Parallelizing the Large-Width learning algorithm

Joel Ratsaby†, Alon Sabaty‡

Department of Electrical and Electronics Engineering
Ariel University
Ariel, Israel
ratsaby@ariel.ac.il†, alonsb90@gmail.com‡

Abstract—We introduce a new parallel algorithm that implements the Large-Width (LW) learning algorithm [3]. The LW algorithm is an instance-based learning procedure which produces a multi-category classifier defined on any distance space, with the property that the classifier has a large sample width (which is similar to the notion of large margin learning). Being instance-based, the LW algorithm spends a majority of the time computing pairwise distances between examples (instances). The parallel version introduced here takes advantage of this fact and processes these computations in parallel. We present pseudo-code and estimate the speedup factor relative to the sequential LW algorithm.

Index Terms—Machine learning, classification, parallel algorithm, big data

I. INTRODUCTION

With the ever growing amounts of data and the expanding variety of domains on which machine learning is applied, the ability of learning over non-Euclidean input spaces becomes more important. This poses a challenge to existing machine learning technology which relies primarily on algorithms that need numerical training data which is structured into predefined attributes that measure different features of data instances. Often a problem domain is a space that can be equipped with a dissimilarity or distance function in which case it is referred to as a distance space. In contrast to a metric, a distance function does not need to satisfy the triangle inequality which makes it be applicable to a richer variety of problem domains. There are many existing distance functions [5] and new ones can be defined easily for any kind of data, for instance, bioinformatic sequences, graphs, images, etc..

The LW algorithm [3] learns multi-category classification over a distance space. It produces a classifier which has a large width on the training sample. The concept of width was introduced by [2] and expanded in various settings (see references in [3]). It is analogous to the ‘margin’ idea (see, for instance [1], [6]) and can be used to obtain sample-dependent error bounds on learning classification. While both width and margin functions represent a form of confidence in classification, width functions are not based on any real-valued function (in contrast to the notion of margin) but instead are always based specifically on functions that measure the distance between a point and some set of points that are labeled oppositely. Learning classification with large width can yield tighter error bounds and therefore more efficient learning (smaller sample sizes).

The current paper introduces a parallel version of the LW algorithm.

II. OVERVIEW OF THE SEQUENTIAL LW ALGORITHM

Let the distance space be denoted \( \mathcal{X} \) and let \( d(x, x') \) denote the distance between points \( x \) and \( x' \). A labeled sample, \( \xi = \{z_i\}_{i=1}^n \) with \( z_i := (x_i, y_i) \), is a sequence of points of \( \mathcal{X} \) together with labels in a set \( \mathcal{Y} = \{1, \ldots, M\} \) (for some fixed integer \( M \)). We call such a labeled point \( z = (x, y) \) an example; and we denote by \( x(z) \) and \( y(z) \) the \( x \) and \( y \) components of \( z \). We will slightly abuse the notation and for any two points, \( z, z' \) in \( \mathcal{X} \times \mathcal{Y} \) we also write \( d(z, z') \) to mean \( d(x(z), x(z')) \). Denote by \( U \) an initial set of unlabeled points in \( \mathcal{X} \) which are to be classified. The LW algorithm classifies this set incrementally. For positive integer \( t \), denote by \( L_t \) the set of unlabeled points at time \( t \), then this set decreases in size by one as time \( t \) increases by one. Denote by \( L_t \) the set of points which have been classified up to time \( t \). We refer to any point in the set \( \xi \cup L_t \) of labeled points or examples as a prototype. For any example \( z \) define

\[
NU(z) = \min_{p \in L_t \cup \xi : y(p) \neq y(z)} d(x(p), x(z)).
\]

It is either a labeled point in \( L_t \) or a labeled example in \( \xi \) which is closest to \( x(z) \) and whose label differs from \( y(z) \). Let the \( NU-ball \) centered at a labeled point \( z \) be the set of all points \( p \) (not just labeled ones) such that \( d(z, p) \leq d(z, NU(z)) \). For an unlabeled point \( p \) and any \( k \in \mathcal{Y} \), define the vote-set \( V_k(p) \subseteq \xi \) to be the following subset of the sample \( \xi \):

\[
V_k(p) := \{ z \in \xi : y(z) = k, d(p, z) < d(NU(z), z) \}.
\]

This is the set of examples in \( \xi \) of category \( k \) whose \( NU-balls \) contain \( p \). Given an unlabeled point \( x \), the LW algorithm classifies \( x \) with the label \( k \) such that the size of the set \( V_k(x) > V_j(x) \) for all \( j \neq k \). If there is no single label that maximizes \( V_k(x) \) over \( k \) then the algorithm uses a slightly different rule (see [3] for details). Once an unlabeled point \( x \) is assigned a label, it becomes a prototype and is used in the next iteration to classify another unlabeled point. The algorithm continues in this manner until all unlabeled points in \( U \) are assigned labels.
The next section introduces a parallel version of the LW algorithm which we denote by Algorithm LWP.

III. ALGORITHM LWP

Let \( m \) and \( n \) denote the size of the sample \( \xi \) and the set \( U \) of unlabeled points to be classified. Both \( m \) and \( n \) are finite so we represent \( \xi \) and \( U \) as sets of natural numbers,
\[
U := \{1, \ldots, n\} \quad \text{and} \quad \xi := \{n+1, \ldots, n+m\}.
\]
Algorithm LWP’s main part is Algorithm 1. It calls several procedures which are listed as Procedure 1 – Procedure 12. We use the terminology which is based on nVIDIA’s parallel computing architecture which is based on blocks of executable threads. The statement ‘Launch parallel blocks’ refers to an operation which deploys several blocks of threads for execution. We use a for-loop but instead write the code to be executed by each thread, in parallel. We write ‘synchronize all blocks’ to mean deploying multiple threads to run on a single block. Here we do not use a for-loop but instead write the code to be executed by each thread, in parallel. We write ‘synchronize all threads’ for the operation to wait for all threads to terminate and we write ‘synchronize all blocks’ to wait for all blocks to finish execution.

Algorithm 1 performs parallel computations for the following: to compute all pairwise distances between examples and unlabeled points (step 4), to compute the distance \( d(z, \text{NUN}(z)) \) for every example \( z \in \xi \) (step 8), to compute the size of all votesets of every unlabeled point (step 13), to compute the size of the largest and second largest votesets for every unlabeled point (step 16), to compare the size of votesets of a single point that is to be classified (step 18), and to adapt the \( \text{NUN}(z) \) for every example \( z \in \xi \) (step 21). We write MAX to denote the largest number representable in the computing platform.

Algorithm 1  
\( \text{LWP}(U, \xi) \)

\textbf{Input:} a set \( U \) of unlabeled points to be classified \( U = \{p_1, p_2, \ldots, p_n\} \), \( U[i] = p_i \in \{1, \ldots, n\}, 1 \leq i \leq n \), a set of labeled examples \( \xi = \{z_1, z_2, \ldots, z_m\} \), \( z_i = (x_i, y_i) \), where \( x_i \in \{n+1, \ldots, n+m\}, y_i \in \{1, \ldots, M\}, 1 \leq i \leq m \).

\textbf{Output:} Classification labels for all points in \( U \)

\textbf{Declare} \hspace{1em} \text{// global variables (common to all procedures)}

\begin{itemize}
  \item \( L := [l_1, l_2, \ldots, l_n], l_i \in \{1, \ldots, M\} \) where \( l_i \) is the classification value assigned to point \( p_i \).
  \item \( r := [r_1, r_2, \ldots, r_n] \) is an indicator vector, \( r_i = 1 \) if \( p_i \) is already classified otherwise \( r_i = 0 \). The entries of \( r \) are initialized to zero.
  \item \( D := [d[i, j]] \) is \( m \times n \) matrix where \( d[i, j] \) is the distance between example \( z_i \) and point \( p_j \). Denote by \( D_k \) the \( k \)th row of \( D \) and by \( D^{(j)} \) the \( j \)th column of \( D \).
  \item \( d_{\text{NUN}} := [d^{(1)}_{\text{NUN}}, \ldots, d^{(m)}_{\text{NUN}}] \), where \( d_i^{\text{NUN}} \) is the distance from \( z_i \) to its closest prototype whose label differs from \( y(z_i) \).
  \item \( V := [v[i, j]] \) is an \( M \times n \) matrix where \( v[k, p] \) holds the size of Voteset \( V_k(p), k \in \{1, \ldots, M\} \). Denote by \( V_k \) the \( k \)th row of \( V \) and \( V^{(j)} \) the \( j \)th column of \( V \).
  \item \( a := [a_1, a_2, \ldots, a_n], a_i \) is size of largest Voteset of point \( p_i \).
  \item \( b := [b_1, b_2, \ldots, b_n], b_i \) is size of the second largest Voteset of point \( p_i \).
  \item \( u := [u_1, u_2, \ldots, u_n], u_i \in \{1, \ldots, M\} \) is index (row number of matrix \( V \)) of the largest Voteset of point \( p_i \).
  \item \( v := [v_1, v_2, \ldots, v_n], v_i \in \{1, \ldots, M\} \) is index of the second largest Voteset of point \( p_i \).
\end{itemize}

1: \hspace{1em} // Build distance matrix \( D \)
2: \hspace{2em} \textbf{Launch} parallel blocks \( B_q, 1 \leq q \leq m \) // one block per example \( z \in \xi \)
3: \hspace{2em} \textbf{for} all \( 1 \leq q \leq m \) \hspace{1em} // one block per example \( z \in \xi \)
4: \hspace{2em} \text{D}_q := \text{compDistFromEx}(z_q) \hspace{1em} // Block \( B_q \) executes this procedure, \( \text{D}_q \) is \( 1 \times n \) vector
5: \hspace{2em} \textbf{end for}
6: \hspace{2em} \textbf{synchronize} all blocks \( B_q, 1 \leq q \leq m \).
7: \hspace{2em} \textbf{Launch} parallel blocks \( B_q, 1 \leq q \leq m \) // one block per example \( z \in \xi \)
8: \hspace{2em} // Initialize vector of minimum distances
9: \hspace{2em} \textbf{for} all \( 1 \leq q \leq m \) \hspace{1em} // one block per example \( z \in \xi \)
10: \hspace{2em} \text{d}_q^{\text{NUN}} = \text{initDmnum}(z_q)
11: \hspace{2em} \textbf{end for}
12: \hspace{2em} \textbf{synchronize} all blocks \( B_q, 1 \leq q \leq m \) // Continued below
### Algorithm 1

**Procedure 1** \( \text{compDistFromEx}(z) \)  
// This procedure is executed by a block
**Input:** \( z \)  
// an element of \( \Omega \). \( \Omega \) and \( U \) are declared in Algorithm 1  
**Output:** array of numbers that are distances between \( z \) and elements of \( U \)  
// length of array is size of \( U \)
1. **Declare:** array \( S := [s_1, \ldots, s_n] \)
2. **Launch** parallel threads of block \( T_i, 1 \leq l \leq n \)  
// \( n \) number of threads in block
3. \( s_l := \text{dist}(z, U[l]) \)
4. **Synchronize** all threads \( T_i, 1 \leq l \leq n \)
5. **Return** \( S \)

### Algorithm 2

**Procedure 2** \( \text{compV}(k) \)  
// This procedure is executed by a block
**Input:** \( V_k \)  
// \( V_k \) is a \( 1 \times n \) vector
**Output:** \( \text{compV}(k) \)  
1. for all \( 1 \leq t \leq n \) do
2. **Launch** parallel blocks \( B_q, 1 \leq q \leq M \)  
// one block per classification category value
3. for all \( 1 \leq k \leq M \) do
4. \( V_k := \text{compV}(k) \)
5. **end for**
6. **Synchronize** all blocks \( B_q, 1 \leq q \leq M \).  
// Find best point to classify
7. **Launch** single block \( B_1 \), and execute the next call on this block
8. \( \{p^*, a, u, b, v\} := \text{findBestPoint()} \)  
// \( p^* \) is best point in \( U \)
9. **Launch** single block \( B_1 \), and execute the next call on this block
10. **Classify** \( p^* \)  
// this sets \( l_{p^*} \) to some category value
11. **Launch** parallel Threads \( T_q, 1 \leq q \leq m \), \( \text{one thread per example} \)
12. **updateDnun** \( z_q, d(z_q, p^*), l_p \)
13. **Synchronize** all threads \( T_q, 1 \leq q \leq m \)
14. \( t := t + 1 \)
15. **end for**  
// for \( t \)
16. **Return** \( L \)  
// \( L \) contains the classification values of all points in \( U \)

### Algorithm 3

**Procedure 3** \( \text{initDnun}(z) \)  
// This procedure is executed by a block. It is used only initially when the only prototypes are the examples in \( \xi \)
**Input:** \( z \)  
// an element of \( \xi \)
**Output:** \( \alpha \)  
// \( \alpha \) minimal distance between \( z \) and \( z'_l \), over all \( 1 \leq l \leq m, \) where \( z'_l \in \xi \) and \( y(z'_l) \neq y(z) \)
1. **Declare:** \( \alpha, S := [s_{l=1}^m] \)
2. **Initialize:** \( s_l := \text{MAX}, 1 \leq l \leq m \)
3. **Launch** parallel threads \( T_l, 1 \leq l \leq m \), in block
4. if \( y(z) \neq y(z'_l) \) then
5. \( s_l := \text{dist}(z, z'_l) \)
6. **end if**
7. **Synchronize** all threads \( T_l, 1 \leq l \leq m \)
8. \( \{\alpha, i\} := \text{min}(S) \)  
// \( \text{min} \) is described in Procedure 9
9. **Return** \( \alpha \)

### Algorithm 4

**Procedure 4** \( \text{max}(S, \gamma) \)  
// This procedure is executed by a block
**Input:** \( S := [s_1, \ldots, s_N], \gamma \)
**Output:** \( \{\alpha, i\} \)  
// \( \alpha := \text{max}\{s_j : 1 \leq j \leq N, s_j < \gamma\}, i \)
// index of \( \alpha \) in \( S \)
1. **Implement** by parallel reduction algorithm (see [4])

### Algorithm 5

**Procedure 5** \( \text{max}_2(S) \)  
// This procedure is executed by a block
**Input:** \( S := [s_1, \ldots, s_N] \)
**Output:** \( \{\alpha, \beta, i, j\} \)  
// \( \alpha \) is largest entry of \( S \), \( i \) is index of largest entry, \( \beta \) is the second largest entry of \( S \), \( j \) index of second largest entry
1. \( \{\alpha, i\} := \text{max}(S, \text{MAX}) \)
2. \( \{\beta, j\} := \text{max}(S, \alpha) \)
3. **Return** \( \{\alpha, \beta, i, j\} \)

### Algorithm 6

**Procedure 6** \( \text{updateDnun}(z, \alpha, k) \)  
// This procedure is executed by a thread
**Input:** \( z, \alpha, k \)  
// \( z \) is an element of \( \Omega \), \( \alpha \) scalar, \( k \in \{1, \ldots, M\} \)
1. if \( d^\text{NUM} > \alpha \) and \( y(z) \neq k \) then
2. \( d^\text{NUM} := \alpha \)
3. **end if**
4. **Return**
2018 ICSEE International Conference on the Science of Electrical Engineering

Procedure 7  CompV(k)
// This procedure is executed by a block
Input: k  // k ∈ {1,...,M}
Output: S  // S := [s_1,...,s_n], where s_i equals size of
Vote set V_k(p_i), 1 ≤ i ≤ n
1: Declare: S := [s_1,...,s_n]
2: Launch parallel threads T_l, 1 ≤ l ≤ n in block
3: s_l := 0  // Initialize counter
   // only if p_i is not yet classified (p_i ∈ U)
4: if r_l = 0 then
5:   for all 1 ≤ j ≤ m do
6:     if D_j^l < d_N^U then
7:       y(z_j) = k then
8:       s_l := s_l + 1
9:     end if
10:   end if
11: end if
12: end if
13: Synchronize all threads, T_l, 1 ≤ l ≤ n
14: return S

Procedure 8  min2(S, T)
// This procedure is executed by a block
Input: S := [s_1,...,s_m], T := [ν_1,...,ν_M]  // M is number of categories
Output: i  // i ∈ {1,...,M}, where i = y(z*) and s_{z*} =
min {s_j : ν(z_j) = 1, z_j ∈ Ξ, 1 ≤ j ≤ m}
1: Declare: E := {e_1,...,e_m}  // E is array on which
   we search for minimum
2: Launch parallel threads in block T_l, 1 ≤ l ≤ m
3: if ν(z_l) = 1 then
4:   if y(z_l) is a relevant category
5:     e_l := s_l
6: else
7:     e_l := MAX
8: end if
9: Synchronize all threads, T_l, 1 ≤ l ≤ m
10: {s_j, i} := min(E)  // call Procedure 9, i is index of
    minimum entry of S
11: // s_i is not used (only i)
12: return y(z_i)  // return the label of example z_i

Procedure 9  min(S)
// This procedure is executed by a block
Input: S := [s_1,...,s_N]
Output: {α, i}  // α := min{s_j : 1 ≤ j ≤ N}, i := argmin_{1≤j≤N}s_j is index of entry with minimum value
   // Implement by parallel reduction algorithm (see [4])

Procedure 10  chooseRandomPoint()
// This procedure is executed by a block.
Output: p_k  // p_k is a randomly chosen point in U whose
r_k = 0
1: Declare array E := {e_1,...,e_n} and initialize it to
   {-1,...,-1}
2: Launch parallel threads T_l, 1 ≤ l ≤ n in block
3: if r_l = 0 then
4:   // only if point is not yet classified
5:     e_l := random()  // draw a random number in
         // range [0,1]
6: end if
7: synchronize all threads, T_l, 1 ≤ l ≤ n
8: {α, i} := max(E, MAX)  // i contains index of
   maximum value
9: return i

Procedure 11  Classify(p)
// This procedure is executed by a block
Input: p  // p is an entry of U
// This procedure sets l_p to some value k ∈ {1,...,M},
l_p is an entry of L where L is defined in Algorithm
1, M is number of categories. The procedure uses ar-
rays a, b, u, v, r and matrix D, defined in Algorithm
1
1: Declare: w := {w_1,...,w_M}  // entries of w are
   binary indicators, u_i = 1 indicates that vote-set
   V_i(p) has size equal to the maximum value a_p
2: Initialize: w := [0,...,0]
3: if a_p > b_p, then
4:   if a_p, b_p are entries of a, b
5:     k := u_p  // u_p is entry of u
6:     Goto 23
7: else
8:   if a_p > b_p and a_p > 0 and b_p > 0 then
9:     // next, search in column D_p for minimal entry
   whose row corresponds to z with y(z) = k, where
   k satisfies v[k, p] = a_p
10: Launch parallel threads in block T_l, 1 ≤ l ≤ M
11: if v[l, p] = a_p then
12:     w_l := 1
13: end if
14: D_p is m × 1 column of D
15: k := min2(D_p, w)
16: Goto 23
17: end if
18: else
19: w := [1,...,1]
20: k := min2(D_p, w)
21: Goto 23
22: end if
23: r_p := 1  // indicate that the point p is now classified
24: l_p := k  // and has a label k
25: return
Procedure 12 findBestPoint()
// This procedure is executed by a block
Output:
1) \( q \) // a point in \( U \)
2) \( \alpha \) // \( \alpha := [\alpha_1, \ldots, \alpha_n] \), \( \alpha_i \) is the size of largest
   voteset of unlabeled point \( p_i \)
3) \( \Upsilon \) // \( \Upsilon := [v_1, \ldots, v_n] \), \( v_i \) is index \( k^*(p_i) \) of vote
   set \( V_k(p_i) \) of maximum size
4) \( \beta \) // \( \beta := [\beta_1, \ldots, \beta_n] \), \( \beta_i \) is the size of the second-
   largest voteset of \( p_i \)
5) \( \Lambda \) // \( \Lambda := [\lambda_1, \ldots, \lambda_n]_{i=1}^n \), \( \lambda_i \) is index of vote set
   of second largest size
1: Declare: \( S := [s_1, \ldots, s_m] \) // \( S \) is array that contains
   the score of each unlabeled point
2: Launch parallel threads \( T_l \), \( 1 \leq l \leq n \), in block //
   one thread per point in \( U \)
3: if \( T_l = 1 \) then
4:  // the point \( p_l \) is already classified
5:  \( \alpha_l := -1 \), \( s_l := -1 \)
6:  Goto step 10
7: end if
8: \( \{\alpha_l, \beta_l, v_l, \lambda_l\} := \max2(V(U)) \) // Obtain maximum
   entry and second largest entry of column \( V(U) \)
9: \( s_l := \alpha_l(\alpha_l - \beta_l) \)
10: Synchronize all threads, \( T_l \), \( 1 \leq l \leq n \)
11: \( \{s_{\max}, q\} := \max(S, MAX) \)
12: if \( s_{\max} > 0 \) then
13:  Goto 23
14: else
15:  \( \{\alpha_{\max}, q\} := \max(\alpha, MAX) \)
16:  if \( \alpha_{\max} > 0 \) then
17:    Goto 23
18:  else
19:    \( q := \text{ChooseRandomPoint}() \)
20:  Goto 23
21: end if
22: end if
23: return \( \{q, \alpha, \Upsilon, \beta, \Lambda\} \)

IV. Speedup

Recall that \( n \) is the number of unlabeled points which are
to be classified. Let us assume that the number of parallel
executing threads is always large enough to handle all the
operations which are to be performed in parallel (the ‘launch’
statements). While this assumption describes an ideal setup,
it is a reasonable approximation of a typical scenario since
standard GPU support execution of thousands of threads
in parallel (for instance, nVIDIA’s Tesla K20c). In this
case Algorithm LWP takes \( O(n(\log M + \log m)) \) time to
execute. The sequential Algorithm LW takes \( O(n^2 \cdot m) \). Thus
the speed up under this ideal setup is \( O(nm/(\log(Mm))) \).
The factor \( nm \) is much larger than \( \log(Mm) \) hence LWP
provides a very significant speedup relative to Algorithm LW.

V. Conclusion

We introduce a parallel version of Algorithm LW [3]
which is an instance-based classification learning algorithm.
It learns over a space equipped with a distance function
which need not satisfy the metric axioms. This makes it
applicable to learning domains in which it is difficult to
formalize quantitative features that are encoded by vector
of numerical variables. Because the LW algorithm computes
distances between all pairs of data instances it is impractical
to apply it to large data sets. The parallel version intro-
duced here computes these efficiently by exploiting standard
parallel computing platforms and is therefore applicable
to learning problems with big data over general distance
spaces.

REFERENCES
  in Large-Margin Classifiers (Neural Information Processing). MIT