2009 15th International Conference on Parallel and Distributed Systems

Implementation of the AdaBoost Algorithm for Large Scale Distributed Environments:Comparing JavaSpace and MPJ

Virginie Galtier and Stéphane Vialle
Supélec
2 rue Edouard Belin
F-57070 Metz
{*virginie.galtier,stephane.vialle*}*@supelec.fr*

Stéphane Genaud
AlGorille Team - INRIA
Campus Scientifique - BP 239,
F-54506 Vandoeuvre-lès-Nancy, France
stephane.genaud@inria.fr

***Abstract*—This paper presents the parallelization of a machine learning method, called the adaboost algorithm. The parallel algorithm follows a dynamically load-balanced master-worker strategy, which is parameterized by the granularity of the tasks distributed to workers. We first show the benefits of this version with heterogeneous processors. Then, we study the application in a real, geographically distributed environment, hence adding network latencies to the execution. Performances of the application using more than a hundred processes are analyzed in both JavaSpace and P2P-MPI. We therefore present an head-to-head comparison of two parallel programming models. We study for each case the granularities yielding the best performance. We show that current network technologies enable to obtain interesting speedups in many situations for such an application, even when using a virtual shared memory paradigm in a large-scale distributed environment.**

***Keywords*-Adaboost; MPJ; JavaSpace; Java; Grid Computing;**

I. Introduction

A. AdaBoost Algorithm Overview

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 [1] is an iterative algorithm that selects an optimal combination of elementary features among a large set of candidates for a two-class separation problem. 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 iteration. To ensure the optimality of the weak classifier combination, the training examples are assigned weights that change from iteration to iteration. The previously misclassified examples are assigned higher weights and the training error of each weak classifier at a given iteration 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 iteration. The selected weak classifiers are therefore more and more focused on the misclassified examples. The complete AdaBoost algorithm is described in Table 1.

Given training examples $\{(x_i, y_i)\}$, $i = 1, .., n$ where:
$x_i \in X$ are examples,
$y_i \in \{-1, +1\}$ label positive or negative examples,
and a set of weak classifiers $\{ h_j\}$, where $h_j : X \mapsto \{-1, +1\}$
Initialize step $t = 1$, $error = 1$, and example
distribution $\forall i,\ D_t(i) = 1/n$
while $error > \xi$ **do**
Train all weak classifiers using D_t and find the best
classifier h_t which produces the minimum error:
$h_t = arg\min_{h_j} \epsilon_j$, where $\epsilon_j = \sum_{i:h_j(x_i)\neq y_i} D_t(i)$
Update distribution for next step using the best weak classifier:
$D_{t+1}(i)=\frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{\sum_i D_{t+1}}$, where $\alpha_t=\frac{1}{2}\ln(\frac{1-\epsilon_t}{\epsilon_t})$
Compute strong classifier from best weak classifiers and error:
$H(x_i) = sign(\sum_t \alpha_t h_t(x_i))$ and $error= \sum_{i:H(x_i)\neq y_i} \frac{1}{n}$
$t = t+1$
Output the final hypothesis $H(x) = sign(\sum_i \alpha_t h_t(x))$

Figure 1. *AdaBoost* Algorithm

B. Motivations

The final goal of the application parallelization is to allow signal processing researchers to quickly test performances of new classifiers trained by the AdaBoost algorithm. In a previous work [2], we have proposed a parallelization using JavaSpace, with a static distribution of the computations. This solution exhibited a good speed-up on a small homogeneous cluster (30 nodes). This approach is summarized in section II-A. JavaSpace is a virtual shared memory Jini service, and the Sun implementation we used (called *outrigger*) only provides a centralized shared memory, which could constitute a bottleneck. When time to move to a larger scale came, we wondered whether the centralized nature of JavaSpace would put a limit on performance, compared to

1521-9097/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPADS.2009.67

some other programming models such as message passing. The execution environments we target must be larger than just a local cluster, and as a result, heterogeneity in the computing resources must be considered. However, our users can get an exclusive usage of these resources, so that the heterogeneity does not vary in a significant way during execution. Finally, while weak classifiers used so far exhibit similar computing requirements from one to another, the framework aims at enabling the plug-in of any kind of classifiers. Some classifiers could be more CPU-consuming than others, making difficult to predict beforehand how computing time will vary. For all those reasons, we propose a second version of our distributed AdaBoost algorithm, performing an adaptive distribution of the weak classifiers. To implement it, we naturally extended our previous JavaSpace development, but to compare its scalability, we also developed the application in the message passing programming model with P2P-MPI.

This paper is structured as follows: section II explains the principles of the parallel algorithm, and its load-balancing strategy using an adaptive distribution. Section III presents some related works. In section IV, we briefly introduce JavaSpace and P2P-MPI, and we explain how they are used to implement the algorithm. In the second part of the paper (section V), we carry out three experiments. The two first experiments show results on a single cluster to validate the adaptive approach and the scalability of the parallelized implementation (up to 256 processors). The third experiment describes results in a large scale distributed environment (120 processors on three distant sites). Finally, concluding remarks and future works are presented in section VI.

II. Adaptive Parallelization Strategy

A. Static Distribution

All our parallel versions of the AdaBoost algorithm follow a master-worker pattern. In the first version, where the P workers are supposed homogeneous and where all classifiers are supposed to require sensibly the same amount of CPU-time, the master divides the N classifiers into P intervals of N/P classifiers. Because all workers are in charge of the same number of classifiers (excepted for one worker that is also assigned the remaining), we call this distribution a *static* distribution.

Both master and workers get a local copy of the database of E training examples. During the initialization step, every worker pre-processes the examples and retrieves a range of classifiers specified by the master. Then, at each step, a worker starts by training the classifiers it is in charge of on the examples database, and next it lets the master know which of its classifiers exhibited the smallest error (this classifier constitutes a *candidate*). It then waits for information from the master stating which of the P candidates was the best. If that classifier was within the range it is in charge of, it removes it from its list. In any case, it updates example weights according to the performance of the best candidate of that step before starting a new step. The master, in addition to the operations already described (setting-up intervals of classifiers, selecting the best candidate and informing the workers) also updates at each step the final strong classifier by combining the best candidates and computes the error of that strong classifier on the training database of examples. When the error falls below a given ϵ value, the master orders the workers to stop and delivers the complex classifier.

B. Adaptive Distribution

Our proposal for a dynamic load-balancing of the application is the introduction of an *adaptive* distribution during the first step. The user can specify the granularity of the distribution by setting the size S ($S \leq N/P$) of the intervals of weak classifiers distributed by the master. If $S = N/P$, the distribution is equivalent to the static distribution. Such an interval is called *chunk* in the following. The initialization and first step of the previous algorithm are modified as follows: each worker pre-processes the examples, and retrieves a first range of S classifiers. It trains those S classifiers, finds out which one performs the best, and tries to retrieve another set of S classifiers. If there are classifiers left, it repeats the previous step and updates its candidate if necessary. If there are no more classifiers left, it informs the master of its candidate. The following steps are similar to the previous algorithm. Thus, during this adaptive step, each worker requests greedily and repeatedly a share of weak classifiers. Consequently, faster workers end up with more classifiers. This assignment of the first step is then kept for the entire computation. Figure 2 depicts the modified version of the algorithm.

Given our target execution platform, we assume that the processing capabilities of the resources do not vary during execution (no external load) and therefore a single adaptive step is sufficient to balance the load over the processors. Note also that a) a given classifier always leads to the same operations, no matter the weight of the examples (which is the only changing data from one iteration to the next), and b) even though at each iteration one of the workers removes one classifier, it has a negligible effect on the computation time given the large number of classifiers.

The remaining causes of load-unbalance, that we address using this method are:

- network heterogeneity: if we distribute workers over several sites, the requests of remote workers will arrive later than those from workers close to the master.
- processor heterogeneity,
- load heterogeneity: all classifiers do not require the same amount of computation, as will be detailed in Section V-A (for instance, in Figure 2, classifiers E and F takes longer to train than classifiers a b c d g or h).

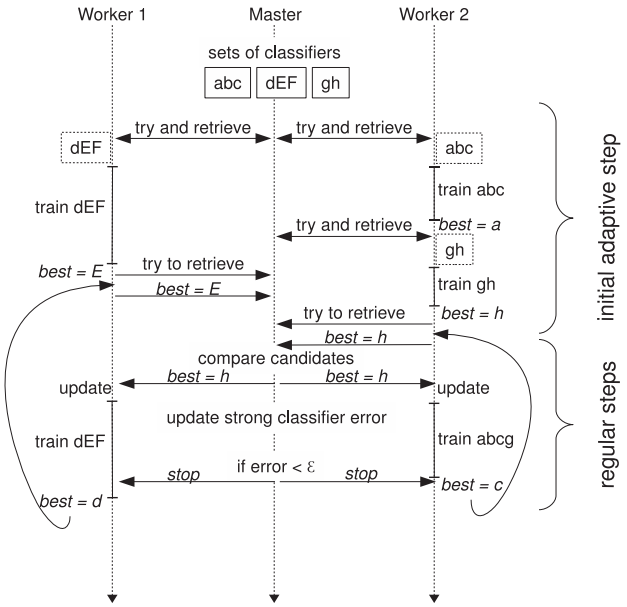

Figure 2. Distributed AdaBoost Algorithm with Adaptive Initial Step

This adaptive scheme results in a load-balancing, whose efficiency depends on the granularity of the distribution (i.e choosing the chunk size S): too large chunks lead to uneven finish times between workers, while too small chunks add a communication overhead caused by the numerous work requests.

III. Related Work

To the best of our knowledge, there are few results regarding the parallelization of AdaBoost to reduce the learning phase running time. In [3], the authors propose a method of parallel boosting that operates on distributed databases. Their objective is to overcome the problem of a massive database that cannot fit into main computer memory by distributing the learning process on disjoints datasets. Another work ([4]) with the same objective proposes to train multiple base learners simultaneously. This work focuses on several heuristic methods to find a set of (statistical) distributions that can be generated independently in advance of the training process. This work outlines general methods but does not concentrate on implementation issues and performances as we do.

Regarding the load-balancing issue, a considerable amount of research work has addressed this problem in the case of master-worker applications. Many algorithms have been produced to compute optimal *static* distributions of tasks to workers. The divisible load theory [5] has established results for various network topologies. However, this theory assumes that the work can be arbitrarily partitioned and that each part has a linear cost. Some other works (e.g [6], [7]) model the computation and communication costs as functions of the processor or network link used, and propose dynamic programming algorithms to compute an optimal static solution.

When the computation cost of each load element varies, the problem of computing an optimal distribution is intractable in practice. Even an exhaustive search of solution cannot tackle the variations bound to system-level fluctuating parameters (e.g cache misses, network cross-traffic). Hence, a dynamic load-balancing strategy is more suitable. A general framework for such load-balancing in grid environments has been proposed early by Goux et al. [8]. We could have used this system if the interface allowed linking with Java.

IV. Parallel Implementations

One contribution of our work consists in a head-to-head comparison of two programming models, starting from the same sequential Java application and parallelization strategy.

A. *JavaSpace and* P2P-MPI

An evolution from the two-decade old Linda system, but benefiting from Jini and Java object-oriented paradigm and portability, JavaSpace [9], is a Jini service enabling programs to exchange objects through a virtual shared memory. Object retrieval is done by matching a template (*associate lookup*). The proposed API is simple, yet rich enough to enable fast and easy development of loosely coupled distributed applications. Several implementations (both commercial and free open-source) proposing various associated tools and demonstrating different performances exist, and we used the Sun's implementation (*outrigger*) provided with the Jini Starter Kit.

P2P-MPI [10] is a framework designed for running message passing programs in large scale heterogeneous distributed environments. On one hand, it is a middleware system which offers system-level services to the user, such as finding requested computing resources, transferring files, launching remote jobs, etc. On the other hand, it provides programmers with a communication library which follows the message passing programming model. More precisely, P2P-MPI contains an MPJ (Message Passing for Java) [11] implementation. MPJ is a recommendation from the Java Grande Forum, and proposes an adaptation of the MPI specification [12] for Java. The API primitives are based on blocking or non-blocking send and receive operations.

B. *AdaBoost Implementation*

To make a fair comparison of the performances, we have developed an MPJ version that mimics the JavaSpace program structure (see Figure 2). Like in the JavaSpace version, the MPJ program has one process exclusively playing the master's role of distributing classifier chunks to workers.

In JavaSpace, the master puts all the chunks in the space, and workers issue a take of one chunk each time they need work to do, consuming the chunks. When a worker has finished to compute the classifiers from its current chunk, it writes the best weak classifier (candidate) to the space. The master reads these best weak classifiers and once all have been returned by workers, the master writes to the space which is the best for the iteration.

In MPJ, the master asynchronously waits for work requests from any worker. When a worker sends a work request, it embeds in the message the best weak classifier it has found in the previous chunk (except for the first work request). Immediately after a request is received, the master ranks the result received, and sends the next chunk to be computed to this worker. This is iterated until all chunks have been distributed. Once all chunks have been computed, the master broadcasts to all workers the best weak classifier for the iteration.

On one hand, JavaSpace proposes a more abstract and concise style for communications. Comparatively in MPJ, we have to use different tags to differentiate the messages. For example, after a work request, a worker must distinguish if the answer received contains a new chunk or the final result for the iteration. On another hand, MPJ offers more control over the way the communications are made. For example, it is noteworthy that MPJ can semantically express a broadcast while JavaSpace cannot. MPJ has a specific primitive (Bcast), which enables the communication library implementation to optimize this collective communication (e.g use of a binomial tree in P2P-MPI).

V. Experiments

The experiments objectives are to observe the behavior of the two frameworks on the AdaBoost application, with two different execution platforms: one single cluster, and a set of processors taken from three clusters distributed at a large geographical scale. Our first and second experiments consist in testing the application speed-up on the single cluster, and then to assess the validity of the adaptive strategy when the CPUs' loads are unbalanced. In the third experiment we evidence the effects of wide-area latency network communications on the execution by using three clusters at different sites. Before describing the experiments in these respective environments, we detail the application dataset used in all experiments.

A. Dataset Characteristics

The proposed algorithm has been tested on a standard and widely used AdaBoost application: the "Viola-and-Jones" face detector [13]. In this application, a set of very simple features such as Haar filters response, i.e binary filters requiring only additions on some adjacent 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 (134,736 in our case) on a large dataset (8,500 24x24-pixel images in our case) and in finding a threshold separating positive and negative examples according to this response at each iteration. 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. In our experiments, the algorithm is run so as to achieve a 3% error rate on the training set ($\epsilon = 0.03$).

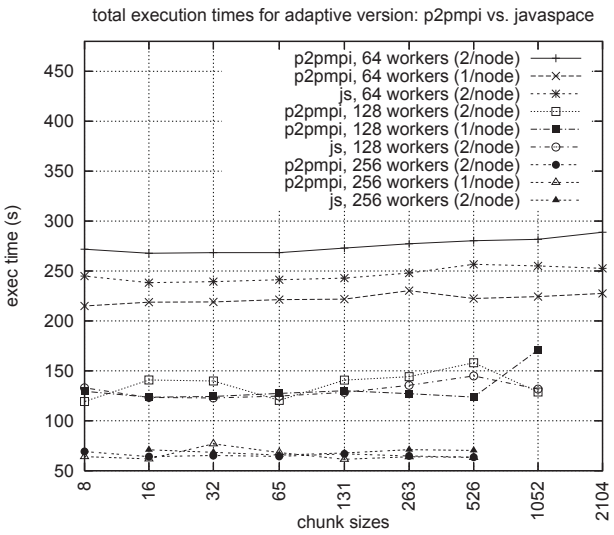

Figure 3. exec. times on a homogeneous cluster.

We must notice that each Haar filter has a different computation cost, depending on the filter type and location in the image. Hence, it is important to have filter types homogeneously scattered over the workers to average the load of chunks assigned to workers and avoid load imbalance. We have examined the input dataset of filters set and found that it was the case. To double-check this, we have also conducted experiments using several shuffled datasets, in which filters are randomly permuted. Overall, runs with shuffled datasets exhibited comparable execution times.

B. Experiment 1: Homogeneous Cluster

For this first experiment, we use a specific cluster to get up to 256 nodes. Each node hosts a bi-core processor Xeon-3075 2.66GHz, 4GB RAM, and the interconnect is a Gigabit Ethernet network with a CISCO 6509 switch. Figure 3 shows the performance of the two frameworks for 64, 128 and 256 workers depending on chunk size (right-most chunk size is equivalent to the static version). In all tests, we map exactly one worker per core. Some tests were run using only one core per node to observe effects of memory contention. We observe the same differences with both P2P-MPI and JavaSpace for 64 workers (for a sake of room

workers	JavaSpace static	P2P-MPI static	JavaSpace adaptive	P2P-MPI adaptive
64	40	56	30/49	56/60
128	88	86	64/94	75/104
256	169	168	123/177	167/207

Table I
SPEED-UPS

we only plot the comparison for P2P-MPI). With twice or four times more workers, either mapping lead to equivalent performances, probably because each worker makes less memory accesses. For each number of workers, we choose different chunk sizes, starting from one equivalent to the static version. For example, for 64 workers we start with $\lfloor 134736/64 \rfloor$=2105 filters per chunk. The chunk size is then recursively halved. We see that using smaller chunks makes very little difference in the performance results. There are three reasons explaining this observation: a) the computation load required by the dataset is well scattered over workers, b) the processors are homogeneous, and c) the chunk sizes we test are all divisors (or nearly) of the total number of filters per worker. We have checked that choosing a bad chunk size on purpose significantly increases the execution time. For example, a chunk size of 2072, which represents an even share for 65 workers, instead of 2105 for 64 workers leads to a 51% increase because one worker is assigned two chunks while all others have one. The speed-ups obtained are reported in Table I, relatively to the sequential time: 11598s (more than 3h). For runs with the adaptive distribution, we report the worse/best speed-ups. These results show a good scalability of the program on this cluster.

C. Experiment 2: Heterogeneous Cluster

We then artificially introduce some heterogeneity in the processors, by permanently running an auxiliary program on half of the CPUs used. When the chunk size is the same as in the static version (right-most chunk size values), the performance (Figure 4) largely suffers from the processor load imbalance. The adaptive version achieves a good load balance with many different chunk sizes: in the experiment, the load balance seems to be reached for chunk sizes from $\frac{1}{4}$ to $\frac{1}{128}$ the chunk size of the static version.

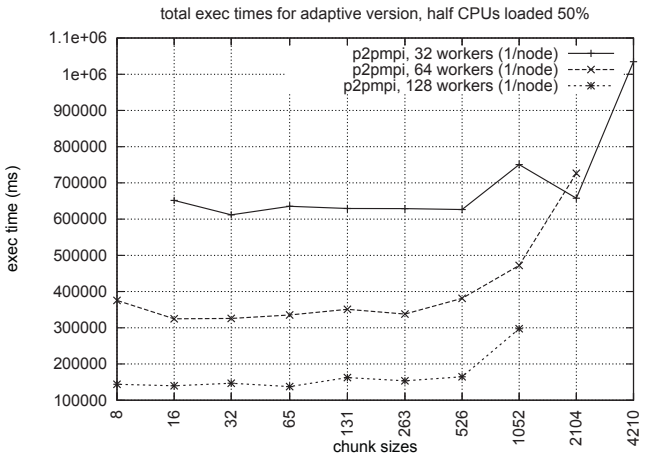

Figure 4. Exec. times on a simulated heterogeneous cluster.

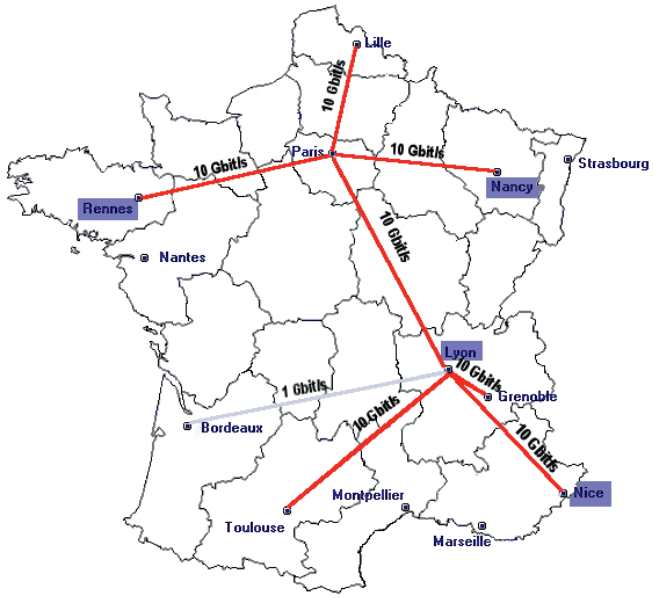

Figure 5. Locations of clusters used in experiment.

D. Experiment 3: Grid

1) Experiment Setup: In this experiment, we want to characterize the effects of wide-area network communications on the execution. We use Grid'5000 [14], a highly controllable experimental grid platform.We place 40 workers processes on each of three remote clusters (Nice, Rennes, Nancy, several hundred kilometers apart on the map shown on Figure 5). The master (and the JavaSpace service in the case of JavaSpace) is in a fourth distant cluster (Lyon). The selected clusters have dual-core nodes, and apart from the node running the master and the Jini services, each node runs two workers, for a total of 120 workers. The four clusters are linked through a 10Gbps backbone. Each node is connected through a 1Gbps link to the router, which can aggregates 10Gbps to the backbone. The RTT (ping) from the site hosting the master and Jini services to the other clusters are 10.5ms, 12.5 and 17ms. We have chosen clusters whose hardware is as similar as possible in order to isolate the effects that are related to network. In each cluster, all selected nodes are dual-core Opteron 2GHz, 2GB RAM, either embedded in a HP ProLiant DL145G2 chipset (clusters in Rennes and Nancy), or in an IBM eServer 325 chipset (clusters in Nice and Lyon). However, the performance of the two hardware differ from about 10%

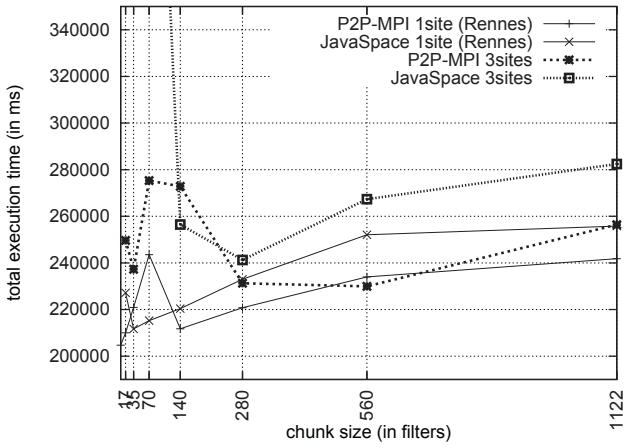

Figure 6. Total execution times

accordingly to the SciMark benchmark,whatever the JVM used.

2) Comparative Performances: Figure 6 presents the total execution times for JavaSpace and P2P-MPI when varying the chunk size. The boldest, dashed lines shows the multi-site execution times. We also show for reference, the times obtained on only one of the three cluster (Rennes), plotted as thin, solid lines. The first observation is that the JavaSpace performance is roughly 10% less than the one of P2P-MPI in most cases. Thus, this result is overall encouraging for using a virtual memory paradigm for this kind of application even at a large scale. The second observation is that performance in the multi-site configuration is competitive with the single cluster configuration. The JavaSpace execution on multi-site could reach 93% of the performance of the single cluster execution (chunk size 280) and the multi-site P2P-MPI executions are very close on average to their cluster counterpart for large chunk sizes. The third observation is that executions with these clusters are more sensible to the chunk size choice. On the Rennes cluster, the execution time can drop by 13% and 17% with P2P-MPI and JavaSpace respectively, relatively to the static distribution (it was only 7% and 6% with 128 workers in Section V-B). The drop is even bigger for P2P-MPI on multi-clusters (up to 20%), while it remains limited for JavaSpace (15%) because the range of chunk sizes involving a performance increase is narrower.

3) Detailed Analysis: However, the above results show that the wide area communications have little impact in this application. Provided we choose the right chunk size, the performance is mostly governed by the CPU power. Yet, a bad choice for the chunk size may lead to a performance collapse, like in the situation observed for a chunk size of 70 in JavaSpace, which takes about 550s.

In order to understand the performance differences for such a chunk size, we must examine the activity of each worker.

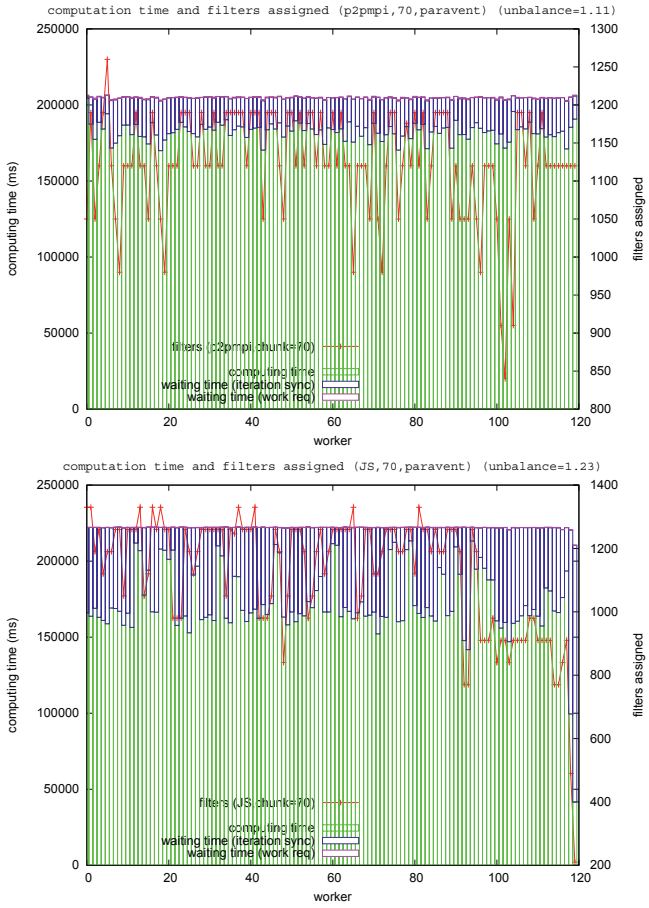

Figure 7. Compared behavior on one cluster, chunk=70

In each graph of Figures 7 and 8, we draw the time each worker (identified by its number on the horizontal axis) spent computing (green, bottom area) and communicating (blue, upper area). These times are on left vertical axis. We superimpose the number of filters assigned to each worker as a red line (to be read on the right vertical axis).

We define an indicator to characterize the load balance of an execution with w workers. Let C_i be the time spent on computations by a worker i, the load balance indicator is $max_w(C_i)/average(C_1, \ldots, C_w)$. The executions on only one cluster is shown in Figure 7. It shows an acceptable load-balance, with 1.11 and 1.23 for P2P-MPI and JavaSpace respectively. P2P-MPI spent less time waiting than JavaSpace, which suffers from its last worker being curiously underloaded (this phenomenon was observed several times). In the multi-site execution presented on Figure 8, we observe huge imbalances for JavaSpace, with a load balance indicator of 2.75. Some workers got as few as 210 filters to compute while some other got more than 3000. The P2P-MPI execution with that chunk size leads to its worst performance but does not reach such a level of imbalance,

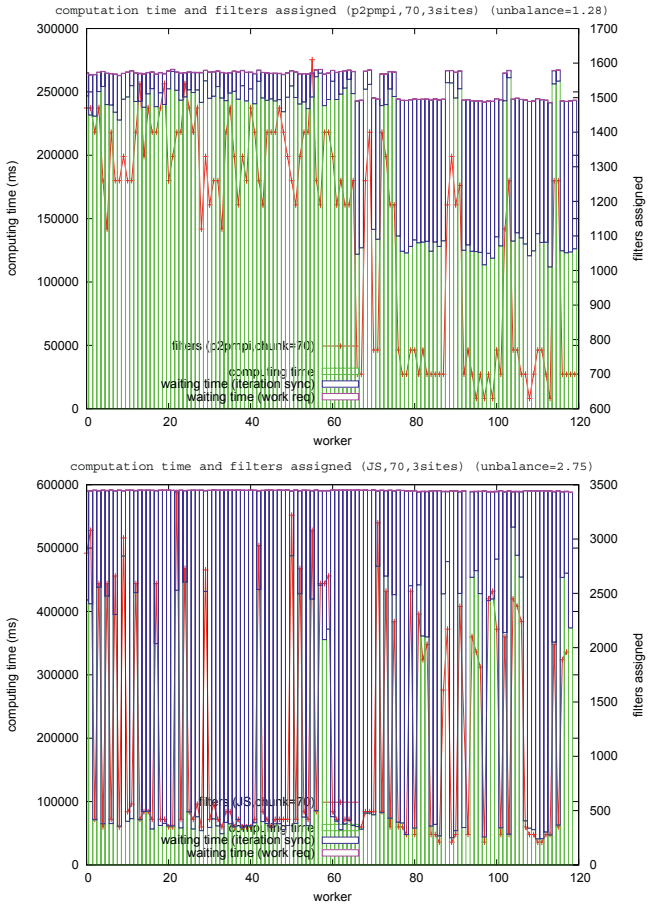

Figure 8. Compared behaviors for multi-site, chunk=70

with an indicator of 1.28. Our explanation for such a situation is that workers access the JavaSpace unequally in the presence of too many wide-area communications. As a result, some workers are starving while some other are overloaded. The difference in the behaviors of the two frameworks can be explained by the different messaging systems they employ. JavaSpace uses RMI while P2P-MPI uses the Java NIO class, which provides the equivalent of the `select` operation of *libc*. With NIO, P2P-MPI can concurrently monitor events occurring on a vector of sockets without creating one thread per socket. Finally, the reason for the sudden execution time increase is not due to an increase in the communication time but to a load imbalance.

E. Multi-site scalability

A quick note from a user point a view. Assuming a user has only 40 available local processors, the running time is for example in Rennes of about $600s$. He can wonder how much additional speedup he can get using extra remote resources. Here, using two remote sites with 40 other processors each, the running time is between $230-250s$. The gain is roughly 2.5 times for three times more processors. Hence we conclude it is worth using such grid resources with this application.

VI. Conclusion

In this work, we have achieved the parallelization of a boosting application through two programming models: one is the message passing model, used through the MPJ implementation P2P-MPI, and the other is the black-board model with the JavaSpace implementation outrigger. We have extended the parallelization to an adaptive version of the application to tackle heterogeneous environments. The application belongs to a large class of problems, for which the computation cost of a single element as well as the computation capabilities are not precisely known, making impossible to compute a schedule. Hence, our observations may apply to a number of similar problems. We have shown that the dynamic load-balancing scheme using an adaptive step is effective in an heterogeneous context. The experiment conducted in a large scale distributed environment shows that the application can be used in such conditions with interesting performances. Our comparison of the performances obtained with P2P-MPI and JavaSpace gives users helpful hints to choose the granularity for the load-balance (chunk size). Whereas we expected the latency to be an obstacle to fine grain load-balancing, we have learnt from this work that the current network technologies enable an unexpected large range of chunk sizes yielding acceptable performances from a user point of view. Applications with a structure similar to the algorithm presented here are hence good candidates for executions in large scale distributed environments.

Future work should study alternative approaches to avoid the bottleneck at the master, which is obviously a limit to the scalability of the application. Two possible directions are foreseen. We can replace the single master by a distributed set of masters coordinated in a hierarchical manner, as proposed in [15], or we can suppress the master and implement an election algorithm among workers so that they decide which one has the best weak classifier at each iteration.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners.

References

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

[2] V. Galtier, O. Pietquin, and S. Vialle, "Adaboost parallelization on PC clusters with virtual shared memory for fast feature selection," in *IEEE International Conference on Signal Processing and Communication*, November 2007, pp. 165–168.

[3] A. Lazarevic and Z. Obradovic, "The distributed boosting algorithm," in *7th ACM SIGKDD international conference on Knowledge discovery*. ACM Press, 2001, pp. 311–316.

[4] F. Lozano and P. Rangel, "Algorithms for Paralell Boosting," in *Proceedings of the 4th international conference on Machine Learning and Applications (ICMLA'05)*. IEEE Press, 2005.

[5] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible load theory: A new paradigm for load scheduling in distributed systems," *Cluster Comput.*, vol. 6, no. 1, pp. 7–17, Jan. 2003.

[6] S. Genaud, A. Giersch, and F. Vivien, "Load-balancing scatter operations for grid computing," *Parallel Computing*, vol. 30, no. 8, pp. 923–946, Aug. 2004.

[7] O. Beaumont, A. Legrand, and Y. Robert, "A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs," in *17th Intl Symp on Computer and Information Sc. (ISCIS XVII)*. CRC Press, Oct. 2002, pp. 115–119.

[8] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, "An enabling framework for master-worker applications on the computational grid," in *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE CS Press, Aug. 2000, pp. 43–50.

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

[10] S. Genaud and C. Rattanapoka, "P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs," *Journal of Grid Computing*, vol. 5, pp. 27–42, 2007.

[11] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox, "MPJ: MPI-like message passing for java," *Concurrency: Practice and Experience*, vol. 12, no. 11, Sep. 2000.

[12] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.

[13] P. Viola and M. Jones, "Robust real-time object detection," in *2nd International Workshop On Statistical And Computational Theories Of Vision Modeling, Learning, Computing, And Sampling*, Jul. 2001.

[14] F. Cappello *et al.*, "Grid'5000: a large scale and highly reconfigurable grid experimental testbed," in *6th IEEE/ACM International Conference on Grid Computing (GRID 2005)*, 2005, pp. 99–106.

[15] K. Aida, W. Natsume, and Y. Futakata, "Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm," in *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003, pp. 156–163.