Impact Assessment on the Parallel Performance of Node-Core Combinations in a Multicore Cluster Environment: A Case of Study

Cesar Fernández, Francisco Saravia, Carlos Valle, and Héctor Allende

Universidad Técnica Federico Santa María
Departamento de Informática
{cfernand,fsaravia,cvalle,hallende}@inf.utfsm.cl

Abstract. Multicore processors have opened new paths for improving the parallel performance in cluster environments. Nevertheless, the selection of different combinations between the amount of nodes and the number of cores per node implies different results in terms of parallel performance. We performed an impact assessment on the parallel performance of node-core combinations using a parallel approach of a machine learning ensemble algorithm. Our results reveal that two key factors for selecting a suitable node-core combination: the network capabilities and the workload distribution. We observed that the network interconnection limits the amount of nodes that can be efficiently used, due to the extra-node communications does not allow to keep scaling as the number of nodes is increased. The best results were obtained by reaching a balance between intra-node and extra-node communications. By the other hand, the parallel performance can be negatively affected when the workload distribution is not homogeneous among nodes.

Key words: Parallel Algorithms, Parallelism and Data Sharing on Multicore Architectures, Ensemble Learning, Local Negative Correlation.

1 Introduction

The new generation of multicore processor architecture delivers new possibilities of exploding the potentials of calculus, multimedia and parallel processing, but one of the limits has been undoubtedly to obtain real benefits respect of speed-up and performance. Although performance has been increased (theoretically), allowing to reach a better use of parallelism with the introduction of new features in assembly functions [13]. Multicore processors are being used in a big percentage by the most important High Performance Computing centers in the world [3], and many others have begun to migrate their single core processors to multicore processors [12].

Multicore technology has enhanced the *shared memory system* through optimizations both in new generation of multicore processors and in software, allowing the execution, debugging and monitoring of applications [2]. However, in

distributed memory system, the advent of compute nodes equipped with multicore processors has allowed to increase the computing capability, by increasing the amount of computing units in each node. By the other hand, this new technology involves to consider the different kinds of communications between processes, depending on whether they are running on different cores within the same node or different nodes [4].

Chai [9] and Zhang [26] present the communication schemes in a multicore cluster, as shown in Fig. 1, where the following kind of communications are described: *intra-processor*, that represents the internal communications among cores inside a processor (dashed lines), *intra-node*, that represents the communication among cores of the same node, but in different processor (dotted lines), and finally the *extra-node* communication among cores in different nodes (solid lines).

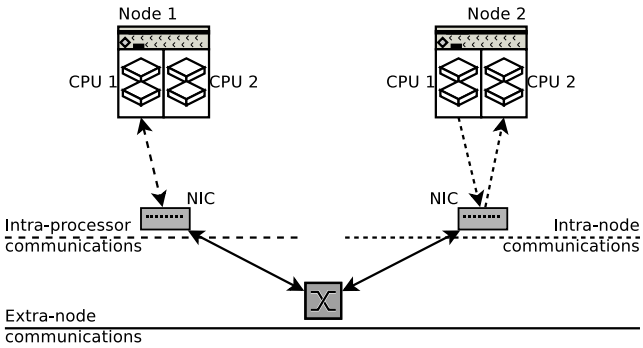

Fig. 1. Communications on a Multicore Cluster

The performance evaluation of parallel algorithms consists in analyzing the behavior of the algorithm from different perspectives [22], considering variations in both dataset size and number of processors used in the execution [16]. Aditionally, there are constraints in real situations that determinate the amount of available processors in a computer cluster, or limit the number of processors that can be used by an algorithm, such as a limited amount of processors in shared clusters, or algorithms with structural features that need to be run in a fixed number of processors.

Inside of a multicore cluster based on distributed memory system, the parallel execution using a fixed number of p processors – with the *Message Passing Interface* (MPI) [14] as the programming model – can be carried out by using several combinations between nodes and cores per node. Considering a homogeneous hardware environment, for a number of parallel tasks λ (homogeneous and independent from each other) executed in a cluster with a maximum number of nodes δ, where each node has the same number of cores μ, the *node-core*

combination for an execution with p processors is composed by the number of used nodes n and the number of cores used per node c:

$$p = n \cdot c \qquad 1 \leq n \leq \delta, \quad 1 \leq c \leq \mu \tag{1}$$

The equation (1) defines the set of combinations available to select p processors in a multicore cluster. This fact involves taking a decision about what node-core combination delivers the best performance, through the evaluation of the key features of the algorithm that can affect – positive or negatively – the expected performance.

For a number of tasks $\lambda \geq p$, two possible scenarios are generated: balanced workload among nodes, as shown in Fig. 2(a), where (a.1) represents the parallel tasks being executed using a single node, and (a.2) represents the same tasks being executed using two nodes, in both combinations the number of used cores is the same, varying the workload distribution. Unbalanced workload is shown in Fig. 2(b), where (b.1) and (b.2) are analogous with the previous case respect of the number of nodes used, but with different number of tasks in each node.

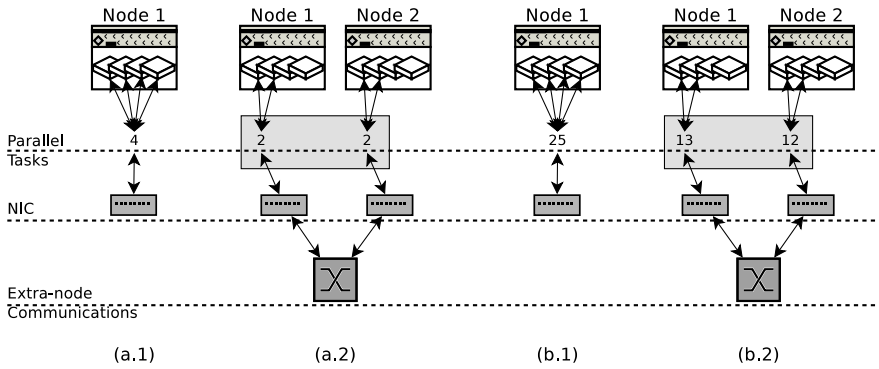

Fig. 2. Load Balance with $p = \{(1, 4), (2, 2)\}$, considering values of $\lambda = 4$ (illustrations (a) and (b)), and $\lambda = 25$ (illustrations (c) and (d))

Considering the previous issue, the selection of node-core combinations with $\lambda \geq p$ can be defined as follows:

$$\begin{aligned} p &= n \cdot c \qquad 1 \leq n \leq \delta, \quad 1 \leq c \leq \mu \\ \lambda &= \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} \left(\left\lfloor \frac{\lambda}{n} \right\rfloor + \varphi_i \right) \end{aligned} \tag{2}$$

where T_i represents the number of parallel tasks assigned to the node i and $\varphi_i \in \{0, 1\}$ represents the additional workload caused by the unbalance obtained when $\frac{\lambda}{n} \neq \lfloor \frac{\lambda}{n} \rfloor$.

According to the definition given by Breshears [6], the granularity is defined as *the amount of computations within a task* or alternatively as *the amount of*

computation done before synchronization is need. Based on previous definition, the granularity of an algorithm is a key factor of the parallel behavior in terms of scalability, according to the trade-off between processing time and overhead time caused by synchronization or communications among processes.

Given the degree of granularity, a parallel algorithm can be classified as follows: A *coarse-grained algorithm*, characterized by a high granularity, spends a minuscule fraction of the overall parallel execution in synchronization or communication, while *fine-grained algorithms*, characterized by a low granularity, spend a higher fraction of the execution time in synchronization. In distributed memory systems, a *coarse-grained algorithm* has better parallel performance due that overhead time represents a small fraction of the overall parallel execution time. By other hand, a *fine-grained algorithm* tends to produce bottlenecks in the network interface due to the high frequency of communications among parallel tasks as the number of processors increases.

To improve the parallel performance on fine-grained algorithms it is necessary to modify their structure through a set of optimizations to reduce the overhead time, thus the algorithm tends to mainly focus on processing instead of synchronization. Optimizations introduce changes on algorithm, such a way its structure can be adjusted to our parallel approach that allow to decrease the amount of synchronization events, reducing the risk of a parallel task does not have enough work to perform before a synchronization event.

On distributed memory environments, the information exchange among parallel tasks are carried out through message passing over the network. Communications in fine-grained algorithms run the risk of spending a high fraction of the execution time, according to their structural features, which can be influenced positively or negatively by the relationship between intra-node and extra-node communications [26] depending of the node-core combination selected. According to the previous issue, the node-core combinations can be a *key factor* to consider in MPI implementations on multicore cluster environments, such a way to reach a successful parallel implementation, both optimizations in the algorithm structure and determining the suitable node-core combination are required.

An ensemble of learning machines consists in the arrangement of N learners which are trained for solving the same problem. The goal of this approach is to obtain a different prediction from each learner for the same data sample, such a way the weighted average of the N predictions has better generalization ability than the obtained from a single learner [7]. The training process of an ensemble is performed by exhibiting to each learner a set of training samples, whose expected outputs (targets) are known. Using a functional for the error, that considers the difference between the obtained outputs and the targets of each sample, where the internal parameters of each learner are adjusted through a *learning rule*, which allows to minimize the error. The algorithms for training ensembles of learning machines often consume considerable computational resources, such that the parallel approach has many benefits in terms of scalability, to achieve large instances of the problem and also a better use of the available resources.

In this work, we perform an impact assessment on the parallel performance, considering a set of node-core combinations available in our testing environment We have selected *Resampling Local Negative Correlation* (RLNC) [19], an algorithm for training an ensemble of learning machines. The goals of this work consist in: to analyze the impact of the node-core combinations on the parallel performance, using RLNC as testing algorithm, and to determinate what node-core combination (n, c) is suitable given a set of processors, parallel tasks and dataset sizes.

The remainder of the paper is organized as follows. Section 2 presents the RLNC algorithm. Section 3 presents our parallel implementation of RLNC using MPI and Section 4 details the experimental methodology and reports the experimental results. Finally, Section 5 presents the conclusions and future works of this paper.

2 Resampling Local Negative Correlation Algorithm

Ensemble algorithms, a technique that uses a set of n learning machines in collaboratively way to solve the same problem, have gained considerable attention from the machine learning and soft computing communities [15], [10], [17]. The basic idea is to build hypotheses by combining a set of simple functions, instead of carefully designing the complete map between the set of inputs $\mathcal{X}$ and the set of outputs $\mathcal{Y}$ in one single step. Ensemble algorithms have demonstrated to be a flexible way of improving the generalization ability of a base learning algorithm in different tasks including classification [21], regression estimation [8], feature selection [20] and clustering [25], also for a broad range of applications such as financial, network security, astronomy and physics, to name a few.

An ensemble consists of a set of machines that solve the same problem: each machine i solves the problem independently, providing an answer f_i and finally these responses are combined using an aggregation function to get the answer F of the ensemble.

For classification tasks, a common aggregation function is the majority voting, while for regression, the convex combination of the individual outputs

$$F = \sum_{i=1}^{N} w_i f_i \tag{3}$$

is commonly used, where N is the number of ensemble machines, w_i and f_i are the weight and the output of the i-th machine respectively.

The problem in designing successful ensemble learning algorithms is how to select the individual hypotheses from the base space H and how those are aggregated. Since replication of exactly the same hypothesis does not represent any advantage, the concept of diversity has appeared as a key characteristic to get better generalization performance and denotes the differences in the behavior of the individual components to be combined. Hence, designing ensembles involves the definition of a strategy to make the individual components different

in a useful way. Many efforts has been made in order to measure and generate diversity, this has permitted the elaboration of a taxonomy of diversity creation methods in classification and regression scenarios [23]. Resampling Local Negative Correlation (RLNC) [19] is an alternative approach to generate an ensemble for regression problems. This technique has performed a generalization ability comparable with well-known ensemble methods as Adaboost [24].

2.1 Local Negative Correlation

A way to train ensembles using explicitly diversity is based in the so-called *Ambiguity Decomposition*, for ensemble F obtained as equation (3) the quadratic loss can be decomposed as,

$$(y-F)^2 = \sum_{i=0}^{N-1} w_i^2 (y-f_i)^2 + \sum_{i=0}^{N-1} w_i (f_i - F)^2 \tag{4}$$

This decomposition states that the error can be decomposed into two terms where the first is the aggregation of the individual error $(y - f_i)$ and the second, called ambiguity term, measure the deviations of these individual predictions around the ensemble prediction. It can be observed that the higher the second term, the lower the ensemble error and so this seems an approach to measure the concept of diversity. This decomposition can be alternatively stated [19] as follows:

$$(F-y)^2 = \sum_{i=0}^{N-1} w_i^2 (y-f_i)^2 + \sum_{i=0}^{N-1} \sum_{j \neq i} w_i w_j (f_i - y)(f_j - y) \tag{5}$$

As the ambiguity decomposition, the first term measures the individual performance of the learners while the second measures the error correlation between the different predictors. From this decomposition it seems natural to train each learner with the training function, $i = 0, \ldots, N-1$ with the training function

$$\tilde{e}_i = (y - f_i)^2 + \eta \sum_{j \neq i} (f_i - y)(f_j - y) \tag{6}$$

where $\eta > 0$ controls the importance of the diversity term versus the individual performance. The diversity term is computed over the correlations among the

entire sets of learners, therefore the approach is computationally expensive due to the communication process among the learners. We can make this objective function local by restricting it to the neighborhood of the i-th predictor

$$e_i^{local} = (y - f_i)^2 + \eta \sum_{j \in V_i} (f_i - y)(f_j - y) \tag{7}$$

We can generate ensemble diversity using a set of locally coupled learners. Each learner f_i is related with a reduced and fixed subset of other learners V_i through the definition of a linear neighborhood function of order ν.

$$\psi(i,j) = 1 \Leftrightarrow (i-j) \bmod n \leq \nu \text{ or } (j-i) \bmod n \leq \nu \tag{8}$$

It can be proved that a minimal degree of overlapping ($\nu = 1$) between the learner is enough to propagate the information about the performance of each learner by all the group [18].

The RNLC algorithm requires a set of previous tasks to prepare the data for each learner through the bootstrap resampling, obtaining N different datasets from the original dataset, generating as many datasets as learners the ensemble has. The algorithm defines a neighborhood V_i for each learner $i \in \{1, \ldots, N\}$. Each learner is trained T times, using the equation (9). The training rule consists in adjusting each learner i in the step t with information based on the predictions made by its neighbors in the step $t-1$. The RLNC algorithm is defined as follows:

Algorithm 1 RLNC

1: Let $D = \{(\boldsymbol{x_i}, y_i); i = 1, \ldots, m\}$ be a set of training patterns
2: Let f_i, $i = 0, \ldots, n-1$ be a set of n learner and f_i^t the function implemented by the learner f_i at time $t = 0, \ldots, T$
3: Let V_i be the neighborhood of f_i as described in equation (8)
4: Generate n new samples D^i, $i = 1, \ldots, n$ randomly sampling with replacement the original set of examples D
5: **for** $t = 1$ to T **do**
6: Perform one epoch on the learner f_i with the learning function

$$e_i^t = (y - f_i)^2 + \eta \sum_{j \in V_i} (f_i - y) \left(f_j^{t-1} - y\right) \tag{9}$$

and the set of examples D^i
7: **end for**
8: Set the ensemble at time t to be $F(\boldsymbol{x}) = 1/N \sum_{i=0}^{N-1} f_i(\boldsymbol{x})$

3 Parallel Resampling Local Negative Correlation

A parallel approach for the RLNC algorithm needs to identify the regions in the algorithm that can be simultaneously executed, which allows: to separate the

parallel tasks, to establish control points, and to determine what information to exchange. We have used an approach based on distributed memory, such the information interchange among parallel tasks is carried out through message passing, which introduces synchronization events. We have selected a distributed memory approach due to it fits better to the features of the algorithm, and has more flexibility to adapt to a particular communication scheme (in our case, a ring) among learners using a graph-based representation. It also allows to perform the processing of enormous amount of data, because it uses a less amount of local copies inside the node than the number of threads, such a way both the amount of synchronization events and the memory usage decreases. By the other hand, this approach is more suitable to our multicore cluster environment, given its features.

Our parallel approach for the training process described by equation (9) can be divided in three stages: *processing*, *synchronization* and *adjusting*. Processing stage consists in each learner i computes the outputs for the dataset $D^j, j \in V_i$, using the state $t-1$. The synchronization stage consists in information interchange among neighbors, based on a ring scheme of communications where 2ν represents the size of the neighborhood V_i of each learner i. The adjusting stage takes the previously received outputs and uses them to adjust each learner using the error function e_i^t. The division on three stages is shown in the Fig. 3.

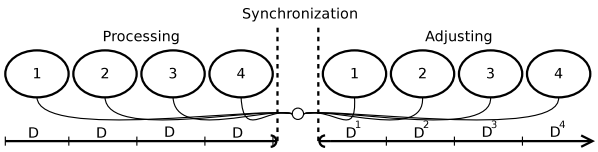

Fig. 3. Processing, synchronization and adjusting stages

In the sequential algorithm, each learner uses the original dataset for processing, thus processing the data of the whole neighborhood is performed without accessing to the resampled dataset D^j of each neighbor, avoiding concurrency problems. Nevertheless, in our parallel approach, each learner i is able to access only to a local copy of the original dataset D and to its own resampled dataset D^i, such a way in the processing stage the original dataset is processed by each learner simultaneously. Then, each learner interchanges the obtained outputs with its neighborhood, and finally the adjusting is performed.

From the point of view of a particular learner i, the parallel implementation of algorithm 1 under a distributed memory paradigm presents two drawbacks: the learner i needs a local copy of the datasets of all neighbors, and by other hand the size of the messages depends of the resampling performed by learner i. Considering the previous issues, our parallel implementation considers that each learn processes the original dataset D and sends the outputs to its neighborhood, in this way, each learner i receives the outputs for the whole dataset D from

$j \in V_i$, and all the messages have the same fixed size m. The improvements of this approach consist in: avoiding the concurrent access to each remote dataset D^j, and avoid handshake messages by using a fixed message length instead of a variable length caused by the resampling. The parallel structure of the implementation is shown in the Fig. 4.

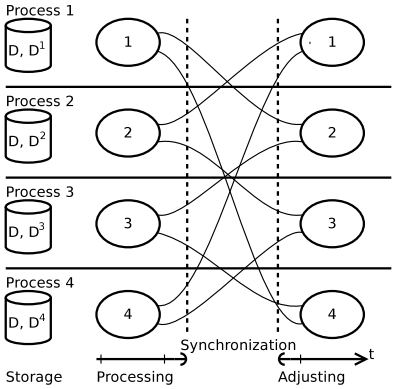

Fig. 4. Processing, synchronization and adjusting stages according to our parallel approach. Interconnections for a neighborhood order $\nu = 1$ are also shown

According to the parallel structure used in the implementation of RLNC, the parallel training of i-th learner is defined in the algorithm 2.

Algorithm 2 Parallel RLNC Training of i-th learner

```
Initialize the learner with random weights
Define V_i as described in equation (8)
Load D and resampled D^i
for t = 1 to T do
    for j ∈ V_i do
        SEND f_i(D) to learner j
    end for
    for j ∈ V_i do
        f_j^{t-1} = RECV from learner j
    end for
    Perform one epoch with learning function in equation (9)
end for
```

The parallel implementation of RNLC is developed using Message Passing Interface (MPI) [6], [16], where the parallel scheme consists in a set of slave

processes which represent the learners of the ensemble, and a master process that only implements the aggregation function.

4 Experimental Results

Our impact assessment of node-core combinations involves a study of the behavior of the RLNC algorithm when the number of processors and dataset sizes are increased. We have proposed a classification of the node-core combinations to carry out the parallel performance analysis and to determinate which is the suitable combination according to each case. In this section, we present and discuss the obtained experimental results, considering the following perspectives: the increasing of the amount of processors and the increasing of the dataset sizes.

4.1 Hardware and Software Configuration

The experiments in this section were run using the following hardware and software configuration:

- Hardware: six compute nodes with the following specifications:
  - Processor: Intel Core2 Quad CPU Q9400 @ 2.66GHz
  - Memory: 4 GB
  - Network Interconnect: Gigabit Ethernet
- Software:
  - Operating System: Centos 5.4 x86_64
  - Kernel Version: 2.6.18
  - Compiler: Intel icc 10.1
  - MPI Library: Mpich2 1.1.1

4.2 Classification of Node-core Combinations

According to the definition about selecting node-core combinations described in equation (2), we have classified the set of feasible combinations as: *coupled*, *combined* and *scattered*. *Coupled* combinations are focused on concentrate the computing in the cores, for maximizing the number of cores per node. *Combined* combinations are focused on a balance between nodes and cores as possible. *Scattered* combinations try to maximize the number of nodes, distributing the workload as scattered as possible. Given the previous definitions and according to the set of nodes and cores feasible in our cluster environment. We have used the combinations shown in Table 1.

Combination	(p, n, c)	Combination	(p, n, c)	Combination	(p, n, c)
Coupled	$(2, 1, 2)$	–	–	Scattered	$(2, 2, 1)$
Coupled	$(3, 1, 3)$	–	–	Scattered	$(3, 3, 1)$
Coupled	$(4, 1, 4)$	Combined	$(4, 2, 2)$	Scattered	$(4, 4, 1)$
Coupled	$(6, 2, 3)$	Combined	$(6, 3, 2)$	Scattered	$(6, 6, 1)$
Coupled	$(8, 2, 4)$	–	–	Scattered	$(8, 4, 2)$
Coupled	$(12, 3, 4)$	Combined	$(12, 4, 3)$	Scattered	$(12, 6, 2)$

Table 1. Classification of node-core combinations

4.3 Execution Set-up

In Table 2 the parameters of the execution and a brief description of them are shown.

Table 2. Algorithm Parameters

Parameter	Description	Values
p	Number of processors (cores)	$\{2, 3, 4, 6, 8, 12\}$
N	Number of machines in the ensemble	50 (fixed)
ν	Neighborhood order	1 (fixed)
m	Dataset size	$\{250, 500, \ldots, 2500\}$
T	Number of training iterations	100 (fixed)
η	Influence of neighborhood in training process	0.95 (fixed)
β	Number of executions of each experiment	10 (fixed)

According to our cluster environment, the values for the number of processors described in Table 2 correspond to those that allow to obtain diverse node-core combinations, for this reason, values over twelve processors are not considered. The number of parallel tasks λ is equal to the number of learners in the ensamble N, being this parameter fixed during execution.

RLNC has been implemented using the Multilayer Perceptron [5] as the learning machine. The synthetic dataset used with the RLNC algorithm is the *Friedman Dataset* which is available in the Delve Datasets [1]. The generation of samples with arbitrary sizes is carried out by using the equation defined by Friedman [11].

The execution time was obtained by the UNIX command "`time -p`". Each experiment was executed β times, such a way the experimental results are based on the average of them. The cluster environment is exclusively used by our experiments and only one experiment was performed at the same time.

4.4 Impact Assessment on Parallel Performance

We have performed an analysis according to the previously proposed classification, considering both the increasing of the dataset sizes and the increasing of the

number of processors. The impact assessment involves to determine the impact of node-core combinations on the parallel performance, focused on the scalability. The previous fact implies to identify the behavior pattern and how the scalability is influenced by the obtained workload distribution, for each number of p and particular node-core combination. Table 3 shows the obtained parallel performance, in terms of speed-up, for the values of number of processors p and dataset sizes m introduced in Table 2.

In the *scattered* combinations, the amount of extra-node communications increases as the number of nodes increases, such a way the throughput of messages generated by the algorithm will reach a point which the network performance does not allow to keep scaling when the number of nodes is increased. According to our results, *scattered* combinations become the best choice when $(2 \leq p \leq 4, 2 \leq n \leq 4, c = 1), \forall m \in \{250, 500, ..., 2500\}$, without matter if the combination generates a balanced or unbalanced workload distribution, due to the strong dependence of network capabilities. From six nodes, the parallel performance decreases due to the increasing of extra-node communications.

In *coupled* combinations using only one node, all communications of the λ parallel tasks are to and from the same node, which produces a bottleneck in the network, increasing the overhead.

In the case of $p = 6$, *combined* combination has better performance than *scattered* due to the high dependence respect on the network capabilities and the amount of extra-node communications, even though both of these combinations are unbalanced. Given that *combined* combination is unbalanced and has more extra-node communications than *coupled*, its performance is negatively affected by those factors, and for this reason the *coupled* combination has the best performance.

The case of $p = 8$ shows that *coupled* combination is the best choice, due to it has less extra-node communications, and intra-node communications are not enough to produce a bottleneck inside each node as seen in *coupled* combinations with $2 \leq p \leq 4$; additionally, the workload is balanced. This result reveals the fact that the use of one node overstresses the network reducing the parallel performance, and reinforces the previous result which shows that the network and the workload balance are key factors, nevertheless, the network capabilities has a stronger influence on the parallel performance.

For $p = 12$, all combinations are unbalanced, such the workload balance affects the performance of all node-core combinations in this case. From 1000 data samples, *combined* combination shows the best results, due to this combination compensates the drawbacks of *scattered* combination, that shows an analogous behavior to $p = 6$, and of *coupled* combination, which tends to increase the intra-node communications in each node. By the other hand, for $m < 1000$ the message passing frequency increases, thereby *coupled* combination is better suited to these dataset sizes. Nevertheless the obtained performance is slightly better to the obtained using *combined* combination.

p	m	Speed-up			p	m	Speed-up		
		Coupled	Combined	Scattered			Coupled	Combined	Scattered
2	250	1.901	–	**1.972**	6	250	**5.231**	5.187	5.101
	500	1.944	–	**1.996**		500	**5.526**	5.471	5.349
	750	1.915	–	**1.991**		750	**5.647**	5.500	5.437
	1000	1.928	–	**1.990**		1000	**5.614**	5.526	5.477
	1250	1.891	–	**1.985**		1250	**5.668**	5.462	5.453
	1500	1.902	–	**1.990**		1500	**5.547**	5.430	5.488
	1750	1.865	–	**1.997**		1750	**5.632**	5.401	5.508
	2000	1.892	–	**1.997**		2000	**5.530**	5.406	5.511
	2250	1.846	–	**1.987**		2250	**5.540**	5.361	5.473
	2500	1.899	–	**1.998**		2500	**5.583**	5.509	5.552
3	250	2.691	–	**2.851**	8	250	**6.538**	–	6.513
	500	2.867	–	**2.906**		500	**7.096**	–	7.006
	750	2.870	–	**2.921**		750	**7.294**	–	7.107
	1000	2.887	–	**2.919**		1000	**7.273**	–	7.141
	1250	2.876	–	**2.922**		1250	**7.345**	–	7.097
	1500	2.869	–	**2.908**		1500	**7.167**	–	7.009
	1750	2.858	–	**2.935**		1750	**7.288**	–	7.041
	2000	2.872	–	**2.937**		2000	**7.124**	–	6.963
	2250	2.815	–	**2.922**		2250	**7.170**	–	6.981
	2500	2.869	–	**2.945**		2500	**7.194**	–	7.056
4	250	3.395	3.627	**3.665**	12	250	**9.007**	8.943	8.833
	500	3.705	3.768	**3.786**		500	9.461	**9.935**	9.636
	750	3.724	3.770	**3.798**		750	**10.562**	10.494	10.070
	1000	3.761	3.776	**3.832**		1000	10.069	**10.487**	10.102
	1250	3.730	3.713	**3.816**		1250	10.718	**10.766**	10.150
	1500	3.718	3.700	**3.794**		1500	9.964	**10.446**	10.001
	1750	3.700	3.682	**3.807**		1750	10.717	**10.775**	10.083
	2000	3.715	3.702	**3.834**		2000	10.221	**10.545**	10.091
	2250	3.632	3.641	**3.809**		2250	10.565	**10.641**	10.022
	2500	3.754	3.766	**3.838**		2500	10.443	**10.621**	10.017

Table 3. Impact of node-core combinations on speed-up

5 Conclusions and Future Work

In this work we studied the impacts of node-core combinations on parallel performance of the RLNC algorithm using synthetic data, using a multi-core computer cluster. Our results were obtained using a fine-grained algorithm, whose kind of algorithm is hard to parallelize according to the features of them, described in the literature [6], [16]; despite of this fact, results show that optimizations on the RLNC algorithm structure reduce the overhead impact on parallel performance.

The algorithm behavior exhibits a relationship between the workload distribution and the selected node-core combination. This algorithm requires to consider both the number of parallel tasks λ and the number of nodes n, because an unbalanced combination obtains a lower scalability than a balanced one when two or more nodes are used. In the case of all combinations have unbalanced workload, it is necessary to compensate the amount of intra-node and extra-node communications.

Our network interconnection can support the extra-node communication in *scattered* combinations using up to four nodes, the extra-node communications does not allow to keep scaling as the number of nodes is increased.

Coupled combinations tend to encourage intra-node communications over extra-node, which might reduce the parallel performance due to communications, because of the bottlenecks that appear in the network, which enhances when the algorithm is run in one node. This drawback can be mitigated by increasing the number of nodes, thus the number of intra-node communications is reduced.

We observed that running experiments in a multi-core cluster – particularly, the RLNC algorithm – implies to take a decision about the selection of a node-core combination that compensates the number of intra-node and extra-node communications, trying to obtain a balanced workload distribution as possible. Moreover, it is necessary to evaluate the network capabilities in order to determinate which is the maximum number of nodes that can be used without negatively affecting the parallel performance.

It would be interesting as future work to perform an analysis using a different network architecture, which will allow to determine the influence of the network on the parallel performance, because our results reveal a clear dependence on the network capabilities, such a way the increasing of the amount of nodes in the cluster environment must have a narrow relationship with the increasing of the network capabilities, to keep the algorithm scalability.

To improve the parallel approach for RLNC it would be interesting to implement a hybrid approach composed by a shared memory approach – such as openMP or Pthreads – that handles the intra-node communications without resorting the network layer, preserving the distributed memory approach only for extra-node communications.

Acknowledgements

This work was supported in part by Research Grant Fondecyt (Chile) 1070220. This work was also supported by CONICYT (Chile) Phd. Grant 21080414 and

by MECESUP2 Phd. Grant and UTFSM-DGIP Phd. Grant. This research uses the resources of Advanced Computing Laboratory of the Informatics Department at Universidad Técnico Federico Santa María, which is supported by the strengthening program of Phd in chile MECESUP2.

References

1. Delve datasets (2008), http://www.cs.toronto.edu/~delve/data/datasets.html
2. Intel high performance computing tools (2010), http://software.intel.com/en-us/intel-hpc-home
3. Top 500 supercomputing site (2010), http://www.top500.org/
4. Alam, S., Barrett, R., Kuehn, J., Roth, P., Vetter, J.: Characterization of scientific workloads on systems with multi-core processors. In: In IISWC. pp. 225–236 (2006)
5. Bishop, C.: Neural Networks for Pattern Recognition. Oxford University Press (1995)
6. Breshears, C.: The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications. O'Reilly Media (2009)
7. Brown, G.: Diversity in Neural Network Ensembles. Ph.D. thesis, School of Computer Science, University of Birmingham (2003)
8. Brown, G., Wyatt, J., Tiňo, P.: Managing diversity in regression ensembles. J. Mach. Learn. Res. 6, 1621–1650 (2005)
9. Chai, L., Gao, Q., Panda, D.: Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In: Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on. pp. 471–478 (2007)
10. Erdem, Z., Polikar, R., Gurgen, F., Yumusak, N.: Ensemble of svms for incremental learning. Multiple Classifier Systems pp. 246–256 (2005)
11. Friedman, J.: Multivariate adaptive regression splines (1991)
12. Gepner, P., Kowalik, M.: Multi-core processors: New way to achieve high system performance. In: PARELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering. pp. 9–13. IEEE Computer Society, Washington, DC, USA (2006)
13. Gochman, S., Mendelson, A., Naveh, A., Rotem, E.: Introduction to intel core duo processor architecture. Intel Technology Journal 10(2), 89–98 (2006)
14. Gropp, W., Lusk, E., Skjellum, A.: Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). The MIT Press (1999)
15. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer (2001)
16. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Pearson Education 2003, Redwood City, CA (2003)
17. Muhlbaier, M., Polikar, R.: An ensemble approach for incremental learning in nonstationary environments. Multiple Classifier Systems pp. 490–500 (2007)
18. Ñanculef, R., Valle, C., Allende, H., Moraga, C.: Ensemble learning with local diversity. In: ICANN (1). pp. 264–273 (2006)
19. Ñanculef, R., Valle, C., Allende, H., Moraga, C.: Local negative correlation with resampling. In: IDEAL. pp. 570–577 (2006)

20. Netzer, M., Millonig, G., Osl, M., Pfeifer, B., Praun, S., Villinger, J., Vogel, W., Baumgartner, C.: A new ensemble-based algorithm for identifying breath gas marker candidates in liver disease using ion molecule reaction mass spectrometry. Bioinformatics 25(7), 941–947 (2009)
21. Nguyen, L., Shimazu, A., Phan, X.: Semantic parsing with structured svm ensemble classification models. In: Proceedings of the COLING/ACL on Main conference poster sessions. pp. 619–626. Association for Computational Linguistics, Morristown, NJ, USA (2006)
22. Quinn, M.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill Science/Engineering/Math (2003)
23. Rokach, L.: Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. Comput. Stat. Data Anal. 53(12), 4046–4072 (2009)
24. Schapire, R.: The boosting approach to machine learning: An overview (2001)
25. Tumer, K., Agogino, A.: Ensemble clustering with voting active clusters. Pattern Recogn. Lett. 29(14), 1947–1953 (2008)
26. Zhang, C., Yuan, X., Srinivasan, A.: Processor affinity and mpi performance on smp-cmp clusters (to appear) (2010)