A PARALLEL ONLINE REGULARIZED LEAST-SQUARES MACHINE LEARNING ALGORITHM FOR FUTURE MULTI-CORE PROCESSORS

Tapio Pahikkala, Antti Airola, Thomas Canhao Xu,
Pasi Liljeberg, Hannu Tenhunen, and Tapio Salakoski
Turku Centre for Computer Science (TUCS) and Department of Information Technology, University of Turku, Joukahaisenkatu, Turku, Finland
tapio.pahikkala@utu.fi

Keywords: Machine learning, Network-on-Chip, Online learning, Regularized least-squares

Abstract: In this paper we introduce a machine learning system based on parallel online regularized least-squares learning algorithm implemented on a network on chip (NoC) hardware architecture. The system is specifically suitable for use in real-time adaptive systems due to the following properties it fulfills. Firstly, the system is able to learn in online fashion, a property required in almost all real-life applications of embedded machine learning systems. Secondly, in order to guarantee real-time response in embedded multi-core computer architectures, the learning system is parallelized and able to operate with a limited amount of computational and memory resources. Thirdly, the system can learn to predict several labels simultaneously which is beneficial, for example, in multi-class and multi-label classification as well as in more general forms of multi-task learning. We evaluate the performance of our algorithm from 1 thread to 4 threads, in a quad-core platform. A Network-on-Chip platform is chosen to implement the algorithm in 16 threads. The NoC consists of a 4x4 mesh. Results show that the system is able to learn with minimal computational requirements, and that the parallelization of the learning process considerably reduces the required processing time.

1 Introduction

The design of adaptive systems is an emerging topic in the area of pervasive and embedded computing. Rather than exhibiting pre-programmed behavior, it would in many applications be beneficial for systems to be able to adapt to their environment. Imagine smart music players that adapt to the musical preferences of their owner, intelligent traffic systems that monitor and predict traffic conditions and re-direct cars accordingly, etc. Clearly, it would be useful in such applications, if the considered system could automatically learn to perform the desired task, and over time improve its performance as more feedback is gained.

1.1 Machine Learning in Embedded Systems

Machine learning (ML) is a branch of computer science founded on the idea of designing computer algorithms capable of improving their prediction performance automatically over time through experience [Mitchell, 1997]. Such approaches offer the possibility to gain new knowledge through automated discovery of patterns and relations in data. Further, these methods can provide the benefit of freeing humans from doing laborious and repetitive tasks, when a computer can be trained to perform them. This is especially important in problem areas where there are massive amounts of complex data available, such as in image recognition or natural language processing.

In the recent years ML methods have increasingly been applied in non-traditional computing platforms, bringing both new challenges and opportunities. The shift from the single processor paradigm to parallel computing systems such as multi-core processors, cloud computing environments, graphic processing units (GPUs) and the network on chip (NoC) has resulted in a need for parallelizable learning methods [Chu et al., 2007, Zinkevich et al., 2009, Low et al., 2010].

At the same time, the widespread use of embedded systems ranging from industrial process control

systems to wearable sensors and smart-phones have opened up new application areas for intelligent systems. Some such recent ML applications include embedded real-time vision systems for field programmable gate arrays [Farabet et al., 2009], personalized health applications for mobile phones [Oresko et al., 2010] and sensor based video-game controls that learn to recognize user movements [Bogdanowicz, 2011]. For a thorough review of the design requirements of machine learning methods in embedded systems we refer to [Swere, 2008].

Majority of present-day machine learning research focuses on so-called batch learning methods. Such methods, given a data set for training, run a learning process on the data set and then output a predictor which remains fixed after the initial training has been finished. In contrast, it would be beneficial in real-time embedded systems for learning to be an ongoing process in which the predictors would be upgraded whenever new data become available. In machine learning literature, this type of methods are often referred to as online learning algorithms [Bottou and Le Cun, 2004]. One of the principal areas of application of this type of adaptive learning systems are hand-held devices such as smart-phones that learn to adapt to their users preferences.

In this work we consider how to implement in parallel online learning methods for (multi-class) classification in embedded computing environments. In classification, the system must assign a class label to a new object given the feature representation of the object. For example, in spam classification the features could be the words in an e-mail message, and possible classes consist of $\{$spam, not-spam$\}$, whereas in optical character recognition features could represent image scans and the set of available classes would encode different characters in the alphabet.

Our method is built upon the regularized least-squares (RLS) [Rifkin et al., 2003, Poggio and Smale, 2003], also known as the least-squares support vector machine [Suykens et al., 2002] and ridge regression [Hoerl and Kennard, 1970], which is is a state-of-the art machine learning method suitable both for regression and classification. Compared to the ordinary least-squares method introduced by Gauss, RLS is known to often achieve better predictive performance, as the regularization allows one to avoid over-fitting to the training data. An important property of the algorithm is that it has a closed form solution, which can be fully expressed in terms of matrix operations. This allows developing efficient computational shortcuts for the method, since small changes in the training data matrix correspond to low-rank changes in the learned predictor.

An online version of the ordinary (non-regularized) least-squares method was presented more than half a century ago by [Plackett, 1950]. The method has since then been widely used in real-time applications in areas such as machine learning, signal processing, communications and control systems. Online learning with RLS is also known in the machine learning literature (see e.g. [Zhdanov and Kalnishkan, 2010] and references therein). In this work we extend online RLS for multi-output learning and present an implementation that takes advantage of parallel computing architectures. Thus, the method allows adaptive learning efficiently in parallel environments, and is applicable to a wide range of problems both for regression and classification, and for single and multi-task learning.

1.2 Future Multi-Core Systems

Many embedded systems suffer from limited processing ability as they usually have only one relatively slow system-on-chip (SoC) processors. Since the beginning of the 21st century, Network-on-Chip (NoC) has become an emerging and promising solution in the Chip Multiprocessor (CMP) field [Dally and Towles, 2001]. This is due to the fact that the traditional design methods such as SoC have encountered critical challenges and bottlenecks as the number of on-chip integrated components increases. One of the most well known and critical problems is the communication bottleneck. Most traditional SoCs have the bus based communication architecture, such as simple, hierarchical or crossbar-type buses. In contrast with the increasing chip capacity, bus based systems do not scale well with the system size in terms of bandwidth, clock frequency and power consumption [Dally and Towles, 2001].

To address these problems and improve the system performance, NoC is proposed and endeavors to bring network communication methodologies to the on-chip communication. The NoC design approach is to create a communication interconnect beforehand and then map the computational resources to it via resource dependent interfaces. Figure 1 shows a 4×4 mesh based NoC topology. The underlying network is comprised of network links and routers (R), each of which is connected to a processing element (PE) via a network interface (NI). The basic architectural unit of a NoC is the tile/node (N) which is consisted of a router, its attached NI and PE, and the corresponding links. Communication among PEs is achieved via the transmission of network packets. Intel [1] has demon-

[1] Intel is a trademark or registered trademark of Intel or its subsidiaries. Other names and brands may be claimed as

strated an 80 tile, 100M transistor, 275mm^2 2D NoC under 65nm technology [Vangal et al., 2007]. An experimental microprocessor containing 48 x86 cores on a chip has been created using 4×6 2D mesh topology with 2 cores per tile [Intel, 2010]. Therefore, in this paper, NoC is chosen to be the platform for the study of the online machine learning algorithm.

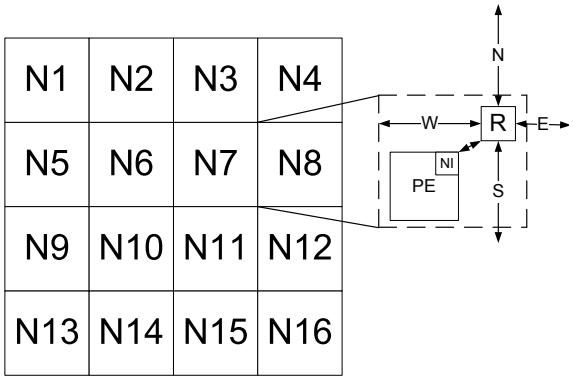

Figure 1: An example of 4×4 NoC using mesh topology.

1.3 Contributions of the Paper

In this work we introduce the parallel online RLS method, and explore its suitability for implementing adaptive systems on the NoC platform. The proposed approach has the following key benefits:

- The system can automatically learn to perform a task given examples of desired behavior, and can incorporate information from new training examples in an online fashion. This process may go on indefinitely, meaning lifelong learning.
- The system learns to predict several labels simultaneously which is beneficial, for example, in multi-class and multi-label classification as well as in more general forms of multi-task learning.
- Learning is parallelized and requires minimal storage and computational resources.
- The system is shown to work well on both on a quad-core platform and a NoC platform with 16 threads.

We expect that methods such as the online RLS have the capability to serve as the enabling technology for a wide range of applications in adaptive embedded systems.

the property of others.

2 Algorithm Descriptions

2.1 Regularized Least-Squares

We start by introducing some notation. Let $\mathbb{R}^m$ and $\mathbb{R}^{m\times n}$, where $m,n \in \mathbb{N}$, denote the sets of real valued column vectors and $m \times n$-matrices, respectively. To denote real valued matrices and vectors we use bold capital letters and bold lower case letters, respectively. Moreover, index sets are denoted with calligraphic capital letters. By denoting $\mathbf{M}_i$, $\mathbf{M}_{:,j}$, and $\mathbf{M}_{i,j}$, we refer to the ith row, jth column, and i,jth entry of the matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$, respectively. Similarly, for index sets $\mathcal{R} \subseteq \{1,\ldots,n\}$ and $\mathcal{L} \subseteq \{1,\ldots,m\}$, we denote the submatrices of $\mathbf{M}$ having their rows indexed by $\mathcal{R}$, the columns by $\mathcal{L}$, and the rows by $\mathcal{R}$ and columns by $\mathcal{L}$ as $\mathbf{M}_{\mathcal{R}}$, $\mathbf{M}_{:,\mathcal{L}}$, and $\mathbf{M}_{\mathcal{R},\mathcal{L}}$, respectively. We use an analogous notation also for column vectors, that is, $\mathbf{v}_i$ refers to the ith entry of the vector $\mathbf{v}$.

Let $\mathbf{X} \in \mathbb{R}^{m\times n}$ be a matrix containing the feature representation of the examples in the training set, where n is the number of features and m is the number of training examples. The i,jth entry of $\mathbf{X}$ contains the value of the jth feature in the ith training example. Moreover, let $\mathbf{Y} \in \mathbb{R}^{m\times l}$ be a matrix containing the labels of the training examples. We assume each data point to have altogether l labels and the i,jth entry of $\mathbf{Y}$ contains the value of the jth label of the ith training example. In multi-class or multi-label classification, the labels can be restricted to be either 1 or -1 depending whether the data points belongs to the class, for example, while they can be any real numbers in multi-label regression tasks (see e.g. [Hsu et al., 2009] and references therein).

As an example, consider that we have a set of m images, each of which is represented by n features. In addition, each image is associated with an array of l binary labels of which the value of the jth label is 1 if the object indexed by j is depicted in the image and -1 otherwise. Our aim is to learn from the set of images to predict what is depicted in a any new image unseen in the set.

In this paper, we consider linear predictors of type

$$f(\mathbf{x}) = \mathbf{W}^{\mathrm{T}}\mathbf{x}, \tag{1}$$

where $\mathbf{W} \in \mathbb{R}^{n\times l}$ is the matrix representation of the learned predictor and $\mathbf{x} \in \mathbb{R}^n$ is a data point for which the prediction of l labels is to be made.[2] The computational complexity of making predictions with (1) and the space complexity of the predictor are both $O(nl)$

[2] In the literature, the formula of the linear predictors often also contain a bias term. Here, we assume that if such a bias is used, it will be realized by using an extra constant valued feature in the data points.

provided that the feature vector representation $\mathbf{x}$ for the new data point is given.

Given training data $\mathbf{X}, \mathbf{Y}$, we find $\mathbf{W}$ by minimizing the RLS risk. This can be expressed as the following problem:

$$\underset{\mathbf{W} \in \mathbb{R}^{n \times l}}{\operatorname{argmin}} \left\{ \|\mathbf{X}\mathbf{W} - \mathbf{Y}\|_F^2 + \lambda \|\mathbf{W}\|_F^2 \right\}, \qquad (2)$$

where $\|\cdot\|_F$ denotes the Frobenius norm which is defined for a matrix $\mathbf{M} \in \mathbb{R}^{n \times l}$ as

$$\|\mathbf{M}\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{l} (\mathbf{M}_{i,j})^2}.$$

The first term in (2), called the empirical risk, measures how well the prediction function fits to the training data. The second term is called the regularizer and it controls the tradeoff between the empirical error on the training set and the complexity of the prediction function.

2.2 Batch Learning for RLS

A straightforward approach to solve (2) is to set the derivative of the objective function with respect to $\mathbf{W}$ to zero. Then, by solving it with respect to $\mathbf{W}$, we get

$$\mathbf{W} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{Y}, \qquad (3)$$

where $\mathbf{I}$ is the identity matrix. We note (see e.g. [Henderson and Searle, 1981]) that an equivalent result can be obtained from

$$\mathbf{W} = \mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{X}^{\mathrm{T}} + \lambda \mathbf{I})^{-1}\mathbf{Y}. \qquad (4)$$

If the number of features n is smaller than the number of training examples m, it is computationally beneficial to use the form (3) while using (4) is faster in the opposite case. Namely, the computational complexity of (3) is $O(n^3 + n^2 m + nml)$, where the first term is the complexity of the matrix inversion, the second comes from multiplying $\mathbf{X}^{\mathrm{T}}$ with $\mathbf{X}$, and the third from multiplying the result of the inversion with the matrix $\mathbf{Y}$. The complexity of (4) is $O(m^3 + m^2 n + nml)$, where the terms are analogous to those of (3). Putting these two together, the complexity of training a predictor is $O(nm(\min\{n,m\} + l))$. It is also straightforward to see from (3) and (4) that the space complexity $O(m(n+l))$ of RLS directly depends on the size of the matrices $\mathbf{X}$ and $\mathbf{Y}$.

One of the benefits of RLS is that the number of labels per data point l can be increased up to the level of m or n until it starts to have an effect on the space and time complexities of RLS training. That is, we can solve several prediction tasks almost at the cost of solving only one. This is beneficial especially in multi-class and multi-label classification tasks, for example.

2.3 Online Learning for RLS

Next, we consider a computational short-cut for updating a learned RLS predictor when a new training example arrives. The short-cut is then used to construct an online version of the RLS algorithm. Similar considerations have already been presented in the machine learning literature (see e.g. [Zhdanov and Kalnishkan, 2010] and references therein) but here we formalize it for the first time for multiple outputs, that is, for the case in which the data points can have more than one label.

First, we present the following well-known result which is often referred to as the matrix inversion lemma or the Sherman-Morrison-Woodbury formula (see e.g. [Horn and Johnson, 1985, p. 18]):

Lemma 2.1. *Let* $\mathbf{M} \in \mathbb{R}^{a \times a}$, $\mathbf{N} \in \mathbb{R}^{b \times b}$, $\mathbf{P} \in \mathbb{R}^{a \times b}$, *and* $\mathbf{Q} \in \mathbb{R}^{b \times a}$ *be matrices. If* $\mathbf{M}$, $\mathbf{N}$, *and* $\mathbf{M} - \mathbf{PNQ}$ *are invertible, then*

$$\begin{aligned} &(\mathbf{M} - \mathbf{PNQ})^{-1} \\ &\quad = \mathbf{M}^{-1} - \mathbf{M}^{-1}\mathbf{P}(\mathbf{N}^{-1} - \mathbf{Q}\mathbf{M}^{-1}\mathbf{P})^{-1}\mathbf{Q}\mathbf{M}^{-1}. \end{aligned} \qquad (5)$$

The main consequence of this result is that if we already know the inverse of the matrix $\mathbf{M}$ and if $b << a$, we can save a considerable amount of computational resources by using the right hand side of (5) instead of computing an inverse of an $a \times a$ matrix.

Assume that we have already trained an RLS predictor from the training set $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{m \times l}$ with a regularization parameter value λ, and hence we have $\mathbf{W}$ stored in memory. In addition, let us assume that during the training process, we have computed the matrix

$$\mathbf{C} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}$$

and stored it in memory. According to (3), we have $\mathbf{W} = \mathbf{C}\mathbf{X}^{\mathrm{T}}\mathbf{Y}$. Moreover, let $\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^l$ be a new data point unseen in the training set and let $\widehat{\mathbf{X}} \in \mathbb{R}^{(m+1) \times n}, \widehat{\mathbf{Y}} \in \mathbb{R}^{(m+1) \times l}$ be the new training set including the new training example. Now, since we already have the matrix $\mathbf{C}$ stored in memory, we can use the matrix inversion lemma to speed up the computation of the matrix $\widehat{\mathbf{C}}$ corresponding to the updated training set:

$$\begin{aligned} \widehat{\mathbf{C}} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \mathbf{x}\mathbf{x}^{\mathrm{T}} + \lambda \mathbf{I})^{-1} && (6) \\ &= (\mathbf{C}^{-1} + \mathbf{x}\mathbf{x}^{\mathrm{T}})^{-1} \\ &= (\mathbf{C} - \mathbf{C}\mathbf{x}(\mathbf{x}^{\mathrm{T}}\mathbf{C}\mathbf{x} + 1)^{-1}\mathbf{x}^{\mathrm{T}}\mathbf{C}), && (7) \end{aligned}$$

where the calculation of (7) requires only $O(n^2)$ time instead of the $O(n^3)$ time required in (6). The predictor $\widehat{\mathbf{W}}$ corresponding to the updated training set can then be computed from

$$\widehat{\mathbf{W}} = \widehat{\mathbf{C}}(\mathbf{X}^{\mathrm{T}}\mathbf{Y} + \mathbf{x}\mathbf{y}^{\mathrm{T}}) \qquad (8)$$

in $O(ln^2)$ time provided that $\mathbf{X}^{\mathrm{T}}\mathbf{Y}$ has already been computed during training with the original training set. If there are altogether m training examples, which are added into the training set one at a time, the overall computational complexity of this online variation would be $O(mln^2)$, which is slower than the batch RLS training if the number of labels per training examples l is large.

Next, we show how to improve the complexity even further. Let us first define some extra notation. Let

$$\begin{aligned} \mathbf{v} &= \mathbf{C}\mathbf{x}, && (9)\\ c &= \mathbf{x}^{\mathrm{T}}\mathbf{v}, \\ d &= (c+1)^{-1}, \text{ and} \\ \mathbf{p} &= \mathbf{W}^{\mathrm{T}}\mathbf{x}. && (10) \end{aligned}$$

Continuing from (7) and (8), we get

$$\begin{aligned} \widehat{\mathbf{W}} &= (\mathbf{C} - \mathbf{C}\mathbf{x}(\mathbf{x}^{\mathrm{T}}\mathbf{C}\mathbf{x}+1)^{-1}\mathbf{x}^{\mathrm{T}}\mathbf{C})(\mathbf{X}^{\mathrm{T}}\mathbf{Y} + \mathbf{x}\mathbf{y}^{\mathrm{T}}) \\ &= (\mathbf{C} - d\mathbf{v}\mathbf{v}^{\mathrm{T}})(\mathbf{X}^{\mathrm{T}}\mathbf{Y} + \mathbf{x}\mathbf{y}^{\mathrm{T}}) && (11)\\ &= \mathbf{C}\mathbf{X}^{\mathrm{T}}\mathbf{Y} + \mathbf{C}\mathbf{x}\mathbf{y}^{\mathrm{T}} - d\mathbf{v}\mathbf{v}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{Y} - d\mathbf{v}\mathbf{v}^{\mathrm{T}}\mathbf{x}\mathbf{y}^{\mathrm{T}} \\ &= \mathbf{W} + \mathbf{v}\mathbf{y}^{\mathrm{T}} - d\mathbf{v}\mathbf{p}^{\mathrm{T}} - cd\mathbf{v}\mathbf{y}^{\mathrm{T}} \\ &= \mathbf{W} + \mathbf{v}((1-cd)\mathbf{y}^{\mathrm{T}} - d\mathbf{p}^{\mathrm{T}}). && (12) \end{aligned}$$

The computational complexity of calculating $\widehat{\mathbf{W}}$ from (12) requires $O(n^2 + nl)$ time. Here, the first term is the complexity of calculating $\widehat{\mathbf{C}}$ with (7) and $\mathbf{v}$ with (9). The second term is the complexity of calculating $\mathbf{p}$ with (10), multiplying $\mathbf{v}$ with $(1-cd)\mathbf{y}^{\mathrm{T}} - d\mathbf{p}^{\mathrm{T}}$, and adding the result to $\mathbf{W}$ in (12). Multiplying the complexity of a single iteration with the number of training examples m, we get the overall training time complexity of online RLS $O(mn^2 + mnl)$. Thus, online training of RLS with a set of m data points is computationally as efficient as the training of batch RLS with (3) and provides exactly equivalent results. Batch learning with (4) is more efficient only if $m < n$ but this is rarely the case in the lifelong learning setting considered in this paper. The space complexity of online RLS is $O(n^2 + nl)$, where the first term is the cost of keeping the matrix $\mathbf{C}$ in memory and the second is that of the matrix $\mathbf{W}$.

Putting everything together, we present the online RLS in Algorithm 1. The algorithm first starts from an empty training set (lines 1-2). Next, it read a feature representation of a new data point (line 4) and outputs a vector of predicted labels for the data point (line 5). After the prediction, the method is given a feedback in the form of the correct label vector of the data point (line 6). Finally, the features and labels of the new data point are used to update the predictor (lines 7-11). The steps 4-11 are reiterated whenever new data are observed.

Algorithm 1 Pseudo code for Online RLS

1: $\mathbf{C} \leftarrow \lambda^{-1}\mathbf{I}$
2: $\mathbf{W} \leftarrow \mathbf{0} \in \mathbb{R}^n$
3: **for** $t \in 1, 2, \ldots$ **do**
4: Read data point $\mathbf{x}$
5: Output prediction $\mathbf{p} \leftarrow \mathbf{W}^{\mathrm{T}}\mathbf{x}$
6: Read true labels $\mathbf{y}$
7: $\mathbf{v} \leftarrow \mathbf{C}\mathbf{x}$
8: $c \leftarrow \mathbf{x}^{\mathrm{T}}\mathbf{v}$
9: $d \leftarrow (c+1)^{-1}$
10: $\mathbf{C} \leftarrow \mathbf{C} - d\mathbf{v}\mathbf{v}^{\mathrm{T}}$
11: $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}((1-cd)\mathbf{y}^{\mathrm{T}} - d\mathbf{p}^{\mathrm{T}})$

2.4 Parallelized Online RLS

Because of the layout of matrices in memory and the nature of the basic matrix operations, it is often possible to gain considerable performance improvements with parallelization. Indeed, the parallelization of the batch RLS has been tested on graphics processing units by [Do et al., 2008] who reported large gains in running speed. Here, we consider the parallelization of online RLS with multiple outputs.

Algorithm 2 Parallel computation for $\mathbf{v} \leftarrow \mathbf{C}\mathbf{x}$

Split the index set $\{1, \ldots, n\}$ into p disjoint subsets $I_1, \ldots, I_p$ of which each of the p processors get one.
for $h \in \{1, \ldots, p\}$ **do**
  Load $\mathbf{x}$, $\mathbf{v}_{I_h}$, and $\mathbf{C}_{I_h}$ into cache/memory.
  for $i \in I_p$ **do**
    $\mathbf{v}_i \leftarrow 0$
    for $j \in \{1, \ldots, n\}$ **do**
      $\mathbf{v}_i \leftarrow \mathbf{v}_i + \mathbf{C}_{i,j}\mathbf{x}_j$
return $\mathbf{v}$

Algorithm 3 Parallel computation for $\mathbf{C} \leftarrow \mathbf{C} - d\mathbf{v}\mathbf{v}^{\mathrm{T}}$

Split the index set $\{1, \ldots, n\}$ into p disjoint subsets $I_1, \ldots, I_p$ of which each of the p processors get one.
for $h \in \{1, \ldots, p\}$ **do**
  Load d, $\mathbf{v}$, and $\mathbf{C}_{I_h}$ into cache/memory.
  for $i \in I_p$ **do**
    $g \leftarrow d\mathbf{v}_i$
    for $j \in \{1, \ldots, n\}$ **do**
      $\mathbf{C}_{i,j} \leftarrow \mathbf{C}_{i,j} - g\mathbf{v}_j$
return $\mathbf{C}$

The two most expensive parts in Algorithm 1 are the lines 7 and 10 which both require $O(n^2)$ time,

and hence we concentrate primarily on those when designing a parallel version of online RLS. The parallelization of the lines 7 and 10 are presented in Algorithms 2 and 3, respectively. In both algorithms, the outer loop corresponds to distributing the work among p processors. The former algorithm is simply a parallelization of a matrix-vector product which is widely known in literature but we present if here for self-sufficiency. The parallelization of the outer product of two vectors considered in the latter algorithm is almost as straightforward. In both cases, there is no time wasted waiting for memory write locks, because every processor is updating different memory locations determined by the index sets. Moreover, if the processors have a sufficient amount of cache memory available, the progress can be accelerated even further, since the different processors require different portions of the matrix $\mathbf{C}$.

Finally, we note that if the number of label per data point l is large, the lines 5 and 11 in Algorithm 1 can be parallelized in similar way as the lines 7 and 10, respectively. This is, because the former contains a product of a matrix and a vector, and the latter contains an outer product of two vectors. Putting everything together, the computational complexity of a single iteration of parallel online RLS is $O(n^2/p+nl/p)$, where p is the number of processors. The complexity of learning from a sequence of m data points is m times the complexity of a single iteration.

3 Experiments

We implement the online RLS method in the C++ programming language, and parallelize it via the OpenMP parallel programming platform. Experiments are run both in a desktop environment with a multi-core Intel processor, as well as on a NoC simulation platform.

3.1 Recognition of Hand-Written Digits

In the experiments we explore the behavior of the parallel online RLS on the MNIST handwritten digit database[3] benchmark. The database consists of 28×28 pixel black and white image scans of handwritten digits. The features of each image consist of pixel intensity values normalized to $\{0,1\}$ range. The task is to be able to predict given the pixel intensity values of an image, which of the digits from range $\{0 \ldots 9\}$ it represents. We follow the original training-test split defined in the dataset, which ensures that the test characters have been written by different authors than the training ones. The training set consists of 60000 examples, and the test set of 10000 examples. The pixel intensity values are directly used as the features for the linear model. We note that developing a state-of-the art handwritten digit recognition system would require more advanced feature engineering and non-linear modeling but constructing such a system goes outside the scope of this paper. The regularization parameter is set to 1 in the experiments.

[3] Available at `http://yann.lecun.co m/exdb/mnist/`

3.2 Runtime and Classification Performance

First, we measure the effect of the parallelization on the runtime required for updating the OnlineRLS predictor with a new example. We first load the data set into the memory, and then compute the average time spent on update over the 60000 training examples. The experiment is run on an Intel Core i7-950 processor machine. The run-times are presented in Figure 2. On 4 cores the update operations are approximately 3 times faster than on a single core, demonstrating that substantial speedups can be gained in parallel hardware architectures for the method. The CPU time spent updating the predictor ranges from around 900 to below 300 microseconds per example, depending on the number of cores, which suggests that the method could be useful in systems, where only minimal computational times can be afforded.

In Figure 3 we plot the classification performance of the learned classifier as a function of number of training examples processed. The prediction performance is especially in the early phases of learning affected by the order in which the training examples are supplied. Therefore, we compute the average results over 10 repetitions of the experiment, where at the start of each experiment the order of the training examples is shuffled. The error rate measures the relative fraction of misclassified test examples. Since all classes are are roughly equally represented in the test set, naive approaches such as predicting always the same class, or choosing the class randomly would lead to around 90% test error. The online RLS method also begins with predictive performance in this range, but as more examples are processed, we see significant improvements, with error rate reaching 14.7% once all the 60000 training examples have been processed. The curve demonstrates the ability of online RLS to improve its performance by adapting to examples provided over time.

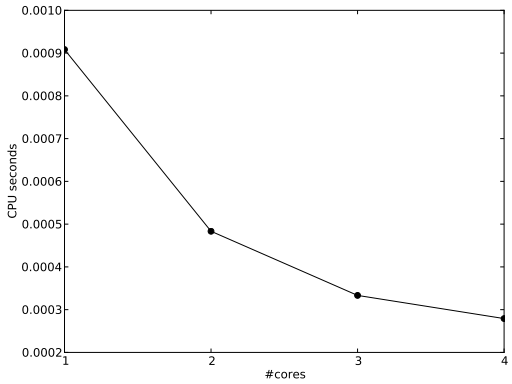

Figure 2: Average computational cost of updating the predictor with new training example on Mnist.

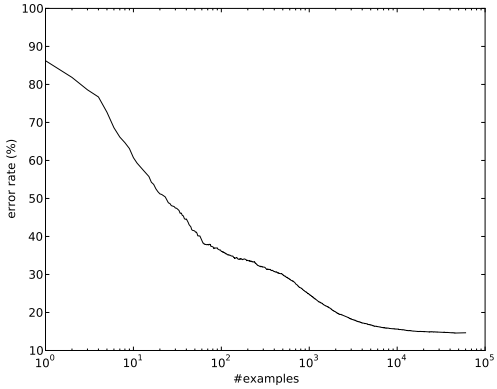

Figure 3: Error rate on the Mnist data as a function of training examples.

3.3 NoC Simulation Environment

To evaluate our algorithm further, we use a cycle-accurate NoC simulator (see Figure 1). The simulation platform is able to produce detailed evaluation results. The platform models the routers and links accurately. The state-of-the-art router in our platform includes a routing computation unit, a virtual channel allocator, a switch allocator, a crossbar switch and four input buffers. Routers in our NoC model have five ports (North, East, West, South and Local PE) and the corresponding virtual channels, buffers and crossbars. It is noteworthy that not all routers in a NoC require five ports, e.g. router of N1 in Figure 1 has only East, South and Local PE ports. Adaptive routing is used widely in off-chip networks, however deterministic routing is favorable for on-chip networks because the implementation is easier. In this paper, a dimensional ordered routing (DOR) [Sullivan and Bashkow, 1977] based X-Y deterministic routing algorithm is selected, in which a message packet (flit) is first routed to the X direction and last to the Y direction.

Table 1: System configuration parameters

Processor configuration	
Instruction set architecture	SPARC
Number of processors	16
Issue width	1
Cache configuration	
L1 cache	Private, split instruction and data cache, each cache is 16KB. 4-way associative, 64-bit line, 3-cycle access time
L2 cache	Shared, distributed in 4 layers, unified 2-16MB (16 banks, each 256KB-1MB). 64-bit line, 6-cycle access time
Cache coherence protocol	MESI
Cache hierarchy	SNUCA
Memory configuration	
Size	4GB DRAM
Access latency	260 cycles
Requests per processor	16 outstanding
Network configuration	
Router scheme	Wormhole
Flit size	128 bits

We use a 16-node network which models a single-chip CMP for our experiments. A full system simulation environment with 16 processors has been implemented. The simulations are run on the Solaris 9 operating system based on SPARC instruction set in-order issue structure. Each processor is attached to a wormhole router and has a private write-back L1 cache. The L2 cache shared by all processors is split into banks. The size of each cache bank node is 1MB; hence the total size of shared L2 cache is 16MB. Each L2 cache bank is attached to a router as well. The simulated memory/cache architecture mimics SNUCA [Kim et al., 2002]. A two-level distributed directory cache coherence protocol called MESI [Patel and Ghose, 2008] has been used in our memory hierarchy in which each L2 bank has its own directory. Four types of cache line status, namely Modified (M), Exclusive (E), Shared (S) and Invalid

(I) are implemented. We use Simics [Magnusson et al., 2002] full system simulator as our simulation platform. The detailed configurations of processor, cache and memory configurations can be found in Table 1.

3.4 NoC Simulation Result Analysis

The normalized full system simulation results are shown in Figures 4 and 5. We use the machine learning algorithm with 16 threads. The problem size for our simulation is 100 inputs. To analyze the temporal locality of our algorithm, we estimate the cache miss rate with different L2 cache sizes. For shared memory CMPs, a large last level cache is crucial, because a miss in the cache will require an access to the off-chip main memory. Figure 4 show that, as the cache size increases, the cache miss rate decreases gradually (From 2.79% to 1.48%). The main reason is that, in our algorithm, the data-set is pre-loaded into the cache-memory first. More data-set and intermediate data can be stored with a larger cache. Average network latency represents the average number of cycles required for the transmission of all network messages. For each message, the number of cycle is calculated as, from the injection of a message header into the network at the source node, to the reception of a tail flit at the destination node. As illustrated in Figure 5, the 16MB configuration outperforms others in terms of average network latency. The latency in 16MB configuration is 1.73%, 3.53% and 4.56% lower than the 8MB, 4MB and 2MB configurations, respectively. This is primarily due to the reduced cache miss rate in the 16MB configuration compared to the other configurations. We notice that, an off-chip access of the main memory will result a significant higher network latency. However, since there are millions of cache/memory accesses, the impact is less significant as a whole.

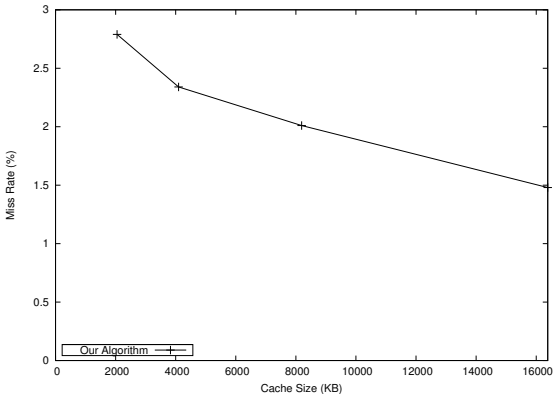

Figure 4: Cache miss rate versus cache size.

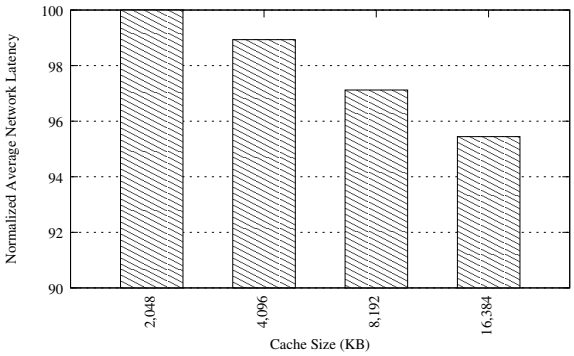

Figure 5: Normalized average network latency with different number of pillars.

4 Discussion and Future Work

The most immediate bottleneck in the considered parallel online RLS algorithm is its quadratic scaling with respect to n, the number of features in the data points, because both the space and time complexities are quadratic in n. Therefore, combining the algorithm with feature selection techniques (see e.g. [Guyon and Elisseeff, 2003]) when the classification tasks involve high dimensional data can be a fruitful research direction. A suitable technique for this purpose would be, for example, the computationally efficient feature selection algorithm for RLS proposed by us in [Pahikkala et al., 2010]. In the future, we plan to develop algorithms which would perform online learning and feature selection simultaneously. There is already some prior work on this type of algorithms (see e.g. [Jung and Polani, 2006]).

5 CONCLUSIONS

We have introduced a machine learning system which is based on parallel online regularized least-squares learning algorithm and implemented on a network on chip (NoC) platform. The system is specifically suitable for use in real-time adaptive systems due to the following properties it fulfills:

- The system is able to learn in online fashion, that is, it can update itself in real-time whenever new data is observed. This is an essential property in real-life applications of embedded machine learning systems, for example, in smart-phone applications that aim to adapt to their owners preferences.
- The learning system is parallelized and works

with limited processor time and memory. This opens the possibilities to deploy the system, for example, into small hand-held devices which operate in real-time.

- The system can carry out complex learning tasks involving simultaneous prediction of several labels per data point. Typical examples of this type of tasks are, for example, in multi-class and multi-label classification.

The run-time performance of the proposed system was evaluated using 1 to 4 threads, in a quad-core platform. It was shown that, as expected according to the theoretical considerations, the performance gain is roughly linear with respect to the number of cores. In an additional experiment, we used NoC platform to test the system in 16 threads. The NoC consists of a 4x4 mesh. The obtained results demonstrated that the system is able to learn with minimal computational requirements, and that the parallelization of the learning process considerably reduces the required processing time.

Altogether, the study sheds light on the possibilities of deploying modern machine learning methods into embedded systems based on future multi-core computing architectures. For example, the machine learning techniques that are able to operate in real time and in online fashion are promising tools for pursuing adaptivity of embedded systems. This is because they enable the real time updating of the system according to the data observed from the environment. While we used a digit recognition task as case study in this paper, the learning system can be applied on a wide range of other tasks such as energy efficiency or control of the embedded systems.

ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland.

REFERENCES

Bogdanowicz, A. (2011). The motion tech behind Kinect. *IEEE The Institute*. Published Online 6. January 2011 http://www.theinsti tute.ieee.org.

Bottou, L. and Le Cun, Y. (2004). Large scale online learning. In Thrun, S., Saul, L., and Schölkopf, B., editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y., and Olukotun, K. (2007). Map-reduce for machine learning on multicore. In Schölkopf, B., Platt, J., and Hoffman, T., editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, Cambridge, MA.

Dally, W. J. and Towles, B. (2001). Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th conference on Design automation*, pages 684–689.

Do, T.-N., Nguyen, V.-H., and Poulet, F. (2008). Speed up SVM algorithm for massive classification tasks. In Tang, C., Ling, C. X., Zhou, X., Cercone, N., and Li, X., editors, *Proceedings of the 4th International Conference on Advanced Data Mining and Applications (ADMA 2008)*, volume 5139 of *Lecture Notes in Computer Science*, pages 147–157. Springer.

Farabet, C., Poulet, C., and LeCun, Y. (2009). An fpga-based stream processor for embedded real-time vision with convolutional networks. In *Fifth IEEE Workshop on Embedded Computer Vision (ECV'09)*, pages 878–885. IEEE.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182.

Henderson, H. V. and Searle, S. R. (1981). On deriving the inverse of a sum of matrices. *SIAM Review*, 23(1):53–60.

Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12:55–67.

Horn, R. and Johnson, C. R. (1985). *Matrix Analysis*. Cambridge University Press, Cambridge.

Hsu, D., Kakade, S., Langford, J., and Zhang, T. (2009). Multi-label prediction via compressed sensing. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, pages 772–780. MIT Press.

Intel (2010). Single-chip cloud computer. http://techresearch. intel.com/articles/Tera-Scale/ 1826.htm.

Jung, T. and Polani, D. (2006). Sequential learning with ls-svm for large-scale data sets. In Kollias, S. D., Stafylopatis, A., Duch, W., and Oja, E., editors, *Proceedings of the 16th International Conference on Artificial Neural Networks (ICANN 2006)*, volume 4132 of *Lecture Notes in Computer Science*, pages 381–390. Springer.

Kim, C., Burger, D., and Keckler, S. W. (2002). An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ACM SIGPLAN*, pages 211–222.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). Graphlab: A new framework for parallel machine learning. In *The 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*.

Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: A full system simulation platform. *Computer*, 35(2):50–58.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.

Oresko, J. J., Jin, Z., Cheng, J., Huang, S., Sun, Y., Duschl, H., and Cheng, A. C. (2010). A wearable smartphone-based platform for real-time cardiovascular disease detection via electrocardiogram processing. *IEEE Transactions on Information Technology in Biomedicine*, 14:734–740.

Pahikkala, T., Airola, A., and Salakoski, T. (2010). Speeding up greedy forward selection for regularized least-squares. In Draghici, S., Khoshgoftaar, T. M., Palade, V., Pedrycz, W., Wani, M. A., and Zhu, X., editors, *Proceedings of The Ninth International Conference on Machine Learning and Applications (ICMLA'10)*, pages 325–330. IEEE.

Patel, A. and Ghose, K. (2008). Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. In *Proceeding of the thirteenth international symposium on Low power electronics and design*, pages 247–252.

Plackett, R. L. (1950). Some theorems in least squares. *Biometrika*, 37(1/2):pp. 149–157.

Poggio, T. and Smale, S. (2003). The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society (AMS)*, 50(5):537–544.

Rifkin, R., Yeo, G., and Poggio, T. (2003). Regularized least-squares classification. In Suykens, J., Horvath, G., Basu, S., Micchelli, C., and Vandewalle, J., editors, *Advances in Learning Theory: Methods, Model and Applications*, volume 190 of *NATO Science Series III: Computer and System Sciences*, chapter 7, pages 131–154. IOS Press, Amsterdam.

Sullivan, H. and Bashkow, T. R. (1977). A large scale, homogeneous, fully distributed parallel machine. In *Proceedings of the 4th annual symposium on Computer architecture*, pages 105–117.

Suykens, J., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J. (2002). *Least Squares Support Vector Machines*. World Scientific Pub. Co., Singapore.

Swere, E. A. (2008). *Machine Learning in Embedded Systems*. PhD thesis, Loughborough University.

Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y., and Borkar, N. (2007). An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *IEEE International Solid-State Circuits Conference ISSCC 2007*, pages 98–589. IEEE.

Zhdanov, F. and Kalnishkan, Y. (2010). An identity for kernel ridge regression. In Hutter, M., Stephan, F., Vovk, V., and Zeugmann, T., editors, *Proceedings of the 21st international conference on Algorithmic learning theory*, volume 6331 of *Lecture Notes in Computer Science*, pages 405–419, Berlin, Heidelberg. Springer-Verlag.

Zinkevich, M., Smola, A., and Langford, J. (2009). Slow learners are fast. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, pages 2331–2339.