Parallel Kernel Computation for High Dimensional Data and Its Application to fMRI Image Classification

Shibin Qiu Terran Lane

{terran, sqiu}@cs.unm.edu
December, 2003

Abstract

Kernel method is frequently used for support vector machine classification and regression prediction. Kernel computation for high dimensional data demands heavy computing power. To shorten the computing time, we study the performance issues of parallel kernel computation. We design the parallel algorithms and apply them to the classification of fMRI images, which have extreme high dimensionality. We first formulate the kernel calculation through linear algebraic manipulation so that the operations can be performed in parallel with minimum communication cost. We then implement the algorithms on a cluster of computing workstations using MPI. Finally, we experiment with the fMRI data to study the speedups and communication behaviors.

1 Introduction

Kernel based methods for regression and classification are powerful statistical tools for machine learning. By transforming data from an input space into a feature space, data that is not linearly separable can be effectively separated and classification accuracy is remarkably improved. Support vector clustering uses kernels to cluster data in high dimensional space for unsupervised learning [1]. Kernel based regression can predict data with better precision than linear methods [7,21]. Kernel based support vector machines (SVM) have been successfully used for many applications, such as speech recognition, face detection, disease diagnoses, genetics, drug design, and text categorization [1,3,8,15,22,23,24,26].

Due to their importance in theory and application, we study kernel based SVMs in this project. Specifically, we study the performance of kernel computation for high dimensional data and apply the algorithms to fMRI image classification. We will formulate the SVM computation linear algebraically so that the algorithms are appropriate for parallel processing. We will implement the algorithms in a cluster and test speedup, accuracy, and communication efficiency with public domain data and the fMRI data.

1.1 Background and related works

While kernel methods are widely applicable, kernel computation is time and space consuming. This is especially true for high dimensional data. The KDD cup thrombin data set has 139,351 features and 1909 data points [20]. The CDK2 drug dataset has 14,223 compounds, each of which has 35,926,557 descriptors [24]. The fMRI image, if represented by its raw voxel data, has 64K features in each image. In the kernel methods, we need to store the training set, we often need to store the kernel matrix, and we need memory space for working data structures and copies of data. Though the kernel matrix itself is not large if the number of data points is not large, the data matrix can take a great deal of space. For such applications, the space requirement for kernel computation can easily exceed the capability of computer's main memory. That is why related works generally use linear SVMs and avoid kernels for large datasets.

The training time for large datasets can also be prohibitive. The solution to a SVM involves quadratic programming, which is generally slow. To make the quadratic search faster, SVM^{light} was invented [9], which selects a smaller working set than the training set at each iteration and reduces the amount of computation. Similar strategies are used to achieve faster performance in the quadratic programming to solve the SVMs [6,10,14,16], and different variants of SVMs were proposed, such as SMO [16], SSVM [12], etc. To reduce space and time consumption, an approximate SVM, the RSVM was introduced [11], which randomly selects a subset from the training set and uses this reduced subset to compute the kernel and make the kernel smaller.

The proximal SVM (PSVM) was invented as an alternative to speedup the computation [4]. PSVM avoids the quadratic programming and solves the SVM with linear computations only. Due to its simplicity and fast performance, PSVM is becoming increasing popular [5,17]. PSVM can save computing time, but its space requirement is high, since it must save the kernel matrix.

Incremental training methods were proposed for PSVM to save space and to unlearn old data [3,5]. But they only considered linear classifiers without kernels. Decremental learning for the linear PSVM was proposed to reduce the contribution of old data [18]. Multi-categorical classification using incremental PSVM was studied [19], but the kernel was not considered. A parallel incremental learning algorithm was proposed for the linear PSVM [17]. An incremental learning algorithm was suggested for high dimensional data, but for linear SVM only [3].

In summary, related works often avoid using kernels for large and high dimensional data. Parallel computation is not widely applied with SVM. The only parallel SVM computation considers linear SVM without kernels [17].

1.2 Functional resonance magnetic imaging

Functional resonance magnetic imaging (fMRI) is an non-invasive neural imaging technique that tracks brain activity. Unlike standard (or structural) MRI, which images anatomical structure, fMRI images the blood oxygen level, or BOLD response, which has been shown to be correlated with neural activity.

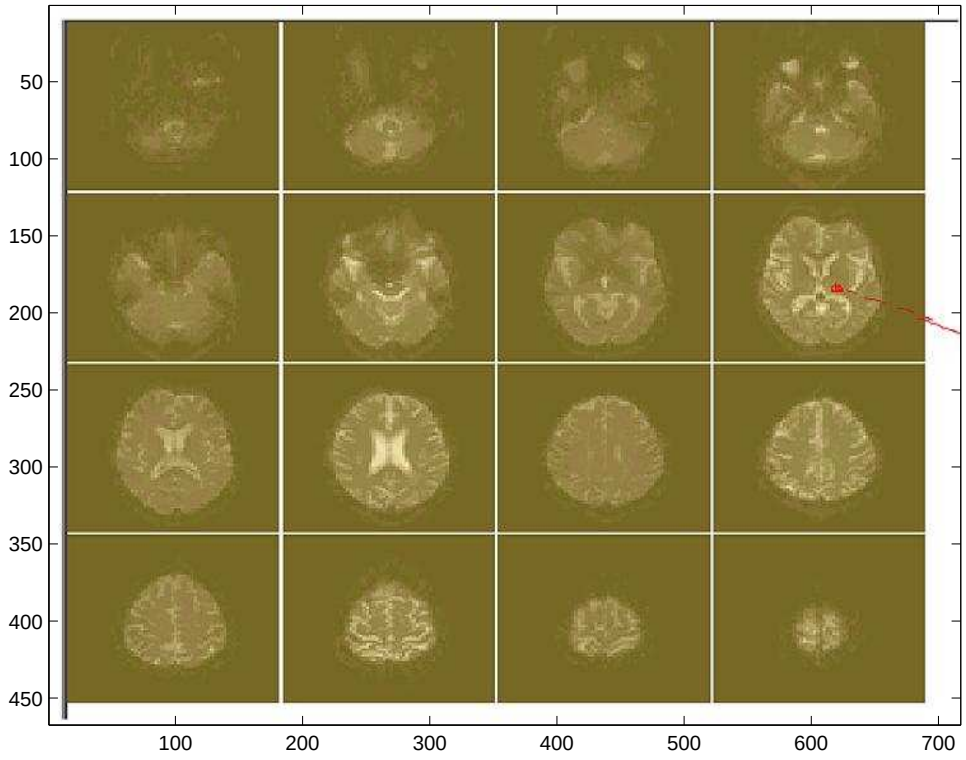

Figure 1: The 16 images are one data point

Each point of the fMRI data contains 16 images (one 3D image group). Each image has 64×64 voxels. Therefore each data point has $64 \times 64 \times 16 = 65,536$ values, which is the dimension of the data. Our purpose is to classify a patient as demented or mentally normal. Figure 1 shows one image group, i.e., one data point.

1.3 Goal of this project

To effectively use kernel methods for high dimensional data, we study parallel and high performance kernel computation. We will use distributed memory on a computing cluster and parallelize the computation across multiple computing nodes. We first formulate the kernel computation for high dimensional data in a way appropriate for parallel computing through linear algebraic manipulation. We then implement and experiment with the algorithms. To experiment our parallel algorithm, we first test it with the KDD cup data for correctness and then apply it to analyze and classify the fMRI data.

The main issues involved in this project are.

- Formulation of the kernel matrix computation so that it is suitable for parallel computing;
- Implementation of the formulated algorithms. Design of a proper communication pattern so that the communication overhead is as low as possible;

- Tuning of parameters, such as variance used in the Gaussian kernel. Weighting parameter in the SVM to minimize empirical error;
- Selection of the training set and the test set. By observing how well the SVM generalize, we will try different kernels with different parameters, and use various combination of data sets;
- Low level implementation issues such as endianness and feeding fMRI data into the cluster's frontier node after reading fMRI raw image by passing its header file and data file.

Section 2 of this report outlines our algorithm formulation. Section 3 describes parallel implementation. Section 4 presents test results and Section 5 concludes the report.

2 Algorithm Formulation

In this section, we briefly present our algebraic manipulation in making the kernel computation parallel, and then point out our core parallel computation functions.

2.1 Nonlinear support vector machines

The nonlinear support vector machine can be formulated as follows, where we follow the notation used by Mangasarian et al [13].

$$min \quad \nu e'y + \frac{1}{2}(u'u + \gamma^2) \tag{1}$$

$$subject \quad to \quad D(K(X, X')Du - e\gamma) + y \geq e \tag{2}$$

$$y \geq 0 \tag{3}$$

Solving the above optimization problem, we get u and γ [13]. For a given data point x, the separating surface (classifier function) is,

$$f(x) = sgn(K(x', X')Du - \gamma) \tag{4}$$

In the above equations, $X \in R^{m \times n}$ is the matrix containing all the training data. m is the number of data points in the training set, and n is the dimension of input space. We consider the case where $n >> m$. We use X' to denote transposition. X_i is the i^{th} row of matrix X, thus the i^{th} data point. $D \in R^{m \times m}$ is the diagonal label matrix, $D_{ii} = 1$, if the label at data point is 1; and $D_{ii} = -1$, if the label is -1. $u \in R^m$ and γ, a scalar, are the parameters determining the classifier function. If the classifier is linear, it uniquely determine a hyperplane. ν is a weighting parameter to emphasize the separation error vector $y \in R^m$. $e \in R^m$ is a vector of all ones. $K \in R^{m \times m}$ is the kernel matrix between X and X',

$$K = K(X, X') \tag{5}$$

2.2 Proximal support vector machine–PSVM

The optimization problem of (1), (2) and (3) is a quadratic programming problem. Its solution is a general nonlinear SVM. The proximal SVM is derived by changing (1), (2) and (3) to (6) and (7) below.

$$min \quad \nu\frac{1}{2}\|y\|^2 + \frac{1}{2}(u'u + \gamma^2) \tag{6}$$

$$subject \quad to \quad D(K(X, X')Du - e\gamma) + y = e \tag{7}$$

The solution to (6) and (7) is purely linear and defines the PSVM.

$$v = (I/\nu + D(KK' + ee')D)^{-1}e = (I/\nu + GG')^{-1}e \tag{8}$$

$$u = DK'Dv \tag{9}$$

$$\gamma = -e'Dv \tag{10}$$

$$y = v/\nu \tag{11}$$

$$G = D(K \quad -e) \tag{12}$$

And the separating surface is

$$f(x) = sgn(K(x', X')K' + e')Dv \tag{13}$$

The optimization problem of (6) and (7) is equivalent to that defined by (1), (2) and (3). Both theoretic proof and geometric interpretation are given [4]. PSVM is much faster to solve than SVMs requiring quadratic programming.

2.3 The parallel kernel algorithm

The commonly used kernels are inner product based. Some of them are listed in the following.

Gaussian kernel:

$$(K(X, X'))_{ij} = exp(-\mu\|X_i - X_j\|^2) \tag{14}$$

Homogeneous polynomial kernel:

$$(K(X, X'))_{ij} = (< X_i X_j >)^d \tag{15}$$

P-kernel (probability kernel):

$$(K(X, X'))_{ij} = p(X_i)p(X_j) \tag{16}$$

where $p(t)$ is the probability density function of t .

We take the Gaussian kernel as an example to formulate the kernel computation and then generalize the formulation to all other kernels. We partition the data matrix X in the following column-wise way,

$$X = (X^1 X^2 ... X^p) \tag{17}$$

Each X^k is of dimension $m \times r$, $r = n/p$, where p is the total number of processing nodes in the architecture. We assume r is an integer. For the Gaussian kernel,

$$(K(X, X'))_{ij} = exp(-\mu \left\| X_i - X_j \right\|^2) \tag{18}$$

$$\left\| X_i - X_j \right\|^2 = (X_{i1} - X_{j1})^2 + (X_{i2} - X_{j2})^2 + ... + (X_{in} - X_{jn})^2 \tag{19}$$

$$\begin{gathered} = (X^1_{i1} - X^1_{j1})^2 + ... + (X^1_{ir} - X^1_{jr})^2 + (X^2_{i1} - X^2_{j1})^2 + ... + (X^2_{ir} - X^2_{jr})^2 + ... \\ ... + (X^p_{i1} - X^p_{j1})^2 + ... + (X^p_{ir} - X^p_{jr})^2 \end{gathered} \tag{20}$$

We define a matrix $N \in R^{m \times m}$ as,

$$N_{ij} = N(X_i, X_j) = (X_{i1} - X_{j1})^2 + (X_{i2} - X_{j2})^2 + ... + (X_{in} - X_{jn})^2 \tag{21}$$

and

$$N^k_{ij} = N^k(X_i, X_j) = (X^k_{i1} - X^k_{j1})^2 + (X^k_{i2} - X^k_{j2})^2 + ... + (X^k_{ir} - X^k_{jr})^2 \tag{22}$$

The N matrix consists of the squared norm-2 values of the training data. We observe that,

$$N_{ij} = \sum_{k=1}^{p} N^k_{ij} \tag{23}$$

and

$$N = \sum_{k=1}^{p} N^k \tag{24}$$

Therefore

$$(K(X, X'))_{ij} = exp(-\mu N_{ij}) \tag{25}$$

We can write the kernel matrix compactly as,

$$K = exp(-\mu N) \tag{26}$$

In equation (22), N^k is only dependent on partition X^k of the sub-data matrix. Equation (24) and (26) allow us to parallelize the kernel computation with minimal communication cost. The algorithm for parallel kernel computation on a producer/consumer model (see Section 3 for details) is outlined below.

```
Parallel_compute_kernel( )
 {
    Producer partition X into sub-matrices according to (17)
    Producer sends sub-matrices to consumer nodes
    Consumer nodes compute sub-matrix N (22)
    Consumer nodes send sub-matrix N to producer
    Produce node sum sub-matrix N, (24)
    Producer node computes kernel matrix, (26)
}
```

Suppose we have a data set of a million dimension, $n = 2^{20}$. Using our partition on a cluster of $p = 128$ nodes, each partition has dimension $r = 8K$, which is much easier to handle. Since all commonly used kernels are inner product based, their parallel computation can be carried out by slight modification of the above algorithm.

It can be shown that if we partition the data row-wise, then we will have a lot of communication during parallel computation. Therefore our partitioning data column-wise to have smaller dimension on each node is a correct choice.

3 Implementation

We implement the parallel algorithm on the LosLobos cluster using MPI and the C++ language. We use the producer/consumer model to design collaboration of the cluster. The producer reads data file, partitions data and dispatches subsets of data to consumer nodes, which compute the N^k matrices and send them back to the producer nodes.

3.1 Partitioning the data

In this subsection we discuss the low level issues and how we partition data. The raw fMRI data is stored in big endian format. When processing the data on a Linux cluster, which is based on little endian architecture, we need to swap the bytes to read proper values. The dataset from Dartmouth College contains data of 41 patients, of which all the young patients are nondemented. Among the 27 adults, data of one of them is not complete. We use records of the 26 adult patients.

Each patient has been tested for 4 sessions, each of which has 128 trials. Since each group of 8 trials has the same function and the first 8 trials are just junk data, we take 15 raw image groups from each session, resulting in 60 image groups per patient. And we have totally $60 \times 26 = 1560$ image groups, i.e., 1560 data points. In summary, we have a data matrix of $m_0 = 1560$ rows and $n = 65536$ columns.

For a cluster of $p = 32$ computing nodes, we partition the dimension into 32 parts, each of dimension $n_{seg} = n/p = 2048$. Among the m_0 rows, $m = 1500$ data points are used for training and 60 (one patient) for testing, to assemble the leave one out cross validation (LOOCV) configuration.

Table 1: Data partitions on different clusters.

# of computing nodes	# of rows	# of columns
4	1560	16,384
8	1560	8192
16	1560	4096
32	1560	2048
64	1560	1024

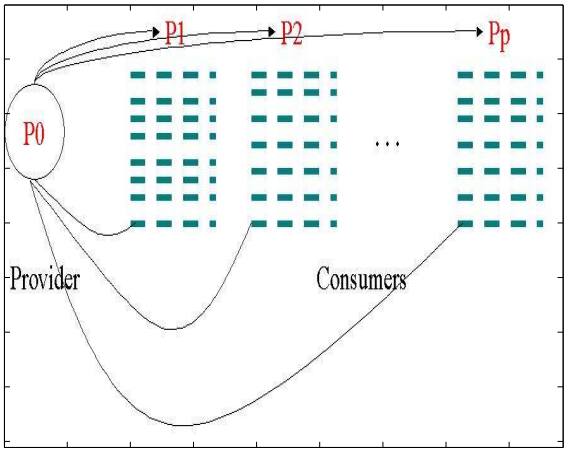

Figure 2: The producer-consumer parallel model

The provider node reads from data files and sends each consumer node a sub-matrix of $R^{m \times n_{seg}}$. Data partitions for clusters of different sizes are listed in Table 2.

On each consumer node, the received sub-matrix is split into training set and testing set. The kernel matrix is computed only on the training set in the training stage. The kernel between a data point and the training set is computed during the classification stage.

3.2 Computing the kernel matrix

On each consumer node, the squared norm-2 of training set of the sub-matrix is computed and sent to the provider node. The following code snippet shows the operations. Due to symmetry, $\frac{1}{2}m^2 n_{seg}$ operations are required instead of $m^2 n_{seg}$ operations.

```
================================================================
for (i=0; i<M; i++){ //i
    for (j=i; j<M; j++){ //j
        sum=0.0;
        for (k=0; k<NSeg; k++)
            sum=sum+ (XTr[i][k]-XTr[j][k])*(XTr[i][k]-XTr[j][k]);
```

```
        K[i][j]=K[j][i]=sum;
    } //j
} //i
MPI_Send(&K, M*M, MPI_FLOAT,...);
```

==========

The provider node receives the squared norm-2 matrices, sums them up and takes the exponential for the Gaussian kernel function. These operations are shown below. The producer-consumer model is illustrated in Figure 2.

```
==========================================================
for (j=0; j<sizeCluster; j++){ // j, consumers
   MPI_Recv(&KRecv, M*M, MPI_FLOAT,...);

   for (k=0; k<M; k++)
       for (L=0; L<M; L++)
           K[k][L]+=KRecv[k][L];
} // j, consumers

for (r=0; r<M; r++)
   for (s=0; s<M; s++)
        K[r][s]=exp((-1.0)*mu*K[r][s])
==========================================================
```

3.3 Solving the linear equations

In this subsection, we show how we save space and time through algebraic manipulation for the linear programming.

To classify a data point according to (13), we need to solve equation (8). We rewrite (8) in the following,

$$(I/\nu + D(KK' + ee')D)v = e. \tag{27}$$

To solve for v, we need to evaluate the matrix

$$P = (I/\nu + D(KK' + ee')D) = (I/\nu + P1), \tag{28}$$

where

$$P1 = D(KK' + ee')D = DQD, \tag{29}$$

where

$$Q = KK' + ee'. \tag{30}$$

Since D is the label matrix, computation of $P1$ can be simplified. We show how to simplify for the case of $m = 3$. Suppose,

$$D = \begin{pmatrix} d_1' \\ d_2' \\ d_3' \end{pmatrix} = (d_1 \quad d_2 \quad d_3) \tag{31}$$

where each d_i is a vector with element i being equal to ± 1, all others zero. Now,

$$P1 = DQD = \begin{pmatrix} d_1'Qd_1 & d_1'Qd_2 & d_1'Qd_3 \\ d_2'Qd_1 & d_2'Qd_2 & d_2'Qd_3 \\ d_3'Qd_1 & d_3'Qd_2 & d_3'Qd_3 \end{pmatrix} \tag{32}$$

We take one element, $d_1'Qd_2$, as an example to show how to simplify the matrix multiplications.

$$d_1'Qd_2 = (d(1) \quad 0 \quad 0) \times \begin{pmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{pmatrix} \times \begin{pmatrix} 0 \\ d(2) \\ 0 \end{pmatrix} = d(1) \times q_{12} \times d(2) \tag{33}$$

Therefore,

$$(P1)_{ij} = DCol(i) \times DCol(j) \times q_{ij}, \tag{34}$$

where $DCol$ is a vector containing data labels. In this way, the square matrix $D \in R^{m \times m}$ is replaced by an array of m elements.

Further more, the intermediate matrices of Q, $P1$, and the identity matrix can be eliminated and their spaces are saved. The code that computes the operations is shown below. The operations for $P = (I/\nu + D(KK' + ee')D)$ is accomplished in one triple loop. If we did not eliminate the D, Q and $P1$ matrices, we need two more triples loops, in addition to more space usage.

```
=================================================================
for (i=0; i<M; i++){ //i
    for (j=i; j<M; j++){ //j
        sum=0.0;
        for (k=0; k<M; k++)
             sum=sum + K[i][k]*K[j][k];     //K[j][k]=KT[k,j]
        P[i][j]=DCol[i]*DCol[j]*(sum+1.0);  //plus one here
        if(i==j) P[i][j]=P[i][j]+ (float)1/nu ;
    } //j
} //i
=================================================================
```

Due to our simplification through linear algebraic manipulation, we save three $m \times m$ arrays, and $2 \times m^3$ times.

The matrix inversion is avoided by using Cholesky factorization. The basic idea of Cholesky factorization is explained below.

To solve

$$Ax = b, \tag{35}$$

where $A' = A$. We can factorize A into

$$A = LL' \tag{36}$$

where, L is a lower triangle matrix. We first solve for y from the following equation using forward substitution,

$$Ly = b \tag{37}$$

Then solve for x from the following equation using backward substitution,

$$L'x = y \tag{38}$$

Thus, matrix inversion is not needed. For our case, $A = P$, $b = e$, and $x = v$. The code for Cholesky factorization is listed in Appendix A. Our implementation of the Cholesky factorization works in place, hence saves space.

3.4 Classification

The separating surface is given is equation (13). Its parallel computation is described in the following. The provider node receives squared norm-2 matrics between test data and the training set, sums them up, and takes the exponential when the Gaussian kernel function is used. It then evaluate the separating function and test on its sign. If it is positive, the data corresponds to a demented patient. Otherwise, nondemented. The code is illustrated below.

```
================================================================
for each test data { //i test data
   for each consumer node j { // j, consumers
       MPI_Recv(&X10, M, MPI_FLOAT,...);
       for (k=0; k<M; k++)
            X1[k]=X1[k]+X10[k];
   } //j
   for (k=0; k<M; k++)
       X1[k]=exp((-1)*mu*X1[k]);
   c=0.0;
   for (ii=0; ii<M; ii++)
       c=c+ KXKA[ii]*V[ii];
   if ( c>0 ){
            demented .....
        }
   else {
            nondemented...
        }
} //i test data
================================================================
```

4 Experiments

In this section we present our test results.

4.1 A small dataset

To verify that our coding for the PSVM is correct, we implemented a sequential version of the software and tested with the ionosphere dataset, $m = 350$,

Table 2: Times spent on different operations.

Operations	Time(sec)	Percentage
Read data	0.1077	1.8 %
Kernel Computation	5.3536	91.1 %
Cholesky factorization	0.414	7.05 %

$n = 35$ [20]. The test ran on the LosLobos cluster using only one node. We have achieved higher than 90% accuracy for cross validation. We have experimented with the Gaussian kernel, the polynomial kernel, the tri-cube kernel and the Epanechnikov kernel. We have also implemented and tested the Cholesky factorization for matrix inversion.

The tests on the 350×350 kernel (Gaussian) for the ionosphere data on single node show that 91% of computing time is spent on kernel computation, as shown in Table 2. Since kernel consumes most of the training time, we are justified to parallelize kernel computation, in addition to parallel classification.

4.2 Slow experiences with the fMRI dataset

Before we present our parallel performance, let's take a look at how slow it could be if we do not use parallel processing. The fMRI dataset is preprocessed to consist of $m_0 = 1560$ data points, among which $m = 1500$ data points are used as training set and the remaining 60 is for leave one out cross validation. The data dimension is $n = 65,536$. We wrote a sequential program and wanted it run on a single CPU computer.

On the PC at my home (P-III 1GHz CPU, 128 MB RAM), it did not run because of insufficient memory. On a PC with more memory, 256 MB RAM, 500 MHz PIII PC, it took 96678 seconds=26.8 hours to train and classify for the 60 data points. On a Sun Ultra Sparc work station with 1 GB RAM, 750 MHz CPU, we ran the same instance and did not get timing information. The code used to get the timing is shown below and the time used is reported to be -1.374 seconds. From the code below, we can see when the end time kept incrementing and crossed zero, it became smaller than the start time. That's why we got negative running time.

```
-----------------------------------------------------------------
clock_t start, end;
start=clock();
  SVM_Kernel();
end=clock();
cout<<"SVM time  in sec="<<(double)(end-start)/CLOCKS_PER_SEC<<endl; \\
$ SVM time in sec=-1.374
-----------------------------------------------------------------
```

With this slow performance, it is hard to continue our experiments.

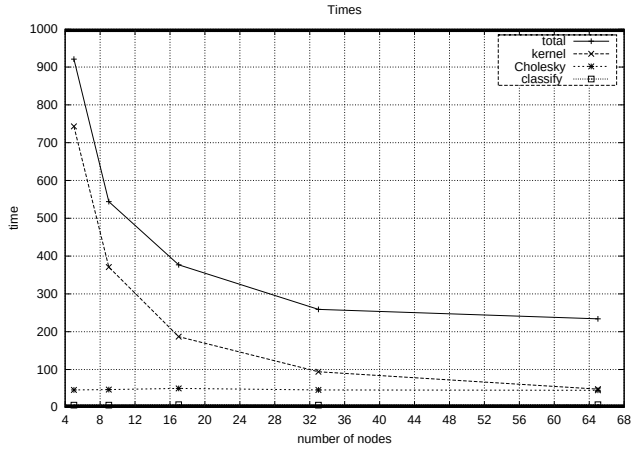

Figure 3: Parallel computing times

4.3 Parallel performance with the fMRI dataset

The fMRI dataset for training and testing is the same as in the sequential case. Namely, it consists of $m_0 = 1560$ data points, among which $m = 1500$ as training set and 60 for leave one out cross validation. The data dimension is $n = 65,536$. Each test run trains the data and classify the 60 data points.

We tested our parallel algorithm on 4, 8, 16, 32, and 64 consumer nodes. The times used is shown in Figure 3. The speedup is shown in Figure 4. With 32 computing nodes, we can run one test in 4 minutes, which is much better than 26 hours.

The correct classification ratio is 91% on the trained data and 62% for cross validation. Figure 3 and 4 are the cases for Gaussian kernels. When we use a 4-degree homogeneous polynomial kernel, we get 10.7% faster performance and the correct ratio is 87% on trained data, and 61% for cross validation . Overall, kernel computation consumes 85% of total computing time. The low correct ratio could be due to the fact that we do not have enough training data. The number of data point compared with data dimension is small and renders the problem less well constrained (please see section 5 on future work to improve accuracy).

The communication times used in all the runs are summarized in Figure 5. The times used to send the sub-data matrices need more attention. The time used for the first node to receive the sub-matrix is large on small cluster and small on large cluster. This is because that on a smaller cluster, each sub-matrix is larger and takes longer to transmit. The opposite situation is true for larger cluster. But the time when the last node received the sub-matrix is almost all the same, between 50 and 60 seconds. This phenomenon implies that communication overhead is not scalable and speedup will not scale up for larger clusters. For clusters large enough, communication cost will dominate total performance. When we increase the cluster size continually, we will get to a point where speedup will decrease.

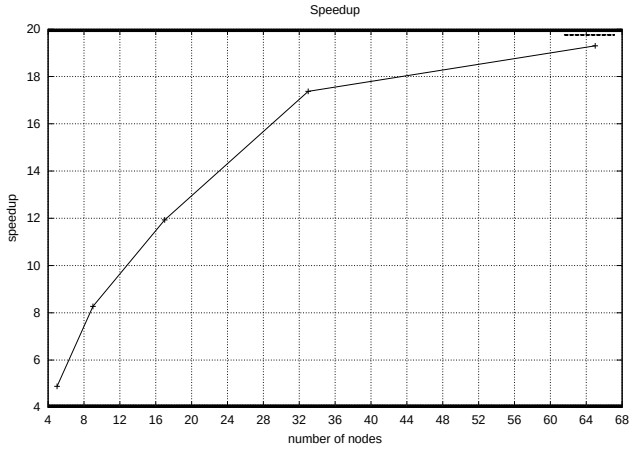

Figure 4: Speedups

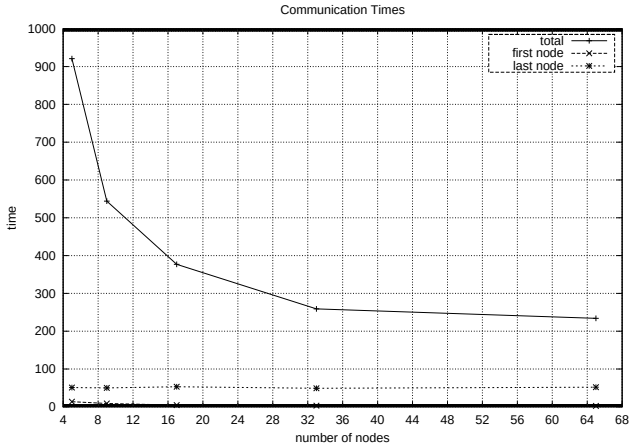

Figure 5: Communication times

5 Conclusions and Future Work

To use kernel based SVMs on high dimensional data and overcome the difficulty of slow computing speed, we propose to build parallel SVMs. We have formulated the PSVM, especially its kernel computation and classification to exploit parallelism (22),(24). We have simplified linear algebraic operations to save space and time (34). The algorithms are tested on a cluster for the fMRI imaging data, which has a high dimension. Substantial speedup has been achieved. Being able to reduce the running time from 26 hours to 4 minutes allows us to test more algorithms and do more meaningful experiments.

Specifically, we have done the following in this project.

- Implementation of the PSVM as a sequential algorithm and profiling its accuracy and running time.

- Implementation of the in-place Cholesky factorization to avoid matrix inversion for numerical stability and space saving.

- Formulation of the kernel based PSVM computation so that it is suitable for parallel computing. Partition data column-wise to reduce communication cost.

- Implementation of the formulated parallel PSVM algorithms. Substantial speedups have been achieved. Running time has been reduce from 26 hours to 4 minutes. Classification accuracies have been tested on the Gaussian and polynomial kernels.

- Simplification of linear algebraic operations so that space and time are saved. (Space saving, $3 \times m^2$; time saving $2 \times m^3$)

In the future we will focus on improving classification accuracy. Since the slow speed problem has been solved and we are able to run an instance in a couple of minutes, more experiments on different algorithms are possible. We will improve accuracy in a couple of ways. One way is to collect more data. We will contact the fMRI data center and see how we can get more data points on the dame dataset. The second approach is to integrate temporal information into consideration, since fMRI is both spatial and temporal. There is a new classification algorithm based on sets of vectors that has been successfully used to classify video surveillance images [25]. The method based on sets of vectors considers the entirety of data collected over the course of time and thereby integrates temporal structure into classification. However, this method involves even larger amount of computation. We will try implement this algorithm in parallel and apply it on the fMRI data.

6 Acknowledgements

We want to thank John Burge of the CS Department, for his ideas on some of the low level implementation issues and helpful discussions. We are also thankful to Mr. Roy Heibach of the UNM High Performance Computing Center, for his support for the LosLobos cluster.

References

[1] Asa Ben-Hur, David Horn, Hava Siegelmann, Vladimir Vapnik, Support Vector Clustering, Journal of Machine Learning Research, 2 (2001), pp. 125-137.

[2] Brown, M., Grundy, W., Lin, D., Cristianini, N., Sugnet, C., Ares, M., and Haussler, D., Support Vector Machine Classification of Microarray Gene Expression Data. Technical Report UCSC-CRL-99-09, Dept. of Computer Science, University of California, Santa Cruz, USA (1999)

[3] Thanh-Nghi Do, Francois Poulet, Incremental SVM and Visualizaiotn Tools for Bio-medical Data Mining, Proceedings of the European Workshop on Data Mining and Text Mining for Bioinformatics, September, 2003, in Dubrovnik, Croatia.

[4] Fung, G., Mangasarian O. L., Proximal Support Vector Machine Classifiers, Proc. of the 7th ACM SIGKDD, Int. Conf. On KDD'01, San Francisco, USA, 2001, pp. 77-86.

[5] Fung, G., Mangasarian, O.L., Incremental support vector machine classification. In Grossman, R., Mannila, H., Motwani, R., eds.: Proceedings of the Second SIAM International Conference on Data Mining, SIAM (2002) 247-260.

[6] P. E. Gill, W. Murray, and M. H. Wright, Practical Optimization. Academic Press, 1981.

[7] Trevor Hastie, Robert Tibshirani, Jerome Friedman, The Elements of Statistical Learning, Springer, 2001.

[8] Huang, J., Shao, X., Wechsler, H., Face pose discrimination using support vector machines (svm). In: Proceedings of 14th Int'l Conf. on Pattern Recognition (ICPR'98), IEEE, 1998, 154-156.

[9] Joachims, T., Making Large Scale SVM Learning Practical, In B. Scholkopf, C. Burges, and A. J., Smola, Ed., Advances in Kernel Methods-Support Vector Learning, pp. 169-184, Cambgidge, MA, MIT Press, 1999.

[10] L. Kaufman. Solving the quadratic programming problem arising in support vector classification. In B. Scholkopf, C. Burges, and A Smola, editors, Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge, USA, 1998.

[11] Y.-L. Lee and Mangasarian, O.L., RSVM: Reduced Support Vector Machines. Technical Report 00-07, Data Mining Institute, Computer Science Department, University of Wisconsin, Madison, Wisconsin, July, 2000. Proceedings of the First SIAM International Conference on Data Mining, Chicago, IL, April, 2001.

[12] Y.-L. Lee and Mangasarian, O.L., SSVM: A Smooth Support Vector Machines. Technical Report 99-03, Data Mining Institute, Computer Science Department, University of Wisconsin, Madison, Wisconsin, Sept., 1999. Computational Optimization and Applications 20(1), October, 2001.

[13] O. L. Mangasarian, Generalized support vector machines, In A. Smola, P. Bartlett, B. Scholkopf, and D. Schuurmans, editors, Advances in Large Margin Classifiers, pages 135-146, Cambridge, MA, 2000. MIT Press.

[14] E. Osuna, R. Freund, and F. Girosi. Support vector machines: Training and applications. A.I. Memo (in press), MIT A. I. Lab., 1996.

[15] P. Pavlidis, T.S. Furey, M. Liberto, D. Haussler, W. N. Grundy, Promoter Region-Based Classification of Genes, Proceedings of the Pacific Symposium on Boicomputing, January, 2001. pp. 151-163.

[16] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines,. In B. Scholkopf, C. Burges, and A Smola, editors, Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge, USA, 1998.

[17] Amund Tveit and Havard Engum, Parallelization of the Incremental Proximal Support Vector Machine Classifier using a Heap-based Tree Topology. Technical Report, IDI, NTNU, Trondheim, Norway, August 2003.

[18] Amund Tveit, Magnus Lie Hetland and Havard Engum, Incremental and Decremental Proximal Support Vector Classification using Decay Coefficients. In Proc. 5th Int. Conf. on Data Warehousing and Knowledge Discovery, DaWak, Springer-Verlag, 2003.

[19] Amund Tveit and Magnus Lie Hetland, Multicategory Incremental Proximal Support Vector Classifiers. In Proc. 7th Int. Conf. on Knowledge-Based Intelligent Information and Engineering Systems, KES, Springer-Verlag, 2003.

[20] UCI KDD data archive, http://kdd.ics.uci.edu. UCI machine learning datasets, http:// www.ics.uci.edu/˜mlearn/MLRepository.html

[21] V. Vapnik, The Nature of Statistical Learning Theory. Springer Verlag, New York, 1995.

[22] Vert, Jean-Philippe, Support Vector Machine Prediction of Signal Peptide Cleavage Site Using a New Class of Kernels for Strings, Proceedings of the Pacific Symposium on Biocomputing, 2002.

[23] Zaki, M., Wang, J., Toivonen, H.: BIOKDD 2002: Recent Advances in Data Mining for Bioinformatics. In SIGKDD Explorations, Vol. 4, Issue 2 (2002) 112-114

[24] Manfred K. Warmuth, Jun Liao, Gunnar Ratsch, Michael Mathieson, Santosh Putta, Christian Lemmen, Support Vector Machines for Active Learning in Drug Discovery Process, Journal of Chemical Information Science, vol. 43(2), pp.667-673, 2003.

[25] Lior Wolf, A. Shashua. Learning over Sets using Kernel Principal Angles. Journal of Machine Learning Research (JMLR), 4(10):913-931, 2003

[26] A. Zien, G. Ratsch, S., Mika, B. Scholkopf, T. Lengauer, K.-R. Muller, Engineering Support Vector Machine Kernels That Recognize Translation Initiation Sites, Bioinformatics, Vol. 16, no. 9, 2000, pp 799-807.

7 Appendix A

Code for the Cholesky factorization.

```
============================================================
void chole2(string strMatrix, string strB, int dimm, int dimn)
 {
//cholesky on matrix P; called in p0
    string strTemp;
```

```
    int flag=0;
    int i=0;
    int m,j,k;
    P[0][0]=sqrt(P[0][0]);
    for (m=1; m<M; m++)
        P[0][m]=P[0][m]/P[0][0];
    for (i=1; i<M; i++){ // i
        for ( k=0; k<=(i-1); k++)
            P[i][i]=P[i][i]-P[k][i]*P[k][i]; //  A(k,i)^2;

        if ( P[i][i] <0 ){
            flag=1;
            cout<< "Pii less than 0 at "<< i<<endl;
            return ;
        }
        P[i][i]=sqrt(P[i][i]);
        for ( j=i+1; j<M; j++){
            for ( k=0; k<=i-1; k++){
                P[i][j]=P[i][j]-P[k][i]*P[k][j];
            }
            P[i][j]=P[i][j]/ P[i][i]; //A(i,j)=A(i,j)/A(i,i);
       }
    } // i
////////////////////////////////  forward substitution
     V[0]=V[0]/P[0][0]; //b(1)=b(1)/A(1,1);
     for ( i=1; i<M; i++){     // i =2:n
         for( j=0; j <=i-1; j++) {  //1:i-1
            V[i]=V[i] - P[j][i]*V[j];   //  b(i)=b(i)-A(j,i)*b(j);
         }

        V[i]=V[i]/P[i][i];      //b(i)=b(i)/A(i,i);
     }
////////////////////////////////  backward substitution
    V[M-1]=V[M-1]/P[M-1][M-1];  //b(n)=b(n)/A(n,n);

    for (i= M-2; i >=0; i--){       //i=n-1: -1: 1
        for ( j=M-1; j>=i+1; j--){  //j=n: -1 :i+1
            V[i]=V[i]-P[i][j]*V[j]; //b(i)=b(i)-A(i,j)*b(j);
        }
        V[i]=V[i]/P[i][i];  //b(i)=b(i)/A(i,i);
    }
}
=====================================================================
```