An empirical study of distribution based on Voyager: A performance analysis

Sérgio Viademonte, Frada Burstein
School of Information Management & Systems
Monash University, Australia
sergio@ponderosa.is.monash.edu.au; frada.burstein@sims.monash.edu.au

Fábio G. Beckenkamp
Software Research Lab
University of Constance - Germany
beckenkamp@acm.org

Abstract

The paper describes the model, implementation and experimental evaluation of a distributed Kohonen Neural Network application (Kohonen Application). The aim of this research is to empirically verify the suitability and the performance of a distributed application based on mobile objects and, in perspective, intelligent agents. This research is aims to provide distribution features in decision support systems. The experiment was based in the Java-ABC project. The Java-ABC project is concerned with flexible software architecture for decision support systems which rely on artificial neural network (ANN) technology. Three parameters: used CPU, used memory (RAM) and time consumed by each Kohonen Application, were taken as evaluation measures in this experiment. Three hardware environments were used: two PCs (one local and one remote) under Windows NT with different RAM capacity and a SUN Ultra under SunOS 5.6. This paper presents the comparison of performance measures from our experimental studies and the analysis of the results. In conclusion, the paper presents the implications of these results for the area of distributed intelligent decision support and future directions of this work.

1. Introduction

During the middle of 1980s the Artificial Intelligent (AI) community began to explore new approaches in which the AI systems were applied to dynamic domains. Generating long-term solutions within a decision support context in dynamic domains needs special care in order to accommodate changes in reaction to the conditions of the outside world. It requires some mechanisms which can respond to rapid change in the environment, and development of a software architecture that allows it to integrate these reactions [1]. Because of its adaptive characteristics, Artificial Neural Networks (ANN) represent an example of a technology that when embedded in a system gives it a capacity to react to the changing environment. Such capabilities of ANN make them attractive for inclusion in decision support system (DSS) frameworks.

Implementation of ANN models requires large computational resources, such as time of processing, CPU capacity and memory allocation; mainly for the training and visualization procedures [2]. Distributed software development paradigms seems to be a suitable way to implement ANN models, providing the means to increase the efficiency of the algorithms by allocating and sharing distributed computational resources.

The motivation of this research is to empirically verify the suitability and the performance of a distributed application based on mobile objects and, in perspective, intelligent agents. This research is investigating the possibility of provision of distributed features to decision support systems. This experiment was conducted as part of the Java-ABC project [3]. The Java-ABC project is concerned with the construction of a DSS, which rely on artificial neural networks technology. The paper is organized as follows: Section 2 presents an overview about the Java-ABC project. Section 3 presents an overview of Kohonen Neural Network model. Section 4 describes the design and implementation issues concerned with distributed ANN (Kohonen Agent and Kohonen Application). Section 5 presents the measurements and the environment used on the experimental evaluation of the implemented components. Section 6 evaluates the achieved results and Section 7 concludes the paper with future directions and proposed work.

2. Brief overview of Java-ABC project

The Java-ABC project is concerned with the object oriented design and implementation of a framework architecture for decision support systems that rely on artificial neural networks technology [3, 4].

One of the aims of Java-ABC project was to be able to manage various ANN models at the same time, trying to solve a specific problem. Besides keeping the design open for supporting various neural network models, a smooth integration of neural network technology into decision support systems is another design goal. In order to achieve these aims several neural models are required to run at the same time in a distributed way. In the long term, the aims of the project include constructing an Object Oriented (OO) architecture that could form the basis of building up hierarchies of ANNs working together; cooperating and acting as intelligent agents in a distributed environment [3].

The project applies object technology to design and implement a flexible framework for building different ANN models. The framework offers a group of classes that represent the core parts of an ANN model such as neurons and synapses. Using those basic classes, it is possible to easily implement various ANN models. Many different ANN models have been implemented using the Java-ABC framework ideas such as the Backpropagation [5] and the ART [6]. The Kohonen Self-Organizing Feature-Mapping (SOM) [7], for which implementation is the subject of this paper, is an important unsupervised learning algorithm.

The Java-ABC project also has developed an ANN simulation environment that allows the simultaneous management of any number of ANN instances, independent of the ANN type. These ANN can be trained or tested at the same time in the simulation environment and should be distributed over networked computing devices. Besides implementing the SOM model using the Java-ABC framework, the experiment presented in this paper is a first attempt to have, in the Java-ABC project, the capability of distributing the ANN objects. Therefore, this work plays an important role in:

- Measuring the hardware requirements for running a distributed ANN application.
- Giving hints on the possible difficulties on having ANN distributed components controlled by an ANN simulation environment.
- Evaluating the possibilities of an actual mobile code environment, namely the Voyager.

3. Overview of Kohonen Neural Network

The Self-Organizing Feature-Mapping (SOM) [7] is concerned with the theory of competition, in which interactions among competitive processing elements (neurons) could be used to construct a network that can classify clusters of input vectors. The SOM acts as a competitive network for classification purposes. The algorithm results in a topology-preserving map of the input data to the output units. As a simplified definition, in a topology-preserving map units (neurons) located physically next to each other will respond to classes of input vectors that are likewise located next to each other. The neurons become selectively tuned to various input vectors during the competitive learning process. In the SOM, all the neurons in the neighborhood that receive positive feedback from the winning neuron participate in the learning process. The locations of the neurons tend to become ordered with respect to each other, building a coordinate system for different input features over the lattice. A self-organized feature map is therefore characterized by the formation of a topographic map of the input vectors, in which the coordinates of the neurons in the lattice correspond to features of the input patterns [8]. Figure 1 shows a two-dimensional SOM with a layer of output neurons.

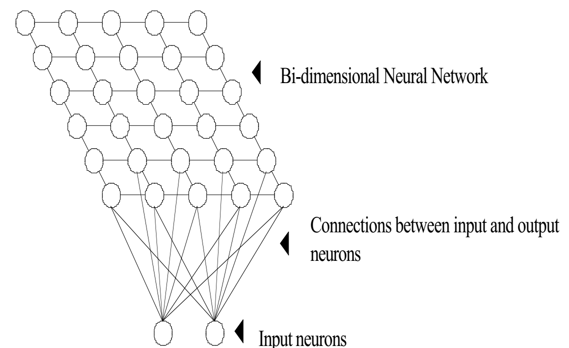

Figure 1: Two-dimensional self-organizing feature map

The next section shows how the Kononen Neural Network was designed and implemented using the object oriented paradigm [9].

4. Design and implementation

The implementation of the Kononen Neural Network was developed in Java language, under MS Visual J++ 1.1 and Voyager 2 Beta 2. Visual J++ is the Microsoft environment for Java development. Voyager is ObjectSpace's product family for distributed computing that unifies the most common industry standards. Full information about Voyager can be obtained at

http://www.objectspace.com/Products/voyager1.htm), the Objectspace home page, including online documentation. As the main purpose of this paper is concerned with performance analysis of the related distributed application, the details of the implementation of the Java and Voyager object hierarchies are omitted. Readers who are not familiar or are more interested in these topics can consult the respective references provided. Also, this paper also does not present explanations of specific Voyager and Java objects such as *Frames, Dialogs, Agents*. [10].

4.1 The Neural Network's class definition

The Kohonen Neural Network model was implemented as a lattice structure with a two-dimensional array of neurons. The basic elements of any artificial neural network model are *Neurons* and *Synapses*. *Neurons* are linked between each other by *Synapses* connections. *Neurons* and *Synapses* have different behaviors. In order to manage the interaction and a particular behavior of *Neurons* and *Synapses,* additional elements must be considered, such as the *Net Manager* object. In this case the basic elements of this implementation are the *KohonenNeuron*, *KohonenSynapse*, and *KohonenManager* objects. The *KohonenNeuron* object is an abstract class, which defines the common behavior of neuron elements: input neurons and output neurons. The input neurons are defined by the *InNeuron* object and output neurons by the *OutNeuron* object. This results in the *KohonenNeuron* object hierarchy (see Figure 2).

The *InNeuron* object defines the specific behavior of the input units of the Kohonen model. It calculates the input activation values for each input neuron. The *OutNeuron* object defines the specific behavior of the output neurons. It calculates the output neurons activate value and the synapses connection values. Figure 3 shows part of the *OutNeuron* object implementation. The figure highlights the class definition, the constructor method and the neuron activate value method.

The *KohonenSynapse* object implements the synaptic behavior, it generates the synapses based on the input neurons and calculates their connection values.

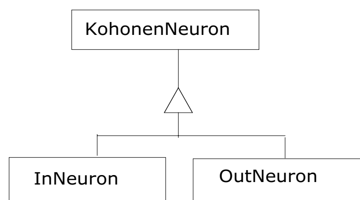

Figure 2: *KonenNeuron* object hierarchy

The major functionality of the Kohonen Neural Network model is provided by the *KohonenManager* object. *KohonenManager* object handles the generation, training and calculations required by the model

```
public class OutNeuron extends KohoneNeuron {
        Vector synapsesVect;
        int outIndex;
        double outActivation;

        public OutNeuron () {
                synapsesVect = new Vector();
                outIndex = 0;
                outActivation = 0;
        }
        public void calcActivation(){ // calculates
the output neuron activate value
                KohonenSynapse synapse;
                double diffSum=0;
                Enumeration e=
synapsesVect.elements();
                while (e.hasMoreElements()) {
                        synapse =
(KohonenSynapse)(e.nextElement());
                        diffSum =
diffSum+synapse.calcDiff();
                }
                outActivation = Math.sqrt(diffSum);
        }
```

Figure 3: *OutNeuron* object implementation

KohonenManager object sets the network parameters; generates the initial network structure; calculates the winner output neuron in each training interaction; computes the activation values for each output neurons; performs the neural network training; visualization and controls the threads execution. Figure 4 illustrates the object diagram for the *KohonenManager* object.

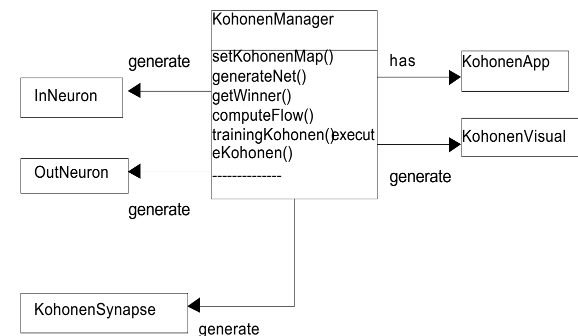

Figure 4: KohonenManager object diagram

The *KohonenApp* is the object that starts the application. It is the initial frame of this application from where all dialogs are initialized. The implementation was developed in a way to provide enough flexibility concerning the network settings, such as the number of input and output neurons, number of lines and columns of the output map

and learning parameters, *i.e.* the number of interactions training, *delta* and *epsilon* factors [2].

4.2 The distributed implementation

In this experiment the distribution is achieved by the execution of several *KohonenApp* objects in different machines at the same time. To implement the distribution two additional objects were defined: *AgKohonen* and *Launcher* objects.

The *AgKohonen* object could be considered the mobile code of this application. It is a derived object from the *Agent* object class defined in Voyager object hierarchy. *AgKohonen* object receives the computer address to where it should move. A computer address normally is set by an IP number, stored in an *Address* object defined in the Voyager object hierarchy. The *AgKohonen* moves to the specific computer and once arrived there, creates an instance of *KohonenApp* object, which starts its execution. Figure 5 shows part the *AgKohonen* object implementation, specifically the *atProgram( )* method, which creates an instance of *KohonenApp* object.

```
import java.util.Vector;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;

public class AgKohonen extends Agent implements  IAgKohonen {

      Vector itinerary = new  Vector(); // vector of address to visit
      int index; // index of current application

      public void addToItinerary( Address  address ) {
          itinerary.addElement( address );
      }

        public void atProgram() {
          KohonenApp  khFrame = new  KohonenApp();
          khFrame.inAnApplet = false;
          khFrame.setTitle ("Kohonen Neural Network Application");
          khFrame.init();
          khFrame.pack();
          khFrame.execute();
          khFrame.show();
          next();
        }

       public void dismiss() throws  VoyagerException {
          System.out.println ( "dismiss" );
          dieNow(); // kill myself and all my forwarders
        }
}
```

Figure 5: *AgKohonen* object implementation

The *Launcher* object handles the *AgKohonen* object. The *Launcher* object receives a machine address, creates an instance of the *AgKohonen* object at the local machine and launches the created *AgKohonen* object to a different machine. It is created by the *LaunchKohonen* object, which is the initial frame from the whole application.

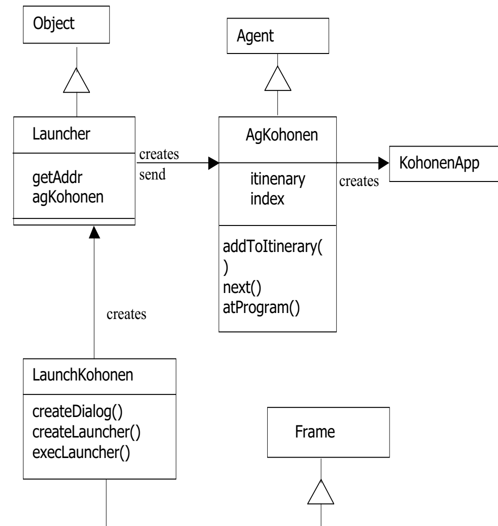

Figure 6: Distribution objects diagram

Figure 6 depicts the relation between *Launcher*, *AgKohonen* and *KohonenApp* objects.

5. The performance measurements

To measure the performance of the distributed Kohonen application the values of used CPU, memory and time consumed by each application, were captured. The CPU was taken as the percentage used, the memory was measured in Kbytes and time in milliseconds. These measurements were taken during the network training and visualization tasks, because these are the most time and CPU consuming tasks of the application. They also involve some threads of control. Regarding the memory allocation, the measures considered in this research refer to the approximate size of memory used by each Kohonen application.

5.1 The experimental framework and settings

The Kohonen Neural Network application was executed with the following configuration:

- Two input neurons in the input layer;
- One hundred output neurons in the output layer;
- The map visualization has 10 lines and 10 columns;
- The learning/training parameters were:
  - Maximum time of interaction: 5000;
  - Delta factor: 2;
  - Epsilon factor: 0.2.

Three hardware environments were used in the performance measurements; two PCs under Windows NT Version 4.0, one with 260 MB and the other with 560 MB of RAM memory and a SUN Ultra under SunOS 5.6 with 64 MB of real

memory. The local PC had 129.856 KB of memory used before the execution and the remote PC had 43.336 KB of memory used before the execution. The SUN workstation had 64 MB of real memory, with 12 MB of free memory before the application execution. This configuration was chosen as it was the available setup in our department, and because it looks representative for a real distributed application: Sun Workstation and PC's with Windows NT operational systems. Other configurations were not tested.

The measurements were taken considering the execution of the Kohonen Agent implementation in the local PC (local mode), in the remote PC (remote mode) and in the SUN workstation (remote mode). The measurements were also taken considering the Kohonen application running without Voyager, it means an implementation in a standalone Java environment under MS VJ++ 1.1, in the local PC.

The amount of CPU and memory used in the PCs were taken by means of the Task Manager utility. In the SUN, the TOP utility in a XTERM window was used. To verify the amount of time consumed by the application, the "STOPWATCH" class was used in the code. The "STOPWATCH" class is defined in the Voyager2, "com.objectspace.timer" package. For detailed references to Voyager2 see the documentation API DOC, available in the Objectspace home page.

The machines were connected to a Local Area Network at the Computer Science Department, at the University of Konstanz, Germany, where the practical part of this experiment was conducted.

5.2 Achieved results and discussion

Figure 7 shows the results for the distributed application:

CONFIGURATION	CPU(%)	MEMORY (Kbytes)	TIME (milliseconds)
PCs			
1 Local Agent	100	11808	55.640
2 Local Agents	100	19644	Agent1=91.51 Agent2=110.52
1 Remote Agent	100	7260	54.41
1 Local and 1 Remote Agent	100 100	Local= 17252 Remote =7444	Local=54.16 Remote=55.27
2 Remote Agents	100	14772	Agent1=110.74 Agent=69.67
SUN			
1 Remote Agent	92.24	9784	59.569
2 Remote Agents	Agent1=48.57 Agent2=48.26	9680 9672	Agent1=118.28 Agent2=118.43
1 Local PC Agent and 1 Remote SUN Agent	PC = 100 SUN= 90.26	PC = 20856 SUN= 9736	PC = 54.92 SUN= 61.28

Figure 7: Measures of performance for the distributed application

Figure 8 shows the results for the non-distributed (stand-alone) Java application:

Non Distributed Implementation	CPU (%)	Memory(KBytes)	Time (milllisec)
1	100	4716	14.41
2	100	4660	13.95
3	100	4636	13.87
4	100	4880	14.09
5	100	4792	14.06
Simple Average	100	4737	14.08

Figure 8: Measures of performance for non-distributed application

Concerning the CPU allocation, as expected, all the tests used nearly 100% of CPU, either in PCs and SUN. This fact does not have a direct relation to distributed aspects, but is more related to the neural network algorithm. It is well known that the training phase of a neural network model is a very CPU and time consuming task [2, 11]. It is clear in the two remote agents tested on SUN, where is possible to take the individual measurements of each agent running, so that each agent used nearly 50% of CPU. This analysis relates to the visualization task, from where the measurements were taken. The visualization task is strongly CPU consuming due to the use of graphical Java drawing objects.

With memory allocation, there are no statistically significant differences on SUN and in the remote PC, with a uniform distribution around the average (see Figure 9 for details). In the local PC there is a significant difference in the memory allocation measurements when there are agents running just in the local PC and when there are additional agents simultaneously running in remote machines. In this last case there is an increase of memory allocation of around 8000 Kbytes. However, there is no significant difference in memory allocation in the remote machines.

The difference in memory allocation on a local PC could be justified by the Voyager *forward* mechanism. Specifically, with the way Voyager deals with distributed objects in order to trace the movements of the agents and forward messages invocation to remote agents. Briefly described, when a remote object is constructed using Voyager, a *proxy* object whose class implements the same interfaces as the remote object is returned to the server machine (from where the remote object was created). Voyager dynamically generates the *proxy* class at a run time. The proxy can receive messages, forward them to the object, receive and return value, and pass the return value on to the original sender. The Local PC in this case is used as a server computer to send the agents. In this way, the machine where the remote objects are created keeps the *proxy* of the remote objects (see Voyager documentation for more details).

CONFIGURATION	Allocated Memory (Kbytes)	NUMBER OF AGENTS	Allocated memory arithmetic average (Kbytes)
SUN	9784	1 remote	9718
	9680 9672	2 remotes, agent 1 and agent 2	
	9736	1 remote	
REMOTE PC	7260	1 remote	7369
	7444	1 remote	
	14772 / 2=7386	2 remotes	
LOCAL PC	11808	1 local	17390
	19644 / 2 = 9822	2 locals	
	17252	1 local + 1 remote PC	
	20856	1 local + 1 remote SUN	

Figure 9: Detailed measurements of memory allocation

Another point to note concerning memory allocation is that the local PC uses more memory than the remote machines. This is because the full Java application remains in the local PC, while the remote machines only have the Voyager server, the Java Virtual Machine and the sent agent; in this case, the Kohonen application. Particularly in this work, the local PC also had the MS VJ++ running, which was not running in the remote machines during the application execution.

Figure 10 shows that the overall average of memory allocation by each running agent application was around **11 Mbytes**.

Allocated Memory Simple Average (Kbytes)			Total Allocated Memory Arithmetic Average (KBytes)
SUN	Remote PC	Local PC	
9718	7369	17390	11492

Figure 10: Arithmetic average of total memory allocation

Figure 11 illustrates the *arithmetic average* of each configuration, concerning the time.

Configuration	Number of Agents	Time (milliseconds) Arithmetic Average
Local PC	1 2	54929 101016
Remote PC	1 2	54837 90204
SUN	1 2	60425 118362

Figure 11: Detailed measurements of used time

There is no significant statistical difference in the PCs' configurations either when 1 agent or 2 agents are considered separately. A difference of 19%, approximately 22752 milliseconds, was observed between SUN and PCs with 2 agents running. Also, a difference of 9%, approximately 5542 milliseconds, was observed between SUN and PCs with one agent running. Accordingly, to the following calculations take place:

SUN (t2) – Local PC (t2) = 118362 – 101016 = 17346 milliseconds
SUN (t2) – Remote PC (t2) = 118362 – 90204 = 28158 milliseconds
_ *(t2)* = (17346 + 28158)/2 = 22752 milliseconds → 19%
SUN (t1) – Local PC (t1) = 60425 – 54929 = 5496 milliseconds
SUN (t1) – Remote PC (t1) = 60425 – 54837 = 5588 milliseconds
_ *(t1)* = (5496 + 5588)/2 = 5542 milliseconds → 9%
where:

SUN (t2) = average time of 2 agents running simultaneously on SUN in milliseconds
Local PC (t2) = average time of 2 agents running simultaneously on local PC in milliseconds
Remote PC (t2) = average time of 2 agents running simultaneously on remote PC in milliseconds
_ *(t2) = average of the difference of the execution time between SUN and PC in milliseconds with 2 agents running simultaneously.*
SUN (t1) = average time of 1 agent running at SUN in milliseconds
Local PC (t1) = average time of 1 agent running at local PC in milliseconds
Remote PC (t1) = average time of 1 agent running at remote PC in milliseconds
_ *(t1) = average of the difference of the execution time between SUN and PC in milliseconds with 1 agent running.*

One must note that these time measurements were taken over the *training* and *visualization* task of the Kohonen application, using nearly 100% of CPU.

Figure 12 shows the *arithmetic average* of used time considering the number of running agents.

NUMBER OF AGENTS	AVERAGE TIME (milliseconds)			USED TIME- arithmetic average (millisec)
	Local PC	Remote PC	SUN	
One	54929	54837	60425	56730
Two	101016	90204	118362	103194

Figure 12: Arithmetic averages of used time with 1 and 2 agents

Figure 11 and 12 show that for a specific number of agents, there is no significant statistical difference in PC configuration, but there is a difference between SUN and PC. It means, that for one agent the used time is almost the same in the

PCs, and approximately 9% more in the SUN. The same could be observed when running two agents simultaneously, with a difference of 19% between PC and SUN.

As it was expected, when increasing the number of agents running simultaneously at the same machine the time spent by each agent has also increased, considering computers with one processor.

Tests with more than two agents running simultaneously at the same machine were not performed. This is because the most important fact in the scope of this work was to investigate the performance of the distributed application among several computers. The performance of the computers when running several applications simultaneously is considered as a side effect of the study. For the purpose of this work, it is more important to have one application running at the same time in different computers than two or three applications running at the same time in the same computer.

6. Analysis

In this section we discuss the results of the measurements presented in the previous section: *amount of CPU*, *memory allocation* and *time used*.

Regarding the amount of CPU required, all the environments used almost **100% of CPU** available. However, it is not true to state that this CPU measurement is related to, or at least only to, the distribution aspects. Neural network models are well known as high CPU consuming applications, mainly for the purposes of the training algorithms [2, 7, 8]. We did not consider any other applications in the test, just the Kohonen neural network model because the experiments presented were performed within a context of distribution in the Java-ABC project, which relies on neural network models as its core technology. For a more general conclusion regarding CPU allocation when running a Voyager agent, some other agents applications should be also considered in the tests.

With memory allocation, the overall average amount of memory allocation was **11Mbytes** per each running application (see Figure 10). The average on SUN was 9718 Kbytes, on the Remote PC it was 7369 and on the Local PC it was 17390 Kbytes. The difference found on the Local PC has already been discussed, and should be advocated to the Voyager forward mechanism and the fact that the Local PC is the server of this application. When there is no remote agent running, the memory allocation on the Local PC is approximately 11Mbytes. It is also important to note the results from Figure 8 related to the non-distributed Kohonen application. In the non-distributed Kohonen application the average amount of allocated memory is **4737 Kbytes**, significantly smaller than the average allocated memory in the distributed application. It implies that the use of Voyager2 Beta2 significantly increases the amount of memory used by the Kohonen application. In this research an average increase of 6755 Kbytes could be observed, it means that the distributed application allocated 58.78 % of additional memory when compared with a non-distributed application, considering the overall memory allocation *arithmetic average.*

Concerning the amount of time used by each application, **no statistically significant differences** were observed between the PC's configurations. A difference of approximately **9%** was found between PC's and SUN when running one agent in each machine, and **19%** when running two agents simultaneously in each machine. In the evaluation of the time spent it must be noted that the PC's used were Pentium desktop computers. In fact, it could be observed that the SUN performance was significantly slower during the drawing of the Kohonen output map.

The number of agents running simultaneously at the same machine with one processor the following observations are worth noting. According to Figure 11, it seems that the time spent by each agent increases in the same proportion as the number of agents running simultaneously. The results in Figure 11 point to a linear relationship between the execution time and a number of agents running simultaneously.

7. Conclusions and comments

The major objective of this research is to verify the suitability and evaluate the performance of a distributed application. In more specific terms, this research is oriented to provide guidelines when using distributed features in decision support systems. In the specific case of the experiment presented in this paper, the motivation comes from the Java-ABC project [3], which is concerned with the construction of a decision support system which relies on artificial neural network technology. ANNs have several interesting capabilities, which make them a powerful technology, especially because of their strong adaptive mechanism. They offer efficient solutions for inductive and deductive machine learning implementations, as well as to the implementation of automatic knowledge acquisition through training procedures. Both features have been used in the decision support field [12, 13, 14, 15]. However artificial neural networks implementation requires large amounts of computational resources, which are not always easy to provide. The distribution of their implementation across several computers running in parallel at same time could bring advantage when using them.

In addition, different concepts have emerged in the decision support arena, as the Intelligent Decision Support (IDS) paradigm. The IDS concept extends the functionality of traditional decision support systems in order to perform knowledge processing in an organization [16], and usually requires the application of some advanced technologies. It is considered that the main role of an IDS is to prepare recommendations for decision-makers based on all the information provided from various sources. Within such paradigm, mobile object technology and in perspective intelligent agents are likely to have an important role. Their inherent capability of mobility within the information space seems to be closely related with the concepts of communication and knowledge processing and sharing. This view is an open way to be explored in future researches.

In the experiments done in this research it was verified that the Kohonen NN application runs in an acceptable time and velocity when one application is running in each machine, except during the training and visualization task. Although the results achieved in this research can not be generalized for every distributed application using Java language and Voyager environment, they can be useful as guidelines regarding the distributed implementations of neural network models under Voyager. Additional tests should be conducted to verify the behavior of distributed applications when using some other technology. An evaluation of the achieved results in this experiment also depends on the application in question, which could be considered appropriate for one application but unreliable for another application. Furthermore this experiment is concerned about providing an idea about the time, memory and computer resources someone could expected to use when building a system in similar environment and platform, and not to make a judgement about the appropriateness of time, memory or processing capacity. Such a judgement is beyond the scope of this experiment and we believe it is relative to each particular application.

Another point to discuss is the number of agents running simultaneously at the same machine, this has been mentioned in a previous section (see section 5.2). Tests with more than two agents running simultaneously at the same machine were not performed because the scope of this work was to investigate the performance of the distributed application among several computers, and not to investigate the performance of the computers when running several applications simultaneously.

The experiment explored the technology for implementing mobile ANNs. The use of proxies is an important design aspect. This makes it easy to turn mobile any object in Java based on the implementation of interfaces. The experiment of the distributed Kohonen corroborated to the construction, on the Java-ABC project, of the interface *NetImplementation* that holds the ANN component core functionality and mobility interfaces. An ANN object that implements this interface can be managed by the Java-ABC simulator and can be moved across the network environment. The *NetImplementation* interface is explained on [4].

Additional topics that could be considered in future work include:

- different ways of implementation of the distribution in the Kohonen neural network model – for instance the distribution of the *KohonenManager* object, or the distribution of input and output neurons
- add knowledge representation and communication features to the object agents using for example knowledge communication languages, such as KIF (http://dutcu15.tudelft.nl/~marcel/mapping/kif/top.html) and KQML http://www.cs.umbc.edu/kqml/papers/kqml-acl-html/root2.html)
- experiment with other distributed development environments.

References

[1] Hendler, James A. "Intelligent Agents: Where AI meets Information Technology". *IEEE Expert,* Vol. 11, No. 6: December 1996, pp. 20-23.

[2] Skapura, David M. 1996. *Building Neural Networks.* ACM Press, 286p.

[3] Pree W.; Beckenkamp F. and da Rosa S.I.V., 1997. *Object-Oriented Design & Implementation of a Flexible Software Architecture for Decision Support Systems*. 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97. Madrid, June 1997.

[4] Beckenkamp F. and Pree W., 1999. "Neural Network Framework Components". Book chapter in Fayad M., Schmidt D.C. and Johnson R. (editors), *Object-Oriented Application Framework: Applications and Experiences*, John Wiley.

[5] Rumelhart, D.E., and McClelland, J.L., 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1. Cambridge, Ma: MIT Press.

[6] Carpenter, G. and Grossberg, S., 1992. Attentive supervised learning and recognition by an adaptive resonance system. *Neural Networks for Vision and Image Processing*. MIT Press.

[7] Kohonen, T., 1982. *Self-organized formation of topologically correct feature maps*. Biological Cybernetics 43, 59-69.

[8] Haykin, Simon, 1994. *Neural Networks A comprehensive foundation*. New Jersey: Prentice Hall, 696 p.

[9] Pree, W., 1995. *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley/ACM Press

[10] Sun 1997: *The Java Language*; *Java Beans*. White Papers at http://java.sun.com, Sun Microsystems

[11] Kosko, B. 1992: *Neural Networks and Fuzzy Systems*. NJ: Prentice Hall, Englewood Cliffs

[12] Medsker, L. R. & Bailey D. L. Models and Guidelines for Integrating Expert Systems and Neural Networks. Em: Kandel A. & Langholz G. *Hybrid Architectures for Intelligent Systems*, CRC Press, 1992.

[15] Leão, B.F. and Reátegui, E. 1993. Hycones: a hybrid connectionist expert system. Proceedings of the *Seventeenth Annual Symposium on Computer Applications in Medical Care - SCAMC*, IEEE Computer Society, Maryland.

[13] Machado R. J., Rocha A. F. & Leão B. F. Calculating the Mean Knowledge representation from multiple experts. In: Fedrezzi M. & Kacprzkyk J. (eds). *Multiperson Decision Making Models Using Fuzzy Sets and Possibility Theory,* The Netherlands: Kluwer Academic Publishers, 1990.

[14] da Rosa, Sérgio I. V., Leao, B.F., and Hoppen, N. 1995: Hybrid Model for Classification Expert Systems. *Proceedings of the XXI Latin American Conference on Computer Science.* Canela, Brazil

[16] Burstein, F. 1995: "IDSS: Incorporating Knowledge into Decision Support Systems". Burstein, F., O'Donnell, P. A. and Gilbert, A. (Eds) *Proceedings of the Workshop on Intelligent Decision Support – IDS'95*, Monash University, Melbourne, 93-96.