A Parallel Algorithm for Optimal Margin Classifiers

Ross Baldick and Craig M. Chase

TR **TR-PDS-1995-005** **1995**

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

A Parallel Algorithm for Optimal Margin Classifiers*

Ross Baldick† Craig Chase‡

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Abstract

In a recent paper, Boser *et al.* describe an algorithm based on quadratic programming for training a classifier that maximizes the classification margin between dichotomous classes of patterns. Boser *et al.*'s algorithm can treat classification problems having parameter spaces of very high dimension because it solves the problem in the dual space. If the margin in the original problem is positive, that is, if there exists a hyperplane separating the two classes, then the dual problem is a convex quadratic program. The dual problem has dimension equal to the number of patterns.

Unfortunately, the number of patterns, while finite, is also very large. Consequently, Boser *et al.*'s algorithm requires considerable computational effort to execute. In this paper we propose a parallel implementation of Boser *et al.*'s algorithm. We demonstrate the implementation on an example database and analyze its performance. We also indicate extensions to other problems of similar structure.

1 Introduction

Boser *et al.* [1] discuss and formulate the problem of finding a classification hyperplane in parameter space that maximizes the classification margin between dichotomous classes, A and B, say, of patterns. The classification margin is the minimum L_2 distance in parameter space between the training patterns and the decision hyperplane. All patterns on one side of the hyperplane are classified as pattern A, while those on the other side are classified as pattern B. For various problems, this max-min criterion has better classification properties than other criteria such as finding the hyperplane that maximizes the sum of the squares of the distances between the patterns and the hyperplane. For a full discussion, see [1, §1].

Boser *et al.* formulate the classification problem as a constrained quadratic optimization with dimension equal to the dimension of the parameter space, and with number of constraints equal to the number of the patterns. Ideally, we would like to find a small number of parameters that, between them, can accurately classify the patterns. The best parameters for classification will, however, differ from application to application. Therefore, *a priori* we may want to explore a very large or infinite number of candidate parameters to find the small number that are best for classifying the particular patterns of interest.

Boser *et al.*'s insight is that, since the parameter space is very high or infinite dimensional, it is better to perform the constrained optimization in the dual space. In the dual space, the parameter space can be handled

*Submitted to the Seventh IEEE Symposium on Parallel and Distributed Processing

†Supported in part by an NSF Young Investigator Award under Grant ECS-9457133

‡Supported in part by NSF Grant 9409736 and by the Jack Kilby/Texas Instruments Endowed Faculty Fellowship in Computer Engineering

implicitly in a problem of dimension equal to the number of training patterns. If the margin in the original problem is positive, that is, if there exists a hyperplane separating the two classes, then the dual problem is a convex quadratic program [4], which can be solved with quadratic programming code. Let us refer to solving for the optimal hyperplane by using a quadratic programming algorithm applied in the dual space as the BGV algorithm.

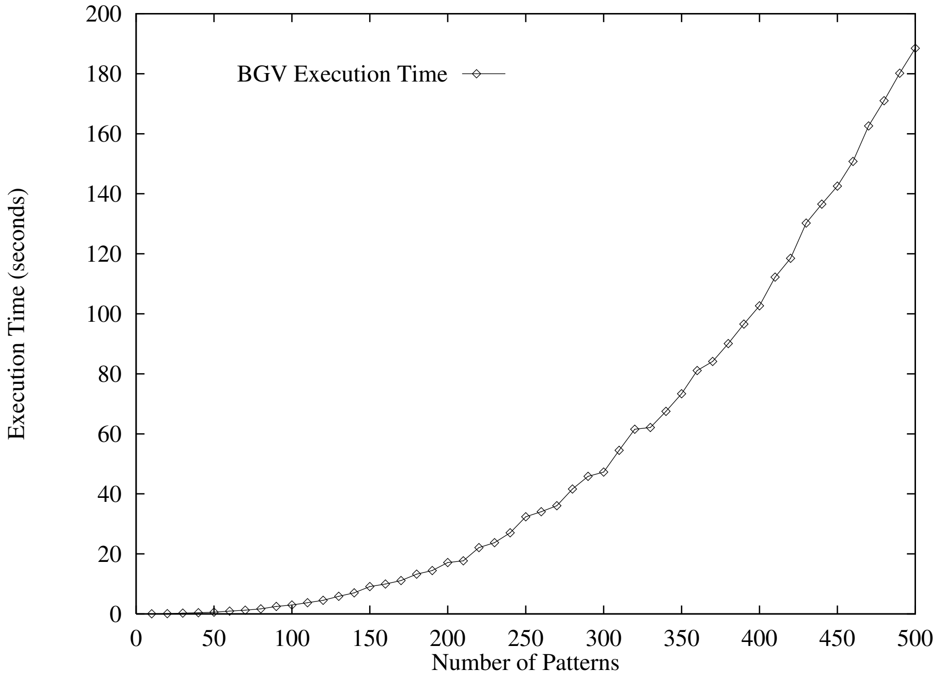

Figure 1: Empirical Estimation of Time Complexity for BGV (SparcStation 10).

Unfortunately, the number of patterns, while finite, is also very large so that applying the BGV algorithm directly to the complete database is unattractive. To see this, note that the major computational effort in the BGV algorithm is the solution of the quadratic programs. The fastest general convex quadratic programming algorithms have an asymptotic requirement of $O(P^3)$ operations, where P is the number of variables [3]. In the case of applying the BGV algorithm directly to the complete database, P is the number of patterns in the database. In our experimentation we use a quadratic programming code in the public domain and written by Nash [7]. Figure 1 shows the execution time for the BGV algorithm implemented with Nash's QP solver when run on a SparcStation 10 workstation. The execution time is $\Omega(P^{2.5})$ for the range of database sizes we analyzed. Although this analysis is far from definitive, it is clear that the computational requirements for a direct application of BGV to very large databases (many thousands or tens of thousands of samples) will be prohibitively expensive.

Instead of applying the BGV algorithm to the complete database, it is therefore promising to try to partition the database and attack only a portion of it at a time. Boser *et al.*'s approach to partitioning follows Vapnik [9] and involves dividing the database into chunks and processing the chunks iteratively using the BGV algorithm. Information from the successive processing of the chunks is used to 'filter' the chunks. After introducing some definitions, we will describe the filtering process in the following paragraphs.

A chunk C is simply a subset of the patterns. The BGV algorithm can be applied to the full database or to any chunk. Assuming that a separating hyperplane exists, the BGV algorithm returns the hyperplane H of

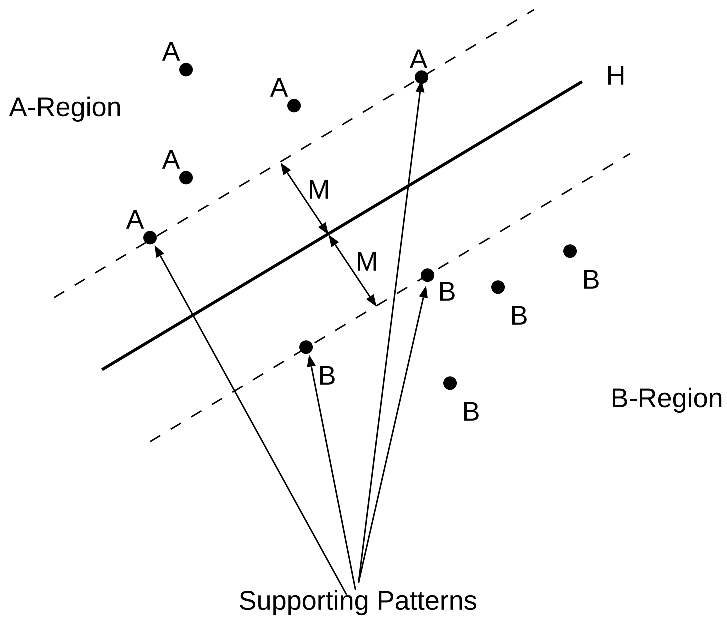

Figure 2: The Result of Applying BGV to a Set of Patterns.

maximum margin for the chunk and the corresponding margin M. This is illustrated in figure 2. Corresponding to H, there will be a subset of the patterns, S, called the supporting patterns, that lie exactly a distance M from H [1]. Naturally, there are some supporting patterns on both sides of H. All the patterns on one side are from class A, while all on the other side are from class B. Notice that the supporting patterns are the critical patterns that determine the separating hyperplane. The BGV algorithm also returns these supporting patterns. Typically, the number of supporting patterns is very small compared to P.

The supporting patterns in each class also define two parallel hyperplanes a distance M on either side of H. Let us refer to the region between these hyperplanes as the separating region defined by S. Now suppose we have a set of supporting patterns, S, from a previous application of the BGV algorithm and an arbitrary set of additional patterns C. We say that Q is the result of filtering C with the supporting patterns S, if Q is the subset of C consisting of:

- those patterns that lie in the separating region defined by S and
- any patterns that would be mis-classified by the separating hyperplane defined by S.

This is illustrated in figure 3.

In Boser's approach, the BGV algorithm is applied to the first chunk in the database and the supporting patterns, S, for this chunk are obtained. The first *two* chunks are then filtered with these supporting patterns, producing a new set Q. The algorithm then iterates by finding the next set of supporting patterns by applying the BGV algorithm to the union of Q and S. At the $i+1$-st iteration, the supporting patterns obtained from applying the BGV algorithm at the i-th iteration are used to filter the union of chunks 1 through $i+1$. After each filtering stage, BGV is used to find the next set of supporting patterns.

The algorithm continues until all the chunks have been processed. Finally, one or more additional iterations must be performed until the filtering operation yields no additional patterns. At this point, the algorithm has converged to the optimal separating hyperplane.

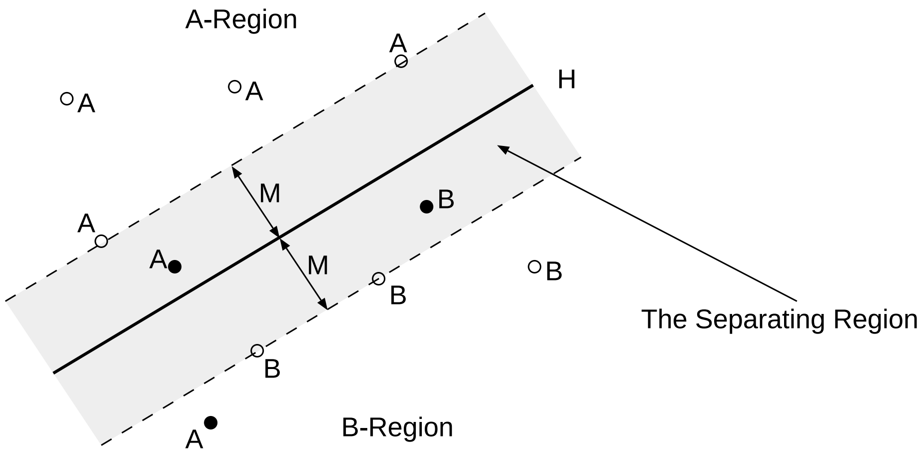

Figure 3: Filtering Patterns: Solid points are the result of filtering.

The intuition behind the chunking approach is as follows. Let $\hat{S}$ be the true set of supporting patterns. The set $\hat{S}$ can be found directly by applying BGV to the complete set of patterns, but at great computational expense. Note that by the definition of a supporting pattern, if $\hat{S}$ is used to filter any part of the database, no patterns (other than those in $\hat{S}$) will be found in the separating region. Furthermore, using any $S \neq \hat{S}$ to filter the entire database yields at least one pattern in the separating region (or misclassified).

Note that if a set S is a poor approximation to $\hat{S}$, then filtering the entire database with S will result in a large number of patterns in the separating region. These patterns will, in turn, require considerable time to process with the BGV algorithm. Boser's chunking strategy is based on the reasonable assumption that distribution of patterns in the dual space is uniform. The first filter that is obtained may be a poor approximation to $\hat{S}$, and is only used to filter the first two chunks. Successive filters are likely to be closer to $\hat{S}$ and are thus used on larger and larger segments of the database.

Figure 4 shows the technique used in [1].

In the next section we describe a parallel algorithm based on the chunking technique. In Section 3 we report on our experimental results with this algorithm. (Details on the experimental method are contained in Appendix A). In Section 4 we give a short description of application of our method to other optimization problems, and Section 5 concludes the paper.

2 Parallel Algorithm

There are two insights for our improved algorithm. First, we note that as a filter becomes a better approximation to $\hat{S}$, fewer patterns will lie in the separating region. We should therefore filter larger and larger chunks on successive iterations. Hence, in Step 1 of Figure 4, the chunks should be selected such that, for example:

$$|C_k| = \sum_{j<k} |C_j|$$

Based on this insight, the number of applications of the BGV algorithm in steps 1–5 can be reduced from linear in the number of patterns to logarithmic, without significantly increasing the number of iterations performed during steps 6 and 7. Qualitatively, we obtained noticeable improvement in computation time com-

Given: A database of patterns and an integer $k \geq 1$.

1 Divide the database randomly into k chunks, $C_1, \ldots, C_k$.

2 Apply BGV to chunk C_1, let S be the supporting patterns for this chunk
let j = 2

3 Use S to filter chunks $C_1, \ldots, C_j$, yielding set Q

4 Let the next S be the supporting patterns for $S \cup Q$, let j = j + 1

5 if $j \leq k$ repeat from step 3

6 Use S to filter the entire database, yielding set Q

7 if $Q = \emptyset$, then exit
otherwise apply BGV to $S \cup Q$ to produce the new S, and
repeat from step 6.

Figure 4: Boser *et al.*'s Method for Finding $\hat{S}$.

pared to using constant size chunks.[1]

Our second insight is that both filtering and the use of BGV to find supporting patterns can be parallelized. Filtering parallelizes perfectly, by simply assigning an equal number of patterns to each processor and broadcasting the filter, S. Although it is difficult to efficiently parallelize the quadratic programming problem, parallelism can still be applied to finding the supporting patterns using a combining tree as follows. Each processor begins with a different set of (already filtered) patterns. Each processor applies the BGV algorithm to their set. Every odd/even pair of processors then combine their supporting patterns into a new set, and BGV is applied to the result. Then pairs of pairs combine supporting patterns, and BGV is again applied. In $\log N$ stages, the root of the tree is reached, and the final set of supporting patterns is broadcast to all processors to be used as the next filter.

Our parallel algorithm is shown in Figure 5. The algorithm initially uses a variation of the optimized serial algorithm on every processor. Locally, the complete chunk is filtered using a sequence of alternating BGV steps and filtering steps. No interprocessor communication is required for this stage.

The second phase of the parallel algorithm involves an iterative procedure in which all processors cooperate. At each iteration, a combining tree is used to find a global set of supporting patterns. This global set is added to all chunks and then used to filter (in parallel) the chunk on every processor. BGV is applied to the resulting patterns to produce a new local set of supporting patterns.

Termination is guaranteed by adding the global supporting patterns, S, to every processors' chunk. This activity ensures that the processors can not endlessly cycle with any fixed sequence of sets of supporting patterns. In the theoretical worst case, eventually every pattern from the database will be in every processor's chunk, and the parallel algorithm reduces to N replicas of Boser's method. In practice, termination in the parallel algorithm occurs after relatively few global iterations (typically on the same order as the number of iterations required by the last two steps of the serial algorithm).

[1]The emphasis of this paper is on parallelizing the classifier algorithm, so we have not presented the results of this improvement to the serial algorithm. Comparison of the two versions of the serial algorithm are available from the authors.

Given A database of patterns and N processors

1 Divide the database into N equal sized chunks, $C_1, \ldots, C_N$

2 on each processor, P_i do:
Use steps 1 through 5 of Boser *et al.*'s approach on C_i (see Figure 4) to find supporting patterns S_i

3 Globally combine all S_i using a combining tree.
BGV is used at each level of the tree to combine two sets of supporting patterns. Let S be the global result of combining.
Broadcast S to all processors.

4 on each processor, P_i do:
let $C_i = C_i \cup S$
Use S to filter C_i, yielding Q_i.

5 If $\forall i : Q_i = \emptyset$ then exit,
otherwise on each processor use BGV on $S \cup Q_i$ to produce a new S_i
repeat from step 3.

Figure 5: Parallel implementation of maximum margin algorithm.

3 Computational Results

We generated a synthetic database that represented the classification of hand-written zeros and ones. We started with 16×16 bitmaps of four examples of zeros and four examples of ones. We randomly altered the bitmaps to simulate degradation of the data. We used a 'linear' classifier in the BGV algorithm, that is, with kernel $K(x, y) = x^t y$ [1, §4].

While this database is not derived from actual hand-writing examples and uses a relatively small parameter space, we believe that the computational results are representative of the results with a non-synthetic database and a larger parameter space. This is because the computational effort is mostly determined by the the number of patterns and the size of the quadratic problems and associated calculations, not by the size of the parameter space.

Our implementation of both the serial and parallel algorithms is in C++ and compiled with the GNU gcc compiler. We use a quadratic programming package in the public domain that was written by Stephen Nash [7]. This package is implemented in Fortran, and was compiled by translation into C by f2c, and then further processed and optimized by gcc. Finally, PVM version 3.3.7 is used for interprocessor communication in the parallel version. More details on the implementation and the experimentation can be found in Appendix A.

3.1 Experimental Results

Importantly, the number of iterations required for convergence did not vary significantly from the parallel version to the serial version. Typically, 4 or 5 iterations were required by both versions of the algorithm once the entire database had been filtered. This value did not change even as the number of processes was varied over the range of 1 to 32. It is difficult to assess how much the convergence behavior we observed was a result of the use of a synthetic database. However, unless the parallel algorithm required significantly more iterations to converge than the serial algorithm, the speedup results we obtained would not be affected.

As we expected, and consistent with Boser *et al.*'s observations, the number of supporting patterns produced by BGV tended to vary little with changes in database size. However, the number of supporting pat-

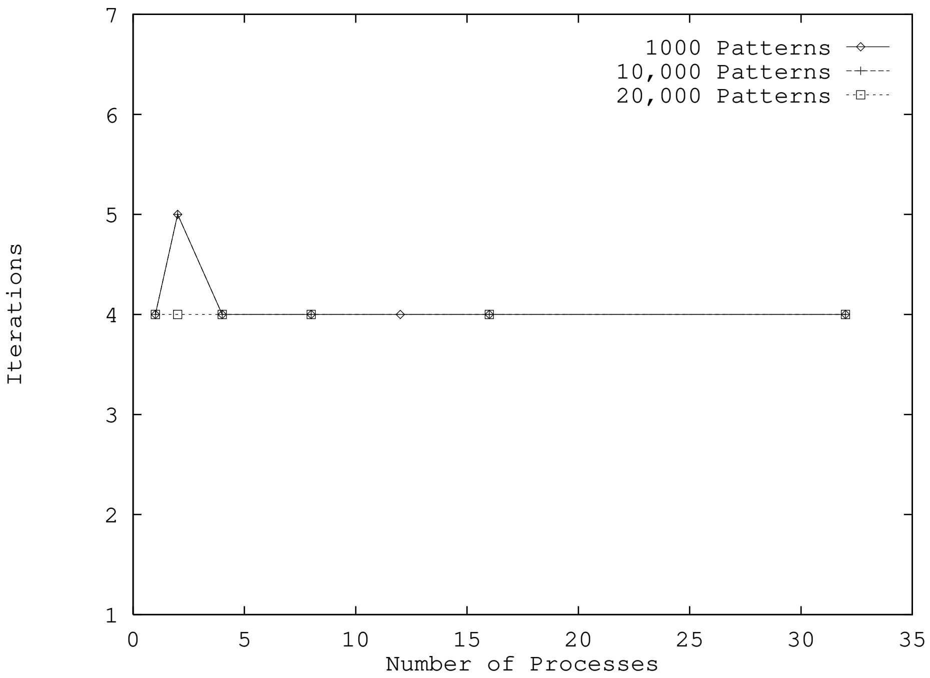

Figure 6: Iterations Required for Convergence.

terns did vary with the amount of degradation of the patterns in the database itself. By increasing the amount of random noise introduced into the patterns, the number of supporting patterns for the database is increased. With a small degree of degradation, typically 25–35 supporting patterns were discovered in any database containing more then 500 patterns (smaller databases contain fewer supporting patterns). This value is consistent with the numbers obtained in [1] for a database of actual handwriting samples (the DB1 database of "clean" samples, [1, 4]). With more degradation we observed approximately 50–75 supporting patterns. With very large amounts of degradation, no separating hyperplane could be found. From the perspective of practical applications of the algorithm, the most interesting situation is the largest amount of degradation that is consistent with the existence of a separating hyperplane.

3.2 Parallel Speedup

Execution speed for the sequential algorithm is determined predominately by the rate at which filtering can be performed. With the optimized chunking strategy, execution time increases approximately linearly with the size of the database. The parallel algorithm must, in addition, perform a global combining operation with supporting patterns. For small numbers of processors (or very large database sizes), the cost of this combining is insignificant with respect to the cost of filtering, and good speedup is obtained. Figure 7 shows the execution time as a function of database size. Each curve in the figure shows the results obtained with a different number of processors. In practical applications, even larger numbers of patterns would be encountered.

As intuition would suggest, the parallel efficiency increases as the database size is increased. Figure 8 shows the parallel speedup obtained with our algorithm. In this figure, each curve represents a particular database size.

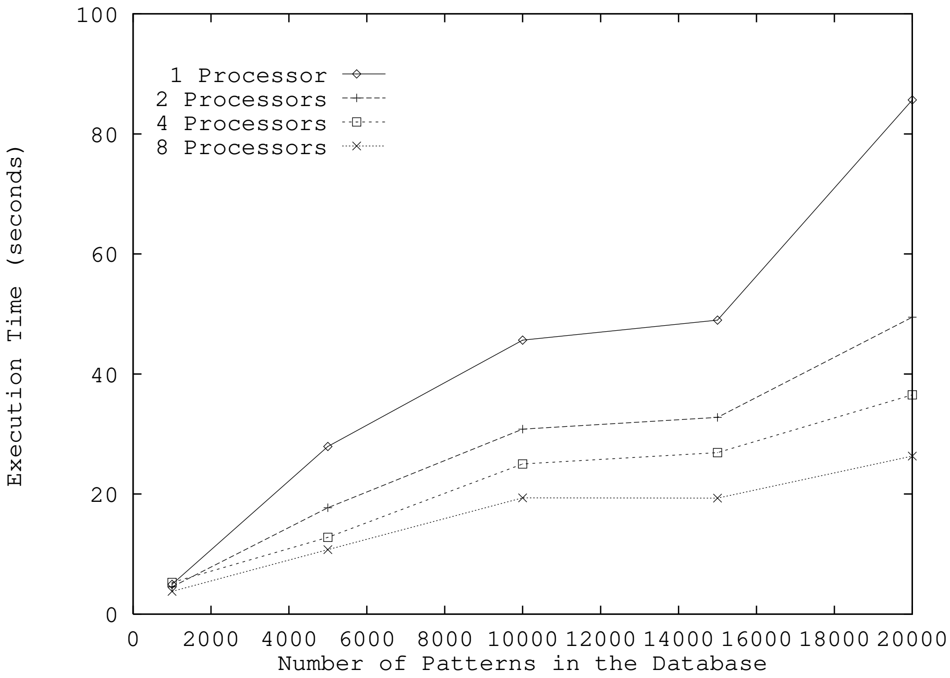

Figure 7: Execution Time as a Function of Database Size.

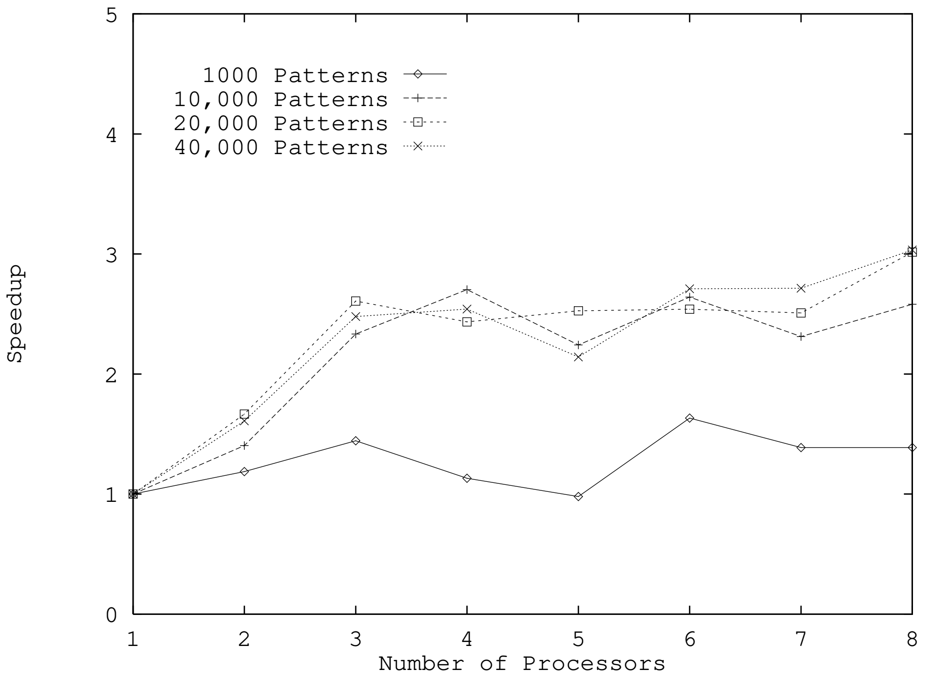

Figure 8: Speedup as a Function of Number of Processors.

4 Generalization to Other Optimization Problems

We have proposed a parallel implementation of the maximum margin classification algorithm. The parallel implementation promises good efficiency for large databases. Other similar problem structures may also be parallelized with this approach. For example, consider the general non-linear optimization problem:

$$\min_{x \in X}\{f(x) : g(x) \leq \mathbf{0}\},$$

where $f : X \Rightarrow \mathrm{R}$ and $g : X \Rightarrow \mathrm{R}^p$ are non-linear functions on the space X. This problem structure is encountered in many applications.

A particularly difficult version of the general non-linear problem occurs in the 'security-constrained optimal power flow' problem in electric power systems [6, 8, 5]. The parallelization of this problem is particularly interesting since it is desirable to solve it in real-time. The dimension of x can be on the order of several thousands in the security-constrained optimal power flow problem and the dimension of g can be extremely large, on the order of tens or hundreds of thousands. Moreover, only a very small proportion of the constraints are binding and it is sometimes possible to identify *a priori* subsets of the constraints that are likely to be binding at the solution. The structure of this problem parallels the structure of the maximum margin problem, but is much more difficult because of the general non-linear objective and constraints.

A parallel algorithm to solve this problem is outlined in figure 9. First, partition the vector function g into $g = (g_i)_{i=1,\ldots,k}$, where each g_i is now itself a vector function $g_i = (g_{i\ell})_{\ell=1,\ldots,p_i}$. To explain the algorithm, we define a filter operation on the constraints. Given a candidate value of the solution $\tilde{x}$, define the filtering of a set of constraints to be those constraints that are violated or satisfied with equality by $\tilde{x}$.

As in the pattern classification algorithm, we first run the optimization on chunks of the constraints. We then identify the binding constraints from all these problems and combine them to form a set T. Let the optimum of the problem subject to just the constraints T be $\tilde{x}$. Then we filter each chunk with $\tilde{x}$.

5 Conclusion

The contributions of this paper are as follows. We present the concept of "Filtering" in constrained optimization problems. We use this notion to clarify earlier work in this area on sequential algorithms and as the basis of a novel parallel algorithm.

The parallelization is akin to Benders decomposition [2] in that information from one problem is used to generate constraint cuts in another problem. It differs from Benders decomposition in that all problems are at the same level. In the particular case of the optimal margin classifier, the optimization sub-problems are solved with a quadratic programming algorithm. The sub-problems can, however, be solved by any suitable algorithm.

We apply our methodology to finding an optimal margin classifier between dichotomous classes of patterns. Our experimental results show that speedup is possible with relatively large databases and small numbers of processors. The high cost of performing additional quadratic programming sub problems in the parallel algorithm limits the speedup achieved with large number of processors.

Acknowledgment

We would like to thank Bernhard Boser of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, for help and discussions during the early part of this work. We would also like to thank Isabelle Guyon of AT&T Bell Laboratories for discussions during the course of this work. We

Given: Integer k, a function f, and a vector function g, with g partitioned into $g = (g_i)_{i=1,\ldots,k}$.

1 For each $i = 1, \ldots, k$, solve

$$\min_{x \in X} \{f(x) : g_i(x) \leq \mathbf{0}\}$$

to obtain x^i.

2 Collect the indices of the binding constraints. That is, for each $i = 1, \ldots, k$ form the set:

$$L(i) = \{\ell : g_{i\ell}(x^i) = 0\}.$$

3 Let $T = \{(i, \ell) : \ell \in L(i), i = 1, \ldots k\}$ be the set of all the binding constraints in all the sub-problems and solve

$$\min_{x \in X} \{f(x) : g_{j\ell}(x) \leq 0, \forall (j, \ell) \in T\}$$

to obtain $\tilde{x}$ with binding constraints S.

4 Use $\tilde{x}$ to filter the constraints in each sub-vector. That is, for each $i = 1, \ldots, k$, find the set: $Q(i) = \{(i, \ell) : g_{i\ell}(\tilde{x}) \geq 0\} \cup S$.

5 For each $i = 1, \ldots, k$, solve

$$\min_{x \in X} \{f(x) : g_{j\ell}(x) \leq \mathbf{0}, \forall (j, \ell) \in Q(i)\}.$$

to obtain x^i and the set of binding constraints

$$L(i) = \{(j, \ell) \in Q(i) : g_{j\ell}(x^i) = 0\}.$$

6 If the binding constraints are unchanged from the previous iteration, then exit.

7 Goto step 3.

Figure 9: Parallel implementation of general convex optimization algorithm.

appreciate the contributions of Md. Sayeed Akmal of the University of Texas at Austin to the development of the programs used during the experimentation.

References

[1] Bernhard Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory COLT*. ACM Press, July 1992.

[2] A. M. Geoffrion. Generalized Benders decomposition. *Journal of Optimization Theory and Applications*, 10(4):237–260, 1972.

[3] Clovis C. Gonzaga. Path-following methods in linear programming. *SIAM Review*, 34(2):167–224, June 1992.

[4] David G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1984.

[5] W. C. Merritt, C. H. Saylor, R. C. Burchett, and H. H. Happ. Security constrained optimization–a case study. *IEEE Transactions on Power Systems*, 3(3):970–977, August 1988.

[6] A. Monticelli, M. V. F. Pereira, and S. Granville. Security-constrained optimal power flow with post-contingency corrective scheduling. *IEEE Transactions on Power Systems*, PWRS-2(1):175–182, February 1987.

[7] Stephen Nash. Newton-type minimization via the lanczos method. *SIAM Journal of Numerical Analysis*, 21:770–778, 1984.

[8] B. Stott, O. Alsac, and A. J. Monticelli. Security analysis and optimization. *Proceedings of the IEEE*, 75(12):1623–1644, December 1987.

[9] Vladimir Vapnik. *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, 1982.

A Experimental Details

A.1 Optimization and Tuning

All experimental results are obtained using gcc with default optimizations enabled (the -O flag). As we are concerned solely with the performance of the classification algorithm, we ignore input/output costs in our analysis. The parallel implementation shares roughly 90% of its code with the serial implementation (the QP code, the BGV implementation, and the code to manipulate patterns). Profiling was used extensively to tune both implementations.

Applying BGV directly to an entire database results in the bulk of the computation spent evaluating the cost function ((17) in [1]), which requires a vector-matrix multiplication (double precision arithmetic is used). The vector contains as many elements as there are patterns in the database and the matrix is square.

Chunking and filtering ensures that BGV is applied only to small subsets of the complete database. In both the optimized serial algorithm, and the parallel algorithm, the bulk of the computation is thus spent performing filtering rather than during the BGV evaluation. Filtering a chunk C with filter S requires computing the kernel $K(x, y)$ for every element $x \in S$ and every pattern $y \in C$. With the linear classifier, this function is the dot product of two 256-element bit vectors.

The dot-product calculation is implemented (as an overloaded operator applied to two instances of the "Pattern" class) in C++ as follows: Each 256-bit pattern is encoded in 8 32-bit integers. Bit-wise AND of the corresponding integers of the two patterns is equivalent to performing 32 simultaneous multiplications (but is significantly faster). Eight of these bit-wise AND operations are performed. Finally, a ones-count is performed to yield a scalar result by repeated shifting and adding. Even optimized in this way (which reduced the execution time by almost two orders of magnitude from our original naive implementation), the dot product computation still accounts for over 80% of the execution time reported in our results.

A.2 General Experimental Method

The experiments were performed on a heterogeneous network of workstations. The machines consisted of three SparcStation 10 and two SparcStation 20 machines each running SunOS 4.1.3 in multi-user mode, and three 90MHz Pentium-based workstations each running NetBSD-1.0 (a BSD4.4-lite derived unix variant) in multi-user mode. Data format conversion was performed transparently to our program by PVM. The machines were on three different subnetworks of the same local area network. Routing time between these networks was determined to be insignificant with respect to the software overhead of PVM (which in turn was insignificant with respect to the computational requirements of our application).

No measures were taken to compensate for the differences in processing speed of these machines. However, as the parallel algorithm is tightly synchronized, the execution rate is essentially limited to that of the slowest machine (the SparcStation 10 in our case). Hence, our results are indicative of what one might expect on a homogeneous collection of SparcStation 10 machines.[2] The results from the serial version of the algorithm are collected on a SparcStation 10 for a more representative comparison. Similarly, each invocation of the parallel algorithm involved at least one SparcStation 10.

Each data point represents the mean result of 5 executions of the program.

[2] The parallel algorithm can be easily optimized for heterogeneous environments by adjusting the number of patterns assigned to each processor according to the relative processing rate.