AdaBoost Parallelization on PC Clusters with Virtual Shared Memory for Fast Feature Selection

Virginie Galtier, Olivier Pietquin, Stéphane Vialle

2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007), 24-27 November 2007, Dubai, United Arab Emirates

ADABOOST PARALLELIZATION ON PC CLUSTERS WITH VIRTUAL SHARED MEMORY FOR FAST FEATURE SELECTION

Virginie Galtier, Olivier Pietquin and Stéphane Vialle

SUPELEC - IMS Research Group
2 rue Edouard Belin, 57070 Metz, France
firstname.lastname@supelec.fr

ABSTRACT

Feature selection is a key issue in many machine learning applications and the need to test lots of candidate features is real while computational time required to do so is often huge. In this paper, we introduce a parallel version of the well-known AdaBoost algorithm to speed up and size up feature selection for binary classification tasks using large training datasets and a wide range of elementary features.

This parallelization is done without any modification to the AdaBoost algorithm and designed for PC clusters using Java and the JavaSpace distributed framework. JavaSpace is a memory sharing paradigm implemented on top of a virtual shared memory, that appears both efficient and easy-to-use. Results and performances on a face detection system trained with the proposed parallel AdaBoost are presented.

1. MOTIVATIONS AND OBJECTIVES

A key issue in the fields of machine learning and pattern recognition is the choice of highly discriminant features keeping the feature space small enough to be processed in a reasonable amount of time. The AdaBoost algorithm [3] is a sequential iterative algorithm that allows selecting the optimal combination of elementary features among a large set of candidates for a two-class separation problem. It has proven its efficiency for building state-of-the-art real-time object detection systems [9]. The fundamental idea underlying AdaBoost is to build a strong classifier which decision is a linear combination of multiple weak classifiers (or base learners) decisions, that is building an accurate decision-making system based on a set of easily computable tests. A weak classifier is usually an elementary learner quickly trained to have performance slightly above 50% on a training dataset. When used for feature selection, the AdaBoost algorithm picks up one weak classifier from a large set of those at each round. To ensure the optimality of the weak classifier combination, the training examples are assigned weights that change from round to round. The previously misclassified examples are assigned higher weights and the training error of each weak classifier at a given round is computed as the sum of the weights associated with the examples it misclassifies. The weak classifier providing the lowest weighted error is selected and a new weight distribution over examples is computed for the next turn. The selected weak classifiers are therefore more and more focussed on the misclassified examples. The complete Adaboost algorithm is described in Table 1.

Our objective is to speed up and size up the AdaBoost algorithm without introducing any modification, in order to help signal processing and machine learning researchers to train classification systems on very large datasets and to test a wide range of candidate features. To achieve high speed up and size up with parallel computers available for many researchers, a classical solution consists in using PC clusters, or Grid of PC clusters (some PC clusters localized on different sites).

Given a training set $\{(x_i, y_i)\}$
where $i = 1, ..., N$, $x_i \in X$ and $y_i \in \{-1, +1\}$
Initialize $t = 1$, example distrib. $D_t(i) = 1/n \ \forall i$ and $E = 1$
while $E > \xi$ **do**
  Train weak classifiers using D_t
  Get weak hypothesis $h_t : X \rightarrow \{-1, +1\}$
  Compute weak class. weighted er. $\epsilon_t = \sum_{h_t(x_i) \neq y_i} D_t(i)$
  Choose $\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$
  Update $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{\sum_t D_{t+1}}$
  Compute strong hypotheses $H(x_i) = sign(\sum_t \alpha_t h_t(x_i))$
  Compute strong classifier error $E = \sum_{H(x_i) \neq y_i} \frac{1}{N}$
  $t++$
Output the final hypothesis $H(x) = sign(\sum_t \alpha_t h_t(x))$

Table 1. *AdaBoost Algorithm.*

Previous works on distributed AdaBoost algorithms on general purpose computers have focussed on designing some modified versions to run independent computations on different computers and to merge the different results at the end of each round with limited communications [6, 7]. This is particularly helpful in the case of physically distributed resources (such as databases) but the aim is not the gain in time. Some of these strategies achieved good performances running less rounds. They are interesting optimizations of the AdaBoost algorithm depending on the kind of application tracked and classifier used, and independently of a sequential or parallel implementation. But we claim it is important to distribute the original AdaBoost algorithm: (1) in order to be able to speed up and size up any kind of boosting applications, (2) to run quickly the reference algorithm when designing an optimized version requiring some long comparisons. Moreover, when designing modified distributed versions of AdaBoost to process multi-site databases, with a fine result merging policy, each database processing can remain long. Then, to parallelize each local boosting algorithm without introducing more change in the algorithm is mandatory to achieve both a low level of errors and reduced execution times.

2. STRATEGIC CHOICES

This project results from a collaboration between researchers in signal processing and parallel computing, in order to design and implement an efficient parallel AdaBoost algorithm. However, future applications using this framework, with various kind of classifiers, will be developed by users: e.g. signal processing researchers, without assistance from expert of parallel programming. So, we opted for an efficient and easy-to-use parallel programming framework: the *JavaSpace*[2], a virtual shared memory on clusters and Grids, associated with the Java programming

language. Our first experiments of JavaSpace (JS) exhibited very good performances and scalability on hundreds of processors in a large cluster or in a multi-site Grid[4, 5], and an ease-to-use parallel programming paradigm.

JavaSpace is a Jini service enabling programs to exchange objects through a virtual shared memory. It is an evolution from a work done nearly two decades ago at Yale University with the Linda system but benefiting from Jini and Java object-oriented paradigm and portability. Several commercial implementations proposing various associated tools and demonstrating different performances exist. The JavaSpace API proposes three main methods: *write* to put an object into the JavaSpace, and *read* and *take* to retrieve a copy of an object or the object itself from the JavaSpace. These methods operate according to two modes: blocking or non blocking. Object retrieval is done by matching a template (*associate lookup*): a program might indicate the class of the desired object, and optionally the value for some of the object's attributes. This API is both simple yet rich enough to enable fast and easy development of a large range of distributed applications, in particular those with frequent and not totally regular communication requirements.

Finally, sequential and intensive computations (run on each processor) can be implemented in Java too, or in C/C++ and interfaced with the Java high level part of the application using JNI. For now, all our experiments have been implemented with Java.

3. SOFTWARE ARCHITECTURE

Our final aim is to use AdaBoost for feature selection, that is training a large number of similar weak classifiers with different input feature vectors so as to select the most discriminant features round after round. We therefore need to run independent training processes on the same dataset and subsequently compare the weighted error of each learner. The software architecture must be as generic as possible so any kind of base learner can be used (even heterogenous learner sets should be possible). In addition, we argue that the Java language is within the reach of most of developers and scientists. This way, researchers willing to test new learning algorithms or new features could just "plug" their java code in the parallel AdaBoost application skeleton presented here.

To do so, one needs to provide a class which extends our `WeakClassifier` class and to implement the methods `train(Vector<Example> examples)` and `detect(Example example)`. One must also provide a class which extends our `Example` class. Eventually, the `Master` and `Worker` classes contain sections which create the examples list and the initial list of weak classifiers. A generic parallel version of the algorithm presented in Table 1 has been implemented this way, separating the application-dependent code from the general AdaBoost code.

4. PARALLELIZATION STRATEGIES

The proposed parallel algorithm is based on a *master-workers* pattern (one *master* and P *workers*). The P *worker* processes have to be deployed on P different *processors*, or cores. In addition, the master process makes some computations in parallel of the workers, and they all concurrently write, read and take objects in the JavaSpace. So, the master process and the JavaSpace services are installed on individual proper processors.

Figure 1 details the parallel and distributed AdaBoost algorithm proposed in this paper. The left part introduces the Master code and the right part introduces the code of a worker node. The central part of figure 1 illustrates the data stored and accessed in the JavaSpace (i.e: the virtual shared memory server). At the early beginning, each process *discovers* the JavaSpace (using a *look up* service). Then the master writes in the JavaSpace the path to the dataset used to train and test the classifiers, and distributes the total amount of jobs on the P worker processes: i.e. writes in the JavaSpace the P intervals of classifier index that the P nodes have to retrieve and to process. Each interval is written in the JavaSpace, and each worker process takes from the JavaSpace one of these interval bounds. Currently, all cluster nodes are assumed homogeneous and all tasks have the same size. In a future version the master process will insure load balancing on heterogeneous clusters, setting on the most powerful nodes more classifiers to train or some CPU-consuming classifiers (for example, writing in the JavaSpace some interval bounds associated to node IDs or to specific node features, in order for each node to retrieve a suited task). The workers end this initialization step building and setting their classifiers and then enter the computation loop of the AdaBoost algorithm.

During one round, each worker trains its weak classifiers on the entire training dataset, identifies its best base learner and writes it in the JavaSpace with its weighted error. The master process waits to retrieves these P best candidates, compares their respective weighted errors and selects the overall best classifier (*competitive policy*). Then, the master process writes the current best classifier, and retrieves from the JavaSpace the best one at the previous round (cleaning up the JavaSpace). Each worker reads this new best classifier, updates its example weights (to focuss on misclassified examples) and enters the next round, training its classifiers again, according to the new examples weight distribution. In parallel, the master process adds the new base learner to the strong classifier aggregate, evaluates this strong classifier on the training set, tests if the targeted error is reached, and decides whether to stop the distributed program or to carry on with the next round.

This parallel algorithm follows a shared memory paradigm, but totally respects the initial sequential algorithm, and deploys it on a distributed architecture (ex: PC cluster) using the JavaSpace virtual shared memory.

5. APPLICATION AND TESTBED

The proposed algorithm has been tested on a standard and widely used AdaBoost application: the "Viola-and-Jones" face detector [9] (Referring to [9], the cascade architecture has not been implemented in this work and the integral images are computed in an initialization phase). In this application, a set of very simple features such as Haar filters response, i.e. binary filters requiring only additions on some connexe pixels intensity, are used to detect faces in real time in an image sequence (typically 24 images/s). The weak classifier training process therefore consists in computing the response (a scalar number) of a large number of such Haar filters (typically more that $130,000$ filters) on a large dataset (typically more than $10,000$ 24x24-pixel images) and in finding a threshold separating positive and negative examples according to this response at each round. The weak learner is therefore a linear separator which input is the response of a Haar filter. The filter (and its associated threshold) providing the minimal weighted error is then selected. The algorithm is run so as to achieve a 5% error rate on the training set ($\xi = 0.05$).

This particular application generates a lot of intermediate results that have to be kept in memory or serialized on disk if necessary. Their number grows with the size of both the data set and the feature set. Some local optimization for the serialization of those results when required should be done by the weak learner designer (i.e. the user), since it is highly application dependent. Yet, this is not in the scope of this paper.

We experimented the boosting application to face detection on a 32-PC cluster (P-IV, 3GHz, 1GB) interconnected with a Giga-

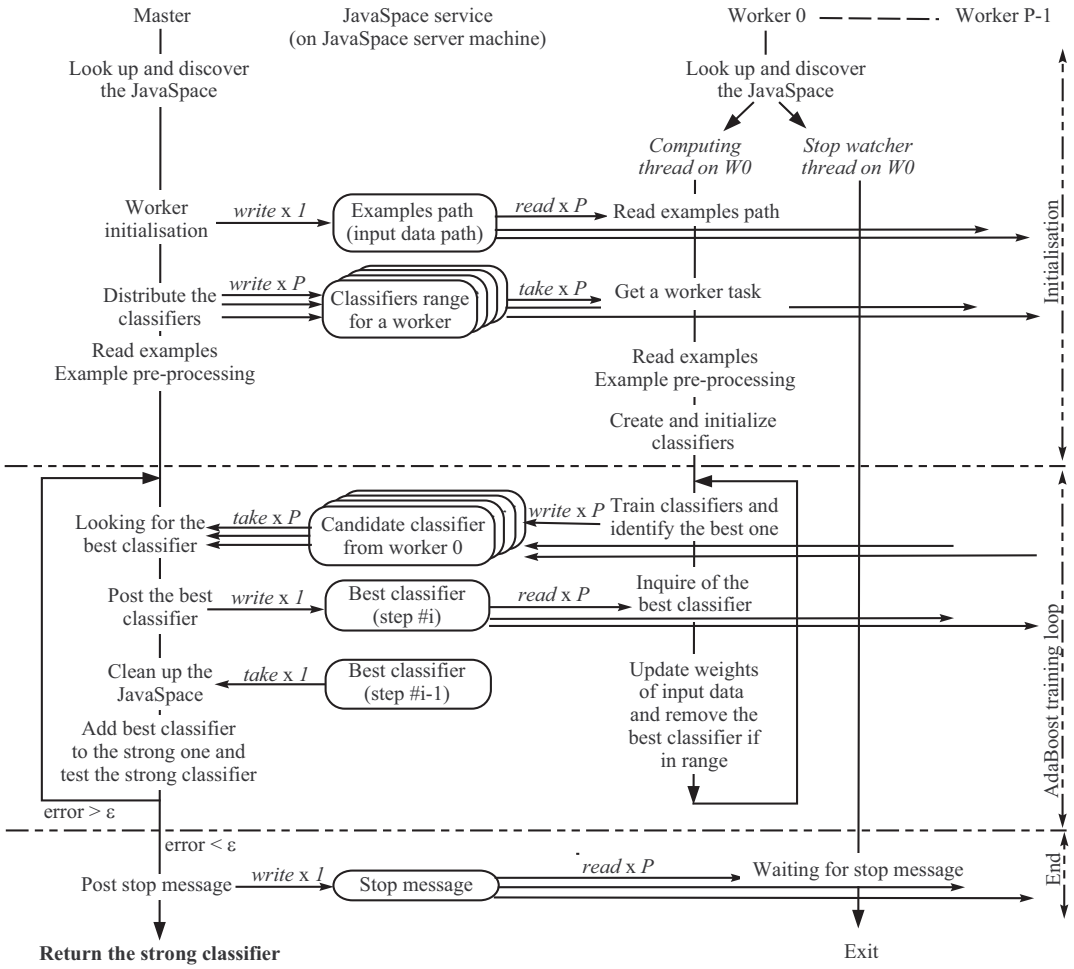

Figure 1. Details of the proposed parallel and distributed AdaBoost algorithm

bitEthernet network (two 24-port switches, with double interconnection links). These PC were running under Linux, and we used the SUN JVM and the SUN JavaSpace (named *Outrigger*). All benches were run in exclusive mode (no other applications running on the cluster), and data files were previously stored on the cluster PC disks.

6. EXPERIMENTAL PERFORMANCES

Figure 2 show the execution time decrease and the speed up increase of the face detection training, applied to a fixed size problem (134, 736 elementary classifiers and a 1000 image dataset), and implemented with a JavaSpace according to the distributed AdaBoost algorithm introduced in section 4. This parallelization uses P worker nodes and 2 other nodes to host the master process and the JavaSpace services. First, we compare the parallel execution time to a purely sequential version to adopt an *end user* point of view, and we observe our parallelization leads to a great decrease of the execution time up to 28 workers, and to a speed up very close of the ideal speed up ($S(P) = P$) considering the number of worker nodes (P). Second, we compare our parallelization using a JavaSpace with P worker nodes (and 2 other nodes) to a version using a JavaSpace with only 1 worker node (and 2 other nodes). This speed up is still very good and the parallelization seems to scale up to 28 workers. So, we have investigated this scalability in detail.

Plot 3 illustrates the execution time of one round when both the problem size and the number of processors increase. This plot focusses on the scalability of the distributed approach.

- When processing a larger problem it is possible to maintain the round duration constant, increasing the computation resources. For example, this parallelization insures the round execution time can remain close to $100s$ when training $134,736$ elementary classifiers per round on a database growing from 1000 up to 8000 images, using from 3 up to 27 processors.
- Moreover, the required computation resources increases linearly with the amount of computations resulting of the increase of data. The algorithm of section 4 shows this amount of computation grows linearly with the dataset size, and figure 3 shows the required number of processors increases linearly as well (approximately). So, our parallelization does not introduce significant losses, and leads to good scalability.

These results help the user to plan his/her experimental setup without requesting systematically too much computing resources.

7. CONCLUSION AND PERSPECTIVES

This paper presents a distributed framework allowing the use of the AdaBoost algorithm for feature selection on a wide range of candidate features. Results are presented on a face detection task

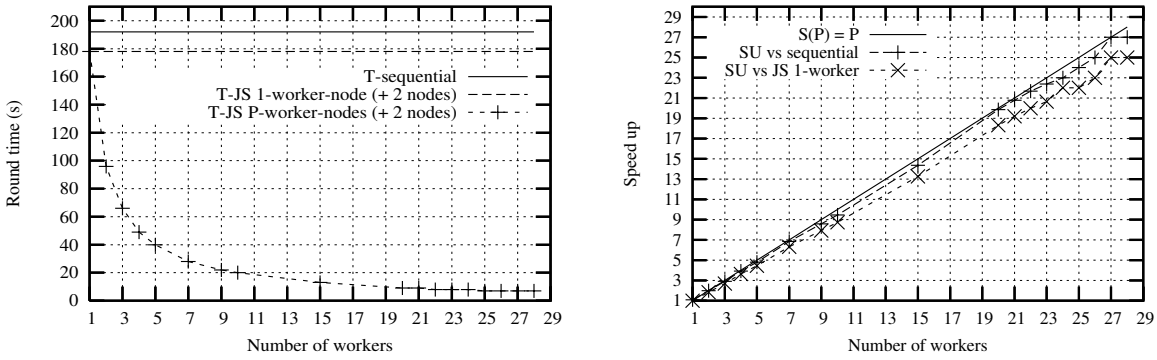

Figure 2. Execution time and speed up of the AdaBoost computation loop

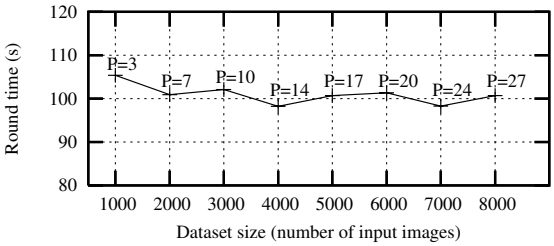

Figure 3. Execution time of one AdaBoost round when increasing the number of processors accordingly to the dataset size

using a linear separator as a base learner and more than $130,000$ Haar filter responses as candidate features. This new distribution strategy and implementation of the AdaBoost algorithm allows the speed up of the selection in a large feature space, using a local database for training. In the near future, we plan to use this framework with more complex base learners (such as ANN or SVM) and more numerous candidate feature vectors. Such experiments are extremely time consuming and can only be envisioned thanks to parallel computing. Thus, this parallelization does not track to process huge databases in production environment as in [6], but to improve the relevance of information extracted from existing databases. This is of great help in object recognition applications, for example, where databases are uneasy to obtain since they are often manually annotated.

From a pure parallel computing point of view, the next steps will consist in experimenting our distributed architecture on larger clusters, and to add some load balancing mechanisms (see section 4) to run simultaneously on several clusters of Grid'5000 (the French experimental Grid). Moreover, we aim at experimenting some variants of the proposed parallel algorithm of section 4, that would have no impact on a small cluster but should improve performances on very large clusters and Grids.

It is likely that a C/C++ based parallel development (using well known MPI library [8] or UPC language[1]) would lead to lower execution times. But, development times would increase, especially when non computer science experts will have to develop and plug their own code in our distributed application skeleton. So, we opted for Java and the JavaSpace framework, that seem to be a good compromise between execution performances and development time.

Acknowledgment:

Authors want to thank Region Lorraine that has supported a part of this research.

8. REFERENCES

[1] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study supercomputing. In *SC02: Proceedings of the Super Computing conference*, Baltimore, USA, 2002.

[2] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practive*. Pearson Education, 1999.

[3] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the International Conference on Machine Learning*, pages 148–156, July 1996.

[4] V. Galtier. Distributing a n-body problem algorithm at large-scale over a multi-sites grid using javaspace. In *Cracow Grid Workshop'06 (CGW'06)*, October 2006.

[5] L. Henrio, V. D. Doan, G. Boss, F. Baude, S. Vialle, V. Galtier, and S.Bezzine. A fault tolerant and multi-paradigm grid architecture for time constrained problems. application to option pricing in finance. In *e-Science 2006*, Amsterdam, Netherlands, dec 2006. IEEE CS Press.

[6] A. Lazarevic and Z. Obradovic. The distributed boosting algorithm. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, NY, USA, 2001. ACM Press.

[7] F. Lozano and P. Rangel. Algorithms for parallel boosting. In *ICMLA '05: Proceedings of the Fourth International Conference on Machine Learning and Applications (ICMLA'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[8] P.S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[9] P. Viola and M. Jones. Robust real-time object detection. In *Proceedings of the Second International Workshop On Statistical And Computational Theories Of Vision Modeling, Learning, Computing, And Sampling*, 2001.