A Parallel Processing Algorithm for Remote Sensing Classification

J. Anthony Gualtieri
Code 606.3 and Global Sciences and Technology
NASA/GSFC Greenbelt, Maryland 20771
gualt@backserv.gsfc.nasa.gov

1 Introduction

A current thread in parallel computation is the use of cluster computers created by networking a few to thousands of commodity general-purpose workstation-level commuters using the Linux operating system. For example on the Medusa cluster at NASA/GSFC, this provides for super computing performance, 130 Gflops (Linpack Benchmark) at moderate cost, $370K. However, to be useful for scientific computing in the area of Earth science, issues of ease of programming, access to existing scientific libraries, and portability of existing code need to be considered. In this paper, I address these issues in the context of tools for rendering earth science remote sensing data into useful products. In particular, I focus on a problem that can be decomposed into a set of independent tasks, which on a serial computer would be performed sequentially, but with a cluster computer can be performed in parallel, giving an obvious speedup. To make the ideas concrete, I consider the problem of classifying hyperspectral imagery where some ground truth is available to train the classifier. In particular I will use the Support Vector Machine (SVM) [1] approach as applied to hyperspectral imagery [16].

The approach will be to introduce notions about parallel computation and then to restrict the development to the SVM problem. Pseudocode (an outline of the computation) will be described and then details specific to the implementation will be given. Then timing results will be reported to show what speedups are possible using parallel computation. The paper will close with a discussion of the results.

2 Supervised Classification of Hyperspectral Data

I will address the problem of supervised classification of hyperspectral images working only in the spectral domain. Because the data is *high dimensional*, in that there are hundreds of channels of spectral information per pixel comprising a spectra and represented as a vector of values, algorithms that use all of the channels can be computationally expensive. The traditional approach to handling this is to perform a pre-processing dimension reduction step, but this removes valuable information. Here we will use an algorithm that uses all the channels.

For supervised classification, the problem separates into a training phase and a testing phase. For the training phase we are given exemplars with class labels, in this case the spectra associated to the ground truth pixels in the scene. These are used to train the classifier. In the testing phase the trained classifier is then applied to pixels not yet classified and not used in the training phase.

[1]The term "machine" in Support Vector Machine is only a name and does not imply a hardware device.

2.1 Simplifying Classification with Many Classes

2.1.1 Training

Given we have an algorithm to classify a pair of classes, how do we build classification algorithms that can handle multiple classes? Let K be the number of classes, where $K \approx 10 \ldots 20$ and more, and where each class contains varying numbers of training vectors (spectra in the case of hyperspectral imagery). A successful strategy to cope with large numbers of classes is to decompose the problem into a number of simpler classification problems that each deal with only *pairs* of classes. One such method is to train over all possible pairs of classes, $K(K-1)/2$, in number, and then use some voting scheme in the testing phase of the classification. Call this the K choose 2 method. An alternative approach is to train K pair classifiers, each of the form 1 versus $K-1$ and then in the testing phase compare results from the K classifiers to render a classification. Call this the 1 vs. $K-1$ method. The second method involves a heavy burden on training the K such classifiers, one for each class, because there will be many training spectra from the $K-1$ classes that are lumped together for training. The SVM training algorithm has computational complexity $O(m^2)$, where m is the number of training vectors in the pair of classes. Computational complexity here means how the time to complete the computation on a single processor scales in terms of the number of units of data, m, to be processed — in this case the number of training spectra. Thus, if time to completion is given by $t = Am^2 + Bm + C$, where t is time to completion, and A, B, C are constants, then $t \sim \mathrm{O}(m^2)$. To obtain an estimate of the complexities of the two methods, assume each training class is the same size, $\frac{m}{2}$. Then the time complexity for the two approaches is given in Table 1. In the left column is the K choose 2 approach and in the right column is the 1 vs. $K-1$ approach. The second row indicates where these results come from.

Table 1: Time complexity on a single processor for performing SVM training for a classifier of K classes each containing $m/2$ exemplars.

K choose 2	1 vs. $K-1$
$\mathrm{O}(K^2m^2) \sim$ $K(K-1)/2{*}\mathrm{O}(m^2)$	$\mathrm{O}(K^3m^2) \sim$ $K{*}\mathrm{O}(((K-1)\frac{m}{2} + \frac{m}{2})^2)$

There is a time complexity advantage to using the K *choose* 2 method. We adopt this approach in the what follows. In addition, there is some evidence that the K *choose* 2 method is more accurate than the 1 *vs.* $K-1$ method [17]. For the K choose 2 method, simple parallelism can be readily obtained by putting each pair-training computation on a different processor on the cluster machine. If there are more pairs than processors, then they are stacked onto processors and solved sequentially. Note we will not parallelize the basic pair-training algorithm as this would require having detailed knowledge of the SVM training algorithm optimization method, though see [28], for results from such an approach for training a pair classifier with very large numbers of training vectors.

2.1.2 Testing

Once we have trained the $K(K-1)/2$ pair classifiers, we obtain the classification in the testing phase by applying all the pair classifiers to each spectra (one per pixel) to be classified, and for each pair noting which of the two classes is found for that pixel. Then all $K(K-1)/2$ results per pixel (think of votes) are placed into one of K bins and the bin with the largest number of votes is declared the class for that pixel. For ties in the number of votes, either a random selection is made or some refinement using a numerical measure from the pair classifier is used. Thus, if we define $V_{\mathbf{r}} \in [1, \ldots, K]$ as the classification label for the pixel at position $\mathbf{r}$, then the winning class label at pixel $\mathbf{r}$ is given by

$$V_{\mathbf{r}} = \arg\max_k \sum_{i,j\ i<j} \delta_{k, v_{\mathbf{r}}(i,j)}, \tag{1}$$

where for pair classifier i, j,

$$v_r(i,j) = \begin{cases} i & \text{if class } i \text{ wins over class } j \\ j & \text{if class } j \text{ wins over class } i \end{cases},$$

and

$$\delta_{m,n} = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

is the Kronecker delta function.

Here parallelism can be used in several ways. We can use a single processor testing code for all $K(K-1)/2$ pairs, and segment the imagery into tiles that cover the whole scene, and process the tiles each on a different processor. Or we may use a single processor code that implements the testing phase for each of the $K(K-1)/2$ pair classifiers applied to the whole image on a single processor, and follow this with a voting scheme to obtain the final classification. The voting scheme part could be run on a single processor as it is likely to be very fast, or if need be, applied on different processors in a tiling of the image.

2.2 The Support Vector Machine Algorithm

The basic algorithm used in this paper to perform supervised classification of hyperspectral remote sensing data is the SVM coupled with the Kernel Method [25]. Previous work using the SVM algorithm together with the Kernel Method derives from the machine-learning community, with methods pioneered by Vapnick, and extended by Boser, Guyon, and Vapnick [2], and Cortes and Vapnick [6]. In particular the SVM approach yields good performance on *high-dimensional* data such as hyperspectral imagery, and does not suffer from the *curse of dimensionality* [1], which limits the effectiveness of conventional classifiers due to the Hughes effect [18]. This because the SVM classifier depends on only a small subset of the training vectors, called the support vectors, which define the classifier as used in the testing phase.

Application of SVM to hyperspectral data was first demonstrated by Gualtieri and co-workers in [16] [15] [14]. I direct the reader to those references for details of the SVM formulation and application to several hyperspectral data sets, including Indian Pines 1992 of image size 145 $\times$ 145 with 200 bands and 16 classes [2] [22], and Salinas 1998 with image size 512 $\times$ 215 with 224 and 16 classes [3] [21]. In the reported work above, all implementations were performed on a single workstation class processor and used T. Joachim's open source SVM^{light} package [19] [20]. The most computationally expensive task within the SVM training requires quadratic optimization, for which A. Smolla's open source PR_LOQO was used [26]. More recent work [29] [23] [4] [3]. has provided further validation of SVM's utility for hyperspectral classification.

3 Parallel Implementation for Supervised Learning by Porting Serial Code

We take as given a serial code for performing SVM training and testing for classification of hyperspectral data. Here we introduce more of the details of the parallel computation and demonstrate how the serial code can be adapted to parallel computation.

[2] A 145 $\times$ 145 pixel and 220 band subset in BIL format of radiance data that has been scaled and offset to digital numbers, DN, from radiance, rad in units $W/(cm^2 * nm * sr)$ according to DN = 500 * rad + 1000 as 2bit signed integers. It is available at `ftp://ftp.ecn.purdue.edu/biehl/MultiSpec/92AV3C`. Ground truth is also available at `ftp://ftp.ecn.purdue.edu/pub/biehl/PC_MultiSpec/ThyFiles.zip`. The full data set is called Indian Pines 1 f920612t01p02_r02 as listed in the AVIRIS JPL repository (`http://aviris.jpl.nasa.gov/ql/list92.html`). The flight line is Lat_Start: 40:36:39 Long_Start: -87:02:21 Lat_End: 40:10:17 Long_End: -87:02:21 Start_Time:19:42:43GMT End_Time:19:46:2GMT. See also `http://dynamo.ecn.purdue.edu/~biehl/MultiSpec/aviris_documentation.html`

[3] AVIRIS Hyperpsectral Radiance Data from f981009t01r_07. The full flight line is 961 samples by 3479 lines consisting of 224 bands of BIP calibrated radiance data that is scaled to lie in [0,10000] as 2bit signed integers. We have studied part of scene 3. The full flight line is called f981006t01p01_r07 as listed in the AVIRIS JPL repository. The flight line is [Lat_Start: +36-15.0 Long_Start: -121-13.5 Lat_End: +36-21.0 Long_End: -121-13.5 Start_Time: 1945:30GMT End_Time: 1950:15GMT.

3.1 Programming Model, Hardware, Message-Passing Interface

For parallel programming, currently the most popular model is Multiple Instruction Multiple Data (MIMD). Here each of the processors can perform different instructions on different data, and there is no synchronization of the processor clocks. Because of the lack of explicit synchronization, any interprocessor communication must be explicitly initialized and and then tested for completion at the sending and the receiving processors, if subsequent processing is to remain synchronized. This overhead of testing for completion is assumed to be the responsibility of the programmer, and is part of what makes this programming model difficult to use.

Such MIMD machines may or may not have a shared memory system among all the processors, but the most typical use a collection of nodes each consisting of a symmetric multiprocessor (SMP) with a shared memory and shared disk. Typically each SMP node may comprise two to four single processors. Here shared memory is only at the nodes. In the sequel we will take the basic unit of computation as the processor. The differences in communication within a node and between nodes will not be considered, which is reasonable given the particular problems we solve. This particular hardware configuration is called a Beowulf cluster and can consist of a few to hundreds of nodes [27].

The software used here to coordinate the processors is the Message Passing Interface (MPI) [10] [8] [13]. MPI extends the C and C++ languages (or Fortran) with a set of subroutines that implement communication and data movement among unsynchronized processors, and provides for methods to test for synchronization. With careful layout of data and communications, the multiple processors can divide up a computation efficiently, although there is a heavy load on the programmer to craft the code. The actual code used in any particular parallel problem is heavily problem dependent.

3.2 Pleasingly Parallel Problems

If we can decompose the serial code into a series of tasks with minimal communication between the tasks, namely at the beginning and ending of the tasks, then the porting of the serial code fits under the rubric of *pleasingly parallel.* [4] As an example, consider problems involving imagery where the spatial coordinates do not enter explicitly into the computation. A decomposition can result from dividing the image into tiles on which the same computation is to be performed. A very simple example would be normalizing an image by dividing by a given constant value. In a serial machine code, a loop over all N pixels is run, and each pixel is normalized one after the other, taking $O(N)$ time. In a parallel setting, if we had as many processors as pixels, then each processor would perform one division operation taking $O(1)$ time. For comparison, a less trivial task would be to find a normalization factor by adding values from each pixel and dividing by the number of pixels. On a parallel machine, explicit communication between nodes would be required as part of the task. In fact, this can be accomplished in $O(log_2 N)$. See Appendix A.

The SVM training and testing problem we address here can be thought of as having a simple decomposition into independent parts that can be processed, with communication occurring only at the start and finish of each task.

3.3 Organizing the Problem

Let us now consider how to organize a cluster computer for a pleasingly parallel problem with n_p number of processors and n_t number of independent tasks. Call the set of tasks the task list. We wish to find an approach that keeps all the processors busy as much of the time as possible. If the size of the task list is $n_t \leq n_p$, then put one task on each processor and wait until the last task completes.

If $n_t > n_p$, a naive solution would be to first put an equal number of tasks, $\lfloor n_t/n_p \rfloor$, [5] on each processor

[4] This is also referred to as *embarrassingly parallel* in that, in principle the decomposition of the problem is trivial.

[5] $\lfloor x \rfloor$ is the floor function, defined as the nearest integer to x that is smaller than or equal to x.

and let them run sequentially on each processor until complete. If there are remaining tasks, $n_{rem} = n_t - \lfloor n_t/n_p \rfloor \neq 0$, then put these last tasks on any of n_{rem} processors. However, this will be efficient only if all tasks take the same amount of time. This solution is like the queuing problem in a bank where a large number of people enter at one time and each go to a teller in a separate queue. If no more people enter and once in a line you cannot shift, then roughly the lines start out at the same length. However, as transaction duration for each person varies, some people will have longer waits than others while eventually some tellers will have no customers. For SVM, the training times will be different because of different numbers of training vectors in the classes to be trained on.

At this point, note that it has not been said where the original complete task list lives, or how the task list is divided up and communicated among the processors. This is taken care of by designating a particular processor of the cluster as the *boss processor*, whose job is to manage the apportioning of the task list and the communication of tasks to the remaining processors, which are called the *worker processors.* Then with the boss processor able to monitor the status of each worker processor, the boss processor can, first using a list of tasks waiting to be done, send one each to all the worker processors and thereafter wait for workers to be done and then send the next task in the list to that worker processor. In this way, all the worker processors are always busy. We have implemented this approach in prototype program called `boss_workers` and subsequently mapped the learning phase and the testing phase of the supervised classification onto it. However, there still remain issues concerning the sending and receiving of communications from the boss to the worker, and from the worker to the boss. We address this below.

3.4 Aspects of Communication in MPI

Here we give an overview of how to handle the asynchronous communication between processors in the MPI language for this particular application. Communications consists of two parts, sending and receiving. To describe this process we use a metaphor based on the postal system and described in Table 2. Here buffer is

Table 2: Postal system analogy to MPI send and receive statements.

Postal System	MPI
sender writes letter	sender loads buffer
sender posts letter and waits for notice of receipt	sender executes MPI_Send on buffer
recipient waits for mail to be received	receiver executes MPI_Receive on buffer
recipient reads letter	receiver unloads buffer

a piece of memory available to the sending and receiving processor. Note that MPI_Send and MPI_Receive do not "simply post the letter" or check for "receipt of a letter", and then continue to do other things. These MPI statements are called blocking send and receives, and they cause the processor to wait for completion before continuing to the next statement in the code. This may have a problematic side effect as illustrated in Table 3, showing deadlocking involving executing MPI_Send and MPI_Receive on processors i and j. The

Table 3: Deadlocking in MPI.

processor i	processor j
MPI_Send(j)	MPI_Send(i)
MPI_Recv(j)	MPI_Recv(i)

problem is that it is possible, depending on buffer size, for neither MPI_Send(i) or MPI_Send(j) to complete because each MPI_Send awaits the execution of the MPI_Recv on the other processor, which cannot be reached.

To handle this problem and to make MPI code *safe*, [6] meaning here that deadlocking cannot occur, MPI provides a second kind of Send and Recv called *non-blocking* designated MPI_ISend and MPI_IRecv. These statements post the send and receive messages, but do not wait for completion. Now to know if the messages are completed, another class of MPI statements called MPI_Wait is needed.

3.5 Boss_workers algorithm in MPI

Now we can can give an abstract version of the `boss_workers` algorithm written in pseudocode. In programming in MPI the approach is to write a *single* code that will be running simultaneously on the boss and the worker processors. Thus, we can have sections of identical code running on all the processors, albeit not synchronized, or we can can have sections that are specific to one or more processors. The overall code makes this distinction by using a function in MPI that returns the number of the processor it is running on. By testing on this number in an *if* or *case* statement, we can designate in the single MPI code, sections that will run only on designated processors. In the `boss_workers` code, one processor, numbered 0, is the boss, and $1 \ldots n_{worker}$ are the workers. Communication issues here only involve the boss communicating with a worker. No worker-to-worker communication occurs.

We present below a simplified code for the `boss_workers` algorithm. The framework for this code is described for the case of each task taking the same computational effort (same time to execute) in the book and on the web site of Peter Pacheco [24], with particular applicaton to matrix vector multiplication. However, for the classification problem, the worker tasks require different computational effort, and also require a generalization. This has been provided by William Gropp in the example of the use of the MPI command `MPI_Waitsome`, [12]. Beside the communications, the computational parts are:

`make_task_list`, where the `boss` generates the tasks.

`boss_make_task`, where the `boss` takes a particular task and creates data for a `worker`.

`worker_work` takes the input data sent by the `boss`, and the `worker` creates an output that is returned to the `boss`.

`consume`, where the `boss` takes the output from the `worker` and completes processing on that output.

In application to the training phase the `worker_work` would be replaced by a module that performs SVM supervised learning, which for training takes as input the pair of two sets of exemplar or training vectors and outputs a file describing the appropriate classifier in terms of the support vectors it finds. For the testing phase, the input is the set of classifiers found in the training phase, and the hyperspectral cube of unclassified data. The output is the classified image. In the following pseudocode, scalar variable declarations are not explicitly included while arrays `a, b` with `n, m` elements, respectively, are explicitly declared with `allocate( a[0:n-1]), b[0:m-1], ...)`.

[6]The problem of *safety* and a related concept *fairness* can be resolved in a variety of ways. See [11].

```
boss_workers( n_tasks, n_procs ) {

    my_id   = MPI_Comm_rank;        % get my own processor number
    n_workers = n_proc - 1;         % don't count boss
    boss = 0;                       % boss is always labeled 0
    i_task_bs = 0;                  % running index of boss task ( 0 based )
    i_task_wk = 0;                  % running index of worker tasks (0 based) labels
                                    % task, but not always  worker doing task

    if ( my_id == boss ) {                 % in this part is the code that runs on the boss
          allocate( buff_bs_snd   [0:n_task_chunk-1],    % buffers for boss
                    buff_bs_snd   [0:n_task_chunk-1],
                    indices_bs_rec[0:n_task_chunk-1],
                     status_bs_rec[0:n_task_chunk-1],
                    indices_bs_rec[0:n_task_chunk-1]  );
          make_task_list( n_tasks, task_list );
          n_tasks_chunk = minimum( n_tasks, n_workers );   % max tasks running at once
          n_tasks_to_do = n_tasks;
          i_done        = 0;                               % 0 based count of completed tasks

          for (i=0;i< n_tasks_chunk;  i++) {                                    % full execution
              boss_make_task(  i_task_bs, n_tasks, task_list,  buff_bs_snd[i] ); % of loop sends
              MPI_Send(  buff_bs_snd[i], i+1,  i_task_bs+1 );                   % out one chunk of
              MPI_Irecv( buff_bs_rec[i], i+1  );                                % tasks and posts
              i_task_bs = i_task_bs + 1;                                        % recv's
          }
          while (n_tasks_to_do > 0 ){
              MPI_Waitsome( n_tasks_chunk,  n_done, indices_bs_rec, senders_array );
              for (j=0; j<n_done; j++ ) {
                    k = indices_bs_rec[j];        % k is index of the buff which received the send
                    sender  = senders_array[j]
                    consume(  buff_bs_rec[k] );
                    i_done        = i_done + 1;   % total completed tasks
                    n_tasks_to_do = n_tasks_to_do + 1;
                    if ( i_task_bs <= n_tasks - 1 ) { % if tasks are left send a new task to
                                                      % processor that finished and post a receive
                         boss_make_task(  i_task_bs, n_tasks, task_list,   buff_bs_snd[k] );
                         MPI_Send(    buff_bs_snd[k],   sender, i_task_bs+1 );
                         MPI_Irecv(   buff_bs_rec[k],   sender, request_bs_rec[k]  );
                         i_task_bs   = i_task_bs + 1;
                    }
              }
          }
          for (i=0; i< n_workers; i++) {
              MPI_Send(0, 0, i+1, DIETAG );  % all done, send DIETAG is a defined constant
          }                                  % to status_wk_rec on worker to notify worker
    }                                        % to shut down
    else{  while (1) {
              allocate( status_wk_rec[0], buff_wk_rec[0], buff_wk_snd[0] );
              MPI_Recv(  buff_wk_rec[0],  boss,  status_wk_rec );
              i_task_wk = status_wk_rec;  %
              if ( i_task_wk == DIETAG) {
                 break;
              }
              worker_work(  buff_wk_rec[0],  buff_wk_snd[0] );
              MPI_Send(   buff_wk_snd[0],  boss, i_task_wk );
          }
    }
}
```

4 Data Input and Output

A secondary issue is how data is to be passed to and from boss and workers. We can either have the boss read all the data and then explicitly pass the data to the workers at the time it is needed or we can make all the data available to the workers who then read the parts they need, based on names of data passed to them by the boss. Because the data set can be large, there are questions of how to best distribute the data to optimize overall computational time.

We elected to have all the input and output be processed by the workers, and have only labels describing the files to be input and output, passed by the boss to the workers. This finesses the passing of large data files between the boss and the workers, and places that burden on the file system of the cluster. For the benchmark computations of SVM training and testing, each node has its own copy of all required input files and writes only to its local disk.

It is possible to have all files shared by workers and boss by using Network File System (NFS), where the operating system passes copies of the needed file to the nodes as they need them. This effectively shares the same file system on all processors, and preliminary results for the SVM problem show similar performance. Note that the total size of the input data and output data in the training phase is 1.6 MB and 6.8 MB, respectively, and for the testing phase 23.6 MB and 2.5 MB, respectively. Given input/output (IO) speeds on the hardware used here, the time for IO of the data used here is small compared to processing times. However, for larger problems, the IO time costs may become significant, and different approaches may be needed for higher performance.

4.1 Details of the MPI Subroutines Used

The explanation of the MPI functions and a definition used in the pseudo code are:

`MPI_Send( buff_bs_snd[i], i+1, i_task_bs+1 )` This posts the data from the boss `buff_bs_snd[i]` into the communication channel to processor `i+1` with message tag `i_task_bs+1` . Only when the buffer is received and emptied does it complete (blocking send).

`MPI_Irecv( buff_bs_rec[i], i+1 )` This accepts the data into `buff_bs_rec[i]` on the boss processor from the worker processor `i+1` and continues without waiting for completion (nonblocking receive).

`MPI_Waitsome( n_tasks_chunk, n_done, indices_bs_rec, senders_array )` Waits for multiple communications to complete on the boss from `n_tasks_chunk` different workers. The quantity `indices_bs_rec` is an array of size `n_tasks_chunk` that contains the index identifying the receiving buffers among all the buffers `buff_bs_rec[ ]` corresponding to the members of the array `senders_array` of size `n_tasks_chunk`, which contains the identities of the `n_tasks_chunk` senders.

`MPI_Recv( buff_wk_rec[0], boss, status_wk_rec )` Blocking receive that accepts the data into the buffer `buff_bs_rec[0]` from the boss processor with resulting status written to the structure `status_wk_rec`.

`DIETAG` is a defined constant to signify work is completed.

In this implementation the buffers passed between boss and workers, `buff_bs_snd`, `buff_bs_rec`, `buff_wk_snd`, and `buff_wk_rec`, are very small structures containing labels describing files to be read by the workers, and timing information to be returned.

4.2 Implementation Details

In this code development for SVM, we opted to use a library of open source routines in C called LIBSVM, written by Chih-Chung Chang and Chih-Jen Lin [5]. In addition to a series of routines for input and output, they have created a library of core subroutines and a set of data structures that we use in our parallel code. To handle hyperspectral data we have written our own IO routines as well as routines to convert from a hyperspectral cube to the data structure they use. These routines are then embedded into the `boss_workers` code.

In vanilla MPI only the single data type, vector, can be defined for the buffers that pass data to and from the boss to the workers, however, it is extremely useful to be able to pass structures or other more complex data types in the buffers. One can use various MPI routines to do this, but a utility called AutoMap [7] automates this and is used in this implementation.

5 Timing and Speedup

To examine the performance of the parallelization of SVM code as implemented in the `boss_workers` algorithm, first we demonstrate the scaling of the speedup (ratio of single code execution time over parallel code execution time) for the `boss_workers` code on a very simple task. Here the work done on each worker processor is to make it sleep for a fixed length of time. Then as a benchmark of a real problem, we apply training and testing of the SVM to the well-studied hyperspectral data set, Indian Pines 1992 [22], of size $[n_l, n_s, n_b] = [145, 145, 200]$ with $K = 16$ classes. In what follows, all times reported are wall clock time, overall elapsed time, rather than actual computation time as performed by the processor. The times reported are based on the nodes in use having only one user. It is possible to have time sharing of parallel jobs, but this is not the case here.

5.1 Simple Validation

Let the work done on each worker processor be to sleep for a fixed length of time, say 5 s. We expect that if the number of processors matches the number of tasks, then total computing time will remain the same independent of the number of tasks. Table 4 shows the total time taken (wall-clock) assuming no other jobs running. Except for the jump of 1 s for $n_p = 1 + 100$, the results are consistent with the notions above. The slight increase in time is presumably due to the boss performing the overhead of managing the workers.

Table 4: Time to completion (s) and speedup for the fixed length computation of sleep 5 s versus nunber of processors.

$n_{proc} = 1 + n_{work}$	1+1	1+2	1+3	1+4	1+5	1+10	1+20	1+30
time (s)	5.04	5.16	5.17	5.18	5.21	5.22	5.30	5.44
speedup	1.00	0.98	0.97	0.97	0.97	0.97	0.95	0.93

$n_{proc} = 1 + n_{work}$	1+40	1+50	1+60	1+70	1+80	1+90	1+100	1+110
time (s)	5.59	5.73	5.96	5.96	6.77	6:96	7.59	6.51
speedup	0.90	0.88	0.85	0.85	0.74	0.72	0.66	0.77

5.2 SVM Training on Indian Pines 1992

For this data set there are 16 classes giving $16 * 15/2 = 120$ pairs. The number of training vectors in the 16 classes is given in Table 5. It is clear that the pair involving classes 2 and 11 will take more training time than any of the other pairs, and it this pair that is the cause of the limiting behavior in the speedup as seen below. Table 6 shows timing results with varying numbers of processors applied to training 120 pair classifiers.

Table 5: Training class sizes for the 16 classes in the Indian Pines 1992 data set. These sizes represent 20% of the available ground truth.

class label	1	2	3	4	5	6	7	8
# train vec.	10	286	166	46	99	149	5	97
class label	9	10	11	12	13	14	15	16
# train vec.	4	193	493	122	42	258	76	19

Table 6: Time to completion (s) and speedup of the training phase for the Indian Pines 92 full data set versus number of processors.

$n_{proc} = 1 + n_{work}$	1+ 1	1+ 2	1+ 3	1+ 4	1+ 5	1+ 10	1+ 20	1+ 30	1+ 40	1+ 50
time (s)	31.8	17.2	13.3	10.8	9.7	8.8	9.0	9.1	9.3	9.2
speedup	1.0	1.9	2.4	2.9	3.3	3.6	3.5	3.5	3.4	3.5
$n_{proc} = 1 + n_{work}$	1+ 60	1+ 70	1+ 80	1+ 80	1+ 90	1+ 90	1+100	1+110	1+120	
time (s)	9.5	9.2	9.2	9.5	9.3	9.4	8.7	9.8	9.9	
speedup	3.3	3.4	3.5	3.4	3.4	3.4	3.6	3.3	3.2	

Here there is the expected result that the speedup improves and then levels out or saturates, (reaches a constant value) around 1+10 processors and then trends slightly down. The trend down for a larger number of processors possibly reflects the communications between the boss processor and the workers occurring in a shorter interval leading to more contention for bandwidth in the network that connects the processors to each other. The small variations up and down for 1+10 and more processors in the speedup are not understood.

The reason that the speedup saturates is because the processor times are not the same. When there is one task that takes substantially longer than any other task, then several faster tasks can be completed before the longest task. Thus, while one processor is tied up with longest task, there will be a given number of processors, fewer than the number of tasks, where cumulatively all the shorter tasks will be completed in the same time as the longest task. Adding more processors than this number will not increase speedup as the time limitation will be the time of the longest task.

5.3 SVM Testing on Indian Pines 1992

For testing, we split the computation into a part that applies each of the 120 classifiers built in the training phase above to the whole data cube, and a part that aggregates these results to get the final classification result. The first part results in 120 arrays, each the size of the image, and where each element is the winning label of one of the two classes from each particular classifier. The aggregation part combines these 120 arrays in a voting scheme according to Eq. 1 on page 2. This second step can be computed quickly, less than 1 s for the 120 145 $\times$ 145 array to be processed, so there is no need to parallelize this step here. In Table 7 are given the results of applying all 120 pair classifiers to the data cube without the aggregation step for varying numbers of worker processors.

Here speedup saturates at around 1+30 processors. The reason for saturation is the same as that for the training phase, a nonuniform distribution of processing times over the tasks. Again the relation of the time for the slowest pair to complete versus the other times to completion determines some number of processors

Table 7: Time to completion (s) and speedup for computation of testing phase (without final voting) for Indian Pines 92 full data versus number of processors.

$n_{proc} = 1 + n_{work}$	$1+1$	$1+2$	$1+3$	$1+4$	$1+5$	$1+10$	$1+20$	$1+30$	$1+40$
time (s)	237.5	120.7	91.3	76.2	67.3	45.8	37.0	32.9	33.6
speedup	1.0	2.0	2.6	3.1	3.5	5.2	6.4	7.2	7.1

$n_{proc} = 1 + n_{work}$ y	$1+50$	$1+60$	$1+70$	$1+80$	$1+90$	1 +100	1 +110	1 +120	
time (s)	35.3	33.6	33.0	31.8	34.6	34.1	33.5	33.7	
speedup	6.7	7.1	7.2	7.5	6.9	7.0	7.1	7.0	

where saturation occurs. This slowest pair determines the fastest time for completion for all tasks that can be accomplished on a cluster of processors with more than the saturation point number of processors.

6 Discussion

In this paper aspects of parallel computation implemented on a Beowulf cluster as applied to the supervised classification of hyperspectral imagery are examined. Because this type of classification is computationally expensive, scaling as $O(m^2K^2)$, where K is the number of classes and m is the number of training vectors per class, the application of parallel techniques is justified. The structure of this problem avails itself of simple parallelization because both the training and the application of the trained classifier to the unclassified imagery, the testing phase, can be decomposed into $K(K-1)/2$ simpler tasks involving only pairs of classes. The problem then is to pass these multiple smaller tasks to the processors of the cluster machine so as to keep all the processors busy. A simple algorithm called the `boss_workers` algorithm implements this. One processor, the `boss`, creates a list of tasks and then passes the tasks to the `worker` processors and awaits their completion at which point a remaining task is sent to the available worker. Implementation of this algorithm in the standard language for cluster computing, MPI with C (or Fortran), must deal with the lack of explicit synchronization among the processors. In particular the various task usually complete in differing lengths of time. MPI handles this using a set of routines to send, receive, and test for completion of communication.

A measure of the effectiveness of parallelization is the speedup. Speedup, S, is defined as the ratio of *time to completion on a single processor to time to completion on multiple processors*. For pleasingly parallel problems where n_t tasks are all decoupled, and where the time to completion is each task is the same, then

$$S = \frac{n_t}{\lceil \frac{n_t}{n_w} \rceil},$$

and [7]

$$S = n_t \text{ if } n_w \geq n_t.$$

This shows that speedup saturates at n_t when $n_w \geq n_t$ if all processes take the same length of time. If there are different times of computation, then lower speedups occur, saturating at fewer numbers of processors. The results obtained above show speedups for supervised classification of hyperspectral imagery for the small problem studied here to be of the order of 3.5 for the training phase and around 7 for the testing phase, much less than the optimal value of 120, obtained only if all the tasks take the same time to complete. The reason for the saturation is due to a non-uniform distribution of times among the tasks. The reason for the training having a lower speedup than the testing phase is because it has a more non-uniform distribution of

[7] $\lceil x \rceil$ is the ceiling function, defined as the nearest integer to x which is larger than or equal to x.

processing times than for testing. If the processing times are ordered from longest to slowest, then roughly the distribution is more non-uniform if the differences in processing times for tasks are greater. Recall that if all the processing times are equal, the speedup saturates only when $n_{tasks} = n_{workers}$. The distribution of task times in descending order for the first thirteen for the training phase, in s, is

$$7.50 \quad 5.79 \quad 3.32 \quad 1.86 \quad 1.85 \quad 1.49 \quad 0.84 \quad 0.74 \quad 0.62 \quad 0.58 \quad 0.58 \quad 0.48 \quad 0.38$$

while for the testing phase, in s, it is

$$31.2 \quad 29.5 \quad 20.7 \quad 15.5 \quad 14.7 \quad 9.9 \quad 8.0 \quad 7.0 \quad 6.9 \quad 6.3 \quad 5.8 \quad 5.6 \quad 4.9.$$

The steeper drop in processing times for the training phase than testing phase causes a more rapid saturation of speedup at a lower value. Note that the time for longest task for the training phase is 7.5 s which is close to the saturated time of 9 s, while for the testing phase the time of the longest task is 31 s which is close to the saturated time of 33 s The task times are only for the worker performing the task without the overhead of the `boss_worker` code creating the tasks and managing them.

In conclusion, though the programming burden of using MPI can be substantial, because all the details of the multiple processor synchronization and communication are the responsibility of the programmer, the low cost and ready availability of cluster hardware suggests that for typical hyperspectral scene sizes of the order 1000 × 10000 with numbers of classes of multiples of 10 can be managed. Of note is that it is currently possible to build very inexpensive low-power clusters [8] [9] that could be made portable.

Because results so far for supervised SVM classification can detect subtle class differences, the approach discussed here provides a means to push the use of SVM techniques to much larger scenes and number of classes than are currently performed for hyperspectral imagery. It remains for future work to see how far this approach can more fully exploit the large amount of information contained in hyperspectral imagery with reasonable times for computation.

Acknowledgments

The author is grateful to Chi-Chung Chang and Chih-Jen Lin [5] for making LIBSVM available for implementing the SVM pair classification. John Dorband, Josephine Palencia, Udaya Ranawake, and Glen Gardner of NASA/GSFC have provided hardware and software support for the cluster system at GSFC used in this work.

Appendix A Simple example of parallel computation

In time step 1 and in parallel let processor p_1 perform the addition of a_1, a_2, p_2 perform the addition of a_3, a_4, ... $p_{N/2}$ perform the addition of a_{N-1}, a_N. In time step 2, let p_1 add the the result from step 1 to the result from $p2$ in step 1, p_2 add the the result from step 1 to the result from $p3$ in step 1, ... $p_{N/4}$ add the result from step 1 to the result from $p_{N/2}$. Continuing like this eventually leads to the last single addition between p_1 and p_2 at time step $O(log_2 N)$. Note that we spend a fixed time at each step to pass values between processors, which itself occurs $O(log_2 N)$ times.

[8]Proteus cluster with 14 nodes, (mini ITX boards), power consumption 170 W, peak demand 438 W. Controlling nodes have 256 MB RAM, gigabit ethernet, and a 40 GB hard drive. The NFS server node is similarly equipped. Computational nodes have 256 MB RAM. Peak performance 5.2 GFLP and 10,400 MIPS with 13 computational nodes. Cost ≈ \$3000.

References

[1] R. E. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, Princeton, New Jersey, U.S.A., 1961.

[2] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM, June 1992.

[3] L. Bruzzone and F. Melgani. Classification of hyperspectral images with support vector machines: multiclass strategies. In *Proceedings of SPIE*, 2004.

[4] G. Camps-Valls, L. Gómez-Choval, J. Calpe-Maravilla, E. Soria-Olivas, J. D. Martín-Guerrero, and J. Moreno. *Support Vector Machines for Crop Classification Using Hyperspectral Data*, pages 134 – 141. Springer-Verlag, Heidelberg, October 2003.

[5] Chih-Chung Chang and Chih-Jen Lin. Libsvm – a library for support vector machines. The demo is available from `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[6] C. Cortes and V. N. Vapnik. Support vector networks. *Machine Learning*, 20:1–25, 1995.

[7] J. E. Devaney, M. Michel, J. Peeters, and E. Baland. Automap: A software tool for the automatic creation of mpi data structures from user code. Technical report, NIST, April 1997. `http://www.itl.nist.gov/div895/savg/parallel/`.

[8] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[9] Glen Gardner, 2004. `http://members.verizon.net/~vze24qhw/`.

[10] W. Gropp. *Tutorial on MPI: The Message Passing Interface.* Argonne National Lab., Argonne, Illinois. Available at `http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html\#Node0`.

[11] W. Gropp. *Tutorial on MPI: The Message Passing Interface.* Argonne National Lab., Argonne, Illinois. See node 87 to node 93 in `http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html\#Node0`.

[12] W. Gropp and E. Lusk. *Exercises for the MPI Tutorial.* Argonne National Lab., Argonne, Illinois. Available at `http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/src/fairness/Cwaitsom%e/solution/.html`.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1996.

[14] J. A. Gualtieri and S. Chettri. Support vector machines for classification of hyperspectral data. In *International Geoscience and Remote Sensing Symposium.* IEEE, 2000.

[15] J. A. Gualtieri, S. R. Chettri, R. F. Cromp, and L. F. Johnson. Support vector machine classifiers as applied to aviris data. In R. Green, editor, *Summaries of the Eighth JPL Airborne Earth Science Workshop:JPL Publication 99-17*, pages 217–227. Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, Feb 1999. Available at `http://aviris.jpl.nasa.gov/docs/workshops/99\_docs/toc.htm`.

[16] J. A. Gualtieri and R. F. Cromp. Support vector machines for hyperspectral remote sensing classification. In R. J. Merisko, editor, *27th AIPR Workshop, Advances in Computer Assisted Recognition, Proceedings of the SPIE, Vol. 3584*, pages 221–232. SPIE, 1998.

[17] C.W. Hsu and C. J. Lin. A comparison of methods for multi-class support vector machines. *IEEE Trans. on Neural Networks*, 13(2):415–425, 2002.

[18] G. F. Hughes. On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, 14(1):55–63, 1968.

[19] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning.* MIT Press, 1998. Available at `http://www-ai.cs.uni-dortmund.de/DOKUMENTE/Joachims_98b.ps.gz`.

[20] T. Joachims, 2001. The SVM^{light} package is written in GCC and distributed for free for scientific use from `http://ais.gmd.de/~thorsten/svm_light/` It can use one of several quadratic optimizers. For our application we used the A. Smola's PR_LOQO package [26].

[21] L. F. Johnson. Aviris hyperpsectral radiance data from: f981009t01r_07, 1998. `ftp://makalu.jpl.nasa.gov/pub/98qlook/1998_lowalt\_index.html`.

[22] D. Landgrebe. Indian pines aviris hyperpsectral radiance data: 92av3c, 1992. Available at `ftp://ftp.ecn.purdue.edu/biehl/MultiSpec/92AV3C`. `ftp://ftp.ecn.purdue.edu/pub/biehl/PC_MultiSpec/ThyFiles.zip`. `http://aviris.jpl.nasa.gov/ql/list92.html`. `http://dynamo.ecn.purdue.edu/~biehl/MultiSpec/aviris_documentation.html%`.

[23] M. Lennon, G. Mercier, and L. Hubert-Moy. Classification of hyperspectral images with nonlinear filtering and support vector machines. In *International Geoscience and Remote Sensing Symposium*, 2002.

[24] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1996. `http://nexus.cs.usfca.edu/mpi/`.

[25] B. Schölkopf, C. Burges, and A. Smola, editors. *Advances in Kernel Methods.* MIT Press, 1998.

[26] A. Smola, 2001. The PR_LOQO quadratic optimization package is distributed for research purposes by A. Smola at `http://www.kernel-machines.org/code/prloqo.tar.gz`.

[27] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, volume I, Architecture, pages I:11–14, Boca Raton, Florida, August 1995. CRC Press.

[28] G. Zanghirati and L. Zanni. A parallel solver for large quadratic programs in training support vector machines. *Parallel Computation*, 29(4):535–551, 2003.

[29] Junping Zhang, Ye Zhang, and Tingxian Zhou. Classification of hyperspectral data using support vector machine. In *Proceedings 2001 International Conference on Image Processing*, volume 1, pages 882–885, October 2001.