NCV: A Machine Learning Development Environment for Parallel Evaluation

Brad Davis, Ethan Sommer, David R. Musicant
Carleton College
One North College Street
Northfield, MN 55057
davisbr,sommere,dmusican@carleton.edu

Abstract

We present NCV (N-fold Cross-Validator), a general purpose C++ library built on top of MPI (Message Passing Interface) that allows one to easily implement cross-validation in parallel for effectively any algorithm. While there are many libraries available for doing parallel processing, and many for implementing machine learning algorithms, NCV is designed for the researcher who needs to do intense cross-validation calculations and wishes to transparently and simply harness the power of multiple computers. All the details of cross-validation, result aggregation, and multiprocessor communication are completely handled by the NCV system. The practitioner is freed from implementing these tasks, and only needs to modify three functions that actually implement the training and testing algorithms.

1 Introduction

Cross-validation is a primary methodology in measuring the success of machine learning algorithms [6, 10]. Techniques such as classification, regression, clustering, and feature selection all make use of cross-validation techniques in predicting generalization success. Cross-validation is inherently parallelizable, as it requires repeatedly breaking the data into different segments, running an algorithm on some of the segments, and using the remaining data to test the results. The desire to do calculations in parallel has become popular in recent years due to the dropping prices of multiprocessor machines, as well as the increasing availability of networked commodity single-processor machines. We present NCV (N-fold Cross-Validator), a general purpose C++ library built on top of MPI (Message Passing Interface) [4] that allows one to easily implement cross-validation in parallel for effectively any algorithm.

NCV fills a specific niche in the machine learning community. It is aimed at the researcher who has access to parallel processing capabilities, either via a parallel machine or a networked collection of individual machines, who does not wish to be concerned with details of parallel programming. Many libraries exist for doing parallel programming, most commonly MPI [4] and PVM [15]. These tools provide support for launching processes on parallel machines and passing messages between them, and are inherently useful to the parallel programmer. NCV provides an extra layer of abstraction on top of these tools to assist the machine learning practitioner.

There are a number of general purpose systems in place for automating parallel process management and communication. Beowulf [14] provides a mechanism for clustering a collection of individual computers to appear as one shared machine. Condor [8] runs silently on machines throughout an installation, and can take advantage of idle machines as requested. While both these environments would also provide the capability to do parallel cross-validation, both require significant setup costs. Moreover, a machine learning practitioner using either of these systems would still have to write code to handle partitioning of data for cross-validation. NCV is relatively easy to install, as the user only needs to install MPI and NCV in a directory that each machine can see. This can be on a shared network drive, or the software can be installed independently on each machine. NCV handles the specific effort involved in partitioning the data for cross-validation, partitioning it out to the individual machines, and aggregating and reporting results.

This environment is certainly not the first "general purpose" machine learning programming environment. MLC++ [7] and Weka [5] are two well-known examples. NCV takes inspiration from these other approaches in providing a straightforward and simple approach for coding up algorithms to run in parallel. While we believe that NCV is quite a useful environment in and of itself, we hope that our architecture might provide direction to parallel generalizations of other existing and future machine learning systems.

We now provide an overview of the rest of the paper. Section 2 reviews cross-validation techniques. Section 3 gives a high level overview of how the practitioner would use NCV. In Section 4 we describe the internals of NCV, and how it is built on top of MPI. We show some experimental results in Section 5 indicating the effective integration and performance of NCV.

Finally, we conclude and provide thoughts for future directions in Chapter 6.

2 Cross-Validation Overview

While cross-validation applies to any number of machine learning problems, we will restrict ourselves to classification problems for purposes of this discussion. The NCV structure is quite general, however, and can be easily extended for other kinds of problems as well. We therefore assume that we have a dataset with m points and k features, indicated by the $m \times k$ matrix A. We also have an $m \times 1$-dimensional vector y, each element of which contains the value that the corresponding row in A is to predict. For classification problems, the elements of y correspond to discrete classes.

In order to test how well a particular machine learning algorithm will generalize to future data, the most common technique is n-fold cross-validation. The matrix A, and the corresponding elements of y, are partitioned horizontally into n separate groups. For each of n iterations, a distinct group (or *fold*) of the data is *held out* from the processes as a *test set*, and the remaining data is used as a *training set*. The machine learning algorithm is run on the training set, and validated via the test set. This yields n estimates of the generalizability of the algorithm as illustrated by its performance on the test set, which are then averaged together to yield the overall test set accuracy. Most often, $n = 10$ (tenfold cross-validation) or $n = m$ (leave-one-out cross-validation).

We note that cross-validation serves a number of purposes in machine learning, beyond measuring the expected success of a single technique. It is often used for comparing multiple algorithms, where statistical tests can be performed for determining whether there are significant differences between algorithms [1]. Cross-validation is also a common tool to use in conjunction with a tuning set for finding optimal parameter settings.

3 NCV Philosophy and Usage

NCV performs cross-validation through a single *master* process and a number of *slave* processes. The master process is responsible for assigning the actual work to be done by the slave processes. On startup, the NCV master process starts up a slave process on each processor to be used for parallel cross-validation. If there are more slave processes than folds to be evaluated, NCV kills off the extra unnecessary processes. The master process then determines all the jobs that need to be run, i.e. the folds to be evaluated, and enters each job into a single job queue. After the queue has been created, slave processes begin asking the master process for work to do. Each slave process receives an assignment that contains information on how the data is to be allocated between training and test sets. As each slave process completes its task, it returns its accuracy rates to the master process for aggregation, signals that it is complete, and asks for another job. When a slave indicates to the master that its job is complete, the master process removes the completed fold from the job queue and sends another fold off to the slave if there are any jobs left in the queue. When the last job in the queue has been completed the master averages the results, computes additional summary statistics, and reports them. Slaves that ask for more work are informed that all the folds are complete, and directed to terminate.

3.1 Coding customized algorithms

Integrating machine learning algorithms in C++ into NCV is quite straightforward, and only requires modification of three functions in the file `ncv_alg.cpp`:

3.1.1 The training algorithm

```
void *ncv_train(const ncv_dset& trnData, int argc, char** argv)
```

The function `ncv_train` is where the code for the actual training algorithm is inserted. The NCV environment does all the data manipulation and segmentation automatically, and provides a ready-made training set for the programmer in the object `trnData`. `trnData` is an object from the class `ncv_dset`, which is used to store datasets. Specifically, here are the elements of the class which are of interest:

```
class ncv_dset
{
    public:
        ncv_npts_t npts;    // # of points in training set
        ncv_ndims_t ndims;  // # of dimensions in training set
        ncv_atr_t** atrs;   // Matrix containing
                            //      numeric attributes
        ncv_cid_t* cids;    // Vector containing class ids
};
```

trnData, therefore, contains all the data in the training set as well as any associated class information. The programmer writes the learning code within the context of this function, using trnData as the data source. The arguments argc and argv provide any optional command line parameters which the user wishes to input to the algorithm. For a neural network back-propogation algorithm [10], for example, these parameters might indicate how many hidden nodes the network has and what the learning rate should be.

The goal of any training algorithm is to build a *rule*. This rule indicates how predictions are to be made on novel data. The rules associated with different algorithms require dramatically different data structures. The rule for a neural network, for example, might consist of a series of matrices containing learned weights, whereas the rule for a decision tree requires a data structure capable of storing a tree. NCV does not impose any particular data structure on the programmer, but rather leaves this completely up to the programmer. The only restriction is that the rule ultimately must be summarized by a single identifier. This can, of course, always be done by grouping all elements of the rule via a C++ struct or class. When training is complete, the programmer casts a pointer to this object as a void*. This ensures that the rule can be adequately manipulated by the NCV system, regardless of type.

3.1.2 The testing algorithm

```
ncv_acc_t ncv_test(const ncv_dset& tstSet, void *pRule,
                   int argc, char** argv)
```

ncv_test is passed a dataset tstSet with class information and a rule. The programmer writes the code to apply this rule to each data point in tstSet to estimate a class, and compare this estimated class with the actual one in order to determine an accuracy rate. This rate is then returned.

3.1.3 Rule destructor

```
void ncv_delete_rule(void*& pRule)
```

ncv_delete_rule is a destructor for the rule which was created by ncv_train. Code is written here to appropriately free up whatever memory has been allocated for the rule.

```
void *ncv_train(const ncv_dset& trnData, int argc, char** argv) {
  ncv_dset* rule = new ncv_dset(trnData);
  return(rule);
}
```

Figure 1: Training code for a basic nearest neighbor algorithm.

```
void ncv_delete_rule(void*& pRule) {
  ncv_dset* pMySet = (ncv_dset*) pRule;
  delete pMySet;
  pRule = NULL;
}
```

Figure 2: Cleanup code for a basic nearest neighbor algorithm.

To reiterate the value of NCV: the only programming work that the practitioner needs to do is to modify these three functions. The rest of the software, unchanged, automatically handles data distribution, parallel execution, and result aggregation.

As an example of how this might be done, Figures 1, 2, and 3 show the code necessary for a simple nearest neighbor algorithm. The learning process is trivial, as the rule simply becomes a data structure storing all the data points and their associated classes. The testing procedure requires looking at each test point, finding the point in the training set which is closest, and using the class of that training point to make a prediction. As a more complicated example, we borrowed the C++ code for the ASVM algorithm (Active Set Support Vector Machine) [9, 12] and were able to integrate it into this environment with a minimum amount of effort. This code is available from the authors on request.

3.2 Parallelizing pre-existing machine learning programs

The above section describes how one can create new source code or integrate pre-existing machine learning source code into NCV. A fortunate but unintended side-effect of our design also allows one to integrate pre-existing

```
ncv_acc_t ncv_test(const ncv_dset& tstSet, void *pRule, int argc, char** argv)
{
  ncv_dset *pTrnSet;
  pTrnSet = (ncv_dset*) pRule;

  ncv_acc_t distSq, minDist;
  ncv_cid_t cidGuess = 0;
  ncv_npts_t nRightGuess = 0;
  for(ncv_npts_t cTstPt = 0 ; cTstPt < tstSet.npts ; cTstPt++) {
    minDist = (ncv_acc_t) ULONG_MAX;
    for(ncv_npts_t cTrnPt = 0 ; cTrnPt < pTrnSet->npts ; cTrnPt++) {
      distSq = 0;
      // Compute Euclidean distance
      for(ncv_ndims_t cDim = 0 ; cDim < tstSet.ndims ; cDim++)
        distSq += pow(pTrnSet->atrs[cTrnPt][cDim]
                      - tstSet.atrs[cTstPt][cDim],2);
      // Keep track of minimum
      if(distSq < minDist) {
        minDist = distSq;
        cidGuess = pTrnSet->cids[cTrnPt];
      }
    }
    // Count guess as correct if matches
    if (cidGuess == tstSet.cids[cTstPt]) nRightGuess++;
  }
  return ((ncv_acc_t) nRightGuess / (ncv_acc_t) tstSet.npts);
}
```

Figure 3: Testing code for a basic nearest neighbor algorithm. This and Figures 1 and 2 are the *only* modifications necessary to NCV to implement parallel cross-validation using nearest neighbor.

machine learning software with no modification to such software. To demonstrate this, we integrated the support vector machine solver within the machine learning tool Weka [5]. Though the source code for Weka is available (in Java), we did not use it at all. We wrote our `ncv_train` function to take the data provided in `trnData`, and write it out to a flat file of the input format that Weka requires. `ncv_train` then issues a system call containing the appropriate command to fire up the Weka tool and dump the output back out to disk. The `ncv_test` routine performs a similar data dump and system call to appropriately call the Weka software to determine a test set accuracy based on the new data and the model that was saved to disk. This is a "messy" solution, and not how NCV was designed to be used; nonetheless, it is a quick and easy way to do parallel cross-validation on external software. Examples of our Weka integration code will be provided with the rest of the software when downloaded.

4 NCV Internals

We now describe in more depth the architecture of the NCV environment. NCV is built on MPI (Message Passing Interface). A number of implementations of MPI are available for use, some of them commercial. We chose to use the free MPICH implementation [3], and we recommend it to anyone who wishes to use NCV. While NCV should work just as well with any implementation of MPI, we have not tried any other. MPICH is freely available for a number of different operating systems. Our implementation was on Redhat Linux 6.2 [13], though the code should be easily portable to other operating systems that have an implementation of MPI available.

We chose to do a single install of MPI on a shared network drive, though this is not necessary. MPI can actually handle machines on a network with local MPI installations, and the machines do not necessarily have to even be the same architecture or be running the same operating system. Since we run NCV on top of MPI, these configuration issues are all transparent to NCV. Installing MPI into a shared directory for a network of homogeneous machines was remarkably simple. See the MPICH installation guide for more details [2].

The next step is to inform MPI which machines are available for use. This is done via direct configuration within MPI at the time of installation, or with a *procgroup* file that is provided as a command line parameter when NCV is started up. Again, both these options are features of MPI which are completely transparent to NCV. The user is free to configure MPI in whichever manner is most appropriate for the network at hand. We chose to set up our procgroup file. Here is a sample:

```
system1.here.edu 0 /Accounts/ncv/ncv
system2.here.edu 1 /Accounts/ncv/ncv
system3.here.edu 1 /Accounts/ncv/ncv
system4.here.edu 1 /Accounts/ncv/ncv
system5.here.edu 1 /Accounts/ncv/ncv
```

Each line specifies a machine which is available for use by MPI. The first column contains the name of the machine. The second contains the maximum number of processes allowed to run on the machine. The first machine has a 0 listed because it is the master machine; it will already be running the main executable. The third column contains the location and name of the executable to run.

An overview of the process can be found in Figure 4. When NCV is invoked on the master machine, MPI immediately fires up and makes connections with all slave machines specified in the procgroup file. If there are more machines available than folds to be evaluated, the master process notes this and only uses a number of machines equal to the number of folds. The master process then transmits the entire dataset to each machine. Next, it creates a *job* associated with each fold, which effectively indicates which points from the dataset are to be considered part of the training set and which are part of the test set. These jobs are stored in a queue. Whenever a machine is available for work (all of them at first), a job is removed from the queue and transmitted to that machine. The job is then pushed back onto the rear of the queue.

Once a slave process has received a job (i.e., an indication of which fold to work on), it executes the training algorithm on that dataset. It then measures the performance of that rule on the training and test sets, and passes this information back to the master process. The master process stores this information for later aggregation, removes that job from the queue, and assigns another job to that slave machine if the queue is not yet empty. When the queue is finally empty, the master process summarizes the results from the cross-validation, reports them to the user, shuts down the slave processes, and ends execution.

This model of re-inserting jobs onto the rear of the queue, then removing them when a job is complete adds some fault tolerance to the system. If a slave process handling a specific job dies for some reason, the job is still present in the queue and another slave will eventually find it and work on it. Moreover, consider a case where termination is "bottlenecked" in a heterogeneous network by all remaining jobs being handled by slow processors. A free fast processor can go to the queue to obtain a job, which in this case is simultaneously being handled by a slow processor. Whichever machine finishes the job first sends its results to the master, and the results are aggregated. The master will then ignore any further results on the same job. In the current release of the NCV software, the master process will immediately report the aggregate results when all jobs have been complete, and terminate when all slave processes are complete. Some of these slave processes may be redundantly finishing jobs which other processors have already completed. In a future release of NCV, we plan to kill the remaining processes at this time.

We were able to make an additional optimization in our code that is worth describing. On shipping the data to each slave, that slave is immediately directed to begin working on the fold corresponding to its own ID number (slaves are numbered with consecutive integers starting at 1). These jobs are therefore immediately placed at the rear of the job queue. This optimization allows each slave process to start training immediately after it receives its data, rather than being forced to contact the master process for its first assignment. Without this technique, it is conceivable that each slave would not actually start to process anything until the master process has completed distributing all the data and can find time to give each slave an assignment.

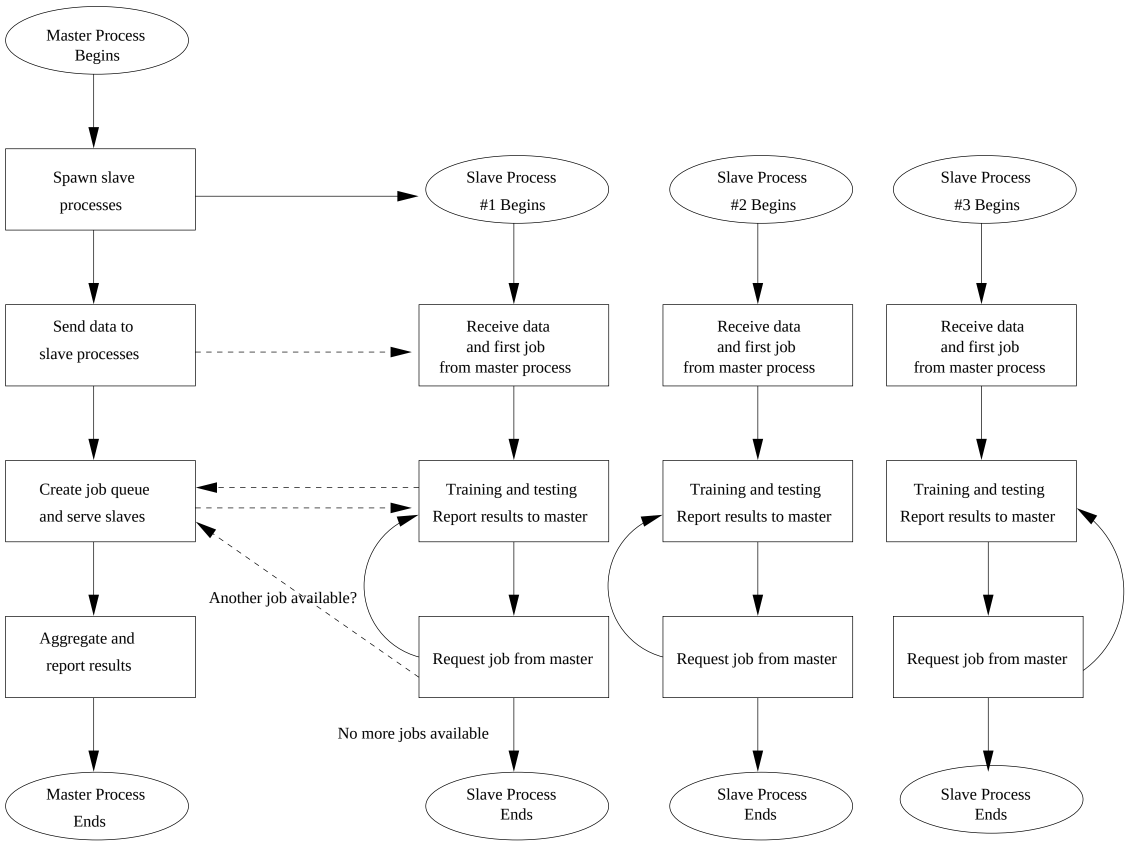

Figure 4: Flow between master and slave processes. Solid lines indicate program flow, and dashed lines indicate communication between processes. The communication arrows shown actually occur between the master and all three slave processes, but communication only with the first slave process is shown for simplicity.

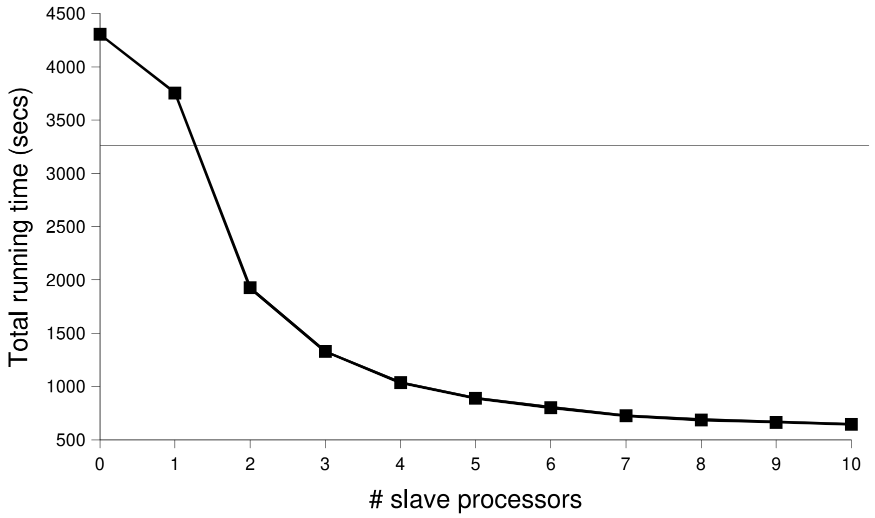

Figure 5: Speedup in parallel cross-validation on single processor 400 MHz Pentium II machines. The horizontal line indicates running time on a single processor without using the NCV environment.

5 Experimental Results

In order to demonstrate the effectiveness of NCV, we ran the Weka support vector machine solver with default parameter settings on the liver-disorders dataset from the UCI repository [11]. As described in Section 3.2, NCV makes this very easy to implement. The liver-disorders dataset contains 345 points in six dimensions, and the goal is to find a classifier that correctly distinguishes between the two classes. We ran 100-fold cross validation on this dataset using up to eleven computers from our instructional labs. We had exclusive access to these machines while running the experiments. Each computer has a single 400 MHz Pentium II processor with 64 MB of memory, and runs Redhat Linux 6.2. We performed our 100-fold cross-validation eleven times, always using one of the machines for the master process and varying the number of machines that were allowed to be used as slaves. The resulting running times are shown in Figure 5.

As is to be expected, significant savings in time can be seen as more processors are added to the task. It should be noted that the NCV "zero slaves" experiment timed above is a slightly different experiment from the rest, in that the master and slave processes ran on the same machine. In all other experiments, each machine ran only a single process.

Using the "zero slaves" or the "one slave" experiment as a base case is not entirely fair, as both of these methods contain significant overhead associated with MPI. The base case for comparison should really be 100-fold cross-validation on a single processor, without any parallel programming overhead. Fortunately, Weka offers its own functionality to do cross validation itself on a single processor. We therefore ran Weka on the same dataset independently of NCV, instructing Weka to do 100-fold cross-validation. The amount of time Weka took to run this is shown as the horizontal line in Figure 5. On a single processor, instructing Weka to do the cross-validation runs faster than NCV. NCV, though, takes the lead once more processors are added.

What is the nature of this "overhead" that MPI adds? When NCV starts up, MPI needs to initiate a connection with each of the slave machines. This can take 5 or more seconds per machine. Next, NCV needs to transmit the dataset to each machine. Again, this takes time. It is clear that it only makes sense to use NCV in situations where it takes considerably longer to train than it does to transmit the data, or when the number of folds to be evaluated is large relative to the number of processors (and each fold does not run so quickly that job communication costs dominate). As each fold is completed on a given slave, that slave needs to contact the master process to send the results and request another job. When there are no more jobs to be done, each slave needs to finish its current job (it may be doing a job redundantly) and shut down its MPI connection. These communications add overhead as well.

NCV can be used to run many processes on a single machine, as was done in the "zero slave" case above. This technique becomes much more useful if one wishes to use NCV on a multiprocessor machine. The procgroup file described in Section 4 can be configured to allow a number of processes to run on a single machine. If that machine contains a number of processors, the operating system automatically allocates those processes to different processors. Running the NCV master and slaves entirely on a multiprocessor machine significantly reduces communication costs, as this information is now transferred internally and not across a much slower network connection. We ran similar experiments to the ones described above on our data mining server, which contains four Pentium Xeon 700 MHz processors and 4 gigabytes of memory. This server is also running Redhat Linux 6.2. Figure 6 shows the results of these experiments, allowing up to four slave processes to run simultaneously. Again, we can see noticeable speedup as we increase the number of processors on the task.

In comparing Figure 6 with Figure 5, we can see that the speeds are

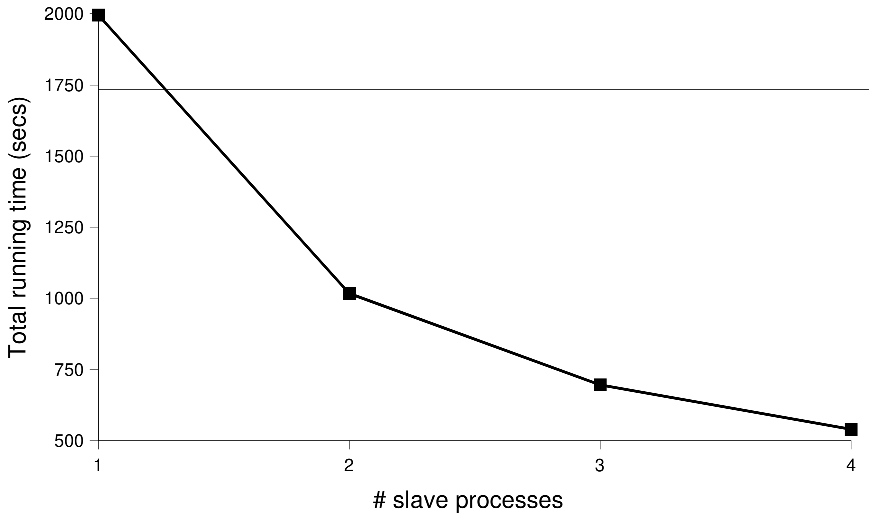

Figure 6: Speedup in parallel cross-validation on server with four 700 MHz Pentium III Xeon processors. The horizontal line indicates running time on a single processor without using the NCV environment.

noticeably faster in Figure 6. This is largely due to the fact that the server on which we ran these experiments is significantly faster than the machines on which we ran the earlier experiments. It is also true that the networked machines in Figure 5 experienced significantly higher latency and lower throughput between processes. One might therefore suggest that MPI on a shared memory multiprocessor machine is overkill. It would likely be faster yet to share memory between processes, rather than bundle up data in messages and pass them around. While this is true, we point out that maintaining the MPI architecture in both environments retains a level of abstraction that provides seamless integration between shared memory machines and networked machines. Additionally, such a system is much simpler to maintain and develop.

6 Conclusions

NCV is an easy to use, highly generalizable tool for performing n-fold cross-validation testing of machine learning algorithms. It scales extremely well, and is easily run on standard Linux machines. For future versions of the software, we hope to implement interfaces to MATLAB [16] and/or Mathematica [17], as well as provide better support for more platforms.

NCV may be obtained from any of the authors by email request.

Acknowledgements

We thank Mike Tie, Technical Associate at Carleton College for his assistance in appropriating and configuring the machines.

References

[1] Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1924, 1998.

[2] W. Gropp and E. Lusk. Installation guide to MPICH, a portable implementation of mpi. Technical Report ANL/MCS-TM-ANL-96/5, Argonne National Laboratory, 1996. Available at `ftp://ftp.mcs.anl.gov/pub/mpi/install.ps`.

[3] W. Gropp, E. Lusk, N. Doss, and T. Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. Technical Report ANL/MCS-P5670296, Argonne National Laboratory, February 1996. Available at `http://www-unix.mcs.anl.gov/mpi/mpich`.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, MA, USA, 1999.

[5] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. In *Proceedings of the Second Australian and New Zealand Conference on Intelligent Information Systems*, pages 357–361, Brisbane, Australia, 1994.

[6] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In C.S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, 1995.

[7] Ron Kohavi, Dan Sommerfield, and James Dougherty. Data mining using MLC++: A machine learning library in C++. In *Tools with Artificial Intelligence*, pages 234–245. IEEE Computer Society Press, 1996.

[8] M. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, June 1988. IEEE Compter Society Press.

[9] O. L. Mangasarian and David. R. Musicant. Lagrangian support vector machines. *Journal of Machine Learning Research*, 1:161–177, 2001. `http://www.jmlr.org`.

[10] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, 1997.

[11] P. M. Murphy and D. W. Aha. UCI repository of machine learning databases, 1992. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[12] D. R. Musicant. ASVM Software: active set support vector machine classification software, 2000. Available at `http://www.cs.wisc.edu/dmi/asvm`.

[13] Red Hat Inc. Red Hat Linux for Intel processors. Available at `http://www.redhat.com`.

[14] 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*, pages I:11–14, Oconomowoc, WI, 1995.

[15] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[16] The MathWorks, Inc. *Using MATLAB*. Natick, MA 01760, 1984-2001. Available at `http://www.mathworks.com`.

[17] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Reading, MA, USA, 1988.