Parallelization of Specific R Functions For the SPRINT Framework

Dipesh Debnath

29th August 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011

Abstract

The Simple Parallel INTerface or SPRINT framework has been used in the field of Bioinformatics that allow researchers to use parallel R functions on high performance computing environments and conduct statistical computations on genetic data bases. In this dissertation, an effort has been made to construct a parallel function for SPRINT called psvm or Parallel Support Vector Machine. The original serial code was present in the R package e1071 and needed to be transferred to SPRINT package before being modified and integrated with R and SPRINT. The objectives of the dissertation was to learn how the serial svm function works, learn how the function can predict and classify test data and then identify areas where parallel code can be introduced. The test results were collected to compare with the pattern of test results to be obtained after running the parallel version of the code. In order to understand the functioning of the svm function, a basic knowledge of the genetic database was necessary. The model uses a set of kernel functions to classify data. The project is an effort to understand the characteristics of the different kernels and determine which part of the function can be parallelized within the project time frame. The serial function was tested with various sets of gene expression data using Golub_Test data set downloaded from the Bioconductor software package and check some of the characteristics of the model as studied from literature review. The performance bottlenecks were identified by measuring the time taken by various functions and I/O operations. Three processes, notably the svmtrain process, the svmpredict process and the cross validation process were investigated for parallel processing. Cross validation function took the least amount of time to execute and was easiest to parallelize whereas svmpredict function and svmtrain function were the most compute intensive operations. The dissertation involved a short study of the mathematical background and data preparation to provide proper explanation of the behaviour of the graphs that were generated. The dissertation ends by identifying the points where the parallel code can improve the performance of the function. The future work would involve the completion of the parallel code. The dissertation also provides the source code of the R test data preparation scripts, R interface functions and the R psvm function that could be used for further work.

Table of Contents

List of Abbreviations

SPRINT	: Simple Parallel R Interface
DNA	: Deoxyribonucleic Acid
EPCC	: Edinburgh Parallel Computing Centre
SVM	: Support Vector Machine
NESS	: Name of the Cluster available at EPCC
CSR	: Compressed Sparse Row
MPI	: Message Passing Interface
ALL	: Acute Lymphoblastic Leukemia
AML	: Acute Myeloid Leukemia
RBF	: Radial Basis Function

List of Figures

.

List of Tables

Acknowledgements

The author gratefully acknowledges the support and guidance of supervisor Michal Piotrowski(EPCC,University of Edinburgh) for providing the necessary support, knowledge and guidance to help make this dissertation a successful effort. The author is also grateful to the writers of e1071 R package whose code was used to modify and design the parallel version of svm function.

Chapter 1

Introduction

The field of Bioinformatics has seen many advancements in the recent years with the emergence of large databases [1] to store medical data and the availability of high precision gene chip making techniques [2] that has opened a new branch of science called DNA Microarray Analysis[3]. Large databases present new challenges to the data manipulation strategy and techniques to learn knowledge from large databases. In recent years, High Performance Computing(HPC) has emerged from a pure scientific technique to a more general way of thinking about computation and handling of large volumes of data. Genetic data is mostly stored in structured databases that can be arranged in matrix formats [4][5], suitable for use in HPC programs where matrix operations are ubiquitous. There are many software tools and services available that enable a biostatistician to conduct data intensive research. Some of the software packages already support parallel execution of tasks[6] that are used in biomedical research. Simple Parallel R INTerface or SPRINT has been developed by Edinburgh Parallel Computing Centre (EPCC) in collaboration with the Division of Pathway Medicine to allow R functions work in parallel. The dissertation is an effort to contribute to the existing work that is being done by the SPRINT team [7] by design and development of parallel functions. Statistics often treats the gene expression data as a multidimensional data where each genetic expression point is a feature [8]. Manual study of large genetic database is time consuming and error prone. Statisticians can therefore use mathematical modeling techniques and solve the problem computationally by analytical methods. Modern DNA microarray analysis techniques has introduced gene chip technology that can store gene expression data in a very small polymer slide[2] that can be scanned and interpreted by image analysis followed by subsequent analysis using statistical algorithms. Often, one gene expression data does not contribute to the overall state of the organism and several gene points need to be simultaneously studied. Genetic data need to be searched for hidden patterns and relations in between discrete gene expression points. Statistics and computational techniques can be combined in the form of standard R functions and High Performance Computing to provide standard and efficient ways to conduct these analytical experiments. The thesis that follows combines the knowledge and involves in the conversion of serial R functions to parallel functions by using MPI. In the present dissertation, **svm** function was chosen from the e1071 R package to be investigated for parallelization support and attempt was made to modify the code to support parallel processing. The svm optimization solver uses an implementation of the SMO algorithm published by Fan et al. [16].

1.1 Motivation

The motivation behind the project is to improve the SPRINT framework and contribute to the academic and scientific community by adding parallel functions to the SPRINT framework. SPRINT is an ongoing project with a list of top priority functions that need to be parallelized and svm function is one of them. The algorithm was developed mathematically by Dr. Vladimir Vapnik [9] and is considered as one of the best supervisory learning algorithm till date. Mastering the proper use of the function requires experience but at the same time, the function provides good prediction and regression efficiency when the parameters are manipulated correctly.

1.2 Objectives

The objective of the present dissertation is to be the starting phase of a research project that aims to learn and implement the svm function in parallel. The svm function has complex mathematical foundation that needs to be thoroughly understood to interpret the test results followed by learning of the kernel functions that are used by the program to generate the model. The points of parallelization need to be identified and the code needs to be implemented in full or in part depending on the time constraints. Data preparation process need to be learnt in order to use freely available databases like Golub_Test[10] and ArrayExpress [4]. In order to understand the correctness of the design and code change, the concept of probability calculation is required as the parallelization might impact the mathematical calculations within the serial code.

1.3 Report Structure

The structure of the report is as follows. The Chapter 2 provides an introduction to the technologies that are involved in SPRINT framework. The Chapter 3 provides the introduction to Support Vector Machine and how classification algorithm works. The Chapter 4 describes the analysis of serial functions and ho the impacted functions can be parallelized.Small code snippets have been explained The chapter 5 discusses about the tests performed on the serial code. Chapter 6 explains the areas of code modifications that are proposed for parallelization. Chapter 7 concludes and discusses the future work that needs to be performed.

Chapter 2

Background: Technologies involved with SPRINT

In this chapter, the key technologies that contribute to SPRINT framework will be discussed. DNA Microarray Analysis, SPRINT framework, R functions and High Performance Computing with MPI have been explained and their suitability to form a coherent platform through SPRINT framework has been described.

2.1 DNA Microarray Analysis

DNA microarray analysis is an emerging area in Bioinformatics used to conduct experiments on large array of gene chips [2] in order to analyze a large number of genetic information of an organism in one single experiment rather than conduct separate experiment on individual genes. DNA Microarray analysis is a post-clinical technique using a combination of statistics and computer imaging techniques to digitally analyze genetic expression and patterns of a large set of genes[11].
Statistical methods are used to understand the collective behaviour by studying expressions of the entire gene set of the organism. The number of genetic data points can be very large, ranging from thousands to millions of data points, statistical methods need to be applied to do data mining to glean knowledge out of the observations. Some typical operations will include classification, and clustering. DNA analysis is further made affordable by the use of freely available genetic databases like ArrayExpress[4] and software that forms a part of the Bioconductor[5] package.

2.2 Genetic data analysis using Mathematics

A Gene expression data set can contain more than 10000 features or gene expression points. Each feature can be mathematically represented in one dimension of data. A series of gene expressions can thus be represented in a 10000 dimensional space or more depending on the number of features in the gene sample. In case of the Golub_Test data sets that have been used in the current thesis, each sample contains 7129 gene expressions in a row of data. Using mathematics and statistics, the data can be viewed as a multidimensional vector and computation can be performed using supervisory machine learning techniques like Support Vector Machine (SVM) for classification and regression problems. A typical classification problem would predict the class of an unknown sample whereas a regression problem would deduce an unknown value given some other known values.

2.3 R Language

The R programming language is an extension of the S programming language[12] that is used by researchers and engineers. R is suitable for data intensive computing and is useful in conducting statistical analysis on large data sets. In the current project, the focus has been placed on the Bioconductor software package that is useful in the field of biomedical research. R supports a wide range of functions that are needed to conduct experiments using DNA microarray analysis. R being a functional programming language benefits from the use of standard functions in application development. Development time is greatly reduced by the use of standard library functions. In the present project, R can be installed along with MPI and a wrapper called 'SPRINT' that allows the scientists to exploit the high performance computing capability. The current project focuses on the support vector algorithm and its implementation in the svm function of R in an effort to create a parallel psvm function for SPRINT.

2.4 MPI support in SPRINT

The SPRINT framework uses MPI to exchange data between nodes in a cluster. The following dissertation has used NESS as the HPC platform to test and analyze the svm function. In the proposed parallel psvm function, MASTER and WORKER method has been used to parallelize the code. The data is read simultaneously by all the processors whereas the parallel computation is done by all the MASTER and WORKER processors combined followed by final collection of data by the MASTER process before the summarized data structures are returned to the calling R object. MASTER /WORKER model was found suitable for the current task as the computation of discreet samples can be performed simultaneously by several processors.

Chapter 3

Support Vector Machine

In the present chapter, the concept of a Supervised Learning Algorithm is described and the working principle of the support vector machine in binary classification has been explained. This is a high level view and detailed explanation is beyond the scope of the current study.

A Supervised Learning Algorithm is a statistical learning method used to predict the class of an unknown sample of data after studying a set of training data that is designed to generate a mathematical model. The quality of training received by the model is directed by the nature of the data present in the training data set.
The model is trained by studying the features of the training data in order to build up a hypothesis about that training set using the features or dimensions of the supplied data. Supervised learning has become successful in recent days because of its ability to solve problems that are computationally difficult to program. The trained model is a mathematical entity that has been programmed to learn by observing test cases [13]. The intelligence is generated by the creation of a hyperplane (a plane with higher number dimensions that is difficult to visualize graphically) that separates the data into two or more classes.

The challenge of creating a good quality model is to focus on the training data that builds up the mathematical model[18]. First step is to scale the features correctly so that all the features represent the classification in a consistent manner. Data that is not scaled properly can lead to highly skewed model that will be difficult to use for classification of test data. Secondly, features must be discovered that completely describe the subject that is being studied. The lesser the number of features used, more limited will be scope of the classification model and the quality of its classification of unknown cases will be affected. The data that is generated from a large variety of sources (e.g. all the hospitals of a country or worldwide) will have a lot of noise in the form of missing data, un-scaled data, noise from excessive coloration of sample etc. The number of samples from both classes should be nearly equal to balance the amount of data points on either side of the separating hyperplane[17]. Lastly, the experience of the user is very important in order to build a successful model.

Chapter 4

Analysis of serial functions for parallel processing

The current chapter discusses some important serial functions have been analyzed during the design phase of the psvm function development. These functions were studied for their scope of parallelization. The functions are part of the Rsvm.c and svm.cpp modules of the e1071 R package. The chapter has used some original code snippets from e1071 package to demonstrate the functioning of the serial functions and whether the code is suitable for parallel processing.
The following sections discuss each of the major functions involved in the analysis phase and each section contains two parts. The Serial functionality part discusses the serial function of the code. The next part is called Scope of parallelization and it is used to define the parallel processing strategy.

4.1 function

Serial functionality
The input data to the #, I (function present in svm.cpp is a problem set and a parameter set. The problem set consists of a data set containing the gene expression values and a classification vector containing binary classification value(+/-1) for each sample row of data.
The #, I (function will check for classification or regression operation that it needs to do by checking the svm_type parameter.
For regression operation, the #, I (function is simpler than classification. In case of regression, the svm needs to calculate the probability of occurrence of the problem set. The #, I # I -(- ' function does this by accepting the problem set and parameter set and calculating probability for the input data and returning it as output. The #, I # I -(- ' function contains two probability arrays. ProbA is the default array that is populated with probability information.
The next step generates the values of the decision function notably, and (' (. The decision function will calculate (' (value for each sample. According to theory, a pair of support vectors can be found for each sample and the evidence is given by a positive (' (value. The next step will calculate the number of (' (values that are greater than zero and assign that value to be equal to the number of support vectors for the sample data set. The next step creates an array of support vectors by checking which sample had positive (' (. The function then deletes all the allocated memory and returns to the calling function.
For classification operation, the first step is to separate the input data set into two groups. The array counts the number of classes per binary type(+/- 1).
Then, a #, 4 array is allocated and the number of rows equal to the number of samples in original data set. The array is reorganized into 2 groups with the two classes separated into to parts within the same matrix. Next, one class is

selected arbitrarily and the pairwise probability is calculated by calling svm_binary_svc_probability function. Then, the (' (and is calculated for first of the binary classes. The same is done for the other remaining class.
The value is populated into the output array of the model.
Next, the total number of support vectors are calculated based on how many samples had a positive the (' (value. The #, (function then deletes all the allocated memory and returns to the calling function

Scope of parallelization
The decision function selects the appropriate solver function by checking the type that was sent as parameter from R . In case of N/ /(or Support Vector Classification), the '# I I # function is called. '# I I # function will then call '# function to solve the optimization problem.
The '# function treats the entire problem set as a single matrix and performs optimization over the entire data set. This problem presents a hurdle to the parallelization in the '# function as the data set cannot be subdivided into multiple segments since the , (+ is unsuitable to be made smaller than the size of the number of sample data(number of rows of data set) . The Qmatrix needs to be further studied if it can be parallelized in the further work of the project. Thus the svm_train function in svm.cpp was found to be difficult to parallelize fully with the current knowledge of its functioning. Qmatrix operation inside the solver functions needs to be studied in further detail to find a parallel decomposition if any.
However, the following code snippet can be parallelized.

```
4444 0N9JL -6?'J331
4444444444444444 0(- 07(' (CF4?491
4444444444444444K4444, '6? )CF4N4 -6?+CFJ
4444444444444444444, '6? #I C9FCF4N47(' (CFJ
4444444444444444443 J>4444444444444444>
```

(Courtesy e1071 R package)
This code snippet can be changed to let all the processors copy the support vectors for the entire problem data set in small segments followed by MASTER process gathering all the vectors at the end of the parallel section.

```
88 4 ( (" '4
   ( 4 4( 4 4 + 4 4 ( 4 5 ( 4 , I, '
  0N- JL J331
4444444444444444 0(- 07(' (CF4?491
4444444444444444K4444 , I, '6? )CF4N4 -6?+CFJ
4444444444444444444 , I, '6? #I C9FCF4N47(' (CFJ
4444444444444444443 J444444444>
488 4 ( (" '4
   . ( 4 4("4 4 4 , '
      4 4( 4 4# 4 4 ! 4( ( 4- 4 # 4 ,
     , 4 * 4 7
      4 , I, '4 4, '
```

4.2 function

Serial functionality

The #, function is part of Rsvm.c program and it will calculate the probability of correct classification by taking the entire set of support vectors as input. It is used to check the efficiency of the model. The function begins by sparsifying the training support vector data set. The (function will add a delimiter at the end of the index component of each support vector data node. The input parameters to this function are obtained from the #, (function of the earlier step in the #, 7 program. The parameters are read into the temporary model that is constructed for the prediction purpose. The next step is to sparsify the sample data set. For each row of sample data, the probability is calculated based on the model. The program calls #, I or #, I I -(-' function depending on input parameters.
The next step is to calculate the decision values for the entire sample data set. This is performed by call to #, I I #(' function.
The function exits after deleting and freeing all the memory that was acquired.

Scope of parallelization

The function has two areas where parallelization can be supported.

Line Code

```
==================================================
        4D4N49J44L42+J4331
444444  4  CF4N4 #, I        I    -(-'  0<, 54 (  CF54  -434424 2  '(      1J
4444    >4 '  4K
44444   4  4D4N49J44L42+J4331
4444444    CF4N4 #, I        0<, 54 (  CFJ
444     4>
```

Listing 4.1 shows the code from serial version of #, 7 program where the calls to #, I and #, I I -(-' functions have the possibility of being parallelized. (Courtesy e1071 R package)

```
88    4 ( (" '4
        4  I  ,  4(  (  4 4 (   4     *    4 4    4 '(   4(- ' 4         4
   (   4     *   74   4 4   ('4 4   4  , -  4 4 ' ,      4N4     6   7
      '(   4   4  4(  ( 4  4   4  I   ,  4(  ( 4(
   ('  '(  4   4     4(   4    4 &4    +   4  4 (   4     *    7
         4   4    4  4 (   4          4              ' 7
```

Line Code

```
==================================================
     "  4$N9J
        4D4N4     J44L4    J4331
4444444    I  ,  C$F4N4 #, I       I    -(-'  0<, 54 (  CF54  -434424 2  '(      1J
```

```
4444444 4$33J
4444    >4 '  4K
4444    4  404N49J44_42+ J4331
4444444 4  I  ,  C$F4N4 #, I      0<, 54 (  CFJ
4444444 $33J
4444    4>
88   4 ( (" '4

     I  ,  4& "4 (  4  , 4   +4 4(  4      4 "4   +4#(' 4            7
  . (  4  4("4          4 4  ,  '  4         7
  %  4  "I  (    4 4  "  4("4  4 '(  4(- ' 4  , 4    *    4 4
       !   7
       4  4(- ' 4  , 4  4       4( (  4 4  4, (    4  4( (  4(   4 '(
     4 ( (4   (" 4 4  4 (,  4    4( 4  4        4 ( 4   -    4  4&  $7
```

Listing 4.2 shows code modification to support parallel processing. (Courtesy e1071 R package)

The code shown by Listing 4. 1 can be parallelized by following the changes to the code that is shown by Listing 4.2.
The next process can be described with respect to the following code snippet

```
Line   Code
================================================

       402       #('    1
4444     404N49J44_42+ J4331
444444   #, I      I #('    0<, 54 (  CF54   43442+2  '(     42402  '(     464, 1484=1J
```

Listing 4.3 shows code from serial version of Rsvm.c program the call to #, I I #(' function. (Courtesy e1071 R package)

```
Line   Code
================================================

88     4 ( (" '4
    (' '(  4    4(  4   4 & 4  4 (  4
       ( (  4  4    4  4 (  4       4(  4       4          '
       402       #('    1
4444   4  404N4     J44_42   J4331
444444   #, I      I #('    0<, 54 (  CF54   43442+2  '(     42402  '(     464, 1484=1J
88   4 ( (" '4
   . (  4  4("4          4 4  ,  '  4  4& $7
     , -   4#('   4 4  "I  (    4 4    4("4 ( (4 4  4   !   4
     , , (    7
```

Listing 4.4 shows code modification to support parallel processing.

(Courtesy e1071 R package)

The serial code shown by Listing 4. 3 can be parallelized by making changes as shown by Listing 4.4.

4.3 function

Serial functionality
The #, I function works differently for regression and classification problems.
In case of regression problems, #, I I #(' function is called and the probability checked. If the probability is > 0, the class +1 is inferred, otherwise class –1 is inferred.
In case of classification problems, #, I I #(' function is called and the decision values are returned. The Classification is performed using the model and training data set as arguments. The decision values determine how many samples are correctly predicted and how many were wrongly predicted. It will be used later to calculate the efficiency of the model.

Scope of parallelization
The function does not need to be parallelized as he calling process will be made to execute in parallel if required.

4.4 function

Serial functionality
The #, I I -(- ' function works differently for regression and classification problems.
In case of regression problems, it will predict the class label for the test data that is sent input to the function.
In case of classification problems, it will find the index of the sample with the highest probability for that particular class to be predicted for a particular input sample.

Scope of parallelization
The function does not need to be parallelized as the calling process will be made to execute in parallel if required.

4.5 function

Serial functionality
The function will calculate the decision values by using the appropriate kernel function. It will also collect information about the number of support vectors and their respective coefficients. There is always a pair of support vectors created per differentiating hyperplane, one for each class of samples. The coefficients are

also created in pairs. The decision value is the sum of all the coefficients of the pair of support vectors for that hyperplane.

Scope of parallelization
This function need not be parallelized and parallel code needs to be written where this function is called as the case may be.

4.6 function

Serial functionality
The do_cross_validation function is a part of the Rsvm.c program will validate the entire data set and check if the kernel can predict efficiently when the problem data set is divided into smaller segments. 5 sub problems are created and each sub problem is independently validated and then the total number of correct and incorrect results is collected.
The function starts with the random shuffle process on the entire problem data set.

```
Line    Code
==============================================

          0N9J4L     - 6?'J4331
444444  K
444444444444  44N4 (     01:  0     - 6?'61J
444444444444        4 #,  I          42 +J
444444444444     - '  4  J

444444444444+4N4     - 6? +OFJ
444444444444     - 6?+OF4N4     - 6? +OFJ
444444444444     - 6?+OF4N4 +J

444444444444  4N4     - 6?  OFJ
444444444444     - 6?  OF4N4     - 6?  OFJ
444444444444     - 6?  OF4N4  J
444444  >
```

Listing 4.5 showing the random shuffling process of sample and classification data sets found in the original code of #, 7 . (Courtesy: e1071 R package)

The listing 4.5 shows the code snippet from original #, 7 program found in e1071 package. The rand() function is used to generate a random index j for each of the serially incrementing i value for all the samples. The i^{th} and j^{th} samples are swapped.

Scope of parallelization

The part of the program shown in the Listing 4.5 is not suitable for parallelizing due to the fact that a uniform random shuffling is more effective on a bigger data set than smaller data sets in each processor. So, the sample and classification data sets need to remain in the original sizes during this process.

Line Code

```
=================================================
        0N9J4L  I  ' J4331
444444 K
444444444444  4     4N42   -6?'8  I   ' J
444444444444  4   4N403;12   -6?'8  I   ' J
444444444444  45$J
444444444444     4 #,  I     - ' ,  4   -     - J
444444444444  -     -74N4   -6?'60   6-      1J
444444444444  -     -7+4N4  ("   0      4 #,  I      25  -     -71J
444444444444  -     -7 4N4  ("   0    - ' 5  -     -71J
444444444444$N9J
444444444444  04N49J44L4      J4331
444444444444K
44444444444444444444  -     -7+C$F4N4   -6?+CFJ
44444444444444444444  -     -7 C$F4N4   -6? CFJ
4444444444444444444433$J
444444444444>
444444444444  04N4    J4L    -6?'J4331
444444444444K
44444444444444444444  -     -7+C$F4N4   -6?+CFJ
44444444444444444444  -     -7 C$F4N4   -6? CFJ
4444444444444444444433$J
444444444444>
```

Listing 4.6: Sub problem creation process in the original code of #, 7 . (Courtesy: e1071 R package)

The code segment given in Listing 4.6 can be parallelized using the rank numbers and number of processors as the variable.

The sub problems are created from the parent data set, by copying samples from rows before the **begin** index and after **end** index. Each processor can copy their individual sets of data for predicting the classes in the process that follows.

Line Code

```
=================================================
;       0 ( ( ,  6?  #,  I      4NN4     "       I   )   4MM
=4444444444444444 ( ( ,  6?  #,  I      4NN4  %I    )    1
@9B4              4
A44444444444444K
```

```
B              L     4         ?
D4444444444444444444  0N-    JL   J331
E4444444444444444444K
G44444444444444444444444444   - '  4#4N4 #,  I        0  - ,      '5   - 6?+OFJ
H44444444444444444444444444   - '  4 4N4   - 6?  OFJ
; 9444444444444444444444444444     43N40#6 120#6 1J
; ; 444444444444444444444444444  ,  #43N4#J
; =             L     4         ?
; @             L         4,      '4(    4     4 ( '   '(     4      ?
; A44444444444444444444444444444>
; B444444444444444>
; D4444444444  '
; E884  '(       (     4
; G444444444444444K
; H             L     4         ?
=9                 0N-      JL    J331
=; 44444444444444444444K
==444444444444444444444444444   - '  4#4N4 #,  I        0  - ,      '5   - 6?+OFJ
=@4444444444444444444444444444 0#4NN4   - 6?  OFI
=A444444444444444444444444444444444433          J
=B444444444444444444444>
=D              L     4         ?
=E              L         4,      '4(    4     4 ( '   '(     4      ?
444444444444444>
```

Listing 4.7: svm predict step for classification and regression.
(Courtesy: e1071 R package)

The Listing 4.7 will shows the lines in the function that can support parallel processing. Each processor has their own begin and end indexes and the processors can predict for (**begin – end)** number of samples of the data set.

Line Code

```
===============================================
88      4  4 (  ( "  '4
           0N-     JL    J331
44444444444444444444K44444   - '  4#4N4 #,  I        0  - ,      '5   - 6?+OFJ
444444444444444444444444444   - '  4 4N4   - 6?  OFJ
444444444444444444444444444     43N40#6 120#6 1J
444444444444444444444444444  ,  #43N4#J
444444444444444444444444444  ,   43N4 J
444444444444444444444444444  ,  ##43N4#2#J
444444444444444444444444444  ,    43N4 2 J
444444444444444444444444444  ,  #  43N4#2 J4444444>
88     4  4 (  ( "  '4
   .  (  4   4( "4           4  4  ,   '
```

```
, '( 4( "4 4 , #54 54 , 54 , ##54 , 4( 4 , # 4#( (-' 4
"I "
```

Listing 4.8: svm predict step modification for parallel processing
(Courtesy: e1071 R package)

The Regression section needs to be modified as shown in Listing 4.8.As shown in the listing, new MPI_Wait clauses can be added for all the processors to finish their processing before performing a global summation of the variables , #5 54 , 54 , ##54 , 4(4 , # .

Line Code

```
================================================
8B 4 4 ( (" '4
0N- JL J331
4444444444444444444K4 -' 4#4N4 #, I 0 -, '5 -6?+CFJ
444444444444444444444444 0#4NN4 -6? CFI
4444444444444444444444444444433 J44>
8B 4 4 ( (" '4
. ( 4 4("4 4 4 , '
, '( 4("4 4#(' 4 , 4 44- 4 "I "
```

Listing 4.9: svm_predict function call modification for parallel processing.

The prediction step needs to be modified as shown in Listing 4.9.Each processor gets its own section of sample data to predict.
The next steps in this function will perform cleanup operation and delete the data structures and free memory before returning exiting the function.

4.7 function

Serial functionality
The function is present in #, 7 module and begins by swapping samples randomly to create a shuffled data set.

Line Code

```
================================================

; 0N9JL -6?'J331
=4444444444K
@44444444444 44N43 ( 01: 0 -6?'61J
A44444444444 &( 0 , CF5 , CFJ
B4444444444>
```

Listing 4.10: Randomization step in svm.cpp.
(Courtesy: e1071 R package)

The Listing 4.10 shows the code snippet that is taken from #, 7 in e1071 package without any modification for explaining the randomization process.
The positions of i and j are swapped across the entire sample data set and an unbiased data set is created.

This region should remain a part of the MASTER process during parallelization because the next step involves splitting up the data set into smaller sub data sets and calculate probabilities independently in each sub data set.

Line Code

```
==================================================
;4        0N9JL  I  ' J331
=44444414444K
@44444444444  4      4N42   -6?'8  I   ' J
A44444444444  4   4N403;12   -6?'8  I   ' J
B44444444444  45$J
D44444444444     4 #, I     -' , 4  -     -J
E44444444444  -     -74N4   -6?'60    6-      1J
G44444444444  -     -7+4N4  ("   0      4 #, I      25  -     -71J
H44444444444  -     -7 4N4  ("   0    -' 5  -     -71J
```

Listing 4.11: 5 step sub problem step in #, I - (I # I -(-' .
(Courtesy: e1071 R package)

The Listing 4.11 shows the next segment of code that can be parallelized. The outer loop starts at line 1 and will loop I ' times, I ' being a fixed value in the program. It is observed that **begin** and **end** are two indexes of the samples that varies with **i**.

Scope of parallelization

This function has a scope for working in parallel in the code that follows. The hard coded I ' value can be replaced by the processor rank based begin and end point calculation code using the total number of samples prob->l and the number of processors **nprocs**.

Line Code

```
==================================================
//begin parallel section
@44444444444  4      4N4   -6?'42(   $8        J
A4            4    4N40   -6?'120(  $3;18        J
B44444444444  45$J
D44444444444     4 #, I     -' , 4  -     -J
E44444444444  -     -74N4   -6?'60    6-      1J
G44444444444  -     -7+4N4  ("   0      4 #, I      25  -     -71J
H44444444444  -     -7 4N4  ("   0    -' 5  -     -71J
```

```
//end parallel section
```

Listing 4.12: sub problem creation step modified for parallel processing.

The Listing 4.11 and Listing 4.12 shows that the size of each sub problem is of the same size for each rank.

```
Line    Code
================================================
            for(j=0;j<begin;j++)
            {
                  subprob.x[k] = prob->x[perm[j]];
                  subprob.y[k] = prob->y[perm[j]];
                  ++k;
            }
            for(j=end;j<prob->l;j++)
            {
                  subprob.x[k] = prob->x[perm[j]];
                  subprob.y[k] = prob->y[perm[j]];
                  ++k;
            }
```

Listing 4.13 original code segment showing sub problem creation.
(Courtesy: e1071 R package)

The Listing 4.13 shows the sub problem being created by copying data from the samples above the **begin** index and from the samples below the **end** index. This region can be a part of the parallel section as the **begin** and **end** points are already obtained by the individual processors.
The next step involves the call to #, I I #(' function for each of the **begin** to **end** samples starting from the sample numbered by index = **begin**. This part can also remain within the parallel section of each processor.
The final steps of the function involves freeing up of the memory for the data structures allocated and the function returns to the calling program.

Chapter 5

Serial benchmarking of svm code

In the present chapter, the serial version of svm code is tested to better understand the functioning of the svm. Two different kinds of tests were performed.

5.1 Test Case# 1:

Create different models with different kernel functions with 8 different sizes of gene expressions and analyze the relative efficiencies of RBF, Linear and Polynomial kernel functions to predict a test case accurately. The test case did not take Sigmoid kernel into some of the test cases because of the limited time available to conduct the testing.

Test case	Number of features	No.of support vectors(Linear)	No.of support vector(Polynomial)	No.of support vector(RBF)	Efficiency (Linear)	Efficiency (Polynomial)	Efficiency (RBF)	Random prediction Pass/Fail
2.1	3	25	30	28	64.70	67.64	76.47	Pass
2.2	10	24	33	30	88.23	79.41	88.23	Pass
2.3	20	24	31	33	91.18	76.47	85.29	Pass
2.4	50	24	33	33	100	70.59	91.17	Pass
2.5	100	28	34	34	100	76.47	94.11	pass
2.6	500	29	34	34	100	79.41	100	Pass
2.7	1000	28	34	34	100	85.29	97.05	Pass
2.8	7000	30	34	34	100	82.35	100	Pass

Table 5.1: The table shows the test results with Golub_Test data set using 3 kernel functions.

5.2 Test Case# 2:

Train the svm function was trained with varying number gene expressions. The table shows some characteristics of the all the kernels. The RBF kernel has high overall efficiency in predicting the entire training set. Linear Kernel shows high prediction efficiency when the number of features increase. RBF kernel shows higher efficiency when the number of features are low. In general, it is better to take a relatively low number of gene expression points for creating a model. The separating hyperplane is more generalized and can predict more accurately.

Benchmarking test was performed on the serial version of svm() function running on NESS. The objective was to understand the relative time taken by the different major functions used to generate the data structure that stores the model. Information available from benchmarking will highlight the need to parallelize the code.

The time is measured in microseconds.

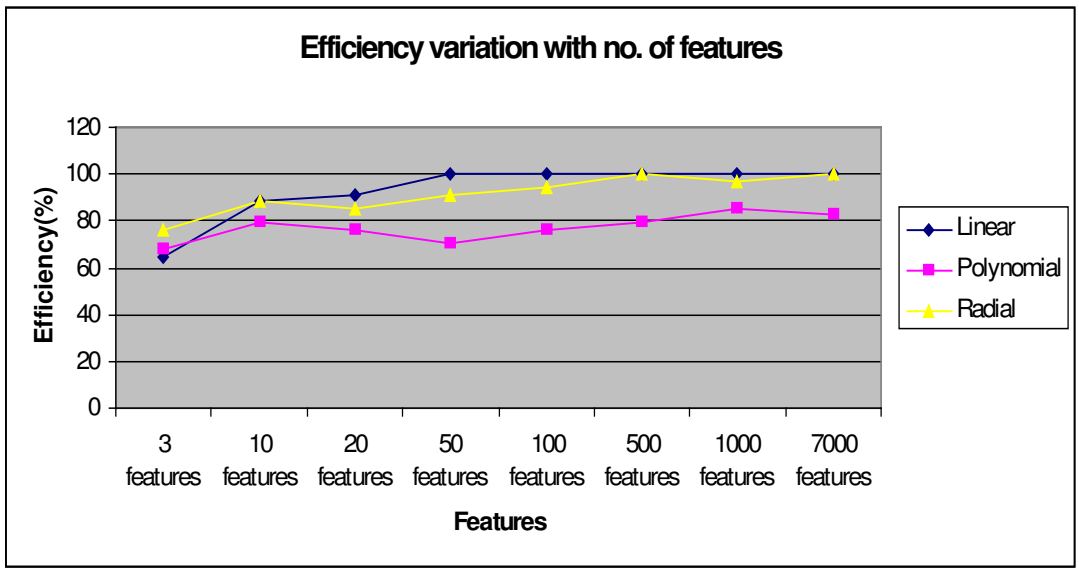

Figure 5.1: A plot of the variation of prediction efficiency of different kernels across different number of gene expression points.

The Fig 5.1 displays the variation of prediction efficiency of the different kernels using default settings. Polynomial kernel works least efficiently for this specific kind of data. The proper choice of the kernel depends upon the type of data that is involved. For the specific case of Golub_Test data, the characteristics of the graph shows that RBF is better when the number of gene expression points are less and linear kernel is better when the number of gene expression points are more. In the present test, default settings were used.

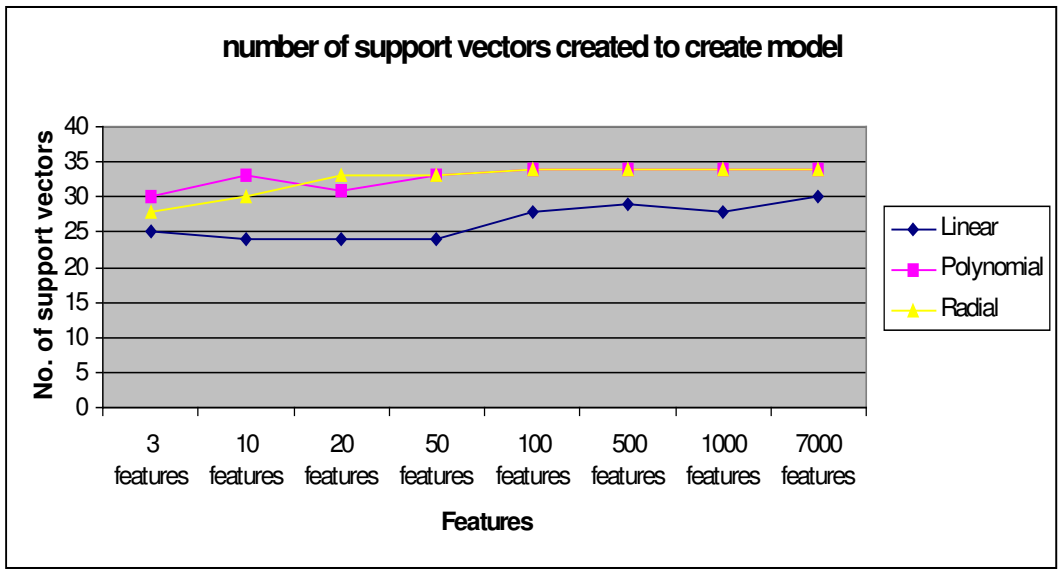

Figure 5.2: A plot of the variation of the no. of support vectors created by various kernel functions to generate a model and classify training data.

In the Figure 5.2, it is understood that RBF and Polynomial can separate the data more widely than linear kernel because the number of support vectors are higher than linear. In any classification problem, there will be a number of support vectors created in the feature space and the objective of the model is to optimize and find that pair of support vectors that can maximize the separation of data along two sides of the hyperplane.

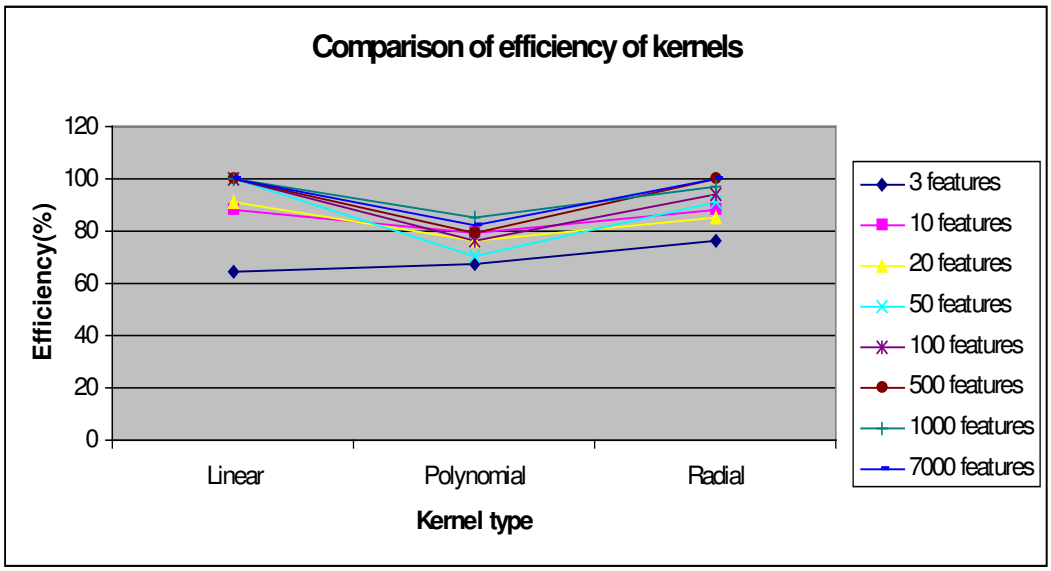

Figure 5.3: A plot showing the comparison of efficiency of different kernels using data sets containing different number of genes.

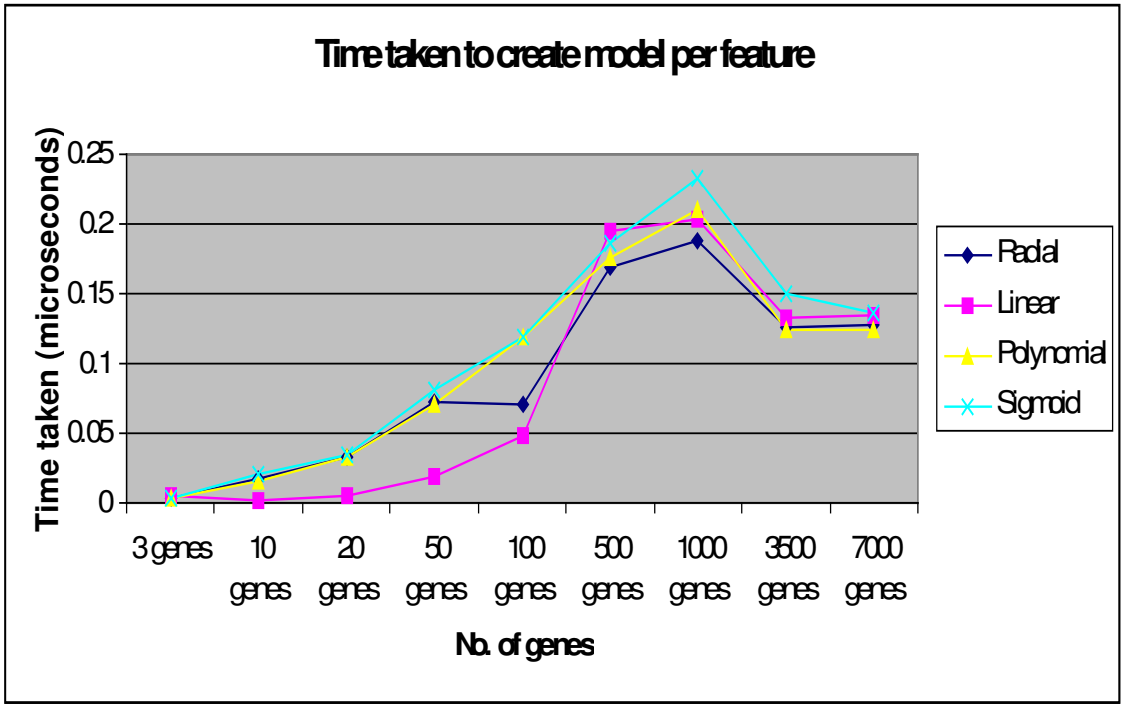

Figure 5.4: A plot of time taken by svmtrain function to create model per feature.

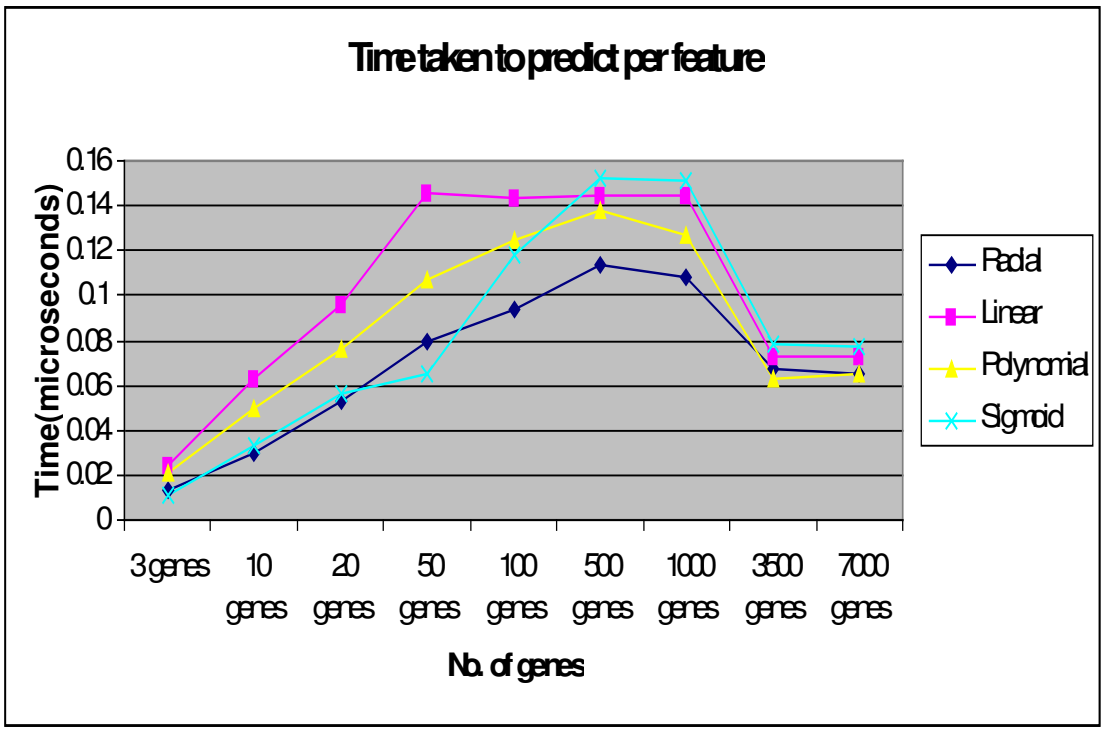

Figure 5.5: A plot of time taken by svmpredict function to predict per feature.

The Figure 5.4 displays the time taken while training a model and Figure 5.5 displays the variation of time in the prediction step per feature while the number of features are increased. The Figure 5.4 shows that the prediction time increases with the rise in dimensions and reaches a maximum somewhere between 1000 to 3500 genes. After this point, the time taken reduces with the rise in the number of features. The reason for this behaviour can be attributed to the curve fitting process. As the number of genes are increased, adding more features overfit the curve of the separating hyperplane.

The Figure 5.5 displays the time taken by the prediction process to conduct prediction per feature with varying number of genes. The general slope of the curves shows the difference between prediction timings of different kernels. Time taken by linear kernel rises and reaches maximum time at around 50 genes and continues till around 1000 genes after which, adding more genes reduces the prediction time. It indicates that the model has reached optimized state and any further increase in features is reducing the efficiency to predict.
The radial kernel shows the time taken to predict per feature is always the lowest among all the kernels.

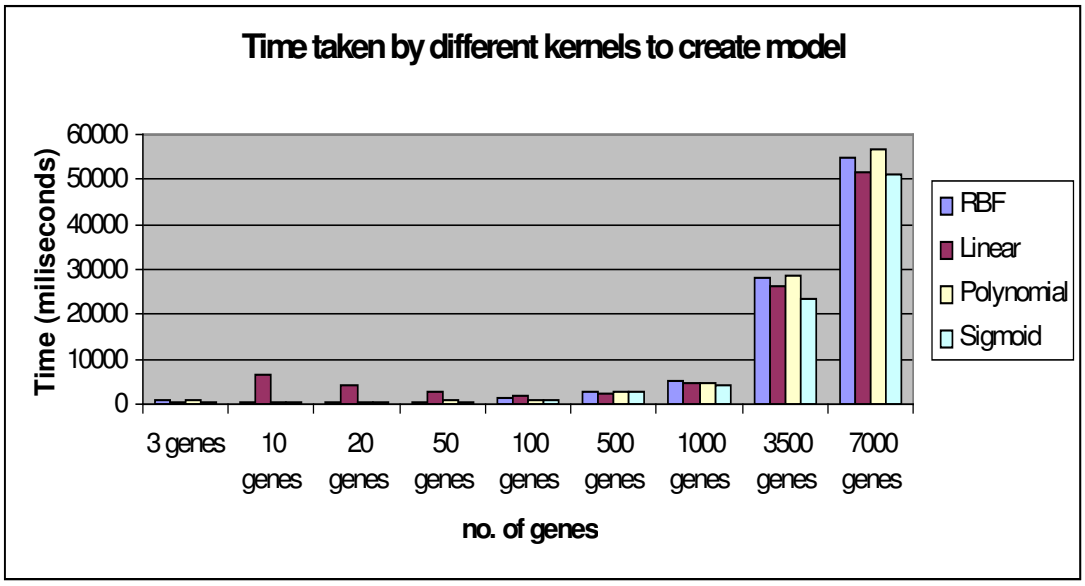

Figure 5.6: A plot showing the time taken by different kernels to create model by using same number of genes and samples.

The Figure 5.6 shows some interesting features about the kernels. The Linear kernel takes longer time when the number of genes are less whereas all the kernels take approximately the same amount of time to create the model as the

number of features increase. The total time taken by kernels seems to be linearly proportional to the number of gene expressions that are present in the data set. When the number of genes are low, linear kernel finds it more difficult to create optimized solution. For non-linear types of kernels, the problem does not occur as the non-linear function can project the data points relatively easily than the linear kernel when there are less number of features. When the number of genes increase, the non-linear kernels take more time and linear kernel seems to be better choice. The exception is sigmoid kernel which being non-linear takes least amount of time throughout the entire test.

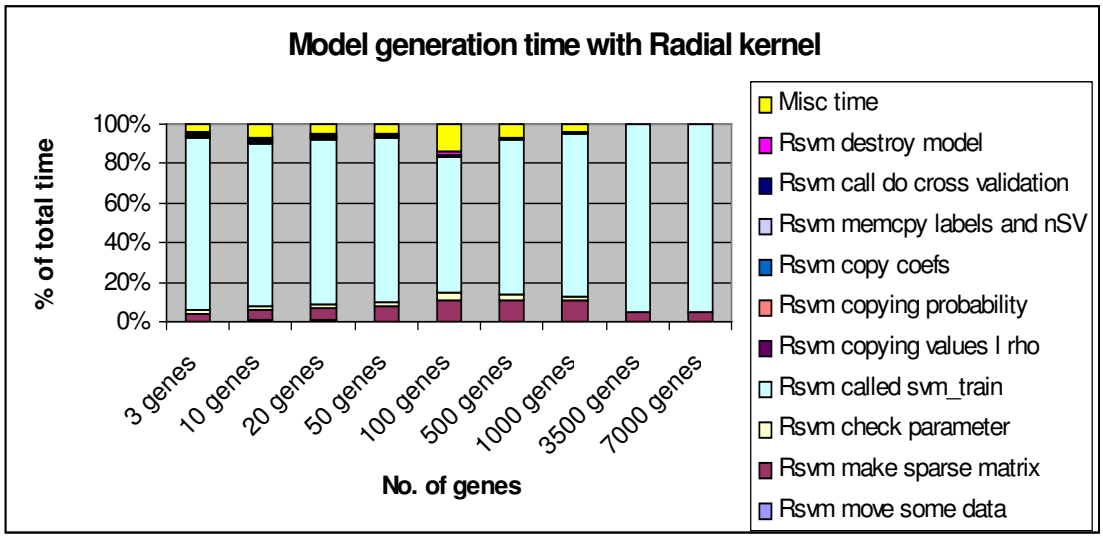

Figure 5.7: A plot showing the % of total time taken by using RBF kernel to generate svm model using 1 processor on NESS.

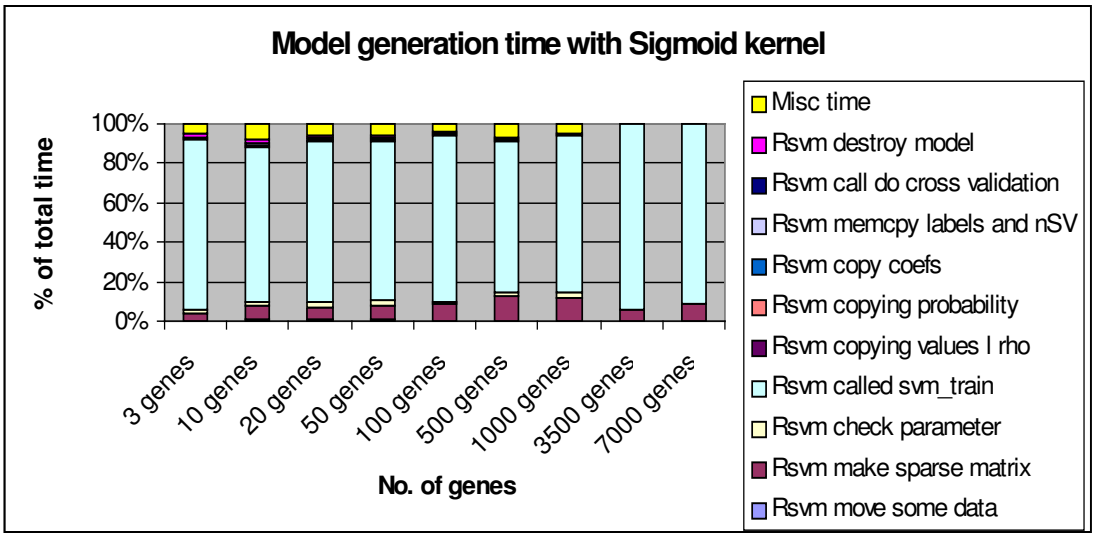

Figure 5.8: A plot showing the % of total time taken by using Sigmoid kernel to generate svm model using 1 processor on NESS.

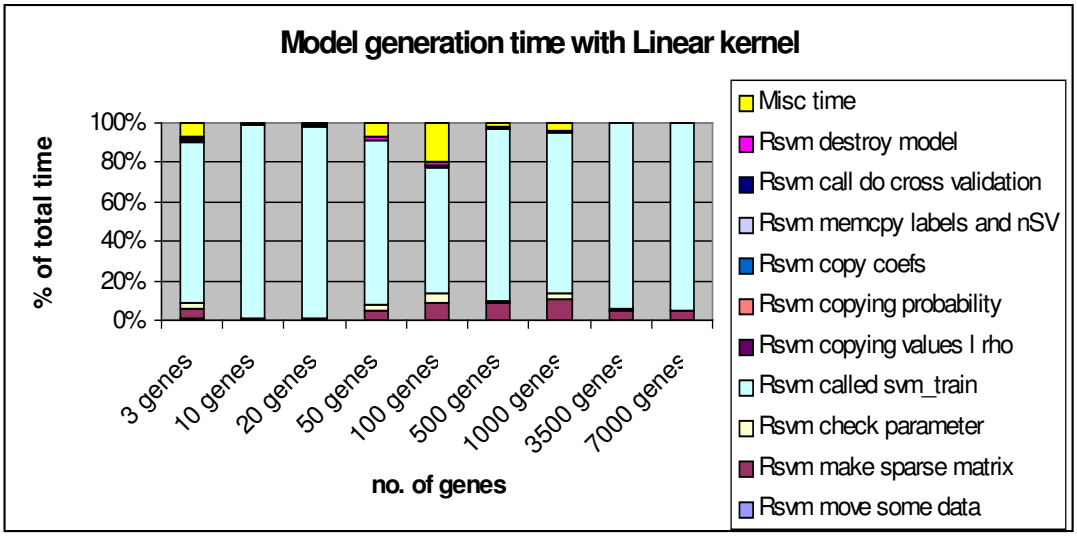

Figure 5.9: A plot showing the % of total time taken by using Linear kernel to generate svm model using single processor on NESS.

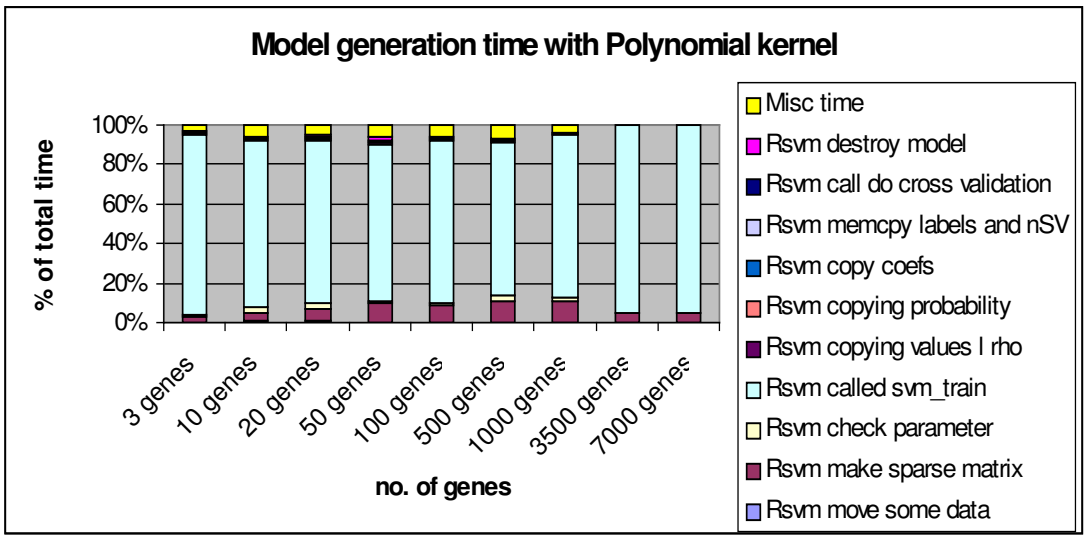

Figure 5.10: A plot showing the % of total time taken by using Polynomial kernel to generate svm model using single processor on NESS.

In the figures 5.7,5.8,5,9 and 5.10, the test was done to collect the execution time of the various operations done by Rsvm function to create a model. The svm_train function call takes majority of the run time. svm_train function has been analyzed in detail in the following chapters and points where parallelization can be performed has been identified.

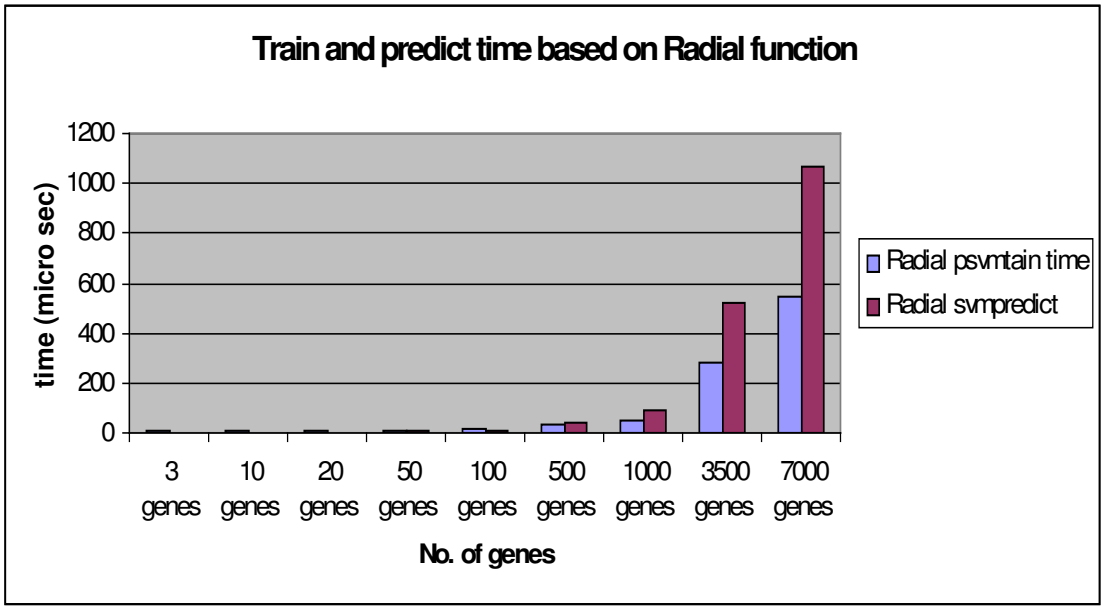

Figure 5.11: A plot comparing the svmtrain and svmpredict time taken during model creation process using (Radial Basis Function).

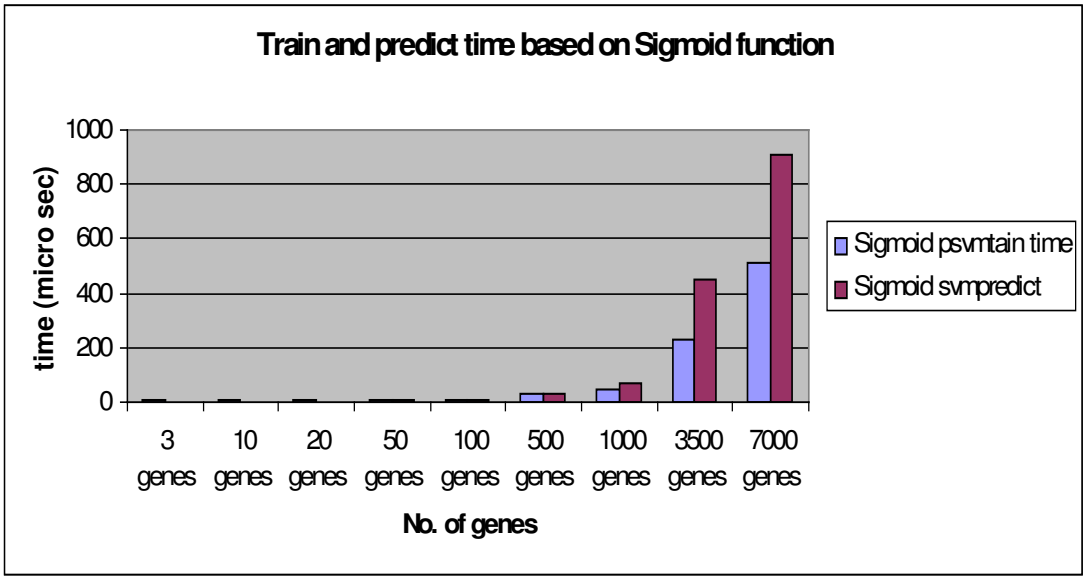

Figure 5.12: A plot s comparing the svmtrain and svmpredict time taken during model creation process using (Sigmoid Function).

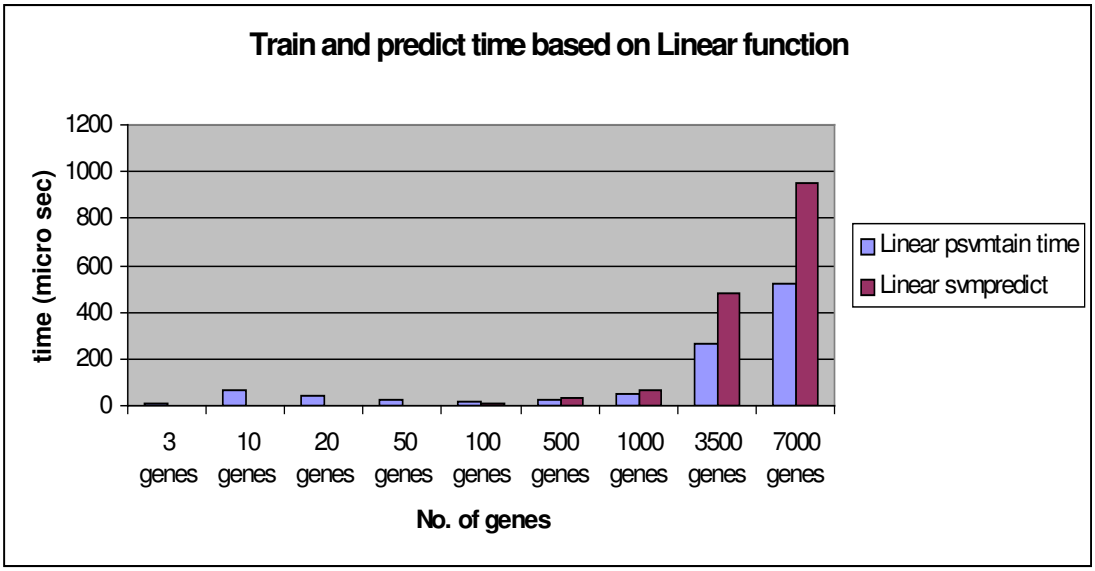

Figure 5.13: A plot comparing the svmtrain and svmpredict time taken during model creation process using (Linear Function).

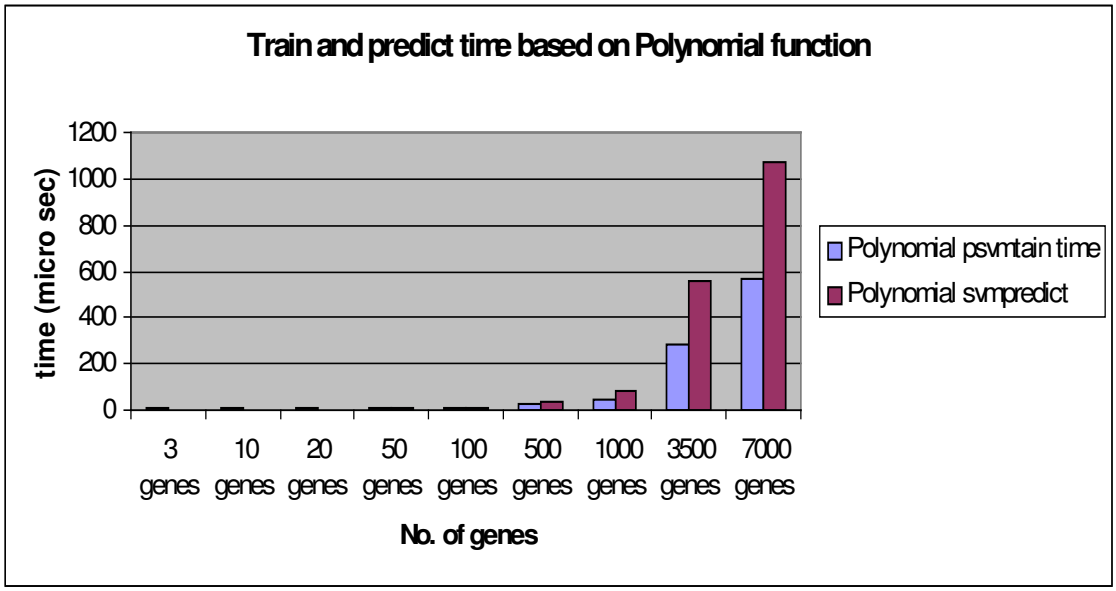

Figure 5.14: A plot s comparing the svmtrain and svmpredict time taken during model creation process using (Radial Basis Function).

The Figures 5.11, 5.12, 5.13 and 5.14 shows the comparison of the execution times taken by the svmtrain and svmpredict steps while creating the model using svm() serial function. The tests were performed on 9 sets of data sets with varying number of genes. It can be observed that when the training data set contains lesser number of genes, the training time is higher than the predict time

for all kernel types. The reason can be found by understanding the Sequential Minimal Optimization algorithm and svm model creation process mathematically. Each gene is a dimension in the feature space. For a 2 feature based input space, the separating hyperplane can be created by adding an extra dimension to the existing 2 dimensions and projecting the input space into a feature space with 3 dimensions. In 3 dimensions, a 2 dimensional plane can become the separating hyperplane. The number of dimensions in input space increase with the rise in number of features (gene expression points) and the space becomes more complex for the hyperplane to fit the plane as well as the support vectors. SMO algorithm will try to find the optimized solution by scanning through the entire feature space data. Each sample is a node of data and the SMO algorithm needs to perform matrix multiplication between node pairs. Analysis of the code shows that data is copied into square matrix and dot products are frequently performed. This results into increased processing time taken to create the optimized model.

Chapter 6

Code modifications

The present chapter will describe the parallel psvm function implementation. The code for psvm.R, psvm.c, Rpsvm.c and psvm.cpp have been taken from the original code of svm.R,svm.c,Rsvm.c and svm.cpp as found in the e1071 package. The present thesis is an enhancement of the original serial code and it is possible to see original code from e1071 for which the credit goes to the writers of e1071. Modifications need to be done following the design strategy discussed below to implement the parallel psvm function.

6.1 PSVM function design

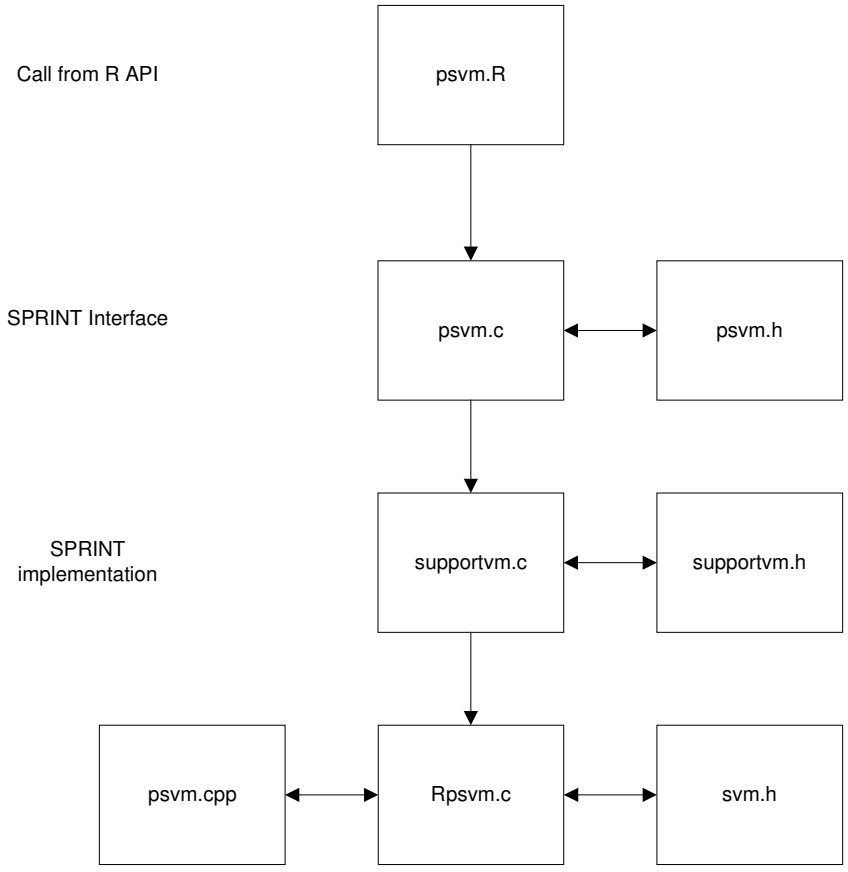

Figure 6.1: High level design of psvm function.

The high level design of the psvm function implementation is describe in Figure 6.1. The call to psvm function will start from within the R API. The list of arguments to be send to psvm function is same as the list of parameters that are sent to the serial function svm.

The command to call psvm classification/regression function using linear kernel will be as follows:
MODEL1 = PSVM(X, Y, type='C', kernel='linear')
where
MODEL1 is the name of the model returned after the svm training,
PSVM is the name of the function to be called,
X is the data matrix,
Y is the classification vector containing binary classes,
type is type of operation (classification/regression) to be performed,
kernel is the kernel function to choose from options.

The call the R object PSVM.R will start the execution of the process. PSVM.R accepts the data and input parameters. The object will validate the data that is sent and send appropriate error messages if invalid data is detected. Within the PSVM.R object, the PSVM.default function will receive and validate the data for null values and send appropriate error message. The input data matrix will be checked if it is a sparse matrix. The object will next convert the data into sparse matrix format. The reason for converting the data into sparse matrix is to compress the data as much as possible. For the genetic databases, sparse matrix may not be beneficial because the data is dense by nature. The sparsification of data will reduce the size of data being transferred and read by the processors. The sparsify function also adds the –1 index value at the end of each row signifying the end of the data of that sample.

The next step is to call the psvmtrain function present in SPRINT interface object called psvm. R object will add new additional parameters that will be required to build the model and ensure the correct passing of data to the SPRINT. The code for this object will remain mostly unchanged as it is mainly derived from e1071 package.

MPI needs to be used to support parallelization in the code of psvm.c.
Each process will execute the MPI commands to get the communicator and rank. The object will call supportvm function to send all the data to the actual C program that will execute in parallel.

supportvm program will receive all the data and MASTER process will distribute it among all the processors. Memory is allocated in all the WORKERs before MPI_Bcast is issued to send data to all processors.

After the psvmtrain function has performed the training, svmpredict function is called as the last step before finishing the psvm function.

6.2 Design of serial and parallel code

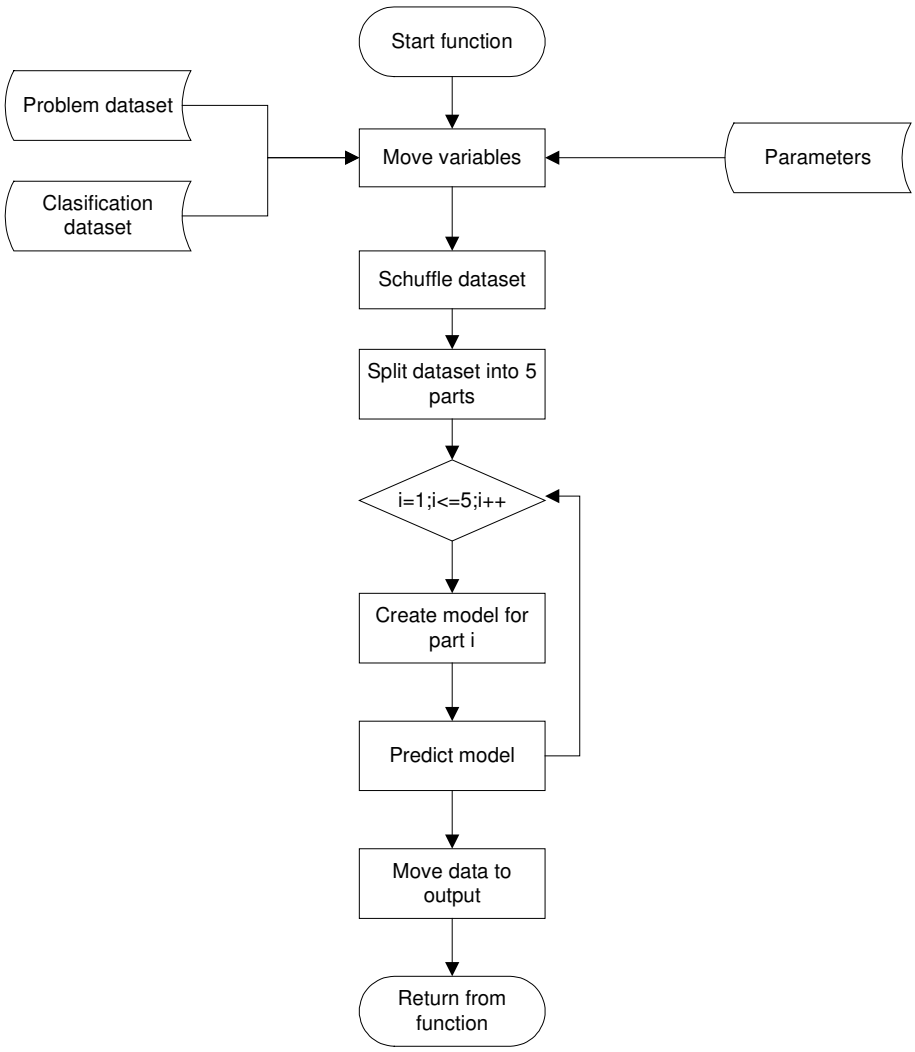

Figure 6.2: Serial program design of svmpredict function.

The Figure 6.2 shows the high level flow diagram of the svmpredict function. The decision box shows the region where the sub problems are predicted separately in 5 segments. This arbitrary segmentation of the data set is independent of the functioning of the algorithm. This point has been studied for parallelization.

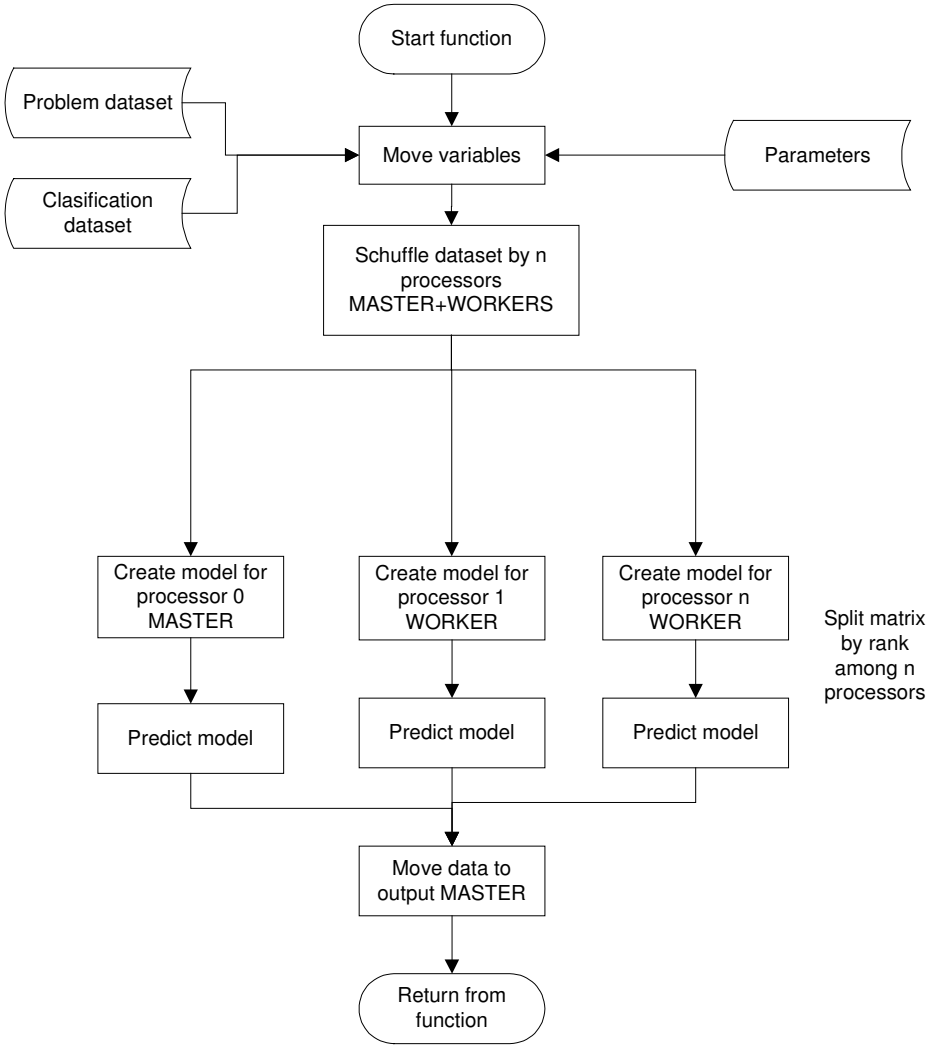

Figure 6.3: Parallel program design of svmpredict function.

The Figure 6.3 shows the process how the svmpredict function can be parallelized using the task farm method of workload distribution. The MASTER and WORKERS can all share the workload and followed by final collection of data by MASTER process.

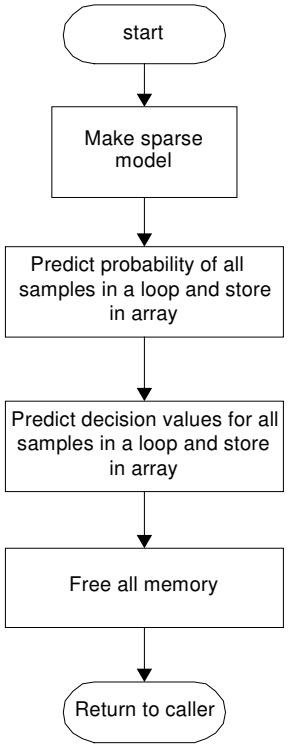

Figure 6.4: Serial program design of do_cross_validation function in Rsvm.c program.

The cross_validation process of Rsvm.c program was analyzed as shown in Figure 6.4 for areas of parallelization. The possibility of parallel processing were observed in the steps where probability was predicted and decision values were deduced.

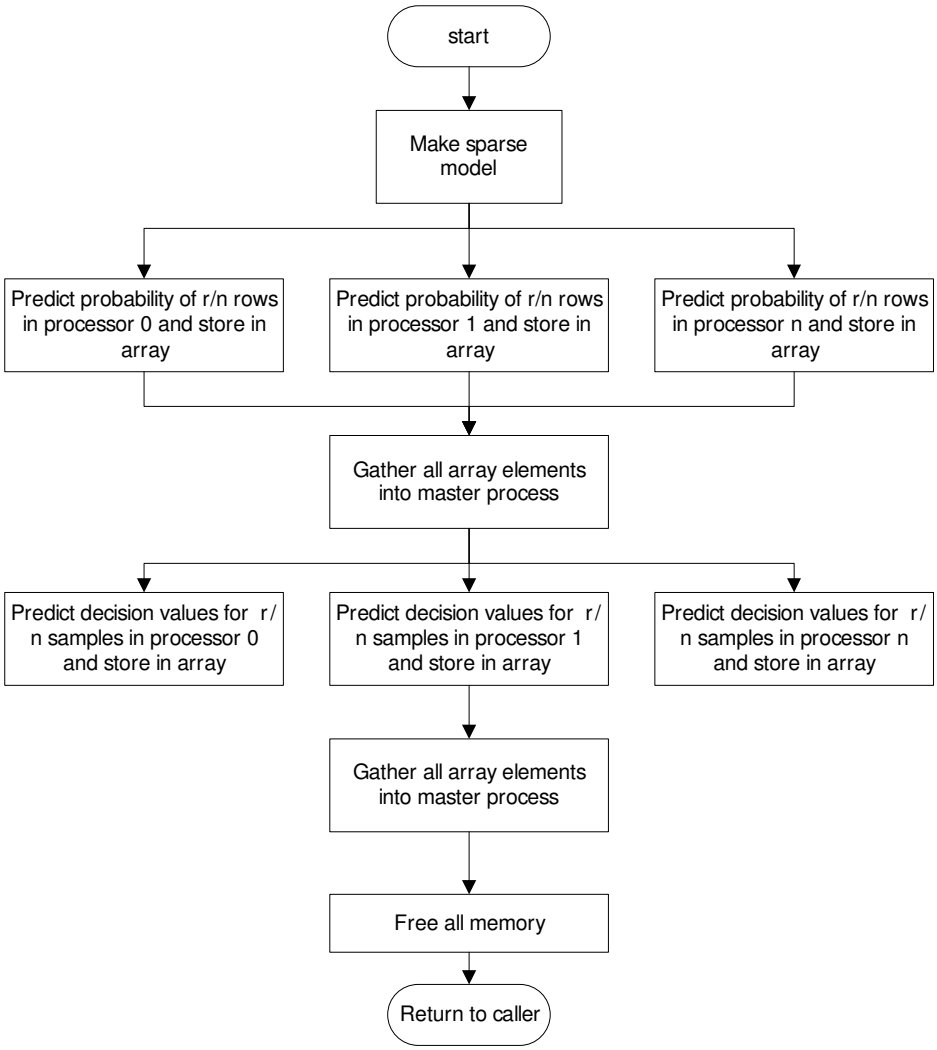

Figure 6.5: Parallel program design of the cross validation step.

In this Figure 6.5,the MASTER and WORKER model has been proposed to be applied so that the workload can be distributed across all the processors and later collected by the MASTER. Two separate sections of parallel code need to be created as global summation steps are required after the calculations are complete. The coding needs has to be completed in the further work on this project.

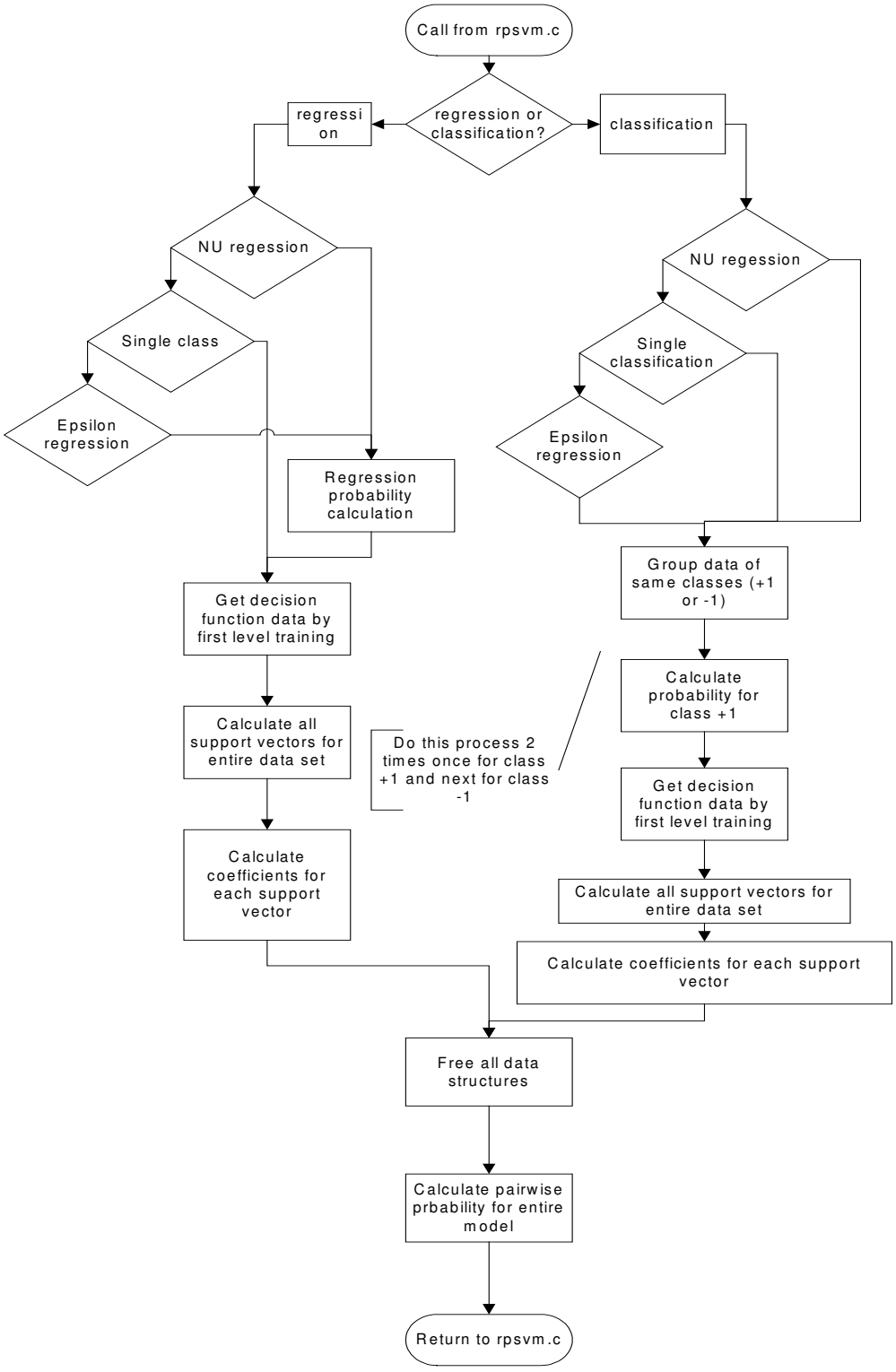

Figure 6.6: svm_train function serial processing design diagram.

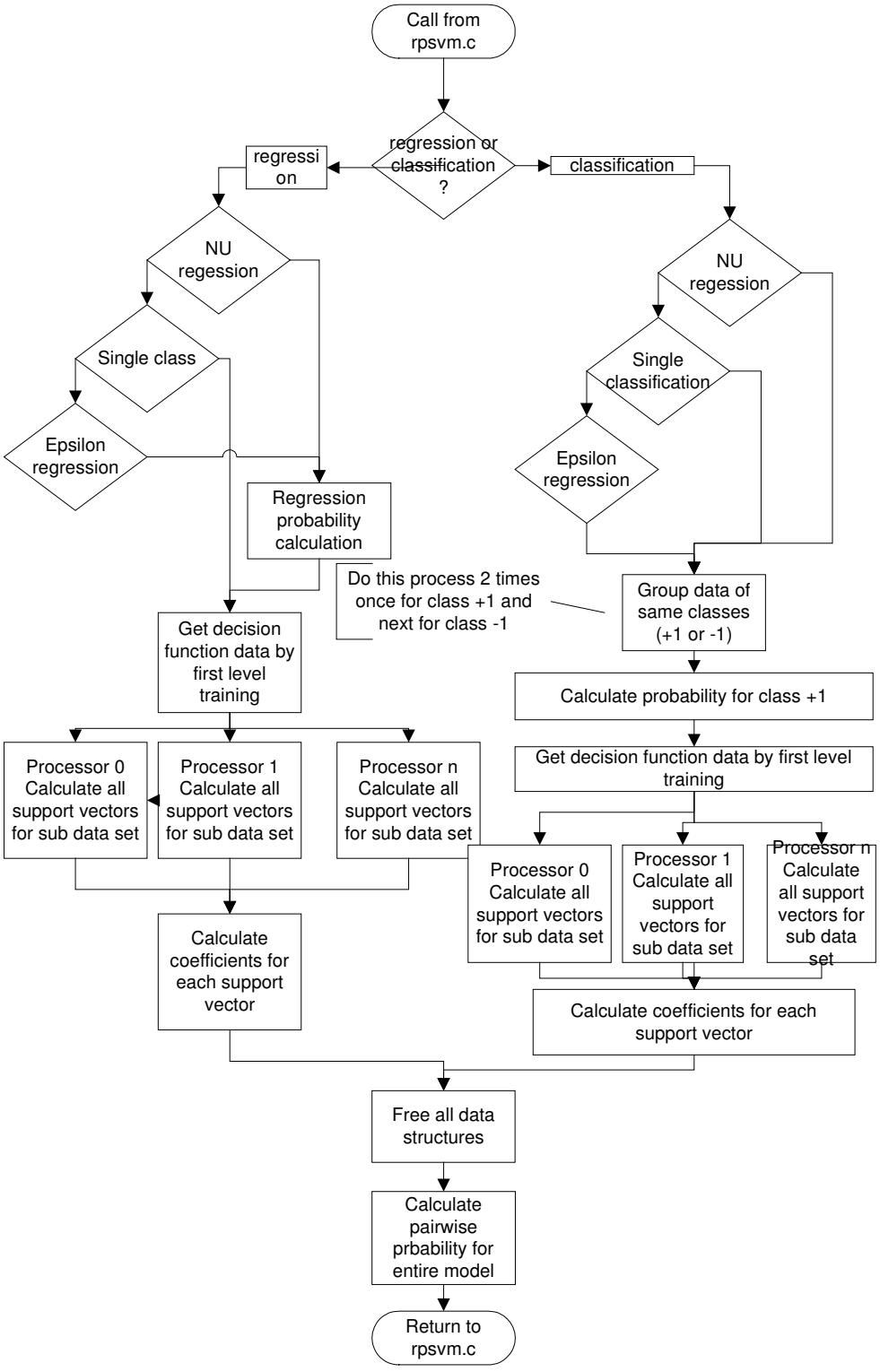

Figure 6.7: psvm_train parallel processing design diagram.

In the figure 6.7,psvm_train function design has been proposed. It can be noticed that the decision function data could not be parallelized as the entire sample data set is needed to calculate decision values.

6.3 Notes on the status of psvm code

The psvm function was written to the level of partial completion and has been supplied in the sprint2.tar.gz file and submitted separately. The code contains the original code from e1071 package and the credit to the written code goes to the original writers of the e1071 package. The writer of the thesis attempted to modify the code as the starting of the code modification phase. Most of the code will be a copy of the original code developed for the serial svm function. The parallel version of svm function needs to retain the original code of the serial svm function to maintain the integrity of the function that is already written in e1071 package. The changes can be seen in the renaming of the svm* functions to psvm* functions without any modification to the functionality in the actual function. The archived file has to be modified to add the parallel code sections using MPI. After completion of the code modification, the contents of the directory needs to be copied to a directory named ‘sprint’ before issuing the R compile command.

Chapter 7

Conclusions and future work

In this chapter the conclusions reached by the project have been discussed and the scope of the future work is explained.

7.1 Summary

The dissertation was approached from a research project point of view with the intention of studying the various contributing factors that will lead to a successful parallel implementation of svm function on SPRINT. The functions were analyzed in detail and the parallelizm was located.

The study on serial svm function has improved our understanding of the mathematical background based on which svm algorithm was written. The graphical patterns of efficiency and the number of support vectors created during model generation should remain the same even if the functions are parallelized. The serial test data can act as regression test data to see the correctness of the modified code. This knowledge of svm algorithm led to understand which functions could be split into multiple processors and which operations should be done by single processor.

The mathematical background of Sequential Minimal Optimization algorithm that was implemented needed to be studied in order to try to find out those areas within the algorithm that can be changed without modifying the actual algorithm and the essence of its design. Several hard code values were encountered in the code and we had to be careful to understand the meaning of each value from the mathematical perspective in order to make sure that changing it will not alter the integrity of the model. It was a challenge to learn the maths component of the literature study but it has greatly helped to improve the ability to understand mathematical representation of algorithm.

The thesis had been partially successful in identifying those areas that can be changed in the several functions as mentioned in earlier chapters and alternative parallel designs have been suggested. Effort has been done to code the parts of the complete code but was not completed by the time stipulated by the thesis.

The coding effort needs to be continued in the future work. Since the project mainly dealt with classification problems, the regression problems were not checked for parallelization support. More investigation needs to be done to check the regression functions for possible regions of parallel processing.

gprof count not be used with R due to difficulty faced while compiling with gprof option. omp.h header file was included to get the wall time by calling the function

omp_get_wtime(). This header file was successfully used during assignments done during the course semesters and was found to be suitable enough to time the steps that was important for analysis.

7.2 Future Work

The author was not able to finish the complete coding of parallel program within the time frame of the project due to the time spent to learn the complexity of the svm algorithm, R language, data preparation technique and analysis of the serial code. Some time should also be spent to learn how to manipulate the parameters and see their results. It can act as a regression testing procedure where old test results must match with new test data from the parallel code.

7.2.1 Knowledge about data preparation

The testing of the model is dependent on the successful tuning of the parameters that is often specific to the type of data that is being handled. The user of the scripts need to learn about data preparation method if external data like those stored in ArrayExpress database is used to build the model. We need to learn how to scale the data set before input to model preparation. Golub_Test data set appears to be a scaled training data set but the real world data sets will have noise and improper scaling. We have to learn how to normalize the data to reduce background noise from impacting the test results and how to clean the data sets before it can be used for training. Some study needs to be done ad using all the various combinations of the parameter. The input parameter list consists of more than 10 parameters and a limited knowledge was gained from the literature studies that were done within the course of the project. Further studies can be performed to understand them thoroughly and create the most optimized model in each case of training data.

7.2.2 Completion the parallel code

In the current thesis, the design of parallel code has been proposed by studying the serial functions. The actual code needs to be completed and run on SPRINT. There maybe more areas of parallelization in the program, especially in the region where a QMatrix is processed where the current thesis could not analyze in detail. An alternative decomposition strategy can be explored in order to use more parallelizm in the Solver section of the code.

7.2.3 Scope of parallel read

The thesis used Golub_Test data sets for testing that had very less number of sample data and gene expression points. The svm function can be used to learn from gene data sets with millions of expression points. Such databases will take enormous amount of time and memory to be loaded into R if the R has to read the data and send to SPRINT. The possibility of reading data directly from a file

should be investigated rather than sending the data through an R object as it is done currently.

7.2.4 Use of better profiling tools

The tests were profiled using wall time function available in omp.h header file. In the further work, further investigation can be done to see if it is possible to integrate gprof or a suitable benchmark program that can collect processing data at a granular level.

7.2.5 Profiling of parallel code on HECTOR and NESS

The tests were run on the serial code only since the parallel code could not be completed on time. The code needs be completed following the parallel design strategy and parallel benchmarking performed on HECTOR and NESS.

7.2.6 Profiling of code using databases from other fields of science

The psvm function was primarily designed to support Bioconductor and SPRINT. The svm algorithm is a general machine learning algorithm. The parallel code, once developed needs to be tested with other databases like image processing data, stock market data or metrics obtained from computer network activity.

7.2.7 Scalability test

The psvm function is designed to work with massive amounts of data and test need to be performed with very high volume data sets. If the parallel svm function can perform well when the data set size is scaled up, very large and intelligent models can be generated using this code.

Appendix A

Brief outline of the Golub_Test database

The database is made up of cases from 34 patients with lymphoblastic leukemia(or AML) or patients with acute myeloid leukemia(or AML)[14].The training data consists of genetic expressions recorded from 7129 genes in each row of data that is termed as 'features' in mathematical model jargon.
Serial number or samples start from 34,35 … and are 34 number in total
11 covariates are named as Samples, ALL.AML, BM.PB, T.B.cell, FAB, Date, Gender, pctBlasts, Treatment, PS, Source

Method to read data:
Reading Golub_Test data into R
> library(golubEsets) command is used to call the specific database from the Bioclite package
golubEsets [15] stands for arrays of genetic Expression data stored in a database.

The following command creates a data frame to copy data from Golub_Test data set :
> data(Golub_Test)
The user can also copy all or a portion of the entire data set by specifying the column indexes of the features that the user wants to copy. The advantage of using this technique is that the user can choose which genes to use to build the model if the user thinks that only a specific genetic sequence is responsible for the disease. Also, the computing facility may not have enough memory to process large data sets, when a smaller data set can be used to get meaningful results. A model, when created with less number of features tend to be more robust that those created by reading many features as in the later case, the model tends to overfit the hyperplane or decision boundary . Overfitted models cannot predict accurately and finds it difficult to cope with error in data. The following command can copy n features taken from the Golub_Test data base into a smaller data set called subgeneset:
> subgeneset<-Golub_Test[1:n,]

To view the data we can use the command
write.table(subgeneset)
The following table has been created to illustrate the nature of the genetic data that is stored in a Golub_Test data set. The values are correct values for that field although they may not have been taken from the same sample number.

Sample #	AFFX.BioB.5_at	AFFX.BioB.M_at	AFFX.BioB.3_at	Sample #	ALL.AML	BM.PB	T.B.Cell	FAB	Date	Gender	pctBlasts	Treatment	PS	Source
39	-342	-200	41	39	ALL	BM	B-cell	NA		F	NA	Success	0.78	DFCI
56	-34	-144	-17	56	ALL	BM	B-cell	NA		F	NA	NA	0.73	St-Jude
42	22	-153	17	42	AML	BM	NA	M4		M	86	NA	0.84	CALGB
47	-243	-218	-163	47	AML	PB	NA	M4	5/12/1997	F	NA	Failure	0.83	CCG

Table A: A Sample data from Golub_Test data set with 4 gene expression. profiles and 11 covariates. Classification of the data is obtained from ALL.AML covariate field. The table has attempted to fill up all the fields with some data to explain a general view a typical gene expression data set.

The creation of a data frame is the starting point of the SVM() classification test. The following explanation of the Golub_Test data set will help understand and solve the problem at hand.

The Golub_Test data set contains 11 covariates or variables that are additional information of the gene expression data but are not necessary a contributing factor to the expressions. The 11 covariates may contain a classification information. In the data set shown above, the leftmost column consists of the sample number. The sample data set contains 3 genetic expression data followed by 11 covariates.

subgeneset<-Golub_Test[1:3,] can create a data set similar to this.

When data is selected from Golub_Test, the covariates are also copied at the end of each row and contains a serial number between the last genetic expression information and the beginning of the covariate data list. The second occurrence of the **sample #** field in the middle of the data set marks the starting point of the list of covariates.

The ALL.AML covariate is of particular interest to the project as it is a binary classification of the leukemia

The data set also displays empty fields(e.g.Date),where data was not provided by the **Source**
The data set also shows the areas where data is unclean from a computational point of view. Occurrence of " " in the data need to be noticed and removed to create valid characters that can be processed.
A binary classifier will have values +1 or -1 denoting the positive and the negative classes of samples. The field AML.ALL contains text data in the form of 'AML' or 'ALL'. The user must write some code to convert this test data into numeric data that can be processed by the model.
The next stage of data preparation is to select the fields that will be used to build the model. The following example will choose the first 3 columns of genetic expressions and the field AML./ALL as the binary classifier.
If the data set contains quotes, it can be removed by using the quote clause as follows:
write.table(subgeneset,quote=FALSE,file="data1")
The data will be stored in working directory.

The data can be then read from working directory by the command:
filename <- c("data1") # assigning filename variable to store file name
dd <- read.table(filename,header=T,sep=" ") # reading data in matrix format.
The next step is to separate the genetic data from the classification into two separate matrices.

The following command creates x matrix or the Instance set with 3 features per Instance
x <- as.matrix(dd[,1:(ncol(dd)-11)])

The following command creates y or the classification set with binary classes AML/ALL for each corresponding sample in x.
y <- as.matrix(dd[5]) # the column index is n+2 where n is the number of features being tested.

The next step will convert text based classification into a binary classification.
The following command will create a binary vector of length 2 because it is expected that the data in y has already been cleaned and processed to contain only 2 classes. The following command will calculate the number of distinct text classes in y matrix.
y_classes <- names(table(y))

y_vector <- vector(length=length(y))

y_vector [which(y==y_classes[1])] <- 1 #replace textual class with integer classes
y_vector [which(y==y_classes[2])] <- -1

ALL is denoted by -1,AML is denoted by +1.The positive and negative values are purely to separate the two classes and the value 1 is the size of the gap between support vector and the separating hyperplane

Finally the vector can be copied into the y_classes array
y_classes <- y_vector
y_classes -> y

Appendix B

SVM Test Procedure

Test objective:Test that Radial Kernel Function better in classifying than Linear classifier when number of features are low.
Data: Data matrix with 3 features ,34 records

This introductory test case has been done to baseline the performance of linear and radial basis functions by using a very low number of features
Training data set x and classification vector y is ready for test after following the previous command sequence.
The following test vector is created by the command
test1 <- matrix(c(-342,-200,41),1,3) # test1 will only contain the features, not the serial number as in the training data set as it is only one test data that we are testing.
The test data is a repetition of the sample #39 and is expected to be classified as ALL by the model.
The efficiency of the model will be tested w.r.t. the data in y shown as follows in R:

```
> y
 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1  1
[26]  1  1  1  1  1  1  1  1  1
```

There are 20cases of ALL and 14 cases of AML in the data set

Test Case# 1

Linear Kernel based classification

In the first training procedure, svm() function is called using linear kernel function

```
> library(e1071)
> model1 <- svm(x,y,type='C',kernel="linear")
> print(model1)
Call:
svm.default(x = x, y = y, type = "C", kernel = "linear")
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  1
      gamma:  0.3333333
Number of Support Vectors:  25
> summary(model1)
Call:
svm.default(x = x, y = y, type = "C", kernel = "linear")
Parameters:
   SVM-Type:  C-classification
```

```
 SVM-Kernel:  linear
       cost:  1
      gamma:  0.3333333
Number of Support Vectors:  25
 ( 13 12 )
Number of Classes:  2
Levels:
 -1 1
```

Then the check is performed to see if the model can classify the training data set

```
predict(model1,x)
39 40 42 47 48 49 41 43 44 45 46 70 71 72 68 69 67 55 56 59 52 53 51 50 54 57
-1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
58 60 61 65 66 63 64 62
 1 -1  1 -1 -1 -1  1 -1
Levels: -1 1
> pred <- fitted(model1)
> table(pred,y)
    y
pred -1  1
  -1 19 11
   1  1  3
```

Result

The sample #39 can be verified from the matrix x as correctly classified as -1 which means the sample is a case of disease named ALL

Now, we can use the model to predict the test data taken from training data set(sample #1)

```
test1 <- matrix(x[1,1:3],1,3)
predict(model1,test1)
 1
-1
Levels: -1 1
```

The prediction is correctly done at -1 which means disease ALL

Error in Linear Kernel was detected at sample # 40 that was ALL but was wrongly classified as AML

2

RBF Kernel based classification

In the next step, the svm() function is called by the using radial basis function(RBF) as the kernel function and using command

```
> model1 <- svm(x,y,type='C',kernel="radial")
> predict(model1,x)
39 40 42 47 48 49 41 43 44 45 46 70 71 72 68 69 67 55 56 59 52 53 51 50 54 57
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1
58 60 61 65 66 63 64 62
 1 -1  1 -1 -1  1  1  1
```

```
Levels: -1 1
> pred <- fitted(model1)
> table(pred,y)
    y
pred -1  1
  -1 20  8
   1  0  6
> predict(model1,test1)
 1
-1
Levels: -1 1
```

The prediction of test1 as disease type ALL is also correct in this case.
No errors in classification was detected for the entire training set

Appendix C

Sample R script to make data from Golub_Test database

The following script is suitable to be used with Golub_Test data set only.
The user needs to change the value n in order to select n genetic expressions or features from the Golub_Test data set.
All selection is done from columns on columns 1 through n (n_{max} =7129)
The script will write the selected data into a file before reading the data again into a matrix.Binary level classification is allowed in this particular R script although svm supports higher levels classification and is out of scope of the current project. The script file should have a .R file extension.

```
library(golubEsets)
data(Golub_Test)
subgeneset<-Golub_Test[1:n,]
write.table(subgeneset,quote=FALSE,file="data1")
filename <- c("data1")
dd <- read.table(filename,header=T,sep=" ")
x <- as.matrix(dd[,1:(ncol(dd)-11)])
y <- as.matrix(dd[n+2])
y_classes <- names(table(y))
y_vector <- vector( length=length(y))
y_vector [which(y==y_classes[1])] <- 1
y_vector [which(y==y_classes[2])] <- -1
y_classes <- y_vector
y_classes -> y
library(e1071)
```

Appendix D

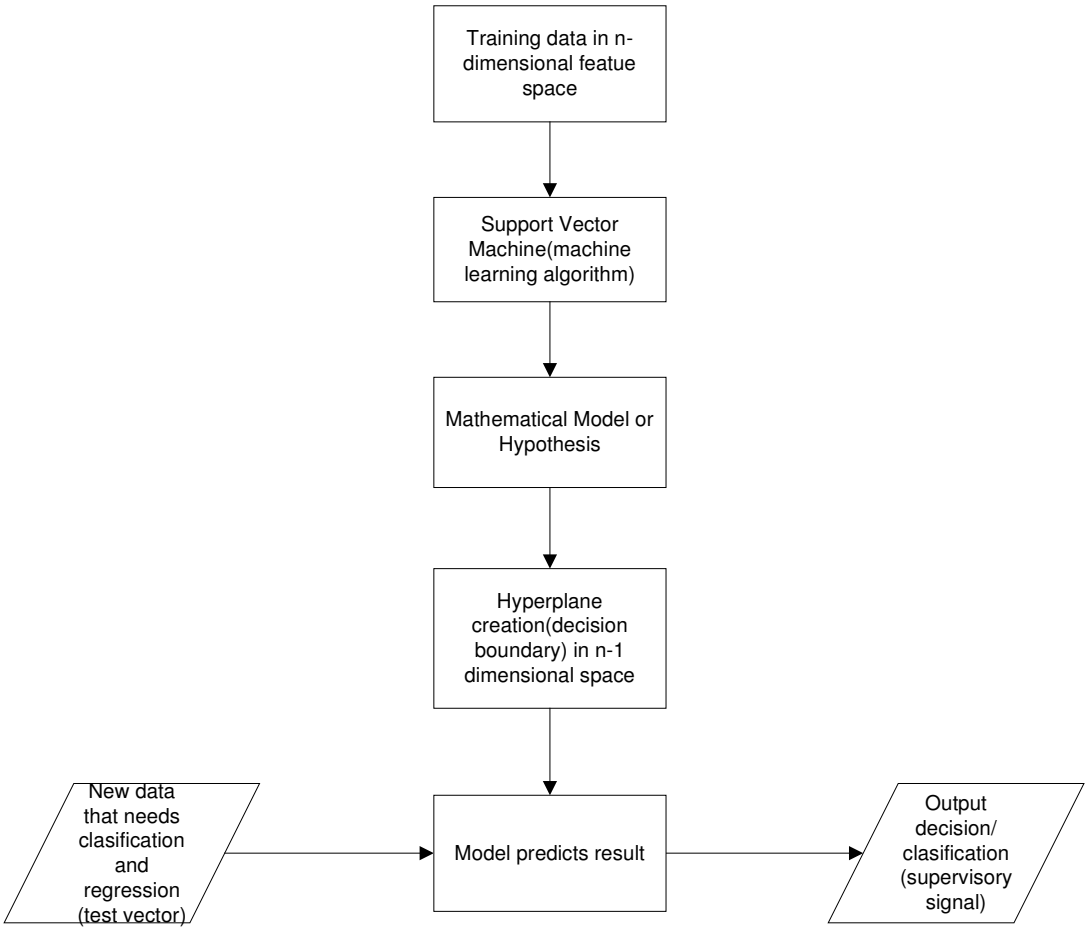

Figure A: Schematic of a typical supervisory machine learning algorithm.

Bibliography

[1] Michael Barton's Bioinformatics and Research Notes,Bioinformatics Zen, http://www.bioinformaticszen.com/software/dealing-with-big-data-in-bioinformatics

[2] DNA Microchip Technology,http://www.genome.gov/10000205

[3] The Basics of DNA Microarrays By Laura Bonetta :www.hhmi.org/biointeical Interactive/genomics/microarray.html

[4] ArrayExpress database basics, www.ebi.ac.uk/arrayexpress/

[5] Gene Expression Atlas, http://www.ebi.ac.uk/gxa/

[6] High-Performance and Parallel Computing with R, http://cran.r-project.org/web/views/HighPerformanceComputing.html

[7] SPRINT overview,http://www.epcc.ed.ac.uk/news/sprint-parallel-statistics-using-r

[8] Chao Cheng,Koon-Kiu Yan,Kevin Y Yip,Joel Rozowsky,Roger Alexander,Chong Shou,Mark Gerstein, A statistical framework for modeling gene expression using chromatin features and application to modENCODE data set, http://genomebiology.com/content/pdf/gb-2011-12-2-r15.pdf

[9] C. Cortes and V. Vapnik. Support-vector network. Machine Learning

[10] Dr.Todd Golub - Expression Sets: http://www.bioconductor.org/packages/2.6/data/experiment/html/golubEsets.html

[11] DNA microarray en.wikipedia.org/wiki/DNA_microarray

[12] R language, http://en.wikipedia.org/wiki/R_(programming_language)

[13] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification

[14] Dr.Todd Golub - Golub database: http://www.bioconductor.org/packages/2.8/data/experiment/manuals/golubEsets/man/golubEsets.pdf

[15] Xiaochun Li:High-dimensional data analysis in cancer research

[16] Rong-En,Kai-Wei Chang,Cho-Jui Hsieh,Xiang-Rui Wang, Chih-Jen Lin,National Taiwan University,Journal of Machine Learning Research 9(2008) 1871-1874,website: http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf

[17] Oracle® Data Mining Concepts 11g Release 1 (11.1), website: http://download.oracle.com/docs/cd/B28359_01/datamine.111/b28129/algo_svm.htm

[18] Jigang Wang, Predrag Neskovic, and Leon N Cooper,Training data selection for Support Vector Machines, Institute for Brain and Neural Systems, Physics Department,Brown University, Providence

Statistical Analysis of DNA Microarray Data in Cancer Research website:orfe.princeton.edu/~jqfan/papers/06/microarray-review.pdf

Bioconductor,website:www.bioconductor.org/

The Basics of DNA Microarrays By Laura Bonetta, website:www.hhmi.org/biointeical Interactive/genomics/microarray.html

Microarray Overview - Genome Resource Facility, website:grf.lshtm.ac.uk/microarrayoverview.htm

DNA Microarray, website:www.scientific-web.com/en/Biology/.../DNAMicroarray.html

Statististical Issues in cDNA Microarray Data Analysis, website:stat-www.berkeley.edu/users/terry/zarray/TechReport/mareview.pdf

John Verzani,Simple R, website:http://www.math.csi.cuny.edu/Statistics/R/simpleR/index.html

R basics,website:www.biostat.wisc.edu/~kbroman/Rintro/

SVM classification, website:http://astor.som.jhmi.edu/~cope/687/pdf/classification.pdf

Alfred Ultsch, David Kämpf, Knowledge Discovery in DNA Microarray Data of Cancer Patients with Emergent Self Organizing Maps,Uni. Of Marburg,Germany

Dr.Todd Golub - eSets, website:ddr.nal.usda.gov/bitstream/10113/20406/1/IND43653225.pdf

e1071 R package,website: cran.r-project.org/package=e1071

C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines, 2001.website: http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Applied Spatial Data Analysis with R by Roger S. Bivand, Edzer J. Pebesma and Virgilio Gómez-Rubio

Software for data analysis Programming with R by John M. Chambers