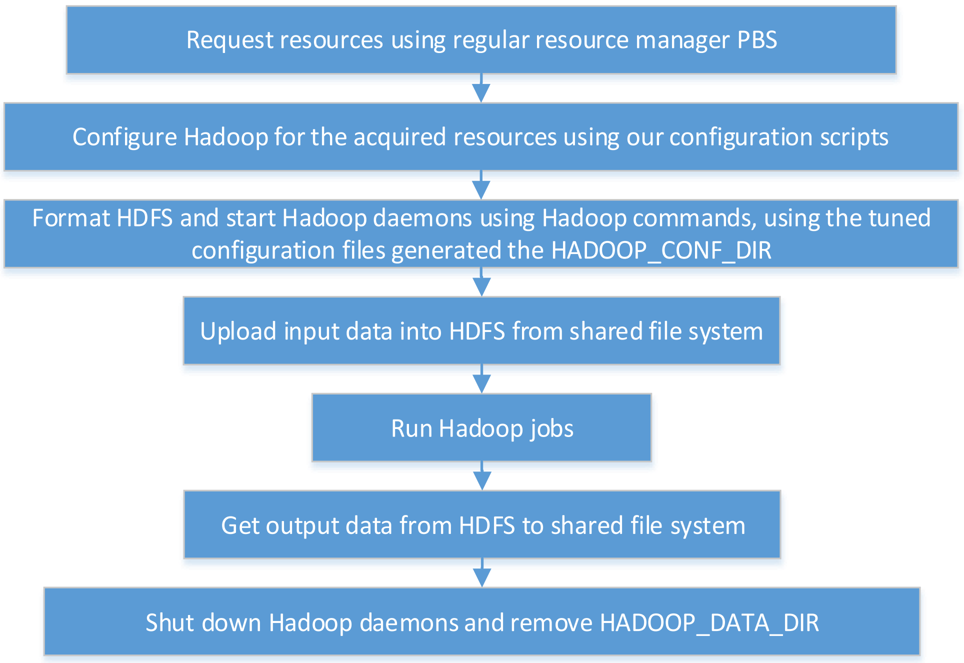
Request resources using regular resource manager PBS
Configure Hadoop for the acquired resources using our configuration scripts
Format HDFS and start Hadoop daemons using Hadoop commands, using the tuned configuration files generated the HADOOP_CONF_DIR
Upload input data into HDFS from shared file system
Run Hadoop jobs
Get output data from HDFS to shared file system
Shut down Hadoop daemons and remove HADOOP_DATA_DIR

Figure 4-2. Hadoop MapReduce jobs execution workflow on Palmetto cluster

## 4.4 Analyzing the Rice Dataset

### 4.4.1 Rice microarray dataset:

We downloaded 1778 rice microarray data using the Affymetrix Rice Genome Array chip (GPL 2025) from GEO (Edgar et al., 2002) for our analysis. The detail of this dataset is listed in Table 4-1.

Table 4-1. Description of rice dataset

| | |
|---|---|
| Number of *.cel* files | 1778 |
| *.cel* file size | 13MB~32MB |
| Total *.cel* files size | 26G |
| *.cdf* file size | 108.8MB |
| Number of probe sets | 57381 |

### 4.4.2 Sequential microarray tools evaluation using Rice microarray data

We run the rice microarray data using the RMAExpress tools on desktop computer in our laboratory, the detail of the desktop computer shown in table 4-2. We tested apt-mas5 on Palmetto Cluster, since apt-mas5 require large memory. We applied 1 node with 30GB memory to run the apt-mas5 program. The detail of the node is shown on table 4-3. The testing results revealed that sequential tools need more time and more memory to process the massive microarray dataset: the apt-mas5 program requires big memory (30GB) to process the results and the RMAexpress requires longer time (12 hours) as shown in Table 4-4.

Table 4-2. Sequential tools running environment on lab desktop

| |
|---|
| • 1 node with 1 cpu and 1 core (4 core per chip, totally 2 chips) Intel i7 2600 @ 3.4GHz HP DL980G7 |
| • 12GB DDR3 1333 MHz RAM |
| • 1TB SATA drives @ 113.24 MB/sec |
| • Fedora release 18 (Spherical Cow) with 3.6.9-200.fc18..x86_64 kernel |
| • Oracle Java(TM) SE Runtime Environment (build 1.7.0_21-b11) with Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode) |

Table 4-3. Sequential tools running environment on Palmetto Cluster

| |
|---|
| • 1 node with 1 cpu and 1 core (8 core per chip, totally 8 chips) Intel Xeon 7542 @ 2.8GHz HP DL980G7 |
| • 100GB of 1 TB DDR2 1600 MHz RAM |
| • *“local_scratch”* 99GB 10000 rpm SATA drives |
| • 10Gb Myrinet network interface |
| • Scientific Linux release 6.1 (Carbon) with 2.6.32-220.4.1.el6.x86_64 kernel |
| • Oracle Java(TM) SE Runtime Environment (build 1.7.0_21-b11) with Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode) |

Table 4-4. Sequential tools running time

| APT (apt-mas5) | RMA preprocess tool | RMA quality control tool |
|---|---|---|
| 3hours 45mins 61sec | 49mins 17sec | 12hours 19mins 9sec |

### 4.4.3 Map/Reduce based microarray analysis evaluation

We tested our Map/reduce based microarray analysis tools on Palmetto Cluster. The details of the nodes we request are shown in Table 4-6. We request 5 nodes, 10 nodes and 20 nodes respectively to test our tools. Our results showed that: (1) for some jobs, like “get intensities” (Table 4-7) and “do median polish summarization” (Table 4-11), when we doubled the number of core to process the data, the running time just got slightly reduced. This is because these tasks write large amount of data to the HDFS or transfer data across the network (mapper send temporary output results to the reducers on other nodes), and lots of the time is wasted in waiting for the I/O operations. (2) for some jobs, such as, “do RNA degradation” (Table 4-15), “compute NUSE & RLE” (Table 4-16), “do background correction” (Table 4-7), even though we doubled the number of total cores, the running time is only decreased a little bit. The reason is that these jobs are running too fast, and most of the time is used to start up job, schedule job, clean up job, these steps taking fixed time. (3) for remain jobs, “compute mean” (Table 4-8), “do quantile normalize” (Table 4-10), “do PLM summarization” (Table 4-13), “do MAS 5.0” (Table 4-14), when we increased to the number of nodes from 5 to 10, 10 to 20, even 20 to 40, the running time is reduced to nearly half.

Based on the quality cutoff threshold, we identified that 278 of 1778 rice microarray data are low quality (having at least one of six measures without satisfying threshold). Table 4-20 lists the number of low quality rice array detected by each measure.

Table 4-5. Map/Reduce based microarray tools testing environment

| |
|---|
| • 8 cores per chip, 2 chip Intel E5-2665 @2.4GHz HP SL250s |
| • 8GB of 64 GB DDR3 1600 MHz RAM |
| • *“local_scratch”* 950 GB 10000 rpm SATA drives |
| • 10/40Gb InfiniBand network interface |
| • Scientific Linux release 6.4 (Carbon) with 2.6.32-358.2.1.el6.x86_64 kernel |
| • Oracle Java(TM) SE Runtime Environment (build 1.7.0_21-b11) with Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode) |

Table 4-6. Comparison of time to “get intensities”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 4mins, 8sec (248sec) | $P_{reduce}$=40 |
| 10 | 3mins 11sec (191sec) | $P_{reduce}$=80 |
| 20 | 2mins 40sec (160sec) | $P_{reduce}$=160 |

Table 4-7. Comparison of time to “do background correction”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 1mins, 26sec (86sec) | $P_{map}$=30 |
| 10 | 1mins 9sec (69sec) | $P_{map}$=60 |
| 20 | 1mins 3sec (63sec) | $P_{map}$=120 |

Table 4-8. Comparison of time to” compute mean”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 6mins, 5sec (365sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 2mins, 58sec(178sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 1mins, 33sec (93sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-9. Comparison of time to “compute mean” (no combiner)

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 9mins, 53sec (593sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 5mins, 36sec (336sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 2mins, 54sec (174sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-10. Comparison of time to “do quantile normalize”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 5mins 5sec (305sec) | $P_{map}$=30 |
| 10 | 3mins, 8sec (188sec) | $P_{map}$=60 |
| 20 | 3mins, 27sec (207sec) | $P_{map}$=120 |

Table 4-11. Comparison of time to "do median polish summarization"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 4mins, 12sec (252sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 3mins, 1sec (181sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 2mins, 30sec (150sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-12. Comparison of time to chain "do quantile normalize" and "do median polish Summarization" together

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 6mins, 47sec (407sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 4mins 51sec (291sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 4mins 58sec(298sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-13. Comparison of time to "do PLM summarization"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 45mins, 5sec (2705sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 22mins, 43sec (1363sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 14mins, 53sec (893sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-14. Comparison of time to "do MAS 5.0"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 27mins 40sec (1660sec) | $P_{map}$=30 |
| 10 | 14mins, 5sec (845sec) | $P_{map}$=60 |
| 20 | 7mins, 38sec (458sec) | $P_{map}$=120 |

Table 4-15. Comparison of time to "do RNA degradation"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 46 sec | $P_{map}$=30 |
| 10 | 41 sec | $P_{map}$=60 |
| 20 | 59 sec | $P_{map}$=120 |

Table 4-16. Comparison of time to "compute NUSE and RLE"

| Number of nodes | Running time | Number of cores used by jobs |
|---|---|---|
| 5 | 1mins, 41sec (101sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 1mins, 2sec (62sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 1mins, 15sec | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-17. Comparison of time to "do quality control methods"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 5 | 59mins, 50sec (3590sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 31mins, 32sec (2006sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 23mins, 41sec (1421sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-18. Comparison of time to "do preprocessing"

| Number of nodes | Running time | Number of cores used by jobs |
|---|---|---|
| 5 | 16mins, 16sec (976sec) | $P_{map}$=30, $P_{reduce}$=40 |
| 10 | 9mins 33sec (573sec) | $P_{map}$=60, $P_{reduce}$=80 |
| 20 | 8mins 22sec(502sec) | $P_{map}$=120, $P_{reduce}$=160 |

Table 4-19. Number of low quality chips being detected by metrics

| RNA degradation | Average background | Scale factor | Present percentage | NUSE | RLE | Total |
|---|---|---|---|---|---|---|
| 49 | 9 | 105 | 21 | 222 | 69 | 278 |

## 4.5 Analyzing the Arabidopsis Dataset

### 4.5.1 Arabidopsis microarray dataset:

We downloaded 9031 Arabidopsis microarray data using the Affymetrix Arabidopsis ATH1 Genome Array (GPL 198) from GEO for our analysis. The detail of this dataset is listed in Table 5-21.

Table 4-20. Description of Arabidopsis dataset

| | |
|---|---|
| Number of *.cel* files | 9031 |
| *.cel* file size | 4.9MB~12MB |
| Total *.cel* files size | 59G |
| *.cdf* file size | 39MB |
| Number of probesets | 22810 |

### 4.5.2 Sequential microarray tools evaluation using Arabidopsis data

We run the Arabidopsis microarray data using sequential tools on Palmetto Cluster in Clemson University. The running environment required is the same as we test the rice dataset. Apt-mas5 program occur unhandled exception while processing the Arabidopsis dataset. The running time of RMAExpress quality control tool excess 72 hours limitation. Thus, we were not able to obtain results for Arabidopsis microarray data using sequential tools.

### 4.5.3 Map/Reduce based microarray analysis evaluation

We tested our Map/reduce based microarray analysis tools using Arabidopsis data on Palmetto Cluster. The details of the nodes we requested are the same as those for running rice microarray data. We requested 10 nodes, 20 nodes, 40 nodes respectively to test our tools. The system configurations for each node are the same as we testing the rice dataset. Since the data size of the Arabidopsis dataset is much larger than that of rice, our experimental results are a little bit different. Our results showed that: (1) The running times of jobs like “get intensities” (Table 4-21) and “do median polish summarization”

(Table 4-25) were decreased slowing when we double the running cores. (2) The running times of jobs such as "do RNA degradation" (Table 4-28), "do PLM summarization" (Table 4-26), were reduced to half when the number of nodes are increased from 10 to 20. However, the running time were not reduced much when the nodes were increased from 20 to 40. (3) The running time of jobs, such as "compute NUSE & RLE" (Table 4-29), "do background correction" (Table 4-22), "compute mean" (Table 4-23), "do quantile normalize" (Table 4-24), "do MAS 5.0" (Table 4-27), were reduced to nearly half when we increased to nodes from 10 to 20, and from 20 to 40.

Based on the quality cutoff threshold, there are 3286 low quality Arabidopsis microarray data (having at least one of six measures without satisfying threshold) of 9031 data. Table 4-33 lists the number of low quality rice array detected by each measure.

Table 4-21. Comparison of time to "get intensities"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 9mins, 32sec(572sec) | $P_{reduce}$=120 |
| 20 | 6mins 18sec (378sec) | $P_{reduce}$=240 |
| 40 | 4mins 50sec (290sec) | $P_{reduce}$=560 |

Table 4-22. Comparison of time to "do background correction"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 4mins 25sec (265sec) | $P_{map}$=20 |
| 20 | 2mins 7sec (190sec) | $P_{map}$=40 |
| 40 | 1mins 9sec (69sec) | $P_{map}$=80 |

Table 4-23. Comparison of time to "compute mean"

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 6mins 20sec (380sec) | $P_{map}$=20, $P_{reduce}$=120 |
| 20 | 2mins 58sec (178sec) | $P_{map}$=40, $P_{reduce}$=240 |
| 40 | 1mins 34sec (94sec) | $P_{map}$=80, $P_{reduce}$=560 |

Table 4-24. Comparison of time to “do quantile normalize”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 6mins 44sec (404sec) | $P_{map}$=20 |
| 20 | 3mins 3sec (183sec) | $P_{map}$=40 |
| 40 | 1mins 37sec (97sec) | $P_{map}$=80 |

Table 4-25. Comparison of time to “do median polish summarization”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 9mins 41sec (581sec) | $P_{map}$=20, $P_{reduce}$=120 |
| 20 | 8mins 22sec (502sec) | $P_{map}$=40, $P_{reduce}$=240 |
| 40 | 7mins 3sec (423sec) | $P_{map}$=80, $P_{reduce}$=560 |

Table 4-26. Comparison of time to “do PLM summarization”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 220mins 21sec (13221sec) | $P_{map}$=20, $P_{reduce}$=120 |
| 20 | 120mins (7200sec) | $P_{map}$=40, $P_{reduce}$=240 |
| 40 | 82mins 40sec (4960sec) | $P_{map}$=80, $P_{reduce}$=560 |

Table 4-27. Comparison of time to “do MAS 5.0”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 41mins 59sec (2519sec) | $P_{map}$=20 |
| 20 | 19mins 36sec (1176sec) | $P_{map}$=40 |
| 40 | 10mins 44sec (644sec) | $P_{map}$=80 |

Table 4-28. Comparison of time to “do RNA degradation”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 1mins 36sec (96sec) | $P_{map}$=20 |
| 20 | 46 sec | $P_{map}$=40 |
| 40 | 30 sec | $P_{map}$=80 |

Table 4-29. Comparison of time to “compute NUSE and RLE”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 3mins 40sec (220sec) | $P_{map}$=20, $P_{reduce}$=120 |
| 20 | 2mins 0sec (120sec) | $P_{map}$=40, $P_{reduce}$=240 |
| 40 | 1mins 5sec (65sec) | $P_{map}$=80, $P_{reduce}$=560 |

Table 4-30. Comparison of time to “do quality control methods”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 224mins 1sec (1344sec) | $P_{map}$=20, $P_{reduce}$=120 |
| 20 | 121mins 9sec (7269sec) | $P_{map}$=40, $P_{reduce}$=240 |
| 40 | 83mins 49sec (5029sec) | $P_{map}$=80, $P_{reduce}$=560 |

Table 4-31. Comparison of time to “do preprocessing”

| Number of nodes | Running time | Number of cores used by job |
|---|---|---|
| 10 | 24mins, 50sec (1490sec) | $P_{map}$=20, $P_{reduce}$=120 |
| 20 | 15mins 29sec (929sec) | $P_{map}$=40, $P_{reduce}$=240 |
| 40 | 9mins 15sec (573sec) | $P_{map}$=80, $P_{reduce}$=560 |

Table 4-32. Number of low quality chips being detected by metrics

| RNA degradation | Average background | Scale factor | Present percentage | NUSE | RLE | Total |
|---|---|---|---|---|---|---|
| 1555 | 1009 | 778 | 34 | 1535 | 453 | 3286 |

# Chapter 5

# TUNNING AND OPTIMIZING

## 5.1 Code Level Optimizing

### Use a combiner

Combiner can decrease the number of data sent to the reducers (White, Tom, 2012). For instance, in one of our Map/Reduce microarray analysis tool, the "compute mean" job, the mapper will send millions of (index, intensity) pairs to the reducer. If we use a combiner to assemble the intensities generated by mappers on one node, we can just send one key-value pair (index, (sum (local_intensities), N)), where N is the number of intensities. N usually is much larger than 1. The Figure 5-1 shows that using a combiner dramatically improved the performance of the job of getting mean.

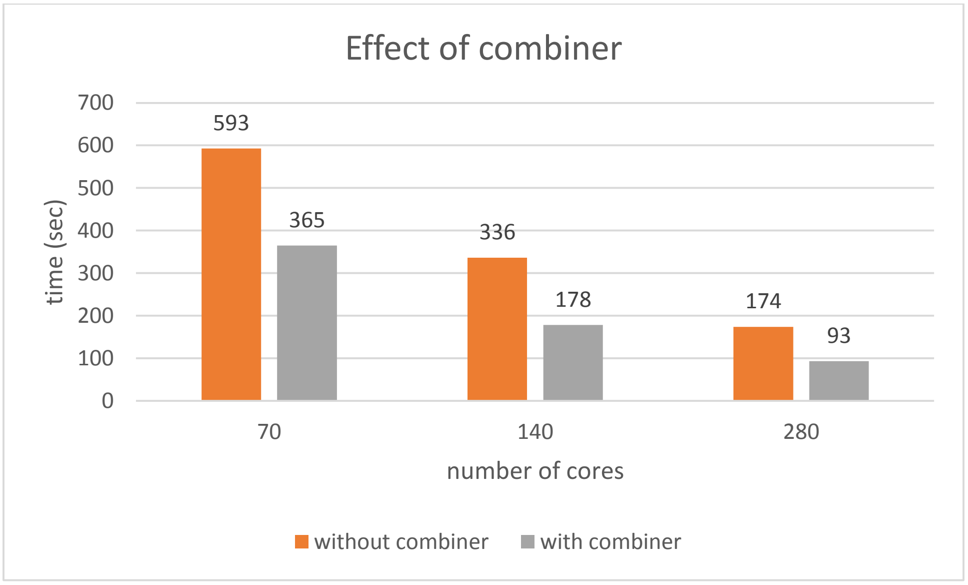

Figure 5-1. Effects of applying combiner on "compute mean"

**Create map-only jobs**

Map-only job means that there is no reducer . Map-only job are efficient, since no data is needed to be transmitted from the mapper to the reducer. Most of the map tasks write output to HDFS directly. In our Map/Reduce based microarray analysis tools we implemented “get positions”, “get background correction”, “do MAS 5.0 algorithm”, “do RNA degradation” as map-only jobs.

**Concurrently submit independent jobs**

In Hadoop 1.x, reducer cannot reuse the mapper slots, so does mapper. Furthermore, to avoid mapper-reducer confliction, we usually require equal number of mappers and reducers. If we have two or more independent jobs, we can submit the jobs at the same time and could utilize the cluster resources better. For example, the “do PLM summarization” job will spend lots of time in the reduce phase. If we concurrently submit PLM summarization job with map-only jobs like “do MAS 5.0” job and “do RNA degradation” job, after map phase of “do PLM summarization” job is finished, we can reuse the mapper slots to run “do MAS 5.0” job and “do RNA degradation” job. In this way we can reuse the cores and decrease the overall running time.

Hadoop provide a *submit* function to submit job for independent jobs:

```
Job job = new Job (new configuration());
job.submit();
```

The Figure 5-2 shows the performance improvement of concurrently submission of “do PLM summarization” job, “do MAS 5.0” job and “do RNA degradation” job.

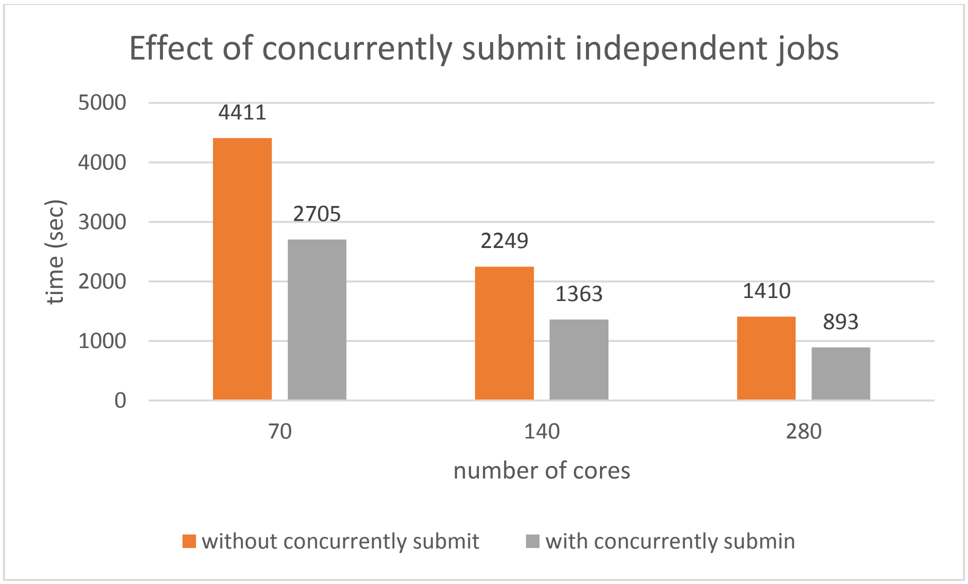

Figure 5-2: Effect of Concurrently submission “do PLM summarization”, “do MAS 5.0” and “do RNA degradation” jobs simultaneously

**Use ChainMapper and ChainReducer**

ChainMapper and ChainReducer are special Hadoop mapper and reducer classes that can be used to chain multiple mappers as one mapper and one reducer with multiple mappers as one reducer (Miner et al., 2012). The output results of each chained map phase are directly sent to the next map phase through the pipeline. In this way, the map-only job would not have to write the results into HDFS and read by the following job later. In our Map/Reduce microarray analysis tools, we use ChainMapper to bind map-only “do compute rank & normalize” job with “do median polish summarization” job. Figure 5-3 shows the using chainMapper can reduce the running time.

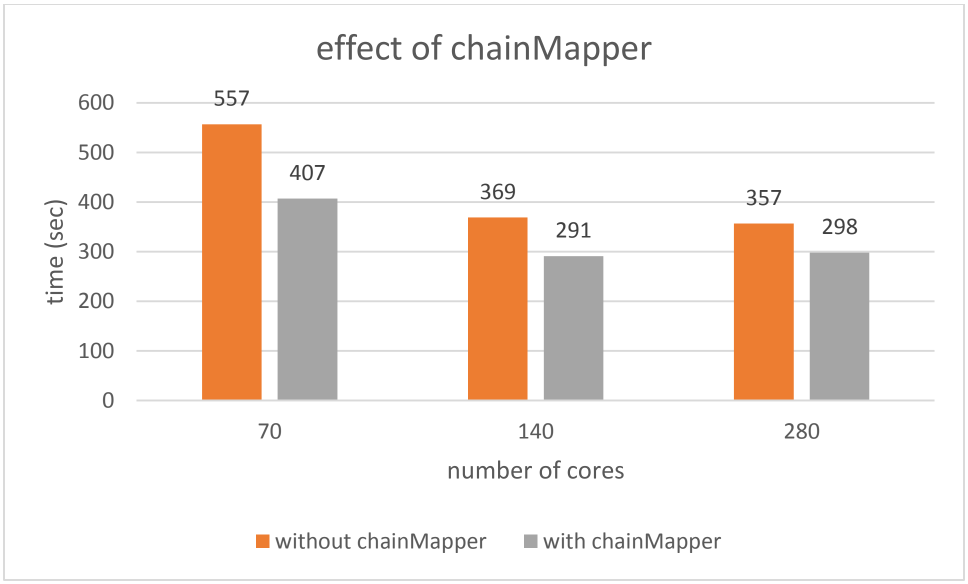

Figure 5-3: Effect of using chianMapper to combine "do normalize" and "do median polish summarization" jobs together

**Use the most appropriate and compact writable type**

Converting numeric data to and from strings is inefficient and can actually use out a significant portion of CPU time . The binary Writable types will cost less space comparing to Text data. Since disk I/O and network transferring will become a bottleneck in large jobs, using VIntWritable or VlongWritable can save transmission time. In our experiments, we learned that using Writable types effectively reduces the network traffic (data not shown).

**Reuse Writables**

One of the coding mistakes is allocating new Writable object for every output from mappers or reducers. For example,

```
for (String word : words) {
        context.write(new Text(word), new Intwritable(1)
}
```

This will lead to the creation of thousands of very short-lived objects. Hence, reusing existing Writables will significantly reduce the usage of the memory and avoid garbage collection. We can rewrite the above code as following:

```
Text wordText = new Text ();
IntWritable outvalue = new IntWritable(1);
for (String word : words) {
   wordText.set(word);
   context.write(wordText, one);
}
```

In this way, we can avoid creating temporary objects, and can greatly save the running time (data not shown).

## 5.2 Hadoop Level Tuning

Since Hadoop and HDFS are complex distributed systems that run arbitrary user code. To take the advantage of the cluster, we need to tune the Hadoop system to achieve optimal performance.

### 5.2.1 Hadoop Configuration Tuning

**Compression**

Hadoop supports compression at 3 different levels – input data, intermediate map output and reduce output data – as well as multiple codecs, like bzip2, lzma, gzip, lzo, snappy, which can be used for compression and decompression (Chen et al., 2010) . Some codecs strike a better compression factor but take longer to compress and to decompress. Some codecs have a fine balance between the compression factor and the overhead of compression and decompression activities.

Compressing reducer output can reduce the usage of HDFS. We can use the method FileOutputFormat to set the properties.

```java
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, Codec,class);
```

where Codec.class can be LzoCodec.class, GzipCodec.class or SnappyCodec.class

Compressing map outputs can reduce the disk and network I/O while increases CPU cycles for compression and decompression temporary output data. If the map outputs are very large, enabling map output compression will surely reduce total job running time. The useful parameters related to intermediate map output compression are *mapred.map.output.compression.codec* (specify the compression codec), *mapred.compress.map.output* (whether to compress the map output, false by default), which can be found in *mapred-site.xml*

The Java code for setting map output compression is

```java
Configuration conf = new Configuration();
conf.setBoolean("mapred.compress.map.output", true);
conf.set("mapred.map.output.compression.codec",
         "org.apache.hadoop.io.compress.SnappyCodec
");
```

**HDFS block size**

Each map task works on a split of input data. Configuration parameters *mapred.min.split.size* and *mapred.max.split.size* in mapred-site.xml and *dfs.block.size* in hdfs-site.xml decide the size of the input split. The total number of map tasks created by the Hadoop framework is determined by both the input split size and the total input data size. For example, we have 1GB input files, the input split size is 64MB, total number of map tasks will be 1GB/64MB = 16. We can change input split size to control the number of map tasks. The easy way to change the input split size is changing the HDFS block size value using *dfs.block.size* parameter. The java code is as follow,

```java
Configuration conf = new Configuration();
conf.setInt("mapred.min.split.size", 512 * 1024 * 1024);
```

As more map tasks means more staring up and tearing down of map JVMs, it prefer to run small number of longer running map tasks.

**Map side spills**

The intermediate output of map tasks is stored in a buffer, which is a chunk of reserved memory in map JVM heap space. The default size of this buffer is 100 MB which is governed by *io.sort.mb* configuration parameter in mapred-site.xml. If the map tasks have large map output, increasing the *io.sort.mb* can decrease execution time. However, our tests indicated that unreasonable large buffer can lead to more failure map tasks.

**Shuffle/sort phase tuning**

Shuffle/sort phase copy and sort the mapper outputs based on the key. The maximum number of parallel map-output copier threads governed by *mapred.reduce.parallel.copies* in mapred-site.xml is set to 5 by default. If there are hundreds of mappers finishing at a same time period and each shuffle can only create 5 threads to download the map output, copy operation of shuffle can be inefficient. If the job have large amount of mappers, increasing *mapred.reduce.parallel.copies* can decrease the reduce phase waiting time. However, unreasonable large parallel copies would lead to JVM error.

# Chapter 6

# CONCLUSION AND FUTURE WORKS

In this thesis, we developed a set of Map/Reduce based Affymetrix GeneChip microarray data analysis tools. This set of tools is based on two widely used algorithms, Affymetrix Micro Array Suite (MAS 5.0) and Robust Multi-array Analysis (RMA). After studying the Affymetrix microarray quality control and preprocess algorithms, we first re-implement the algorithms in Java language, then we developed parallel versions of these algorithms using Map/Reduce framework.

We successfully deployed our tools on Hadoop and Palmetto Cluster high performance computing infrastructures. To achieve higher performance and scalability, we tuning the tools in three levels: the code level, the Hadoop level and the Java Virtual Machine (JVM) level. We tested our tools and compared with the existing tools using rice and Arabidopsis microarray dataset. The experimental results showed that our tools can efficiently utilize Palmetto Cluster resources to achieve high speed-up and can process massive dataset that existing microarray analysis tools cannot deal with.

In conclusion, our Map/Reduce based Affymetrix microarray analysis tools will provide biologists a new way to process and analyze increasing volume of Affymetrix microarray dataset with higher efficiency and lower costs.

In the future, we will add more Affymetrix microarray analysis algorithms to our Map/Reduce based microarray analysis tool kit.

# REFERENCES

Lockhart, David J., et al. "Expression monitoring by hybridization to high-density oligonucleotide arrays." Nature biotechnology 14.13 (1996): 1675-1680.

Lipshutz, Robert J., et al. "High density synthetic oligonucleotide arrays." Nature genetics 21 (1999): 20-24.

Affymetrix (2002) Affymetrix Microarray Suite User Guide, Version 5 edn. Affymetrix Santa Clara, CA

Affymetrix. GeneChip Expression Analysis: Data Analysis Fundamentals. Santa Clara, CA. 2002

Schena, Mark, et al. "Quantitative monitoring of gene expression patterns with a complementary DNA microarray." Science 270.5235 (1995): 467-470.

Alizadeh, Ash A., et al. "Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling." Nature 403.6769 (2000): 503-511.

Rocke, David M., and Blythe Durbin. "A model for measurement error for gene expression arrays." Journal of Computational Biology 8.6 (2001): 557-569.

Hubbell, Earl, Wei-Min Liu, and Rui Mei. "Robust estimators for expression analysis." Bioinformatics 18.12 (2002): 1585-1592.

Huber, Wolfgang, et al. "Variance stabilization applied to microarray data calibration and to the quantification of differential expression." Bioinformatics 18.suppl 1 (2002): S96-S104.

Gautier, Laurent, et al. "affy—analysis of Affymetrix GeneChip data at the probe level." Bioinformatics 20.3 (2004): 307-315.

Gentleman, Robert C., et al. "Bioconductor: open software development for computational biology and bioinformatics." Genome biology 5.10 (2004): R80.

Bolstad, B. M., et al. "Quality assessment of Affymetrix GeneChip data." Bioinformatics and computational biology solutions using R and bioconductor. Springer New York, 2005. 33-47.

Lukk, Margus, et al. "A global map of human gene expression." Nature biotechnology 28.4 (2010): 322-324.

Irizarry, Rafael A., et al. "Exploration, normalization, and summaries of high density oligonucleotide array probe level data." Biostatistics 4.2 (2003): 249-264.

Bolstad, Benjamin M., et al. "A comparison of normalization methods for high density oligonucleotide array data based on variance and bias." Bioinformatics 19.2 (2003): 185-193.

Irizarry, Rafael A., et al. "Summaries of Affymetrix GeneChip probe level data." Nucleic acids research 31.4 (2003): e15-e15.

Edgar, Ron, Michael Domrachev, and Alex E. Lash. "Gene Expression Omnibus: NCBI gene expression and hybridization array data repository." Nucleic acids research 30.1 (2002): 207-210.

Apache Software Foundation. Hadoop MapReduce. http://hadoop.apache.org/mapreduce. 2011.

A. S. Foundation., "Hadoop Distributed File System," http://hadoop.apache.org/hdfs/, 2011.

The Google File System. http://labs.google.com/papers/gfs-sosp2003.pdf

Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." (2007).

Chen, Yanpei, Archana Ganapathi, and Randy H. Katz. "To compress or not to compress-compute vs. IO tradeoffs for mapreduce energy efficiency." Proceedings of the first ACM SIGCOMM workshop on Green networking. ACM, 2010.

Miner, Donald, and Adam Shook. MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems. O'Reilly Media, Inc., 2012.

Gunarathne, Thilina, et al. "Cloud computing paradigms for pleasingly parallel biomedical applications." Concurrency and Computation: Practice and Experience 23.17 (2011): 2338-2354.

Olston, Christopher, et al. "Nova: continuous pig/hadoop workflows." Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011.

Langmead, Ben, et al. "Searching for SNPs with cloud computing." Genome Biol 10.11 (2009): R134.

Pfister, Gregory F. "An introduction to the InfiniBand architecture." High Performance Mass Storage and Parallel I/O 42 (2001): 617-632.

CCIT group. http://www.clemson.edu/ccit/rsch_computing/

Palmetto Cluster. http://citi.clemson.edu/palmetto/

J. S. Sarma. Hadoop - Facebook Engg. Note. 2011. http://www.facebook.com/note.php?note id=16121578919.

Yahoo Inc. "Hadoop at Yahoo!", http://developer.yahoo.com/hadoop. 2011.

Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/. 2011.

TORQUE Resource Manager, http://www.clusterresources.com/products/torque-

resource-manager.php. 2011.

Stonebraker, Michael. "The case for shared nothing." IEEE Database Eng. Bull. 9.1 (1986): 4-9.

Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

Ross, Robert B., and Rajeev Thakur. "PVFS: A parallel file system for Linux clusters." in Proceedings of the 4th Annual Linux Showcase and Conference. 2000.

Krishnan, Sriram, Mahidhar Tatineni, and Chaitanya Baru. "myHadoop-Hadoop-on-Demand on Traditional HPC Resources." San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego (2011).

T. White, Hadoop: The Definitive Guide, 3st ed., O'Reilly Media, 2012.

Donald, Miner and Adam Shook. MapReduce Design Patterns, 2012

Gentleman, Robert C., et al. "Bioconductor: open software development for computational biology and bioinformatics." Genome biology 5.10 (2004): R80.

Oberhumer, M. F. X. J. "LZO real-time data compression library." User manual for LZO version 0.28, URL: http://www. infosys. tuwien. ac. at/Staff/lux/marco/lzo. html (February 1997) (2005).