# Applications in Parallel MATLAB

Brian Guilfoos, Judy Gardiner, Juan Carlos Chaves, John Nehrbass, Stanley Ahalt, Ashok Krishnamurthy, Jose Unpingco, Alan Chalker, Laura Humphrey, and Siddharth Samsi
*Ohio Supercomputer Center, Columbus, OH*
{guilfoos, judithg, jchaves, nehrbass, ahalt, ashok, unpingo, alanc, humphrey, samsi}@osc.edu

## 1. Introduction

The parallel MATLAB implementations used for this project are MatlabMPI[1] and pMatlab[1], both developed by Dr. Jeremy Kepner at MIT-LL. MatlabMPI is based on the Message Passing Interface standard, in which processes coordinate their work and communicate by passing messages among themselves. The pMATLAB library supports parallel array programming in MATLAB. The user program defines arrays that are distributed among the available processes. Although communication between processes is actually done through message passing, the details are hidden from the user.

## 2. Objective

The objective of this PET project was to develop parallel MATLAB code for selected algorithms that are of interest to the Department of Defense (DoD) Signal/Image Processing (SIP) community and to run the code on the HPCMP systems. The algorithms selected for parallel MATLAB implementation were a Support Vector Machine (SVM) classifier, Metropolis-Hastings Markov Chain Monte Carlo (MCMC) simulation, and Content-Based Image Compression (CBIC).

## 3. Methodology

### 3.1. SVM

SVMs[1] are used for classification, regression, and density estimation, and they have applications in SIP, machine learning, bioinformatics, and computational science. In a typical classification application, a set of labeled training data is used to train a classifier, which is then used to classify unlabeled test data. The training of an SVM classifier may be posed as an optimization problem. Typically, the training vectors are mapped to a higher-dimensional space using a kernel function, and the linear separating hyperplane with the maximal margin between classes is found in this space. Some typical choices for kernel functions are linear, polynomial, sigmoid, and Radial Basis Functions (RBF). SVM training involves optimizing over a number of parameters for the best performance and is usually done using a search procedure.

The chosen implementation of the SVM classifier is the "Sequential Minimal Optimization" algorithm.[7,8] SMO operates by iteratively reducing the problem to a single pair of points that may be solved linearly. The problem is parallelized by passing subsets to nodes in the cluster, and recombining the solved subsets to give a partially solved problem. After reaching the "tip" of the Cascade, the trained data set is passed back to the original nodes to test for convergence. This process is repeated until the support vectors have converged on the solution to the problem as defined by all training points satisfying the KKT conditions.

This deliverable demonstrates that toy datasets can be run through an SVM classifier implemented with the SMO algorithm with a Cascade parallelization approach[3] as shown in Figure 1, completely within native MATLAB and MatlabMPI.[5] An example output is shown in Figure 2. Figure 3 shows the median SVM algorithm execution time versus number of processors for five trials on a Pentium 4 cluster using 10000 samples in the checkerboard pattern of Figure 2. Execution is fast in all cases, with some possible benefit from added processors. However, it appears that communication overhead may start to outweigh this benefit as a larger number of processors are used. There are clear improvements that could be made to the codes (including tweaking the Cascade code to encourage faster convergence and adding implementations for alternative kernel functions), but this is an encouraging proof-of-concept that should scale reasonably well to larger problems.

### 3.2. MCMC

The MCMC algorithm is used to draw independent and identically distributed (IID) random variables from a

HPCMP Users Group Conference (HPCMP-UGC'06)
0-7695-2797-3/06 $20.00 © 2006 IEEE

distribution $\pi$ (x), which may be defined over a high-dimensional space. In many cases of interest, it is impossible to sample $\pi$ (x) directly. MCMC algorithms play an important role in image processing, machine learning, statistics, physics, optimization, and computational science. Applications include traditional Monte Carlo methods, such as integration, and generating IID random variables for simulations. MCMC also has numerous applications in the realm of Bayesian inference and learning, such as model parameter estimation, image segmentation, DNA sequence segmentation, and stock market forecasting, to name a few.

We have successfully implemented the MCMC algorithm in MATLAB, MatlabMPI, and pMATLAB. The algorithm lends itself naturally to parallel computing, so both the MatlabMPI and pMATLAB versions may be run efficiently on multiple processing nodes. The code is modular and designed to be customized by the user to simulate any desired probability distribution, including multivariate distributions.

The MCMC algorithm works by generating a sequence of variables or vectors $X^i$ using a Markov chain. The probability distribution of $X^i$ approaches some desired distribution $\pi(x)$ as $i \to \infty$. The update at each step is accomplished using the Metropolis-Hastings algorithm.[6] After some "burn-in" period, the current value of $X^i$ is taken as a sample of $\pi(x)$. The only requirement on $\pi(x)$ is that a weighting function, or unnormalized density function, may be calculated at each value of $x$.

The algorithm has two parts: 1) proposing a value for $X^{i+1}$ given the value of $X^i$, based on a proposal density function, and 2) accepting or rejecting the proposed value with some probability. The proposal density function in the general case takes the form $q(a,b)$, where $a$ is the value of $X^i$ and $b$ is the proposed value for $X^{i+1}$. In our implementation, proposals are restricted to the form $b = a + R$, where $R$ is a random variable with a probability density function $p(r)$. The function $p(r)$ need not be symmetric.

The proposed value of $X^{i+1}$ is accepted with probability $\alpha$, where $\alpha = \min\left(\frac{\pi(b)}{\pi(a)} \cdot \frac{p(-r)}{p(r)}, 1\right)$. If the value is not accepted, then $X^{i+1} = X^i$. Note that $\alpha$ is calculated using the unnormalized density function for the target density $\pi$ (x). If the proposal density is symmetric, $p(r) = p(-r)$ and the second factor disappears.

In our implementation, the user provides functions for the target and proposal density functions, as well as the number of burn-in iterations to use between samples, and the number of samples to be drawn. One of our goals was to make it easy for users to customize the code to generate other probability distributions, use different proposal densities, and change various parameters. Two examples are included with the code. The first example simulates samples from a bivariate normal distribution using a uniform proposal density. The second example simulates samples from a Rayleigh distribution using an asymmetric proposal density.

The problem of generating independent samples of a random variable is naturally parallel. Both of the parallel versions of the algorithm simply divide the number of samples to be generated among the available processors. Each processor generates its share of the total samples separately from the others. The samples are then combined into one output file. Figure 4 shows an example output. The left graph shows the desired distribution: a two-dimensional normal distribution with zero mean and a non-diagonal covariance matrix. The right graph shows 1000 samples generated by the MCMC routine using a uniform proposal density function. Figure 5 shows the median time over five trials to generate these 1000 samples using 1000 burn-in iterations between samples on a Pentium 4 cluster. The execution time decreases with diminishing returns as more processors are added.

### 3.3. CBIC

A typical image compression system applies the same compression strategy to the entire image, effectively spreading the quantization error uniformly over the image. This effectively limits the compression ratio to the maximum that can be tolerated for the important and relevant portions of the image. However, in many applications, the portion of the image that is of interest may be small compared to the entire image. For example, in a SAR image taken from an aircraft, only the portion of the image containing the target of interest needs to be preserved with high quality; the rest of the image can be compressed quite heavily. This leads to the idea of CBIC, in which selected portions of the image are compressed losslessly, while the rest of the image is compressed at a high ratio. In many applications, a sensor may be acquiring a stream of images, and compressing each rapidly for storage or transmission is essential. Parallel CBIC algorithms are useful in such situations.

The initial code for this project was taken from an incomplete copy of thesis work by a graduate student. The code, written in MATLAB, implemented serial wavelet compression[9] of a segmented image. Unfortunately, it was poorly written, with difficult to decipher function names, poorly thought out file I/O, and virtually zero comment lines. A significant effort was undertaken to document the existing source before parallelization could be attempted. Additionally, the code did not determine the areas of interest in the image to be compressed—it required a pre-existing mask on disk.

A very rudimentary serial image segmentation code was written to provide the mask for the compression step. The areas of interest are designated by pixels that are a certain percentage above or below the average brightness of the entire image. For most of our test images, this was surprisingly effective at picking out targets, such as tanks in a SAR image. A better option for a mask generator might be to use a trained, parallelized Support Vector Machine. Other options include a mask generator based on attention and perception, such as through symmetry.[2] Due to the modularity of the implementation, a new mask generator should be trivial to mate to the image compression routine. Once the mask is generated, it is written to disk in a PGM format. Due to the simplicity of this particular segmentation method, effort was not taken to parallelize it.

The parallelization efforts centered upon using pMatlab.[4] This implementation is limited due to the inherently serial nature of the pre-existing code, and not as efficient as it could be.

Testing on HPC systems was successful and Figure 6 shows that selective compression of an image was achieved. The left image is a sample SAR image showing two columns of tanks flanking a road. The right image shows the results of a parallel CBIC operation on the source image using the simple masking algorithm described earlier. Figure 6 shows that the masking algorithm successfully selected the tanks as objects of interest, as well as portions of the road and other objects that "stick out" from the background. The background portions of the image are then heavily compressed.

Figure 7 shows median execution times over five trials for the image shown in Figure 6 using a Pentium 4 cluster. A small benefit is seen by adding a second processor; however, execution times increase afterwards. More efficient code may produce better results.

## 4. Results

The authors now have three rudimentary applications that demonstrate that the algorithms can be implemented in parallel MATLAB. While the implementations may be limited (the SVM classifier, for example, only has one available kernel), they provide a proof-of-concept as well as a starting code base should a DoD user wish to apply any of these algorithms to their problem.

## Acknowledgements

This publication was made possible through support provided by DoD HPCMP PET activities through Mississippi State University under contract. The opinions expressed herein are those of the author(s) and do not necessarily reflect the views of the DoD or Mississippi State University.

## References

1. Burges, Christopher J.C., "A Tutorial on Support Vector Machines for Pattern Recognition." In *Data Mining and Knowledge Discovery*, volume 2, Kluwer Academic Publishers, Boston, MA, 1998.

2. Gesú, Vito Di and Cesare Valenti, "Detection of Regions of Interest Via the Pyramid Discrete Symmetry Transform." In *Advances in Computer Vision*, Springer, New York, 1997.

3. Graf, H.-P., E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel Support Vector Machines: The Cascade SVM." In *Advances in Neural Information Processing Systems*, volume 17, MIT Press, Cambridge, MA, 2005.

4. Kepner, Jeremy, "pMatlab: Parallel Matlab Toolbox." http://www.ll.mit.edu/pMatlab/, accessed May 19, 2006.

5. Kepner, Jeremy, "Parallel Programming with MatlabMPI." http://www.ll.mit.edu/MatlabMPI/, accessed May 19, 2006.

6. Byron J. T. Morgan, *Applied Stochastic Modelling*. New York: Oxford U. Press Inc., 2000, ch. 7.

7. Platt., J., "Fast Training of Support Vector Machines Using Sequential Minimal Optimization." In *Advances in Kernel Methods – Support Vector Learning*, MIT Press, Cambridge, MA, 1998.

8. Platt., J., "Using sparseness and analytic QP to speed training of support vector machines." In *Advances in Neural Information Processing Systems*, volume 13, MIT Press, Cambridge, MA, 1999.

9. Topiwala, Pankaj, ed., *Wavelet Image and Video Compression*, Kluwer Academic Publishers, Boston, MA, 1998.

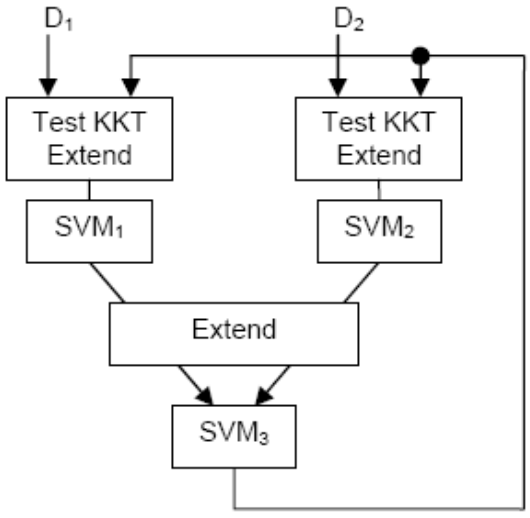


**Figure 1. Simple two-layer cascade SVM**

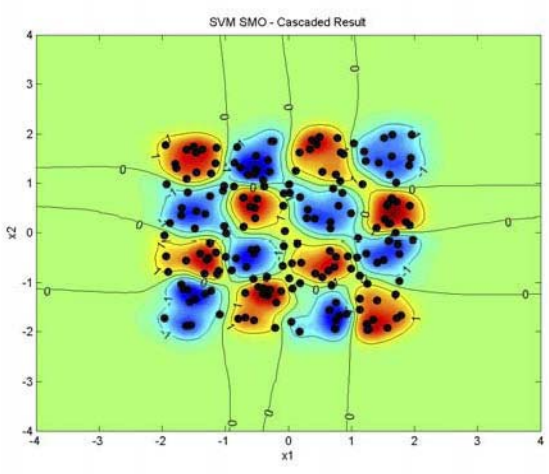


**Figure 2. Sample output of a trained SVM with training vectors in a checkerboard pattern**

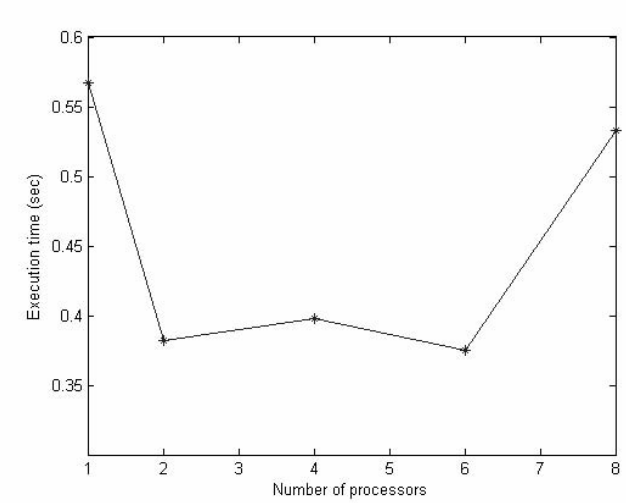


**Figure 3. SVM execution time in seconds versus number of processors for 10000 samples from the distribution shown in Figure 3**

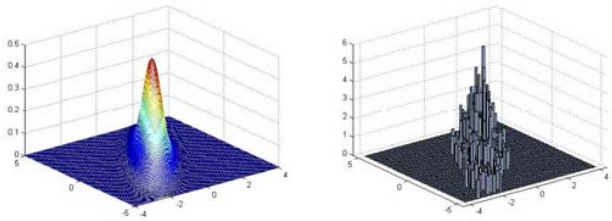

**Figure 4. Sample MCMC output for a 2-D normal distribution**

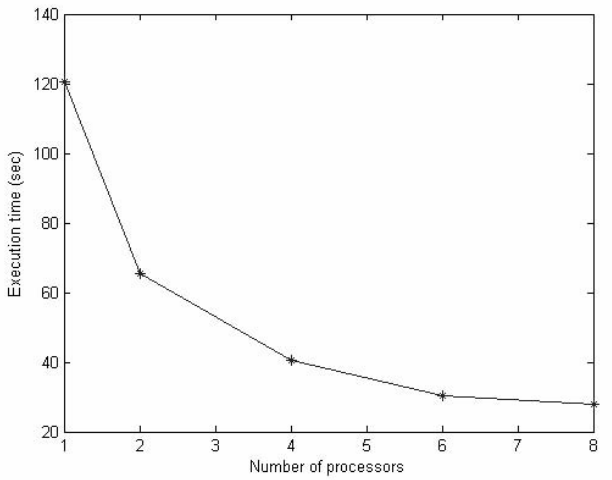


**Figure 5. MCMC median execution time in seconds versus number of processors for 1000 samples with 1000 burn-in iterations from the distribution shown in Figure 4**

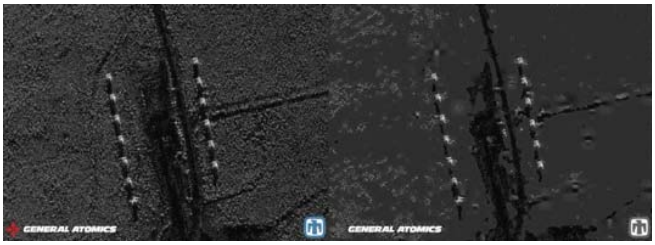


**Figure 6. CBIC compression of a sample SAR image**

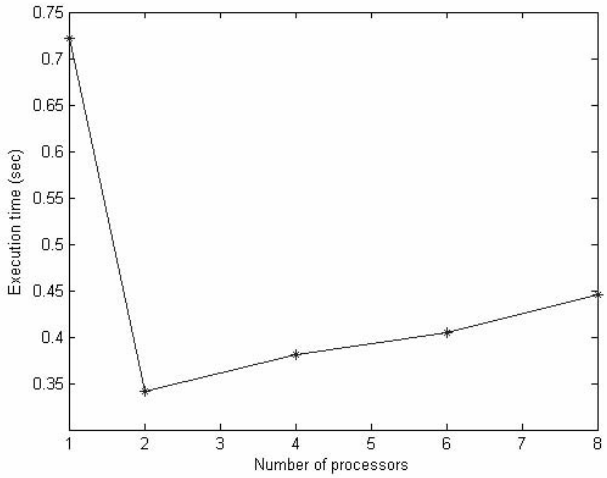


**Figure 7. CBIC median execution time in seconds versus number of processors for the picture shown in Figure 6**