Implementation of
Self-Organizing Maps Neural Networks
on
Network Parallel Computers
by
Captain G.M.M. Quintin, CD, BEng

Royal Military College of Canada
Kingston, Ontario
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

0-612-44855-X

Canadá
Implementation of
Self-Organizing Maps Neural Networks
on
Network Parallel Computers
by
Captain G.M.M. Quintin, CD, BEng
Canadian Forces
A thesis submitted to the Department of Electrical and Computer Engineering
Royal Military College of Canada
Kingston, Ontario
In partial fulfilment of the requirements for the degree
Master of Engineering
21 April 1999

Copyright © by G.M.M Quintin 1999
This thesis may be used within the Department of National Defence but copyright for open publication remains the property of the author.
Abstract

The purpose of this applied master's thesis is to develop and implement a Self-Organizing Map neural network on a distributed computer composed of heterogeneous workstations in order to speed-up the computation and to make better use of available computing resources.

Self-Organizing Map neural networks are systems that can be trained to learn, but to complete this training within a reasonable time, high-performance computers are required. By distributing the computation on different workstations over a network, significant performance was gained. The developed application is characterised by its dynamic load balancing schemes and its Java implementation.

Work on this research entailed the evaluation of the currently available technologies for distributed computing, the implementation of the neural network, the development of load balancing schemes, the solving of typical travelling salesman problems (TSPs) and the analysis of the technology and its limitations.

Keywords: parallel and distributed computing, network parallel computing, loads balancing, self-organizing maps (SOMs), neural networks, Java, parallel virtual machine (PVM), travelling salesman problem (TSP).
Acknowledgement

I would like to express my appreciation to my supervisor, Dr Gilles Labonté, for having provided a most interesting project to work on, for his wise advices and guidance on how to tackle the task and for reviewing this thesis. I would also like to thank Captain Jean Dolbec for his assistance on multiple occasions in regard to the Java programming language.
Dedication

This thesis is dedicated to my mother, to thank her for helping and motivating me throughout my elementary school years, for those lengthy hours she spent making me learn my lessons and overcome my animosity for school.
# Table of Contents

Abstract .................................................................................................................. ii

Acknowledgement ................................................................................................. iii

Dedication ................................................................................................................ iv

Vita ........................................................................................................................... v

Table of Contents ................................................................................................... vi

List of Figures ........................................................................................................ xi

List of Abbreviations ............................................................................................. xiv

1. Introduction ........................................................................................................ 1
   1.1 Goals .............................................................................................................. 1
   1.2 Background .................................................................................................. 1
   1.3 Scope ........................................................................................................... 2
   1.4 Thesis organisation ..................................................................................... 3
2. Network Parallel Computing ........................................... 4
  2.1 Purpose ................................................................. 4
  2.2 Parallel architecture .............................................. 5
  2.3 Programming Interfaces ........................................... 12
  2.4 Client/Server architecture ........................................ 13
  2.5 Parallelism ............................................................ 17
  2.6 Performance calculation .......................................... 19
  2.7 Summary ............................................................... 21

3. Self-Organizing Maps .................................................... 22
  3.1 Anatomy of SOM ...................................................... 22
  3.2 Competitive-learning processes .................................. 23
  3.3 Ordering process ..................................................... 26
  3.4 Batch Processing .................................................... 29
  3.5 Distributed Batch Processing .................................... 30

4. TSP ............................................................................. 33

5. Distributed Computing Technologies ................................. 37
  5.1 PVM ....................................................................... 37
  5.2 JAVA ..................................................................... 41
List of Figures

Figure 1 - Uniform-memory access ........................................ 6
Figure 2 - Non-uniform-memory access ................................. 6
Figure 3 - Distributed memories ........................................... 7
Figure 4 - Internetworking .................................................. 9
Figure 5 - Programming Interface ......................................... 12
Figure 6 - Objects Interaction ............................................. 14
Figure 7 - Distributed Application ....................................... 15
Figure 8 - Transfer function ................................................ 24
Figure 9 - A generic SOM algorithm .................................... 26
Figure 10 - Table: Animals and their attributes translated into input vectors. 27
Figure 11 - Map of animals arrangement. ............................. 28
Figure 12 - SOM batch processing algorithm. .......................... 29
Figure 13 - Distributed SOM batch processing algorithm. .......... 31
Figure 14 - Distributed SOM batch processing algorithm (continued). 32
Figure 15 - 100 cities Euclidean TSP after 100 cycles. ................. 35
Figure 16 - 100 cities Euclidean TSP after 200 cycles. ................. 35
Figure 17 - 100 cities Euclidean TSP after 300 cycles. ................. 36
Figure 18 - 100 cities Euclidean TSP after 361 cycles. ................. 36
Figure 19 - PVM architecture layers ...................................... 39
Figure 20 - Java architecture layers ................................................. 43
Figure 21 - Distribution of the applications ........................................ 55
Figure 22 - Class Diagram (core classes only) ..................................... 57
Figure 23 - Message Sequence .......................................................... 59
Figure 24 - Object layout. ................................................................. 60
Figure 25 - Class Diagram ................................................................. 63
Figure 26 - Load balancing scheme for Case 1. .................................... 67
Figure 27 - Equations for case 1. ....................................................... 69
Figure 28 - Equation for case 2. ....................................................... 70
Figure 29 - Load distribution minimizing wasted time. .......................... 71
Figure 30 - First Load Distribution scheme for Case 3. .......................... 72
Figure 31 - Equation for case 3's first load distribution scheme. ................ 73
Figure 32 - Second load distribution scheme for Case 3. ........................ 74
Figure 33 - Equations for Case 3's second load distribution scheme. ........... 74
Figure 34 - Equations for case 4 using the same load distribution as the second one proposed in case 3. ......................................................... 76
Figure 35 - Load distribution in regard to workstations’ speed. .................. 77
Figure 36 - Case 4 optimal load distribution scheme. ............................ 78
Figure 37 - Equations for Case 4 optimal load distribution. ...................... 79
Figure 38 - Load Balancing Algorithm. .............................................. 81
Figure 39 - Time measurement methodology. ....................................... 87
Figure 40 - Obtained solution for 442 cities TSP. .................................. 91
Figure 41 - Absolute execution time for one cycle for 3038 neurons. ................. 92

Figure 42 - Relative Speedup for a neural network of 3038 neurons. ................. 93

Figure 43 - Relative Speedups for neural networks with 5915, 3038, and 1173 neurons. ........................................................................................................... 94

Figure 44 - Execution times for a neural network with 1173 neurons on different computer networks. .................................................................................. 95
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DAEMON</td>
<td>Disk And Environment MONitor</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical Users Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transport Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PVM</td>
<td>Parallel Virtual Machine</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SOM</td>
<td>Self-Organizing Maps</td>
</tr>
<tr>
<td>TCP</td>
<td>Transport Control Protocol</td>
</tr>
<tr>
<td>TSP</td>
<td>Travelling Salesman Problem</td>
</tr>
<tr>
<td>UDP</td>
<td>Unreliable Datagram Protocol</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Goals

The purpose of this applied master's thesis is to develop and implement a Self-Organizing Maps neural network on a distributed computer composed of heterogeneous workstations. Self-Organizing Maps (SOMs) neural networks are systems that can be trained to learn, but to complete the training within a reasonable time, high-performance computers are required. By implementing the SOM on a distributed computer, significant improvement in performance can be gained thus making better use of available computing resources and better satisfying the needs of SOM users. This type of application has already been developed [GUA96][GUA97][LAN96][VAS95]. One of the most recent projects was carried out at the New University of Lisbon [LOB98]. The intent of our work is through the use of better technologies and algorithms to produce a more versatile product than those created previously.

1.2 Background

Today supercomputers are very expensive and unaffordable for many organisations. On the other hand, distributed computing is based on relatively cheap workstations. A distributed computer consists of multiple workstations that share a
common communication medium such as a single Ethernet bus or a token ring network. Most commercial, educational, and government organisations have such computer networks to meet their current computing needs. This kind of distributed computing can support parallel computation and thus can provide additional computer power. When distributed computers are used for parallel computation, they are referred to as network parallel computers. Since many organisations cannot afford supercomputers, network parallel computing becomes the technology of choice to fulfil their needs for fast computing. Accordingly SOM users can take advantage of these widely available computer networks to obtain the desired computing power required by their application.

1.3 Scope

In order to implement the SOM on a distributed computer, it is first necessary to master the principles that make up SOM neural networks. We also need to study presently available distributed computing technology in order to select the best technology for this specific application. The next step is to develop and implement the algorithm best suited to the selected technology. Finally the application should be tested by executing some demanding computational problems like the well-known Travelling Salesman Problem (TSP). The resulting computing performance must be measured and analysed. The program should be able to execute on a network of heterogeneous platforms and should be able to adapt to some network load variations. Finally the end results should be
analysed to determine if network parallel computing is viable considering the currently available technology.

1.4 Thesis organisation

This thesis is written on the assumption that the reader already has some understanding of SOM neural networks but wishes to learn more about distributed and parallel computing. Consequently, chapter 2 to 4 provide background knowledge required to understand the research results as a whole, with marked emphasis on the distributed and parallel computing aspect. Chapters 5 to 9 concern the implementation and results of the application. This part also provides an analysis of the performance limiting factors for network parallel computing. And finally chapter 10 presents closing comments.
2. Network Parallel Computing

2.1 Purpose

The principal motivation to perform computing in parallel is performance, i.e. to reduce the computational time required to solve a problem. Year after year computer users have constantly demanded faster computers either to solve more complex problems or to solve their current computational problems faster. In either case, the time required to solve these problems on a sequential computer remains too long to satisfy the end users. Parallel computing provides the solution to this performance impasse.

Parallel computing consists mainly in breaking up a large computational task into many smaller tasks. Tasks are then allocated to different processors and worked on simultaneously. Processors communicate among themselves to find the solution to their computational problem. On a parallel computer, problems are solved in a fraction of the time required by one processor. However parallel computing introduces an element that did not exist in sequential computing: communication between processors. Consequently parallel computing speedups can only be achieved on the premise that the reduction in time to perform parallel computations is greater than the increased overhead introduced by the overall communication needs and redundant computations. This is one major issue related to parallel computing, but there are other issues such as how multiple processors
are linked together, and how tasks are broken down. This chapter provides a high level overview of what parallel computing involves so basic concepts can be understood. Core elements of parallel computing like parallel architecture, distributed client/server applications, programming interfaces, parallelism, and performance are discussed.

2.2 Parallel architecture

This section situates network parallel computers in the general context of parallel computers as a whole. There are different ways to categorise the various types of computer architectures. For the purpose of this paper we categorise them into three generic groups according to the way they are interconnected [PAT96]: uni-processor (single-processor), multiprocessor, and distributed computer. Since it requires more than one processor to perform parallel computing, only the last two groups will be examined.

The multiprocessor category refers to Massively Parallel Processors (MPPs) also commonly refer to as supercomputers. These computers combine multiple processors in a number of cabinets which may take up an entire room. These computers normally use identical processors. The group can be further subdivided based on their memory access scheme which can be either uniform-memory access (UMA)(see Figure 1), non-uniform-memory access (NUMA) also referred to as share-distributed memory (see Figure 2), or distributed memories also referred to as multicomputers (see Figure 3)[HAM96][PAT96]. The first two memory-access schemes are based on a shared-memory architecture where
memory access is closely integrated with the processors which translates directly into processor performance. The distributed-memory scheme is based on a message-passing architecture and generally uses switches to link the processors.
that are considered hybrid since their memory-access scheme is a mixture of the ones listed above. These multiprocessor computers may have various interconnection topologies which can be fixed or dynamically changed: hypercube, tree, linear array, pyramid, ring, hybrid, etc.[PAT96]. Multiprocessor computers are very expensive and are therefore beyond the means of many organisations and users.

Distributed computing involves the use of a collection of workstations that are all connected to a network (LAN, WAN) that can be distant of a few metres to thousands of kilometres. The workstations are interconnected to facilitate resources sharing and cooperative work. In this case the collection of workstations could include uniprocessors, MPP and mainframes, and possibly all manufactured by different vendors. Distributed computer architecture is very similar to the multicomputer architecture of the multiprocessor group since both rely on message-passing to communicate. The distinction lies more in the interconnection technology. Distributed computers are based
on LAN technology which is typically implemented on either a bus or a ring topology and more recently on ATM switches [AND95] while multicomputers rely normally on high-end communication links.

It is interesting to note the similarity between multiprocessor computers and distributed computers. Both type of computers, at one point, share a single communication link. The difference is at which level the communication link is actually shared. In multiprocessor computers based on a shared-memory architecture (UMA, NUMA), the interconnections take place at the memory-access level where either a memory bus or switches are shared. In distributed computing, it is the network bus that is mainly shared. In both cases the shared communication channels become the source of bottlenecks. By locating the bottleneck at a lower level where it can be managed by hardware components and closer to the processor, relatively good performance can be obtained. On the other hand, when software gets involved, at a higher level, it is harder to maintain comparable performance levels.

Network Parallel computing takes place when processing tasks are distributed between the workstations available in a distributed computing environment and these workstations are working in parallel toward solving a common problem. Network Parallel Computing is a special case of distributed computing. For example one central database on a single server and multiple stations (clients) that can query to that database, is a distributed computing setup but not a network parallel computing setup. Although
the setup is decentralized, the operation of querying, the actual search on the database and finally the replies are all performed in a sequential manner. If different portions of the database are distributed between multiple servers and when clients make queries, the different parts of the database are concurrently and simultaneously searched in parallel, then, this latter setup would fall within the domain of network parallel computing.

Because most networks today are based on the Ethernet bus or token ring technology, workstations on such networks share the same communication channel and must thus communicate sequentially, i.e., only one computer communicates at any given time. This configuration restriction means that although the processing can be parallel, the communication remains sequential. The end result is a sort of “sluggish parallelism” from a purist point of view. Despite this weakness, as long as the overall computation...
has a significant computing/communicating ratio, distributed computing can still provide considerably enhanced performance. Internetworking allows additional parallelism since communication on one network may not interfere with communication on another network (see Figure 4).

The business community has bought into the world of distributed computing thus making it widely available and cheap. There are other advantages to distributed computing: cost/performance ratio, efficiency, reliability, scalability, and flexibility are among the major ones. An application developed on a distributed computer can be designed so that even if a few machines fail, the application can still function, although performance might be degraded. Distributed systems can easily grow by incrementally adding machines to the network. Another interesting aspect of distributed computing is that networks can be interconnected in what is referred to as internetworking.

Internetworking combined with the TCP/IP protocol has given birth to the Internet. The business community initially used computer networks to access data at a central point (central server), and to share resources (printers, storage facilities, servers), and eventually to manage the control of their business information in a more decentralised fashion. Most applications were still running on a single workstation whenever they were executing. Only later in the history of distributed computing did users start to seriously consider the sharing of CPU time in the same manner that they shared the other previously mentioned resources.
It is believed [AND95] that network parallel computers will take a chunk of the supercomputer market for the same reason that networked PCs took a chunk of the mainframe and minicomputer market. One of the main reasons is related to the performance/price ratio. Because of high volume manufacturing, the development cost for personal computers and workstations can be amortised over a large sales volume. Berkeley's researcher [AND95] estimated that the price-performance ratio is improving at 80% per year for PCs and workstations, while the one for supercomputers is improving at only 20-30% per year.

"The field of distributed and parallel computing will grow and become synonymous with mainline computing. Distributed and parallel computing is all of computing." [ELRE98]

Speedups can only be achieved in any parallel computing environment on the premise that the reduction in time to perform parallel computations is greater than the increased overhead introduced by the overall communication needs and redundant computations. The impact of the communication overhead is reduced if communication is handled at a lower level. Programmers do not however need to know all the specific aspects of computer architecture when developing an application. To hide the underlying hardware architecture, programmers refer to application programming interfaces which are discussed in the next section.
2.3 Programming Interfaces

To better manage computer software complexity, software is developed according to the concept of service layers. In this framework, programmers make use of a lower service layer to program their application by making service calls to lower service interfaces. This approach to software development allows piggybacking on previously developed software thus accelerating the development of more complex programs. In this approach, programmers ignore details of exactly how these lower services are implemented. For example, today's programmers normally write their code by referring to already developed libraries. The code is translated into machine-dependant instructions by a compiler (see Figure 5). The programmers of these applications do not have to understand how the hardware actually works since the programmers that have coded the compiler have already done that work for them. Consequently, well-defined plug-in interfaces are required, so new applications can be easily connected to lower

![Figure 5 - Programming Interface](image-url)
service layers. These plug-in interfaces are generally referred to as application
programming interfaces (API).

Applications for distributed computers are also developed in the way, described
above, with the additional condition that they require a library allowing the creation of a
connection with another application on another workstation. Prior to going further, we
need to describe the client/server concept.

2.4 Client/Server architecture

Whenever an application requests a service (invokes a method or procedure) from
another application that resides on a different workstation, the requesting application is
called a ‘client’. At the other end, the application that is responding to that service
request is called a ‘server’. The server always sits passively idle waiting for requests.
The client and the server communicate among themselves by sending messages. These
messages can only make sense to both of them if they use a common communication
protocol. Protocols consist of rules that determine the subject of a message and the
format for the transmitted data.

Consequently, to develop a distributed application, programmers have to actually
develop two applications: one residing on the client side and one residing on the server
side. Using an API, the programmer develops the application so that the two applications
connect and communicate with each others by sending message back and forth. But prior to that stage, a common approach is to confirm that the application functions as a single unit prior to splitting it and porting it to different workstations. In this approach, a client object and server object are created within a single application to ensure correct functionality. The client object makes requests to the server object directly through the server interface (see Figure 6).

![Diagram of Objects Interaction](image)

**Figure 6 - Objects Interaction**

Once the functional aspect of the application has been validated, the two objects are then ported to two different workstations as distinct applications with the addition of two new objects which are the stub and the skeleton (see Figure 7). The stub resides on the client side and the skeleton on the server side. Now the client object deals with the stub object which replicates the server object’s interface. The client object views the same interface as if it were dealing with the server object directly, consequently no change is required for the client object itself. The role of the stub and skeleton is to pack and unpack the data provided to them by either the client or server object and to forward it to the opposite side. On a request of the client, the stub forwards the request to the skeleton which interprets the request and makes a call to the server application. Upon
completing the call, the skeleton replies to the stub which returns the reply to the client. The client has been waiting all along until the stub returns with the reply. This scenario is called a blocking request [TAN95][CHO97]. This prevents the client from carrying on with other tasks and requests. Blocking requests constrain the application to behave in a sequential manner. To break that sequential pattern, non-blocking request schemes are used. In the context of a non-blocking request, the client after having made its request is free to carry on with other tasks or requests. In the latter scenario, the request is non-blocking and the stub and skeleton act the same way as in the blocking scheme with the exception that the stub has to store the reply until the client requests it. Consequently the client has the responsibility to get back to the stub and pull the reply. With non-blocking request schemes, the client may make requests to multiple servers prior to waiting for the first reply. The servers are then working concurrently and simultaneously on the clients' requests which means parallel computing. In both cases, the stub and skeleton use lower services to communicate. The communication goes through the different services layers.

![Figure 7 - Distributed Application](image-url)

15
(OS, hardware, I/O, etc.). However, the stub virtually communicates directly with the skeleton making the details of the actual connection and communication transparent to the client application (see Figure 7).

The above-described way of developing distributed applications relies on the message-passing framework, i.e., the different processors have to communicate through messages. There are other frameworks for developing distributed applications. The two major alternatives are the share-memory and the publish/subscribe frameworks. Both are built on top of the message-passing services thus requiring an additional service layer. In the share-memory framework, the programmer defines variables which are recorded as share variables. Clients developed using an adequate API only need to read and write to those variables. When one client wants to communicate with another, it writes the desired information to a variable (which acts as a mailbox) and the other client has the responsibility to periodically read the variable. The details of the messages traffic is hidden from the programmer when working in a share-memory framework. Similarly in the publish/subscribe framework, the publisher (a client) writes to a mailbox (the server); on the other hand, the subscriber (other clients) are informed whenever the mailbox information is changed.

These more sophisticated services impose a computing overhead. However, they reduce program complexity from the programmer point of view. Because the functions are reduced to two operations: read and write for the share-memory framework, and
publish and subscribe for the publish/subscribe framework, it is easier for the programmer to design and develop complex applications. In a message-passing framework, the programmers need to know the interconnection topology between the processors that form the computer environment. It is a bit more cumbersome for the programmer to work in this framework since he has to manage the communication links.

Service frameworks with their respective programming interfaces, help programmers to hide details of computer architecture and help develop parallel applications. However, the most important element for programming parallel applications is whether or not the application can be parallelized. This depends on the degree of parallelism of the computational problem.

2.5 Parallelism

Parallelism must be an inherent characteristic of a computational problem for it to gain anything by parallel computing. Most computational problems can be said to have some parallelism. Parallelism is defined as the quality of being able to be subdivided. These subdivisions have to be mutually independent. A computational problem can be either subdivided with respect to its data (data partitioning) or its tasks (task partitioning). The granularity of the data or tasks will determine if the problem can be efficiently solved through parallel computing.
Different kinds of parallelism have been categorised into four groups according to the types of data and instruction streams. Although these categories were originally defined to classify computer architectures [PATT96][HAM96], they can be used to group the different types of computational tasks. The four categories are as follow:

a. Single instruction stream, single data stream (SISD). In this category there is actually no parallelism due either to a lack of process parallelism or to an insufficient quantity of processing data.

b. Single instruction stream, multiple data streams (SIMD). In this case, the same data manipulation is performed by multiple processors on various data sets.

c. Multiple instruction streams, single data stream (MISD). This category describes a pipeline approached to parallelism. All the data go through all the processors in a sequential fashion where each processor performs a specific data manipulation. By the time the data goes through the last processor, the data has undergone all the required processing. All processes are mutually dependent. Each processor assumes that the received data has previously undergone the required transformation and manipulation by the preceding processors. The data latency is longer but the throughput is higher.
d. Multiple instruction streams, multiple data stream (MIMD). In this category each processor executes its own instructions and operates on its own data set. Different data manipulations are performed based on the nature of the data itself.

The parallel nature of the problem to be solved is the driving factor in how the task (or the data) will be partitioned and distributed across a network of workstations. The SOM will be explained in more details in a later chapter. However, the nature of SOMs puts them in the SIMD group. The SOM consists of a very small set of instructions to be performed on a very large set of data. Consequently each processing unit will have the same set of instructions and different subsets of data.

2.6 Performance calculation

There are theoretical laws concerning parallel computing limitations. But many of these theoretical formulas do not account for the detailed characteristics of a specific problem. For example the two well-known laws, the Amdahl and Gustafson-Barsis Laws, ignore communication and load balancing overhead [ELRE98]. These theoretical laws can not be relied on to identify computing performance for specific programs. Consequently as per [ELRE98], it is necessary to leave these generic theoretical laws aside and enter the world of experimentation. To really find out how a program behaves on a parallel or distributed system, it is necessary to derive and implement a model in
order to validate it through experimentation. Such a model will have to take into consideration the particularities of the application and perhaps the architecture on which it is implemented. In any case, models can be characterised by three main attributes: expected speedups, utilisation rate and efficiency [ELRE98][AKL97].

The speedup is defined as the ratio of execution time required to run the sequential version of an application (which implies a single processor) over the time required to run its parallel version on a certain number of processors (i):

$$\text{Speedup} = \frac{Time_{i}}{Time_{1}}$$

Processor utilisation is a measure of individual processor (i) usage and is expressed as:

$$U_i = \frac{\text{computeTime}_{i}}{\text{computeTime}_{i} + \text{idleTime}_{i}}, \quad 1 \leq i \leq N$$

Efficiency is a measure of the average contribution of each processor to execution of the parallel application when a certain number of processors (i) are used. It is defined as follow:

$$E_i = \frac{\text{speedup}_i}{i}, \quad 1 \leq i \leq N$$

These three parameters could then be used to characterize the performance and behaviour of a parallel application and can be used for comparison between similar applications and implementations.
2.7 Summary

Although supercomputers are powerful, they are very expensive and therefore very few organisations can afford them. On the other hand, most organisations have a computer network of either PCs or workstations of some type which makes network parallel computing readily accessible. To ease the development of distributed applications, programmers make use of the application programming interfaces (API) of already existing products or libraries. The fundamental way of developing distributed applications is based on the message-passing framework. Prior to attempting to parallelize an application, the nature of the problem has to be analysed for parallelism. A characterisation of the parallelism combined with the available API will guide the programmer in how to tackle the task. Programmers need to apply mathematical models that adequately consider all pertinent details to faithfully simulate the behaviour of their specific application. The models must then be validated. The foregoing concepts are the major elements required to understand network parallel computing.
3. Self-Organizing Maps

A Self-Organizing Map (SOM) neural networks is particular kind of artificial neural network that was formalised by Teuvo Kohonen in the early ‘80s [KOH97]. Kohonen introduced SOMs in an attempt to model the brain. It has been known for some time that the different areas of the brain respond to different sensory organs and the topography of these subareas corresponds to the arrangement of their respective sensors. For example, the area of the brain that corresponds to eyes movements is located next to the area for vision. Consequently the brain is a projection of the different body sensors. SOM tries to model this topology-preserving neural projections found in different parts of the brain. Like the brain, Kohonen’s SOMs create abstractions and organise dimensionally complex data according to its inherent relationships [ROG97]. SOMs are well suited for dynamic signal processing, non-linear control, adaptive control, and information ordering. SOM may not always produce optimal solutions, but they are known for their rapidity and for producing solutions that are among the best.

3.1 Anatomy of SOM

SOMs are designed to process (learn) information presented to them. The external information is referred to as inputs and this input is presented in vector form. For example, if a SOM has to learn the coordinate position of multiple cities, the
coordinate values, x and y, for each city would have to be encoded into a series of two-dimensional vectors.

The SOM itself consists of an array of neurons. A neuron is characterised by a weight vector and a position identifier. The weight vector is a one-dimension array of real numbers with the same dimension as the input vectors. The weight vector could be initialised with any values. The neuron position identifier is used to identify the neuron position within the neural network. Through network dynamics, all the weight vectors are gradually changed to fit and approximate the input vectors. The fitting technique is the core element of the SOM. This is achieved through a competitive-learning process.

3.2 Competitive-learning processes

The competitive process consists in comparing each input vectors \((x_i)\) to each neuron’s weight vectors \((w_n)\). The neuron, whose weight vector is at the shortest geometric distance from an input vector, becomes the winner for that input vector. Consequently each input vector would eventually be associated with a specific neuron. One neuron may win for more than one input at first. In our project, the Euclidean distance is used for this calculation. The distance is calculated as follows:

\[
dist = \|\bar{x}_i - \bar{w}_n\| = \sqrt{\sum_{j=1}^{\text{dim}} (x_{i,j} - w_{n,j})^2}
\]
Subsequent to this competition, the learning process takes place. During the learning process, each winning neuron and its surrounding neighbour neurons gradually change the value of their weight vectors in an attempt to match the input vector for which the winning neuron has won. Their weight vectors are changed according to a transfer function which characterises the learning rate of the neural network. The transfer function provides a scalar that decreases as the distance between the winning neuron and its neighbours increase. This neuron distance is a property of the neural network. The transfer function is often taken to be half of a Gaussian:

![Figure 8 - Transfer function](image)

Depending on the distance between the subject neurons (n) and a winning neuron (v), the subject neuron changes its weight vector values to come closer to the input vector \(x_i\) associated with the winning neuron. The calculation is as follows:

\[
\hat{w}_n = \overline{w}_n + \text{GaussianFnct}(v,n) \cdot [\hat{x}_i - \overline{w}_n]
\]
Each neuron that is within the neighbourhood boundary of a winning neuron gets to change the value of its weight vector. The neighbourhood boundary is defined by the limit of the support of the transfer function. Neurons beyond that boundary from a winning neuron get a scalar value of ‘zero’, and therefore their weight vectors will not change. Because one neuron might be in the neighbourhood of multiple winners, its weight vector is changed accordingly under the influence of many inputs. After each cycle which consists of one competition and one learning process, the transfer function is changed. The curve is contracted on its longitude and often decreased on its latitude toward a small value. Consequently the neighbourhood size decreases throughout the learning life of the network.

This fitting technique is sometimes described as a hypothetical elastic network, in which each neuron is tied to its immediate neighbours with an elastic band. When the weight vector of a winning neuron moves toward its closest input vector, it also pulls in the same direction, to a certain extent, its neighbours. As time passes, the elastic fatigues, stretches and loses its pull. Consequently the winning neurons lose some of their pulling force as the network learns. The above processes can be condensed into the following algorithm:
3.3 Ordering process

As previously mentioned, each neuron that is within the neighbourhood boundary of a winning neuron gets to change the value of its weight vector. This results in a local smoothing effect of all weight vectors in that neighbourhood. Also, a neuron may be in the neighbourhood of multiple winners. These two situations, in continued learning, lead to a global ordering. Kohonen [KOH97] has provided a very simple and effective example of this ordering process, where the inputs are a list of animals which were encoded based on some attributes as in Figure 10. For example the dove is represented by the input vector \( \{1,0,0,1,0,0,0,0,1,0,0,1,0\} \). The end result is displayed in Figure 11.
as a two-dimensional map of weight vectors after the neural network had been trained
with these inputs vectors. Notice the emerging grouping according to similar attributes.

![Figure 10 - Table: Animals and their attributes translated into input vectors.](image)

An interesting characteristic of the SOM is that the initial values of the weight
vectors for each neuron do not change the end result, as long as the learning rate is not too
fast and the neighbourhood boundary does not diminish too fast either. Consequently the
weight vectors can be initialised with random values. However, it is more advantageous
to start with a uniform distribution. This favours convergence toward a solution and
consequently the learning process can be accelerated.
<table>
<thead>
<tr>
<th></th>
<th>duck</th>
<th>horse</th>
<th>cow</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>zebra</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tiger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goose</td>
<td></td>
<td>wolf</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>hawk</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>owl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dove</td>
<td></td>
<td>dog</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>eagle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hen</td>
<td></td>
<td>fox</td>
<td></td>
<td>cat</td>
</tr>
</tbody>
</table>

*Figure 11 - Map of animals arrangement.*
3.4 Batch Processing

One of the variations of the above described process is the batch processing approach [KOH97][VAS95] which we chose for our implementation. In batch processing, weight vectors are not changed until the respective changes induced by each input has been calculated. The resulting process can be expressed in the following algorithm:

```
for (Neuron n = 1 to N)
    RefVectorBuffer(n) := 0
end_for
for (InputVector, i = 1 to I)
    DiffVectorMin := infinity
    WinNeuron := null
    for (Neuron, n = 1 to N)
        DiffVector := InputVector(i) - WeightVector(n)
        if (|DiffVector| < DiffVectorMin)
            DiffVectorMin := DiffVector
            WinNeuron := n
        end_if
    end_for
    for (Neuron n = 1 to N)
        ratio := TransferFunction( WinNeuron - Neuron(n) )
        RefVectorBuffer(n) := RefVectorBuffer(n) + ratio * (InputVector(i) - WeightVector(n))
    end_for
end_for
for (Neuron n = 1 to N)
    WeightVector(n) := WeightVector(n) + RefVectorBuffer(n)
end_for
```

Figure 12 - SOM batch processing algorithm.
3.5 Distributed Batch Processing

In the distributed version of the batch algorithm, the task is distributed between a set of workers and a coordinator. The neural network is partitioned into subsets of neurons and distributed to the workers. And, each worker also receives a copy of all the inputs. The workers could have received a copy of each neuron and receive a subset of inputs, however, the former partitioning and distribution was preferred and was also demonstrated to provide better performance [LAN96].

For each input vector, each worker has to find the winning neuron among its subset thus making a list of local winning neurons for the set of inputs. Subsequently these lists of local winners are sent to the coordinator. From these lists the coordinator makes a consolidated list of winners among all the neurons, which is referred to as the list of global winners. The coordinator then sends this list of global winners to each worker, which in turn changes the weight of each of its neurons accordingly. Figures 13 and 14 describe the algorithm.

Although the SOM concept is mathematically relatively simple, the resulting mechanism is powerful for sorting and identifying patterns. Despite their simplicity, SOMs remain very demanding in terms of computing resources, since the process, while simple, is applied to a very large number of inputs and neurons. The neurons’ weight vectors may also be of dimensions numbered in the thousands. This is to be expected
considering that SOMs are designed to emulate the human brain, which is composed of billions of neurons that process very complex information.

### done by each worker
for (Worker w = 1 to W) do in parallel
  for (InputVector, i = 1 to I)
    DiffVectorMin(i) := infinity
    WinNeuron(i) := null
    for (Neuron, n = 1 to N)
      if (Neuron(n) != null)
        DiffVector := InputVector(i) - WeightVector(n)
        if (|DiffVector| < DiffVectorMin)
          DiffVectorMin(i) := |DiffVector|
          WinNeuron(i) := n
      end_if
  end_for
send_local_winners_to_coordinator (WinNeuron, DiffVectorMin)
end_for

### done by the coordinator
for (InputVector, i = 1 to I)
  global_DiffVectorMin(i) := infinity
  global_WinNeuron(i) := null
end_for
for (Worker w = 1 to W)
  receive_local_winners_from_workers (WinNeuron, DiffVectorMin)
  for (InputVector, i = 1 to I)
    if (DiffVectorMin(i) < my_DiffVectorMin(i))
      global_WinNeuron(i) := WinNeuron(i)
  end_for
end_for
for (Worker w = 1 to W)
  send_global_winners_to_workers (global_WinNeuron)
end_for

Figure 13 - Distributed SOM batch processing algorithm.
### done by each worker

for (Worker w = 1 to W) do in parallel
  receive_global_winners_from_coordinator (global_WinNeuron)
  for (Neuron n = 1 to N)
    RefVectorBuffer(n) := 0
  end_for
  for (InputVector, i = 1 to I)
    for (Neuron n = 1 to N)
      if (Neuron(n) != null)
        ratio := TransferFunction(global_WinNeuron(i) - Neuron(n))
        RefVectorBuffer(n) := RefVectorBuffer(n) + ratio * 
        [InputVector(i) - WeightVector(n)]
      end_if
    end_for
  end_for
end_for

for (Neuron n = 1 to N)
  WeightVector(n) := WeightVector(n) + RefVectorBuffer(n)
end_for

Figure 14 - Distributed SOM batch processing algorithm (continued).
4. TSP

The travelling salesman problem (TSP) is a classical problem in the field of combinatorial optimisation, and is actually one of the most widely studied problems in that field [LAW85]. Combinatorial optimisation involves searching a large space of possible solutions to a problem for a specific optimal solution.

In the TSP, a salesman is to visit a fixed number of cities, which in the simplest case lie in a two-dimensional Euclidean plane. While passing through each city only exactly once, the salesman has to end his tour in the city where he started, completing a full circuit. Although there are multiple solutions, a good solution would be a route that is among the shortest.

Although it is easy to state, the travelling salesman problem is very hard to solve because the number of possible routes increases exponentially with the number of cities [DUR87]. The number of possible valid routes is given by $N!/2N$ where $N$ is the number of cities [SHA97]. For example, for seven cities, the solution space consists of 360 valid paths. This problem could be solved in a reasonable length of time by evaluating the length of each path. However, with a slightly larger number of cities, 20 for example, the number of possible routes explodes to more than $6 \times 10^{16}$. As stated by [SHA97] even if one could calculate the length of each path in 100ns (using a very fast processor) it would
require more than 190 years to measure all possible route lengths. Such exhaustive computational approaches are impractical, especially since in most real situations, the number of cities is in the order of several thousand.

This research project is only interested in the TSP as a benchmark. It is hoped that by using an easily understandable and well-known problem, the computational challenge will be better appreciated. SOM neural networks are well suited to solve TSPs. They provide good routes without having to evaluate all possible paths. A SOM requires no more time to solve a large TSP than a small one, leaving aside the selected learning rate.

Following is an example that demonstrates the evolution of the SOM ordering process with a 100-cities Euclidean TSP after 100, 200, 300 and 361 cycles respectively.
Figure 15 - 100 cities Euclidean TSP after 100 cycles.

Figure 16 - 100 cities Euclidean TSP after 200 cycles.
Figure 17 - 100 cities Euclidean TSP after 300 cycles.

Figure 18 - 100 cities Euclidean TSP after 361 cycles.
5. Distributed Computing Technologies

Although the concept of Network Parallel Computing is not new, only in the recent years has this type of distributed computing drawn a lot of interest from the computing community, specifically among scientific and engineering users. Like distributed computing in general, network parallel computing is currently in full expansion. Consequently, there are an overwhelming number of products, papers, research projects and conferences related to the topic. In order to find the desired technology to implement the SOM, we therefore proceeded to screen the material that we found throughout the world wide web (WWW) at different sites specialised in parallel computing, proceedings of international symposiums on the topic, etc. A list of some of these web sites is provided in Appendix A. After the screening, only two technologies were retained for further appraisals since they had the best potential to fulfil our requirements. These two technologies are the Parallel Virtual Machine (PVM), and the Java programming language together with its Java Virtual Machine (JVM).

5.1 PVM

Parallel Virtual Machine [GEI94] is a message passing API that enables network parallel computing for a wide range of computer architectures. The individual computer can be anywhere on the Internet as long as the user has an account and login access.
PVM started in 1989 as a project mainly funded by the US government and has been distributed freely since. Today with its latest version 3.4, PVM remains mainly a collaborative research project between Emory University, Oak Ridge National Laboratory, and the University of Tennessee.

With PVM, users write their application as a collection of co-operating tasks in any languages that can make function calls to routines written in C, e.g., C, C++, Fortran, Java and Lisp. A task is defined in PVM as a unit of computation analogous to a Unix process. It is often a Unix process, but not necessarily so. Tasks access PVM resources through a library of standard interface routines (written in C). These routines allow the initiation and termination of tasks across the network as well as communication and synchronisation between tasks. Any process may communicate and synchronise with any other. By sending and receiving messages, multiple tasks of an application can co-operate to solve a problem in parallel.

Someone may ask why use PVM when communication libraries already exist for most computer architectures. This is true. But on a heterogeneous network, this gets complicated as communication protocol formats may be incompatible between different types of workstation architectures. For example on some architectures integers are represented by 32 bits and on others by 64 bits. In such a case programmers would have to take this encoding difference into consideration in all their code. The virtual machine part of PVM allows programmers to hide details of computer architecture and these
incompatibilities by transparently handling all messages routing, data conversion, and task scheduling. In other words, PVM is a complete programming interface that supports process management, message passing, synchronisation, and also miscellaneous status checking and housekeeping tasks.

The PVM system is composed of two parts [GEI94]. The first part is a daemon, called pvmd, that executes on all the computers making up the virtual machine. Pvm is designed so any user with a valid login can install this daemon on a machine. A user wishing to use PVM first configures a virtual machine by specifying a host pool list; the daemons are started on each host, and cooperate to emulate a virtual machine. The second part is the PVM console which is actually a shell command line prompt. The console can be opened on any of the computers that make up the virtual machine. PVM applications can then be started from this console (Figure 19). Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

![Figure 19 - PVM architecture layers](image)
PVM provides other benefits. It is fairly portable since it includes message passing libraries for almost any type of platform. However, PVM is not entirely portable since PVM and the application code have to be compiled individually on each type of platform. PVM is also scalable since it can use computer resources as they become available and can take advantage of the latest computational and network technologies. PVM provides flexibility and can support fault-tolerance if programmers build appropriate logic into its code and dynamically add and delete of hosts/processes as faults are detected.

PVM has gained widespread acceptance in the high performance scientific computing community and is being used in computational applications around the world. Because of its popularity, PVM became the de facto standard for this kind of application in the early '90s. Consequently there are many applications readily available developed as add-ons to PVM [PVM].

There are also different flavours of PVM. The standard PVM [PVM] which was discussed in the above paragraphs distinguishes itself by the fact that it can be run on several different processor architectures as long as each processor gets its own compiled version of the executable program. jPVM [jPVM] is a java interface to be used in conjunction with the standard PVM. In this flavour, performances are reduced since another service layer has been added [YAL98]. JPVM [FER96](to be distinguished from jPVM by the capital ‘j’) is a full implementation of PVM in Java that is not compatible
with the standard PVM. JPVM provides less performance than PVM and jPVM [YAL98]. WPVM [SIL95] is a MS-Windows-based implementation of PVM that is compatible with the standard PVM. Although WPVM was developed for the most popular OS for PC, MS-Windows, it is platform dependent and therefore can not be ported to other platforms.

PVM continues to interest the computing community as a popular and widely used API for concurrent computing. At the time of writing this thesis, a yearly international conference is still focussing on the latest developments of PVM and its applications. The next conference, EuroPVM/MPI'99, is scheduled for September 1999 in Barcelona [PVM]. Because of its popularity, one may expect that PVM will continue to mature and further evolve in the future.

5.2 JAVA

5.2.1 Java in general

Java [JAVA] is a recent modern programming language developed by Sun Microsystems. Java was made available to the public around 1995. Java has a full collection of features that fulfil today's programming needs addressing such major issues as networking with heterogeneous systems, tackling today's complex computational problems, and programming for the World Wide Web. Java is thus more than just
another programming language; in a sense, Java is an enabling technology for today's computing needs.

Java is a fully object-oriented programming (OOP) language. It is now recognised that OOP permits better management of complex software projects. Java not only provides the capability to implement object-oriented principles: it enforces those principles contrarily to other language that only provide the artefacts. For example C++ supports object-oriented programming but does not enforce it. Consequently programmers using C++ may develop an application by using a burdensome structured design approach. Java is a better language also in regard to other features. Java has no pointers, and it provides exception handling, stronger type definition, and multithreading. Java helps developers to extract their documented comments from the code. Java does automatic garbage collection that frees up programmers from having to manage memory permitting them to focus on the problem to solve.

One major feature of Java is its virtual machine. The Java Virtual Machine (JVM) is an application that runs above an operating system (Figure 20). The JVM provides the programmer with a consistent application programming interface (API) irrespective of the underlying operating system. Java source code is pre-compiled into Java bytecode which is interpreted by the JVM. This is similar to PVM, but unlike PVM, the Java compiler is architecture independent. The JVM is a virtual hardware architecture implemented in software. This virtual machine is architecture dependent, but the
bytecode is platform independent. It allows the developer to ignore details of the different operating systems and types of computers that could reside on a network. For example, when sending data between two different types of computers that have different data format representation, the developer does not have to worry about it. The JVM takes care of it. Consequently Java applications are platform independent. This allows Java bytecode to be transported across heterogeneous platforms that have their respective JVM. This concept allows applications to be developed easily on a network of heterogeneous workstations. As a Sun slogan say: "Write (code) once, run everywhere."

As of Nov. 17, 98, JVM has been ported on 22 different operating systems which include the major ones: Microsoft Windows (95/98/NT), Macintosh and several varieties of UNIX (including Linux).

The drawback of using Java is its poor performance due to the slow JVM execution speed. The JVM dynamically interprets the Java bytecode. However, it is
believed that this current shortcoming is only transitory. Since its introduction, Java's performance has continuously been enhanced as improved technologies such as code optimisation, Just-In-Time (JIT) compilation and HotSpot [GRI98] are being developed. JIT compilation is a hybrid interpretation/compilation technique in which a stream of bytecodes is cached during the interpretation process to provide better performance on repeated invocations of a given method. Initially when Java came out on the market, it was 20 to 50 times slower than C/C++. Things have improved since and Java is now about two to five times slower than C/C++ [JOVE].

The performance of Java can also be improved if the Native Interface API (JNI) is used. JNI is a mechanism to allow to access to the OS below the JVM by coding in C. Furthermore, Java source code can be compiled to create native executable files. However, these two approaches defeat the JVM’s purpose of keeping the code platform-independent. Nevertheless, it is understood that it may provide the performance required by some types of applications, e.g., static server applications. Compiled java is said to perform as well as, if not better than compiled Fortran or C++ code [JOVE].

Sun is currently working on JavaOS and JavaChip [JAVA] to eventually provide alternative mode to efficiently execute Java bytecode. JavaOS will be an operating system that directly interprets Java bytecode avoiding the current JVM layer in Figure 20. By having fewer layers in the operation, execution is speeded up. This would also allow optimisation for individual platforms. The JavaChip is a microprocessor that will
directly interpret Java bytecode without the requirement of either an operating system or JVM. This is actually equivalent of implementing the JVM in hardware. This would again reduce the number of layers and allow for some optimisation. These two technologies are not yet widely available, but have been promised by Sun for the near future. These technologies are anticipated solutions to the current lack of execution speed.

Careful consideration was given to networking in Java’s design. Java has five built-in features that help the development of network-based applications: bare socket, RMI, CORBA, serialisation and Classloader Class. The first three allow connectivity and communication between distributed applications. These features will be described in more detail in the next section. The object serialisation mechanism provides a way for objects to be written as a stream of bytes and then later recreated from that stream. This is handy when it comes time to copy and send objects to other Java programmes across the network. Finally the Classloader class dynamically migrates Java code across the network.

Sun is dedicated to make Java the prime programming language of the future. This is why Java API is not governed by an international standard, but benefits from a copyright protection. However, the JVM specification was submitted in March 1997 to ISO/IEC for standardisation. In the mean time, it allows Sun to make changes to the API as they please to improve the API effectively. In the same order of ideas, on December 8,
1998, Sun introduced the Java Community Process for the development and revision of Java technology specifications. The Java Community Process allows the broader Java community to participate in the proposal, selection, and development of Java APIs. Within that context there are groups like Grande Forum [JGO] dedicated to make Java the programming language of choice for large applications. Part of their goal is to recommend changes to the Java language to make it the preferred language for parallel computing, and scientific and engineering application development. For example, they support the development of mathematical libraries for scientific applications to replace current Fortran libraries. It is to be expected that Java will evolve to fulfil developers’ needs in computing, including at least distributed computing, database connectivity, Web applications, standalone applications, reusable code and rapid code generation. Java is thus a subject of active research and a rapidly-evolving language. We can confidently predict that Java will occupy an important niche in tomorrow’s computing world.

Any of the above-mentioned features, taken alone, would not justify the brouhaha over Java. However considering them as a whole, it makes Java, not only a modern language, but an enabling technology to solve today’s computing problems.

5.2.2 Distributed Java

The most interesting aspects of Java for this project are its features for distributed applications. There are four main sub-technologies in Java for developing distributed
applications. These are support for bare sockets, support for CORBA, its RMI technology, and its Javaspace framework. The following paragraphs describe these subtechnologies.

Java has a built-in API that allows easy creation of socket connections between computers. Java's bare sockets are based on the Internet Protocol (IP) socket which is the major networking protocol in use today on the Internet and on most LANs. Before communicating with another computer, the identifier of that computer must be known. The identifier is a logical addressing scheme that corresponds to a computer location on a network. That logical addressing scheme consists of a hostname and a port number. The hostname could be either in a textual or numeric form. The textual hostname is called its Domain Name Services (DNS) name which could be thought of as a kind of alias of its numeric form.

Java supports the different types of protocol: TCP (Transport Control Protocol) and UDP (Unreliable Datagram Protocol) and HTTP (Hypertext Transport Protocol). TCP is a reliable protocol in which data packets are guaranteed to be delivered, and delivered in order. If a packet expected at the receiving end of a TCP socket does not arrive within a set period, then it is assumed to be lost, and the packet is requested from the sender again. The receiver does not move on to the next packet until the first packet is received. Most applications use TCP because of its reliability. UDP makes no guarantees about delivery of packets, or the order in which packets are delivered. UDP
sockets are typically used in bandwidth-limited applications (real-time network audio or video applications). HTTP is a specialised protocol for manipulating HTML documents (HyperText Markup Language).

According to [OAK98] IPs are popular and useful because they incur none of the overhead of other services such as RMI and CORBA which are discussed next. The takeoff is that the application developer is responsible for defining and programming the complete protocol that will be used among the distributed applications. However, this may be considered an advantage occasionally if developers need to have full control of performance for communication between applications.

Common Object Request Broker Adapter (CORBA) is not a distinct feature of Java. It is a middleware that sits between a client and a server application. The concept is meant to be platform and language independent, in the sense that the client stub interface to the object, and the server skeleton implementation of these object interfaces, can be specified in any programming language. The stubs and skeletons for the objects must conform to the specification of the CORBA standard in order for any CORBA-BASED object to access them. Then, this middleware has to be running on each workstation that makes up the overall application. When a client application uses an object, it does not need to know the object’s location, programming language or type of platform because the Object Request Broker (ORB) masks these details. CORBA has various types of interfaces, but a discussion of their nature is beyond the intent of this
thesis. Suffice it to say that CORBA is well/better suited to integrate legacy systems or to provide interfaces among different vendors’ products. And, Java provides support to ease the development of applications that wish to interact with CORBA.

The Java Remote Method Invocation (RMI) is a packaged part of the core Java API. It provides a means to develop distributed applications where all objects are written in Java. RMI is considered as one of the easiest ways to build distributed, cross-platform software that works transparently across heterogeneous networks. With RMI there is a registry server that fulfils the role of the object broker and naming service. The RMI registry is a special Java programme that is delivered as part of the RMI package. It runs as a standalone process on the workstation that is serving objects. Unlike CORBA, the RMI registry is only required to be running on the server of a remote object. To develop a RMI-BASED application, the first thing that has to be done, is to create the remote object. Once a remote object has been written and compiled, it has to register itself in the RMI registry. Once started, the RMI registry programme is always running, waiting inactively for incoming RMI connections. When a client calls a method on a remote object, it is not actually calling the method but connecting to the RMI registry. The registry then looks into its database of registered objects, finds the object that the client wants, and passes the method call to the object, using the parameters received from the client. When the method exits, the registry passes the returned value back to the client. This is completely automatic and transparent. All there is to RMI are the three basic steps: create the remote object; register the remote object, and call the remote methods.
The advantage of RMI is its simplicity. However, its disadvantage is its poor performance. [HAR98] points out that RMI should not be the choice for distributed computing if speed is the ultimate concern. RMI's real power is in its simplicity. Developers do not need to learn a special interface definition language as with CORBA, nor need to learn a complex API as with other protocols. However, it was reported in [HAR98] that if the parameter sizes are small (that is, a few integers are passed instead of a large array of them), the performance of RMI is fairly close to socket-based protocols.

The last Java sub-technology for distributed computing is Javaspace. Javaspace, as a beta version, was only made available to the general public in December 1998. Javaspace is developed on top of Java Jini (another Java sub-technology) that relies on RMI. It provides a shared-memory framework API to develop distributed applications. RMI simplicity has already been commented on in previous paragraphs and the shared-memory framework was also discussed in chapter 2 as a more simple framework for programmer to work-in than the message-passing framework. In that regard Javaspace favours the simplest way of developing distributed application.
5.3 Selection

The choice between PVM and Java sub-technologies, was based on the six criteria that were identified as our requirements: performance, portability, management, interface, support and prospects.

The prime concern was to select a technology that provided the best performance, i.e., minimise communication overhead. PVM and Java’s bare socket are the two technologies that scored highest in that regard. In terms of portability, both PVM and Java fulfil the criteria of the virtual machine concept. However, with PVM, the virtual machine and the application code must be compiled for each different architecture. Although this allows selection of compilers and debuggers that are better tuned for individual architectures, it adds a management burden since different compilers and debuggers for each type of architecture have to be obtained and learned by the programmers. On the other hand with Java, although a different virtual machine is required for different architectures, the Java virtual machines are already compiled when distributed, and the application code is compiled into bytecode with the same compiler for all architectures. In that respect Java is easier to manage. Also about managing software development and maintenance, Java scored better because it is an object-oriented language.
Regarding support, after being on the market for many years, PVM remains a by-product of research with no formal support. Consequently, bugs get fixed on an as-time-allows basis. Also, PVM is not sanctioned by a standard organisation, which partially creates uncertainty about its future compatibility and continuity. Although Java is not yet sanctioned by a standard organisation, it benefits from a copyright protection for Sun Microsystem which is also dedicated to promote its Java technology. Already within a few years of its release, Java has been widely adopted world-wide. In that regard Java scores better than PVM. As for prospects Java also scores better since it is more than a technology for developing distributed applications, it is a new and modern language that offers much more. For example, it is easy to integrate a GUI into a Java application since Java is very well equipped to support the development of GUIs. And Java can distribute applications across the WWW in a secure fashion.

For the above reasons, Java was retained as the technology of choice for developing the distributed SOM. In addition, one of the Java sub-technologies had also to be selected. CORBA was not picked because it is more suited for static applications and for binding legacy systems together. Javaspace could not be selected since it was not available at the beginning of the project. The TCP bare-socket approach with Java was selected as it promised better performance than RMI, particularly, for messages with relatively large payloads which is more likely to be the case with our distributed SOM. With sockets, a complete message-passing protocol has to be developed. However, the protocol is simple in our case.
Although Java with its bare-socket was found to be the best overall technology to choose, Java’s lack of performance when compared with compiled products still is a concern. After all, network parallel computing is about performance. But since its release, Java’s performance has been continuously enhanced, and Sun Microsystems is still aiming at improving it with HotSpot [GRI98] which will be available some time in 1999. In the event, that desired performance is not achieved, Java can always be compiled which will be equivalent to the PVM technology. Consequently the doubts about Java in that regard were considerably dissipated.
6. Design & Implementation

This chapter provides a general description of the design, covering the predominant aspects, but not specific implementation details. The purpose is to provide only a general understanding, sufficient to appreciate the nature of the application. Consequently the following sections describe the major structural entities that make up the application with an explanation of their respective behaviours. For the reasons discussed in the previous chapter, Java was selected over PVM to implement the distributed SOMs. Nevertheless, a bit of the PVM philosophy was incorporated into the design.

6.1 Architecture & Behaviour

The application was developed using an object-oriented approach. A subset of the unified modelling language (UML) [ALH98] is used to facilitate the description of the application. Nevertheless, rather than using class diagrams, a type of deployment and object diagram will be used to present a more concrete description.

The application is made up of three distinct sub-applications which are named: the coordinator, the worker and the daemon. The coordinator application sits on the client side while the worker and the daemon sit on the server side (Figure 21). Most
client/server applications are built with one server and multiple clients. In our application, it is the opposite. The application has one client, and therefore only one coordinator, and several servers which correspond to multiple workers and daemons. All communications between the client and the servers are done through message passing. All workers and daemons are identical copies of the same application but running on different workstations (Figure 21).

![Figure 21 - Distribution of the applications](image)

In general terms, the overall application executes as follows: prior to any attempt to execute the application, the daemon application has to be installed and running on each server. These daemons remain alive as long as the workstation is powered. The user starts the coordinator application on the client. The coordinator application contacts each daemon requesting that a worker application be started. Except for these requests, the daemon applications are asleep and not using any CPU time (since they are waiting for I/O). Once the application completes its overall execution, the coordinator and its workers cease to exist. Only the daemons remain alive awaiting future runs. The
daemon application only exists to ease the automatic start-up of worker applications. There is an option to start manually the worker without daemons. In the manual starting mode, the coordinator application is started first and waits for connection from the worker applications that are subsequently manually started. The manual start of multiple worker applications could be tedious if there are 10 or more, and not practical if the workstations are in different rooms or buildings. On a Unix-based system, the workers could be started using a script, but this is not possible on a PC-based network using Microsoft Windows. The daemons make the process possible and simple for both Unix and PC networks and also make it transparent to the users.

Figure 22 displays the core classes that make up the three applications. The coordinator application includes the Coordinator, Dispatcher, LoadBalancer, and the Client classes. The worker application includes the Server, Worker, and Neuron classes. And, the daemon application only holds the Daemon class.

Besides these sub-applications, four files have to reside on the client side since they are required by the Coordinator. The first file is the list of workstations by hostname on which a daemon has been installed and is running. The second and the third file respectively hold the input values (input file) and the neurons’ weights values (weight file). The fourth file is the configuration file. It holds all the prerequisite information required during execution such as the name of the three other files, the total number of
neurons that has to be created, the number of workstations to be used, the maximum number of iterations, other stopping criteria, enabling/disabling the load balancing feature, mode of load balancing, port numbers to use, etc. All these files are in ASCII format enabling users to examine them. Appendix B offers some samples.

Next we describe each of these core classes in general terms. The Coordinator class’s main role is to ensure synchronisation between the workers and makes the central decision on the global winners after it has collected all the local winners from the
different workers. The Dispatcher is responsible for maintaining a list of workstations where daemons are running. The LoadBalancer contains the algorithm that calculates the quantity of neurons individual workers should have. The Client and Server classes act as mediators between the Coordinator and the Worker classes acting respectively as stub and skeleton. The Worker class is the one that holds the neurons and does all the hard computing. The Neuron class consists of an identification number and an array of real numbers representing its weight vector. The Daemon class monitors incoming communication requests.

A typical execution of the program follows the sequences of messages shown in Figure 23. At the top of the figure are the different objects and below is the message sequence. Initial messages are on top and subsequent messages are added below. The entire process starts at the coordinator's end beginning with the initialisation phase. The initialisation phase consists of creating workers and distributing the workload (inputs and neurons) as follows:

a. Upon creation, the coordinator causes the dispatcher to create workers;

b. The dispatcher then contacts all the different daemons using the information in the configuration and host files;
Figure 23 - Message Sequence

c. Upon contact, daemons receive the hostname and the port number where the dispatcher is located. With that information in-hand, the daemon starts a worker application locally as a separate process;
d. On instantiation, the server object contacts the dispatcher using the information passed along to make a socket connection. Once the socket connection is made, on the dispatcher side, the socket object is passed to a newly created client object. Consequently the client and the server are bound through the socket. The socket connection is not released until the program completes its execution;

e. Subsequently the server creates its workers;

This message sequence is repeated as often as there are workers to be created on the different workstations. Consequently there are as many client objects residing on the client side as there are workers created on the different workstations (see Figure 24);

![Figure 24 - Object layout.](image-url)
After creating all the workers and associated clients, the coordinator reads the attributes of all the individual neurons from the weight file and distributes them evenly to all the workers. Then from the input file, it makes a copy of all the inputs available to each worker. The inputs and neurons' weights are distributed to the workers through the client and server objects.

After completing the initialisation phase, the coordinator goes into the learning phase which is a loop calling the same messages repeatedly until some stopping criteria are met. Within the loop the following actions are carried out:

a. The coordinator requests the local winners from each client object. Each client object contacts their respective server objects with this request. This message is non-blocking which means that once the request has been received and acknowledged, the coordinator is free to carry on with similar requests to another client. Once all these requests have been made, the coordinator waits for the first reply to be received.

b. Concurrently each server upon reception of the request forwards it to their worker. The workers compute and return the answer to their server which in turn forwards it to their associated client.
c. The coordinator pulls out all replies sequentially. Once all the replies have been received, the coordinator gathers the data to identify which neurons are the global winners.

d. Before sending the list of global winners, the Coordinator requests the LoadBalancer to redistribute the neurons between the workers as required. This is accomplished by two consecutive messages; mesureSpeed() and balance(). On receiving these messages, the loadBalancer queries each client for parameters that will be used to compute the load balancing. Depending on the result of calculation, the loadBalancer may decide to remove neurons from some workers and add neurons to other workers. When the load balancing is completed, the loadBalancer returns control to the coordinator.

e. The coordinator then sends each worker the list of global winners. This message is also non-blocking. Upon receiving the list of global winners, each worker executes the required calculation to change the value of each neuron’s weight vector. This completes one iteration of the learning loop.

Once one of the stopping criteria is met, for example a maximum number of iterations set in the configuration file, the loop ends and the coordinator requests a copy of the neurons from all workers. On reception, the coordinator writes the attribute values of each neuron into the weight file.
6.2 Additional classes

The preceding paragraphs have explained the core structure and functionality of the developed application. However, there are some additional classes constructed to encapsulate better some of the application functionality. Figure 25 provides a class diagram of all the developed entities and their respective relationships. The Executive class actually contains the main method for the coordinator application. It first gives control to the Auditor which performs an audit on the four ASCII files. The Auditor checks for the files existence, proper formatting and data validity. Upon a successful
audit, the Executif passes control to the coordinator object. The way the coordinator reads vectors from the ASCII files is encapsulated in the class VectorReader. The class SomException encapsulates exceptions that could be generated across the application. All exceptions are propagated to the Coordinator which deals with them according to their nature. The mathematical equations required by the worker for calculating the distance between neurons and vectors are encapsulated in their respective classes, NeuralDistance and VectorUtilities. And the transfer function that controls the learning rate is encapsulated in the AffinityFunction class.

This chapter explained the main aspects of the design necessary for a general understanding of the structure and functionality of the application. Detailed information on implementation of each class can be found in the source code as it has been heavily commented using JavaDoc features and formatting. As per Appendix D, this thesis comes with a CD-ROM that contains the source code.
7. Load Balancing Schemes

This chapter describes how we have modelled communication patterns and load balancing schemes over a network of workstations, more specifically how to distribute the neurons to the workers. The purpose of load balancing is to improve performance or in other words to reduce the time required to perform a cycle of the SOM algorithm.

There are different possibilities and different ways to perform the tasks. For this work, three parameters are used for modelling a network of workstations: the number of workstations, the different speeds of each workstation, and how much time required for communication between two workstations. Given these three parameters, we have discerned six possible scenarios which we refer as cases. The first four scenarios are analysed and six load balancing schemes are developed, while for the last two, only comments are made. The following sections provide an analysis of each case. These load balancing schemes might also be used for other applications that display the same communication patterns as our implemented SOM.
7.1 Cases

Here is a list of definitions used throughout this chapter.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>number of workstations</td>
</tr>
<tr>
<td>n</td>
<td>number of neurons</td>
</tr>
<tr>
<td>p</td>
<td>time of execution of a partial task</td>
</tr>
<tr>
<td>P</td>
<td>total time for executing the parallel part of a task</td>
</tr>
<tr>
<td>a</td>
<td>time of communication for the request</td>
</tr>
<tr>
<td>g</td>
<td>time for combining replies</td>
</tr>
<tr>
<td>c</td>
<td>time of communication for the reply</td>
</tr>
<tr>
<td>I_j</td>
<td>time required to perform a cycle for case 'j'</td>
</tr>
<tr>
<td>i</td>
<td>worker's identifier</td>
</tr>
<tr>
<td>S</td>
<td>time for executing the sequential part of a task</td>
</tr>
<tr>
<td>s</td>
<td>speed of a workstation</td>
</tr>
<tr>
<td>SU</td>
<td>speedup</td>
</tr>
<tr>
<td>SU_m</td>
<td>maximum expected speedup</td>
</tr>
<tr>
<td>N_m</td>
<td>number of workstations that provide SU_m.</td>
</tr>
<tr>
<td>W</td>
<td>worker</td>
</tr>
<tr>
<td>Coor</td>
<td>coordinator</td>
</tr>
</tbody>
</table>

7.1.1 Case 1

This case describes a network of workstations where all the workstations have the same execution speed (homogeneous network). The intent of the load balancing scheme here is to distribute evenly the load (neurons) so the time each workstation actually works is the same.
Figure 26 - Load balancing scheme for Case 1.

Figure 26 shows the timeline of events that take place between the coordinator and his ‘N’ workers. The small circles represent communications between the coordinator and his workers. The communication time includes the transmission, propagation and receiving time. The first message is sent to the first worker and similar messages are subsequently sent to the other workers. This first set of messages would be the signal that the coordinator sends to the workers to start the computation. This first message, the request, takes “a” time unit. All the messages consist of a message time overhead “O” plus the time to transmit the payload “L”. However, at this stage we can omit this as unnecessary details. Then each worker proceeds with its respective computation that lasts “p” time units. Assuming an even load on each worker with the same execution speed, this computation time is the same for all workers. Once the first worker has finished its computation, it sends its reply message to the coordinator. This reply message takes “c” time units for its transmission and propagation. All the other workers do the same upon completing their computation. Because all the messages are
sent sequentially on a shared communication medium (the network bus), a worker that is ready to communicate is required to wait until any ongoing communication is completed. This is due to the fact the coordinator can only entertain one communication at the time. Once the coordinator has received the last reply message then it compiles the final result from all the replies. To perform this final computation the coordinator takes \( Ng+S \) time units. The time \( g \) corresponds to the time required by the coordinator to combine all the replies, i.e., take all the lists of local winners and make a single list of global winners. Using the above figure as a help, the following principle can be derived. If \( N \) workers are assumed, then the time corresponding to the sequential computation is \( Np + S \), where \( (Np) \) corresponds to the portion of the computation time performed in parallel and \( S \) corresponds to the remaining computation that can not be performed in parallel. Consequently the total time for the sequential computation is \( S + Np \) while in parallel it is \( a+p+Nc+Ng+S \). The speedup can be derived by calculating the time required for the sequential and parallel computations.

The Figure 27 shows the steps needed to carry out this calculation. Based on this load balancing scheme, equation 1, the maximum speedup that can be obtained corresponds to equation 5. It is to be noted for the implementation that if \( n \) is not a multiple of \( N \) then \( n_i \) is not an integer, and additional adjustment is required to account for these fractions.
Since, by hypothesis, the speeds of the workstations are equal,
\[ s_i = s, \forall i \]
and each worker computes for the same time, as per figure 26.

(eq. 0)

\[ p_i = p_1, \forall i \]

\[ p_1 \] corresponds to the execution time of the first worker.

The total number of neurons, \( n \), is always fixed,

\[ n = \sum_{i=1}^{y} n_i = \sum_{i=1}^{y} s_i p_i = Nsp_i \]

consequently, \( p_1 = \frac{n}{sN} = \frac{p}{N} \)

therefore, neurons are distributed as follows

\[ n_i = s_p_i = sp_i = \frac{n}{N} \]  

(eq. 1)

The time required to complete a cycle in this first case is

\[ l_i = a + p_i + Nc + Ng + S = a + \frac{n}{sN} + N(c + g) + S \]  

(eq. 2)

the associated speed-up is

\[ SU = \frac{S + P}{a + \frac{n}{sN} + N(c + g) + S} \]  

(eq. 3)

by deriving the speed-up in regard to the number of workstations, we establish the number of workstations required to provide a maximum speed-up.

\[ \frac{\partial SU}{\partial N} = \left[ - (S + P)(- \frac{p}{N^2} + c + g) \right]^{1/2} \]

\[ \frac{\partial SU}{\partial N} \] is equal to zero when

\[ N_a = \sqrt{\frac{P}{c + g}} = \frac{p_1}{c + g} \]  

(eq. 4)

by inserting \( N_a \) back into the speed-up equation, we define the maximum speed-up that can be obtained in this case

\[ SU_a = (S + P)/\left( a + S + 2\sqrt{P(c + g)} \right) \]  

(eq. 5)

if we neglect \( a \) and \( S \), then \( SU_a \approx \frac{1}{2} N_a \)  

(eq. 6)

Figure 27 - Equations for case 1.
This case makes the same assumption as in case one except that each workstation

\[ n = \sum_{i=1}^{N} n_i = \sum_{i=1}^{N} p_i s_i = p_i \sum_{i=1}^{N} s_i \]

consequently, \( p_i = \frac{n}{\sum_{i=1}^{N} s_i} \)

Neurons are then distributed such that:

\[ n_i = p_i s_i = p_i s = \left( \frac{n}{\sum_{i=1}^{N} s_i} \right) s_i \quad (\text{eq. 7}) \]

The time to complete an iteration in this second case is:

\[ I_2 = a + p_i + Ng + S = a + \frac{n}{\sum_{i=1}^{N} s_i} + N(c + g) + S \quad (\text{eq. 8}) \]

the corresponding speed-up is:

\[ SU = \frac{S + P}{a + \frac{n}{\sum_{i=1}^{N} s_i} + N(c + g) + S} \]

Note: Speedup is a relative measure, and in this case the execution time of the single machine would have to be the fastest one among the heterogeneous set of workstation.

As per the first case through derivation, the optimal number of workers to provide the maximum speed-up is found:

\[ N^*_w = \frac{n}{(c + g) \sum_{i=1}^{N} s_i} \quad (\text{eq. 9}) \]

which results in a maximum speed-up of

\[ SU^*_w = \frac{S + P}{a + S + \frac{2n}{\sum_{i=1}^{N} s_i}} = \frac{N^*_w}{2} \quad (\text{eq. 10}) \]

Figure 28 - Equation for case 2.
has a different execution speed (heterogeneous network). Therefore loads are distributed unevenly so that all workers can compute for the same amount of time. Consequently the related equations are shown at Figure 28. The load balancing is done as in equation 7. The speedup equation remains the same as in case one (eq. 3) however with a different value for \( p_f \). In equation 7 if all \( s_i \) are equal to a constant 's', then this case becomes equivalent to case one.

7.1.3 Case 3

Both in case 1 and 2, if we refer to Figure 26, there is wasted time because the workers are required to wait for the coordinator to be available for communication. To remedy this inefficiency, it is required that the communications time be contiguous, as per Figure 29. This communication pattern prevents the workers and the coordinator from waiting while they could be computing. The load balancing scheme proposed in this case 3 and the one that is going to be proposed in case 4 are to remedy to this inefficiency.

Figure 29 - Load distribution minimizing wasted time.
In case 3, we consider that all the workstations have the same speed (homogeneous network). The load balancing scheme here distributes the load unevenly to each workstation, so that one workstation finishes executing when the previous one finishes communicating with the coordinator. Then no workers are waiting to communicate with the coordinator at any given time. In this case the load balancing and the time to complete a cycle are as per Figure 30. And Figure 31 shows the associated equations.

Figure 30 - First Load Distribution scheme for Case 3.
Since, by hypothesis, the speeds of the workstations are equal,
\[ s_i = s, \forall i \]
and individual worker, \( i \), computes for the time
\[ p_i = p_i + (i-1)(c-a) \] as per figure 30.
As always, the total number of neurons \( n \) is fixed
\[ n = \sum_{i=1}^{N} n_i = \sum_{i=1}^{N} p_i s_i = s \sum_{i=1}^{N} p_i = s \left( N p_1 + (c-a) \frac{N^2 - N}{2} \right) \]
This yields
\[ p_i = \frac{n}{sN} - (c-a) \frac{N-1}{2} \]
The fact that \( p_i \) must be greater than zero puts an upper limit on the value of \( N \).
\[ 1 \leq N < \frac{1}{2} + \frac{1}{2} \sqrt{1 + \frac{8n}{s(c-a)}}, \text{ with } a < c \]
neurons are distributed as follows:
\[ n_i = s_i p_i = s \left[ p_i + (i-1)(c-a) \right] = s \left[ \frac{n}{sN} + (c-a) \frac{2i - N - 1}{2} \right] \]
The time to complete an iteration is then,
\[ I_3 = a + p_i + N(c + g) + S \]
\[ I_3 = \frac{n}{sN} + \frac{(N+1)}{2}(c+a) + Ng + S \] \hspace{1cm} (eq. 11)
and, the speed-up is as follows.
\[ SU = \frac{\frac{n}{sN} + \frac{(N+1)}{2}(c+a) + Ng + S}{S + P} \]
The maximum speed-up occurs when
\[ N_\ast = \sqrt{\frac{p}{c+a+2g}} \]

Figure 31 - Equation for case 3's first load distribution scheme.

It can be proved by simple arithmetic that \( I_3 \) (eq. 11) is always smaller than \( I_1 \) (eq. 2) as long as \( a < c \). Better performance can be achieved if 'g' is executed concurrently.

Then, this second load balancing scheme with its associated equations can be applied for case 3.
Since, by hypothesis, the speeds of the workstations are equal.

\[ s_i = s, \forall i \]

and individual worker, \( i \), computes for the time

\[ p_i = p_i + (i-1)(c-a+g) \] as per figure 32. (eq. 12)

As always, the number of neurons \( n \) is fixed.

\[ n = \sum_{i=1}^{n} n_i = s \sum_{i=1}^{n} p_i = sNp_i + s(c-a+g) \left( \frac{N-1}{2} \right) \]

This yields

\[ p_i = \frac{n}{sN} - (c-a+g) \left( \frac{N-1}{2} \right) = \frac{p}{N} + (1-N) \left( \frac{c-a+g}{2} \right) \]

The fact that \( p_i \) must be greater than zero puts an upper limit on the value of \( N \):

\[ \frac{1}{s} \leq N < \frac{1}{2} + \frac{1}{2} \sqrt{1 + \frac{8P}{c-a+g}} \] (eq. 12a)

neurons are distributed as follows

\[ n_i = p_i t = sP_{\text{neuron}} \frac{n}{N} + s(c-a+g) \left( \frac{2i - N - 1}{2} \right) \]

The time of one iteration would then be

\[ t_i = a + \frac{p}{N} + (1-N) \left( \frac{c-a+g}{2} \right) + N(c+g) + S \] (eq. 13)

and, the speed - up is then as follow

\[ SU = \frac{S + P}{t_i} \]. \( SU \) is maximal when \( N_u = \sqrt{\frac{2P}{c+a+g}} \) (eq. 14)

derived as in case 1, the maximum speed - up is

\[ SU_u = \frac{S + P}{a + S + \sqrt{2P(c+a+g)} + \frac{1}{2}(c+a+g)} \] (eq. 15)

if we neglect \( a \) and \( S \), then \( SU_u = \frac{N_u^2}{2N_u^2 + 1} = \frac{N_u^2}{2} \) (eq. 16)

---

Figure 32 - Second load distribution scheme for Case 3.

Figure 33 - Equations for Case 3's second load distribution scheme.
Again by simple arithmetic, we can prove that $I'_3$ (eq. 13) is always smaller than $I_3$ (eq. 11). Consequently we can demonstrate that $SU_3$ is larger than $SU_1$ by approximately square root of two if we use eq. 5 and eq. 15.
7.1.4 Case 4

This case is like case 3 except that the speeds of the workstations differ. We also use the same communication pattern as in Figure 32 since it provides better performance.

Individual worker, $i$, computes for the time,

$$p_i = p_1 - (i - 1)(c - a + g) \quad \text{as per figure 32.} \quad \text{(eq. 17)}$$

and, as always, the total number of neurons is fixed,

$$n = \sum_{i=1}^{N} n_i = \sum_{i=1}^{N} s_i p_i = p_1 \sum_{i=1}^{N} s_i + (c - a + g) \sum_{i=1}^{N} ((i - 1)s_i)$$

by isolating $p_1$, we get the following

$$p_1 = \frac{n - (c - a + g) \sum_{i=1}^{N} ((i - 1)s_i)}{\sum_{i=1}^{N} s_i}$$

again, $p_1 > 0$ which puts an upper limit on $N$.

Time to complete one iteration is:

$$I_4 = a + p_1 + N(c + g) + S \quad \text{(eq. 18)}$$

associated speed-up is:

$$SU = \frac{(S + P)}{I_4}, \text{ however } SU_m \text{ and } N_m \text{ can't be evaluated}$$

without knowing the individual worker's speed.

Figure 34 - Equations for case 4 using the same load distribution as the second one proposed in case 3.
than the first one proposed for case 3. Again if all workstations have the same speed then \( I_4 \) (eq. 18) becomes equal to \( I'_1 \) (eq. 13).

### 7.1.5 Optimal communication pattern for Case 4

In a case like this one where the workstations have different execution speeds, should faster workstations be receiving more neurons? The answer lays in the following reasoning. Consider the two workstations numbered 'i' and 'j' which have adjacent final communications, as illustrated in Figure 35.

![Figure 35](image)

**Figure 35** - Load distribution in regard to workstations' speed.

Clearly they can interchange the time at which they communicate without affecting the rest of the workstations. As it is, station 'j' can treat \( c \cdot s_j \) neurons while station 'i' communicate, i.e., between \( t_i \) and \( t_j \). If they were to interchange their time of communication, station 'i' would treat \( c \cdot s_i \) neurons between \( t_i \) and \( t_j \). Thus, evidently the
fastest station should be the one computing between \( t_1 \) and \( t_2 \), i.e., be the last one to communicate. A similar argument holds with respect to the communication time ‘a’, implying that the fastest workstation should first receive the communication.

The arguments of the previous paragraphs imply that the following communication pattern favours a greater speedup. The first worker \((W_1)\) has the fastest execution speed while the last worker \((W_N)\) has the slowest execution speed. This results in the communication pattern in Figure 36. An additional benefit of this scheme is that the work load of the last worker \((W_N)\) can be assigned to the same workstation which hosts the coordinator since both computational intervals complement each other. This arrangement makes the most efficient use of the workstations.

![Figure 36 - Case 4 optimal load distribution scheme.](image-url)
The $i$'th worker computes for the time

$$p_i = p_1 - (i-1)(c+a+g) \quad \text{as per figure 39.} \quad \text{(eq. 19)}$$

As always, the total number of neurons $n$ is fixed:

$$n = \sum_{i=1}^{N} n_i = \sum_{i=1}^{N} s_i p_i = p_1 \sum_{i=1}^{N} s_i - (c+a+g) \sum_{i=1}^{N} s_i (i-1)$$

Thus solving for $p_1$,

$$p_1 = \frac{n - (c+a+g) \sum_{i=1}^{N} (1-i)s_i}{\sum_{i=1}^{N} s_i}$$

Again, $p_1 > 0$ puts an upper limit on $N$, but this limit can't be evaluated without knowing the individual worker speeds. Neurons are then distributed as follow:

$$n_i = p_i s_i = \frac{ns_i}{\sum_{j=1}^{N} s_j} + (c+a+g) \left( \frac{\sum_{j=1}^{N} j s_j}{\sum_{j=1}^{N} s_j} - i \right) s_i$$

The time required to complete a cycle is then:

$$I'_i = a + p_1 + c + g + S \quad \text{(eq. 20)}$$

The maximum speed-up $SU_m$ and number of stations $N_m$ can't be evaluated without knowing the individual worker speed.

Figure 37 - Equations for Case 4 optimal load distribution.
7.1.5 Case 5 and 6

Case 5 and 6 apply to situations of load balancing on a network where the time of communication for identical messages is different from one workstation to another. To implement a load balancing scheme for this type of situation, we would need to measure individual workstations for their speed and communication time. We chose not to invest energy in developing a scheme for that type of network for two reasons: First, we take for granted that within a single LAN the communication time for all workstations is approximately the same. Second, in the event that the workstations are located across multiple LANs and these LANs are interconnected there could be different communication time depending on the workstation location relative to the coordinator. In such situations it would be better to develop a different application architecture that would support multiple coordinators located on each LAN. Within each LAN, workers would be able to communicate with their coordinator without interfering with other workers on the other LANs thus allowing for additional parallelism and inevitably providing additional speedups. Each of these coordinators would be managed by a master-coordinator.
7.2 Load Balancing Algorithm

In each of the above cases there was an equation that defines the best number of neurons \( n_i \) that should be assigned to each worker. If at a time workers are currently holding more neurons than this ideal number then the appropriate number of neurons is removed. If workers are missing neurons with regard to the best number, then neurons are added. This task is described by the following algorithm:

\[
\begin{align*}
&\text{for } (\text{Worker } w = 1 \text{ to } W) \\
&\quad \text{currently} (w) := \text{get\_current\_nbs\_of\_neurons(} \text{worker}(w)) \\
&\quad \text{desired}(w) := \text{calculate\_desired\_nbs\_of\_neurons}(w) \\
&\quad \text{if}(\text{currently}(w) > \text{desired}(w)) \\
&\quad\quad \text{buffer} := \text{remove\_neurons}(\text{currently}(w)-\text{desired}(w), \text{worker}(w)) \\
&\quad \text{end-if} \\
&\text{end-for} \\
&\text{for } (\text{Worker } w = 1 \text{ to } W) \\
&\quad \text{if } (\text{currently}(w) < \text{desired}(w)) \\
&\quad\quad \text{add\_neurons}(\text{desired}(w)-\text{currently}(w), \text{buffer}, \text{worker}(w)) \\
&\quad \text{end-if} \\
&\text{end-for}
\end{align*}
\]

**Figure 38 - Load Balancing Algorithm.**
7.3 Performance Limiting Factors

Now that many load balancing schemes have been evaluated and their respective speedup equations derived, we can further analyse the limiting performance factors. On several occasions we have suggested that the parameters ‘a’ and ‘S’ could be neglected. By neglecting ‘a’, we were reducing the precision of the equations. However the omission of ‘S’ does not introduce any error, since ‘S’ is always equal to zero in a SOM. SOM neural networks inherently have no sequential parts; all its processes can be parallelized. The parameter ‘S’ was only retained for form and completeness.

It can be observed in equation 4 and equation 14 that the factor limiting the speedup is the ratio of the processing time over the communication overhead. If we express that time ratio in terms of number of neurons and inputs using equation 4, we get equation 21 that provides some understanding of how performance may be affected by factors characterising a neural network. The parameter ‘n’ stands for the number of neurons, ‘i’ for the number of inputs and ‘d’ corresponds to the dimension of the neurons’ weight vector. The parameter ‘k’ is fixed by the type of workstations. The parameters ‘l’ and ‘L’ are related to the communication. The parameter ‘l’ contributes to increase the communication time based on the message load which is proportional to ‘i’. The parameter ‘L’ is the communication overhead of a message without any load. Finally ‘q’ is the factor based on the number of inputs and workstation’s processing speed ‘s_p’ that provides the value for ‘g’. The speed by which communication take place is annotated by
The parameters 'd', 'n', 'i' are set by the SOM problem; 'k', 's_p', 's_c' and 'q' are characteristics of the network of workstations. By increasing 's_p', the number of workstations required to produce the maximum speedup will be reduced. Also if 'n' or 'd' is increased, so is 'N'. This means that, for a given network, the larger the size of the SOM problem, the greater the maximum speedup, excluding consideration of the maximum number of workstations available on the network. Finally, for a smaller communication speed 's_c', N will decrease.

Although 'l' is fixed by the hardware, it could be virtually reduced. For example if the message load is compressed by 'zipping' it, this would be the equivalent of reducing 'l'. If a compression ratio of 50% is obtained, then 'N' is increased approximately by the square root of two (if 'q' is neglected). We have previously proven

\[
N_* = \sqrt{\frac{P}{c + g}}
\]

\[
N_* = \sqrt{\frac{k \cdot (i \cdot n \cdot d)}{s_p}}
\]

\[
N_* = \sqrt{\frac{l \cdot i}{s_c} + L + \left(\frac{q}{s_p} \cdot i\right)}
\]

\[
N_* = \sqrt{\frac{k(n \cdot d)}{s_p + L s_p + q}}
\]

since the overhead becomes negligible as \(i\) increases.

\[
N_* \approx \sqrt{\frac{k \cdot n \cdot d}{s_p + q}}
\]

(eq. 21)
that if uneven load balancing is performed instead of even load balancing, an additional square root of two is gained in terms of speedup. By performing both the compression and the uneven load balancing, the speedup can almost be doubled from the even load balancing scheme with its message load uncompressed. This increased speedup should be verified through actual experimentation.

7.4 Summary

This chapter has documented various load balancing schemes developed to fit various communication patterns and to maximise the speedup. Although not all speedup equations have been derived, each load balancing scheme was evaluated to determine if cycles were executed in a shorter period, which implies a greater speedup. The optimal number of workstations for each case is outlined in the following table. It is shown that the maximum speedup on a distributed computer is approximately half the optimal number of workstations used. On a homogeneous network, the equations demonstrate that an approximate speedup of 1.4 is further gained by adapting the load balancing scheme to fit communication patterns of a typical Ethernet network bus. This demonstrates that load balancing has a considerable performance impact on network parallel computing. The mathematical equations also provide an understanding of how network parallel computing performs for the SOM and what are the performance limiting factors.
<table>
<thead>
<tr>
<th>Models</th>
<th>Strategies</th>
<th>Execution time for one cycle</th>
<th>Optimal nbs of stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous Network</td>
<td>CASE 1 neurons evenly distributed to obtain same computing time among workstations.</td>
<td>$a + \frac{p}{n} + N(c + g)$</td>
<td>$\frac{p}{\sqrt{(c + g)}}$</td>
</tr>
<tr>
<td></td>
<td>CASE 3 neurons unevenly distributed to fit communication patterns.</td>
<td>$\frac{p}{N} + \frac{1}{2} \frac{1}{2}(c + a + g)(N + 1)$</td>
<td>$\frac{2p}{\sqrt{(c + a + g)}}$</td>
</tr>
<tr>
<td>Heterogeneous Network</td>
<td>CASE 2 neurons unevenly distributed to obtain same computing time among workstations.</td>
<td>$a + \frac{n}{N} + N(c + g)$</td>
<td>$\frac{n}{(c + g) \sum_{i=1}^{N} s_i}$</td>
</tr>
<tr>
<td></td>
<td>CASE 4 neurons unevenly distributed to fit optimal communication pattern.</td>
<td>$\frac{n}{\sum_{i=1}^{N} s_i}$</td>
<td>can’t be evaluated without knowing individual workstation speeds.</td>
</tr>
</tbody>
</table>

Execution time for one cycle Optimal nbs of stations
8. Results

8.1 Methodology

The developed application was tested for functionality and performance. The application was used to solve a typical TSP problem using a predefined set of TSP inputs available at [TSPLIB]. By using a predefined set of input data, comparing our results with already existing solutions was possible. The results were then plotted and visually inspected to confirm their validity. These two tests were designed to confirm that the appropriate functionality had been implemented. To make it easier to inspect the results visually, we chose a problem with a small number of cities, “pcb442.tsp.”

The performance test consisted of time measurements to validate the mathematical models developed for different load balancing schemes. Time measurements provided in milliseconds were obtained by making system calls through the Java API. Workstation time clocks were not synchronised across the network, so only time measurements on individual machines are meaningful. The measurement of the communication overhead time between two workstations is derived as follows:

a. the host (client) sending a message records its local time (Si);
b. upon reception, the host (server) records its local time (Ri);

c. when finished processing the request but before sending the reply, the server extracts its local time again (Rf), subtracts ‘Ri’ from it and records the result as the ‘processing time’ (Pt = Rf-Ri);

d. the server sends the reply to the requesting client followed by the processing time (Pt);

e. the client upon receiving the reply records its local time (Sf) a second time.

f. the overhead time corresponding to the above communication is (Sf - Si -Pt).

Figure 39 - Time measurement methodology.
Referring to chapter 7, the above approach would correspond to measuring the parameters 'a' and 'c' together as a single value. Although the above measurement scheme produces time measurements, it remains approximate. The experiment is carried out on a general purpose network environment, where neither the individual workstations nor the network is dedicated to the experiment. As a result, raw performance or speedup is hard to measure. Even in a dedicated networked environment, with no external use, it remains difficult since operating system activities, window and file system overheads, and administrative network traffic can distort measurements. Nevertheless these experimental conditions remain acceptable since it was possible to duplicate the results on separate occasions, thus validating the methodology.

To validate the mathematical equations 3 and 14 in chapter 7, the size of the TSP problem had to be restrained, so the optimal number of workstations for the problem had to be close to, but less than, the number of available workstations. After some preliminary trials, it was determined that a problem with a set of approximately 3000 neurons and inputs was appropriate. Consequently problem ‘pcb3038.tsp’ was selected from [TSPLIB]. By validating the equations developed for a homogeneous network, we assumed the equations for the heterogeneous network would also turn out to be correct considering all equations were derived from the same theory and following the same methods. This alleviates the requirement to measure individual workstations’ speed. Nevertheless, only the load balancing scheme proposed in case 1 and 4 were actually implemented into the source code. In the event all workstation speeds are similar, the
case 4 becomes equivalent to case 3 as demonstrated mathematically in chapter 7. In the same manner, if the parameters 'a', 'c' and 'g' are set to zero for the load balancing calculation then this is equivalent to case 2.

The SOM algorithm produces a sequence of identical cycles. Therefore, measuring a reduced number of cycles is sufficient to evaluate the relative overall execution time for the given problem. The application was thus configured such that only 15 cycles were executed. Time measurements were extracted from output results produced at each cycle by averaging the last 10 cycles. The first five cycles were not considered since they are irregular due to the initial load balancing.

The computational time required by a single workstation with the communication time associated with one communication was to be measured to be able to compare with the theoretical computational and communication delay. These measurements were accomplished by using two workstations. The first workstation was acting as the coordinator and the second as the worker. The worker was getting all the neurons. The worker measured its required computational time, then sent back to the coordinator the results. At the same time the coordinator was able to measure the communication overhead of one communication as per the above described measurement scheme. With these two time measurements it was then possible to compare with the theoretical speedup curves.
8.2 Experimental Results

There were four homogeneous networks available for the testing. The first network was composed of 30 Sun Microsystems workstations, model Ultra30 running the operating system Solaris 2.6 in conjunction with the JVM 2. The second network was composed of 18 Sun Ultra1 workstations also running the operating system Solaris 2.6 with the JVM 2. The third network was composed of 11 PCs, Pentium II (266 MHz CPU) running Microsoft NT 4 and JVM version 1.1.6. And the fourth network was composed of 12 PCs Pentium (90 MHz CPU) running also Microsoft NT 4 and JVM version 1.1.6. The workstations in all four sub-networks were interconnected by a typical 10Mbps Ethernet network bus. Most testing was done on the first mentioned network since it had the greater number of homogeneous workstations.

8.2.1 Functionality

Figure 40 shows the provided solution to the problem ‘pcb442.tsp’. The 442 cities are represented by dots and the course is represented by the links between cities. All neurons were able to converge toward a city providing a solution to this TSP. The travel distance of this course is 59592 units which is 17% more than the best path, 50778 units, ever recorded for this problem [TSPLIB]. Through visual inspection, it is easily seen that the solution is not optimal. A better solution might be obtained by modifying the SOM algorithm parameters such as the learning rate. For example, slower learning
rates normally provide better solution. However, trying to optimise the results is beyond the scope of this research. Nevertheless the current solution shows that all the neurons’ weight vectors converged toward a city, demonstrating the functionality of our implementation.

Figure 40 - Obtained solution for 442 cities TSP.
8.2.2 Performance

Figure 41 shows the execution time for one cycle, for a problem where 3038 neurons are evenly distributed among workstations according to load balancing scheme 1. The graph that shows execution time decreases as additional workstations are added, up to a limit of 22 workstations, after which additional workstations cause execution time to increase. The graph shows also the execution time for one cycle when the 3038 neurons are unevenly distributed as in the second scheme for case 3. It shows that the execution
time decreases monotonously until the limit of 28 workstations has been reached. This behaviour is as predicted by equation 12a in chapter 7.

Figure 42 - Relative Speedup for a neural network of 3038 neurons.

Figure 42 shows the speedups obtained for the 3038 neurons when distributed as per schemes 1 and 3. The theoretical curves for both schemes 1 and 3 were drawn using the mathematical models and parameters obtained when minimizing the total quadratic error. For case 1, the theoretical and experimental curves are overlapping. The straight line was added to show the progression for a linear speedup.
Figure 43 - Relative Speedups for neural networks with 5915, 3038, and 1173 neurons.

Figure 43 shows the different speedups obtained for a neural network of 1173, 3038 and 5915 neurons. The network composed of Sun Ultra30 was used for this performance test. The three bottom curves use load distribution scheme 1 for the respective neural network; the top curve is the neural network with 5915 neurons distributed according to the second load distribution scheme proposed for case 3.
Figure 44 - Execution times for a neural network with 1173 neurons on different computer networks.

And the last graphic, Figure 44 shows the execution speeds for the different types of workstations found on each individual network available for testing. In all instances the different networks of workstations were computing for the same neural network of 1173 neurons. All these curves were derived from the actual experimental measurements provided in Appendix C.

8.3 Analysis

Figure 41 and 42 have made the proof that the application did conform to the different models developed in chapter 7. It was established that the load balancing scheme used in case 3 provided a better speedup than the load balancing scheme used in
case 1. The model for case 3 did not account for the overhead related to performing load balancing. In the up-left corner of Figure 41, the curve for case 1 goes slightly under the one for case 3 which suggest an adverse effect of load balancing. However, the effect is negligible and only encountered when a reduced number of workstations is used.

The models also predicted that between the two load balancing schemes the optimal number of workstations would approximately differ by the square root of 2, or 1.4, when the parameter ‘a’ was neglected from the equation. In case 1 the optimal number of workstations was 22. The corresponding number of workstations for case 3 should have been 30. The experiment was able to reach 28 which corresponds to 1.27 instead of 1.4, a 9% difference. This gap is reasonable considering that approximation made by neglecting ‘a’. The gap can also be explained by the fact that the model was developed on the assumption that the load could be subdivided indefinitely when actually it has a limited granularity, i.e., fractions of neurons can not be distributed. According to the theoretical model all workstations would have to be able to process fractions of neurons and the last workstation would have an infinitely small number of neurons (or a fraction of a neuron) which is not possible. Similarly, the models also predicted a speedup by the same 1.4 factor and now we can understand why this could not be achieved. Nevertheless if the 1.27 factor is used instead then the models predict that based on the speedup of 10.9 obtained in case 1, the load balancing scheme of case 3 should provide a speedup of 13.8. This corresponds to a 0.7% difference from the actual experimental value of 13.7.
The parameters obtained as result of adjusting the theoretical curves are 99.42 msec/neuron and 667.72 msec for the communication delay for case 1. In regard to case 3, the parameters are 102.11 msec/neurons and 838.07 msec. The two speeds values are consistent with the one measured $309190 / 3038 = 101.77$ msec/neuron (see Appendix C). The measured communication delay was 687 msec which is relatively close to the derived value of 667.72 in case 1. However, this is less consistent with 838.07 found for case 3. This is attributed to a slightly different implementation than the mathematical model. Despite the small gap between the theoretical and experimental curves, these results demonstrate that the mathematical equations are in effect valid models that can approximate and predict speedups for a given neural network.

Figure 43 shows that the first part of the speedup curves follows the linear speedup line which also corresponds to almost 100% efficiency. Larger neural networks have a longer zone where a linear speedup can nearly be obtained. This was again predicted by equation 21. Considering that normal SOM problems involve very large numbers of neurons and that typically available computing resources within an organisation are limited, users are more likely to work in this linear zone. This is definitely encouraging being able to use available resources close to 100% efficiency. The linear speedup zone exists because within that range, the communication overhead has no detrimental effect on the overall speedup. Consequently the selection of a technology with more communication overhead may not adversely impact the speedup performance within that range. Therefore technology which we had initially put aside
because of their large overhead, such as Java RMI and Javaspace, may become viable choice if the user expects to work within that range. However due to the problem size and available computing resources, if an application operates near the top of the speedup curve then the current technology would still provide the best performance. The linear speedup zone may suggest that load balancing is not advantageous considering that both case 1 and 3 provide the same performance. This would be true only for a homogeneous network. In the context of a heterogeneous network load balancing is still a requirement for optimal performance. Also the linear zone is extended when performing the second load balancing scheme proposed for case 3.

If the problems are smaller or users are not limited by the number of workstations, maximal speedups are obtained at half the number of workstations (on a homogeneous network) which corresponds to 50% efficiency (demarcated by the dashed line in Figure 43). Such poor efficiency may be disappointing, but not if the resources were unused. Using them at 50% efficiency is better than at 0%. For example in most organisations desktop workstations are not used at night. Then, the efficiency would be increased from 0% to 50%.

Now that the mathematical models have been proven valid, the hypotheses supported by equation 21 of chapter 7 in terms of limiting performance factors are looked at more closely. The current testing methodology does not attempt to measure the different parameters found in equation 21 to validate the equation. Nevertheless this
equation provides some understanding of how network parallel computing behaves with neural networks. In that regard the equation suggests that if faster workstations are used then the maximum speedup would be obtained with a reduced number of workstations. This behaviour was slightly observed in Figure 44 between the workstations Pentium II, Ultra 30 and Ultra 1. In that instance we would have expected that the Pentium (90 MHz) would have a greater number of workstations before levelling off. This deviation is because the Pentiums (90 MHz) were equipped with poorly performing network interfaces (network cards, drivers, etc.). Such effect was also predicted by equation 21 when either the parameter ‘l’ is increased or ‘s_c’ is decreased. This suggests that as workstation performance improves, one may project that network parallel computing will become less attractive because the computation-communication ratio would decrease. We believe that this will be rapidly counterbalanced by the fact that network communication technology will also improve to support the current need for faster networking for general computing. An even more important factor is that there will always be bigger computational problems to solve which are less affected by the communication overhead. Consequently, network parallel computing definitely has good prospect. Also equation 21 predicted that compressing message payload (reducing the parameter ‘l’) would increase the speedup. As with the load balancing overhead, it is believed that the overall speedup gained by compression will outweigh the compression time overhead, nevertheless, this should eventually be experimentally demonstrated. Equation 21 is only a quick and dirty mathematical model for neural networks on network parallel computers. Future research should be carried to refine equation 21 and to
validate it fully. In the mean time, equation 21 provides a general understanding of how network parallel computing behaves with neural networks.

These curves also demonstrate that the communication overhead is more detrimental than using less-performing workstations. This observation goes along with Berkeley’s [MAR98] findings which identified that applications are most sensitive to communication overhead. Although this may be seen as bad news, it is also reassuring that poor performance of the JVM is not actually a major obstacle. Java’s lack of performance has been a concern throughout this project. But now that we have a better understanding of how network parallel computing performs for SOM neural networks, this concern is mitigated by the fact that lower performance technologies provide higher computation / communication ratios which in turn allow for more workstations to be used in parallel. According to Figure 44 the execution times on the different single workstation notably differ. However, the parallel implementation permits to a certain extent to reduce that disparity between different type of workstations. This does not eliminate the problem, but it compensates somewhat for it. This suggests that old Pentium should not be thrown away, they only required to be equipped with good performing network cards.

This observation may not hold true for network parallel computing at large. SOMs have inherently a very fine granularity (a single neuron) which allow for ample data partitioning. For computational problems that do not have similar granularity, the
optimal number of workstations that can be used would be reduced. So faster
workstations will still benefit the overall performance in such situations. If the
application operates in the linear zone (predominant with large SOMs) then the same
thing applies, faster workstations remain beneficial.

8.4 Summary

The experimental results have demonstrated that the developed mathematical
equations are valid models which predict network parallel computing performance for
SOMs, and linear speedup can be achieved for large SOMs. As depicted by the
mathematical models, performance is proportional to the ration of computation over
communication time, and the results show that performance is more sensitive to
communication overhead. In that respect, network parallel computing performance
benefit from faster workstations, but benefit even more from faster network
communication. Although equation 21 formulated in chapter 7 provides a good
understanding of how network parallel computing behaves with SOMs, further research
work should be carried to validate the equation fully.
9. Improvements

After an overall review of the currently developed product, several improvements that can be incorporated into the product are proposed. They are either performance or functional improvements.

9.1 Performance

In terms of performance, few improvements can be easily incorporated into the product. This first set of proposed refinements was to be part of the baseline; however, due to time constraints, incorporating these features was not possible. They are therefore proposed as recommended enhancements to the current product baseline.

a. Dynamic Speedup Optimisation. Currently the workload is dynamically distributed among a predefined number of workstations. The number of workstations is set at the beginning of a work session as a parameter in the configuration file. Except in mode 4 of the load balancing control scheme, the number of workstations during a work session does not change. Now that the load balancing equations have been validated, the application should be able to dynamically calculate and identify if additional workstations would be beneficial to speed-up the computation. Or if some should be removed
because the computation is actually slowing down. If additional workstations need to be added, the LoadBalancer would request this to the Dispatcher. The Dispatcher would in turn go through its internal list of available workstations, and add some to the computation as required. Currently, only in mode 4 of the load balancing control scheme might a workstation be dropped out of the computation. To implement this added feature will require finishing some existing code in the Dispatcher and the LoadBalancer classes.

b. Message Payload Compression. The mathematical equations demonstrated that the speedup is really constrained by the ratio of computation time over communication time. This ratio can be increased if the payload of the messages is compressed. This is particularly true for the reply messages coming from the workers. Assuming a compression ratio of two to one, an approximate factor of 1.4 can be expected in the maximum speedup. This would require additional code to be inserted in the Client and Server classes.

c. Input Vectors Compression. In the event that workstations run out of RAM memory because of the large number of inputs of the SOM problem, the input vectors can be compressed into a buffer. This improvement would require changing only the Worker class.
d. Two new features are now available in the Java 2 language that were not in previous Java versions. There is the possibility of setting the socket buffer size and enabling a TCP no-delay variable. These two new features should be examined to determine if they could be used to improve application performance. This would require a better understanding of sockets and TCP. If it turns out that performance can be enhanced by these features, minor code changes would be required in the Client and Server classes.

e. Finally the entire application code should be reviewed for optimisation.

Although the entire application may benefit from such a review, the only area of the code that would make a noticeable difference would be within the learning process loop. Consequently if optimisation is to be performed, it should be concentrated on the methods called during the learning process cycles.

Two other possible improvements can be integrated into the current product to improve performance, but a significant amount of additional analysis and code would be required. The first one is that the current load balancing schemes assume a stable network where CPUs usage and communication bandwidth are steady. If there are bursting fluctuations, the current application would carry out unnecessary load balancing activities. Consequently additional intelligence should be built into the LoadBalancer class to avoid these inappropriate tuning attempts.
The second refinement that would provide additional performance is related to internetworking. If a user has access to more than one interconnected LAN, the architecture of the current application should be changed so that each LAN would have its own coordinator object that would communicate the partial results of its workers to a ‘master’ coordinator located on one of the LANs. Because concurrent communications on different LANs do not interfere with each other, additional parallelism can be incorporated into the product, allowing for a greater speedup. All the above proposed refinements are designed to improve performance. There are other proposed refinements, but they are concerned with improving the product functionality.

9.2 Functionality

Following are four possible improvements to the product’s functionality:

a. The application should be allowed to solve a SOM problem in a series of computing sessions. This feature would be useful if a computation takes a few days to complete but the computing resources are only available during a limited number of hours per day, e.g., at night. Consequently the computation could be started at night, interrupted in the morning, and carried out the next night at the point where the computation was left off that morning. To implement this feature would only require to save the state of the neural network at which it was interrupted.
b. Currently only one type of worker has been developed. The current worker only deals with one-dimension problems that are circular in nature, e.g., TSP. The application should have a collection of workers able to deal with different types of problems. Since SOM results are typically displayed on two-dimension maps, a worker that can deal with a two-dimensional matrix of neurons would definitively be a requirement. Other types of workers can be developed as requirements arise.

c. Presently the application behaves appropriately if every workstation is up and running. In the event a workstation stops responding, the application will not recover from such a failure. To make the application more robust, some additional code to handle these types of exception would be required.

d. At this stage the application is operated from a command line prompt. The user has to enter all desired parameters into a configuration file. The name of the program is given in an online command with the configuration file as an argument. For an improved connection between the users and the application, a Graphical User Interface (GUI) would be a desirable add-on. The current Executif class could encapsulate that functionality.
This completes the proposed improvements to the product developed for this research project. However, we suggest the development of another application as a complement to this project: the development of a resource-manager. Currently the Dispatcher has to be provided with a list of workstations available for the computation. Then it assumes that these workstations are actually available, not used, nor about to be used.

It would be more effective if the Dispatcher requested another application such as a resource-manager to provide a list of currently available workstations waiting for tasks. The resource-manager would have to be regularly in contact with all the daemons (a more intelligent daemon than the one developed for this project) installed on all workstations across an organisation’s network. Whenever a daemon detects that its workstation is not used, it would contact the resource-manager server, to be added to its list of available workstations. Whenever an application, like the one described in this thesis, contacts the resource-manager, it can query how many workstations are available for its computation. The resource-manager would apply all the administrative rules that govern which, what, where, who and when workstations can be made available to other applications. Consequently, if there are hundreds of workstations within an organisation, this resource-manager would make it easier for the user, actually his application, to know which workstations are available at any given time. If there is more than one user, then the resource-manager would coordinate the sharing of these resources.
This resources-manager utility would be a 24-hour service available to the entire organisation and even to other organisations that have received an appropriate password to access the service. The resources-manager could track usage of all the organisation's workstations. This would be almost administration-free for the computer services staff. The Java Virtual Machine (JVM) technology makes it safe to run unknown applications on workstations that could be PCs. Such an application would allow for a better usage of currently dormant resources. Resource-managers similar to the one described herein have already been developed [COD] [CON] [DQS] [LSF], but there is always the possibility of creative improvements to existing technology.
10. Conclusion

In this research project, we were able to develop and implement a Self-Organizing Map neural Network on a distributed environment of homogeneous and heterogeneous workstations. Although the speedup it provides is constrained by the communication overhead, network parallel computing nevertheless exhibits linear speedups for large SOM neural networks. We have shown that even poor performing workstations can provide excellent overall performance if more of them are used in parallel. We demonstrated that our parallel virtual machines are able to reduce substantially the computational time required by SOMs and solve problems like TSPs in a fraction of the time taken by a single computer.

By selecting a technology like Java, operating on heterogeneous networks was easy. Our parallel implementation of the SOM was run on Unix-based or on MS-Windows-based operating systems. Not only does Java offer better portability, it also enforces the paradigm of object-oriented programming with favourable consequences.

The developed product also applies different load balancing modes to fit various communication patterns and to adapt dynamically to variations of workstation execution speeds. These characteristics of the product distinguish it as an improvement over previous implementations. For example PVM-based implementations presented at [GUA96][GUA97][LAN96][VAS95] and [LOB98] did not offer dynamic nor optimal
load balancing, and [LOB98] was incapable of operating on a heterogeneous network. Consequently we consider the application presented in this paper to be a much more versatile, powerful, and superior product. Although the product may still benefit from some refinements, the current version is entirely functional.

The main goal of speeding up the computation was achieved. The secondary goal which was to make better use of available computing resources was also achieved. Obtaining supercomputing-like performance at no additional cost was thus possible. However, computing resources might not always be readily available, and this is why it is recommended that parallel applications be used in conjunction with other applications that perform resource management.

Our results lead us to believe and concur with [AND95] that because its availability and low cost, network parallel computing at large is likely to become the primary infrastructure and environment for parallel computing.
References


_COD_ CODINE is a Resource-Management System targeted to optimize utilization of all software and hardware resources in a heterogeneous networked environment. The product is offer by Genias (Germany). See [http://www.genias.de/menu_e.html](http://www.genias.de/menu_e.html)

[CON] Condor project is to develop, implement, deploy, and evaluate mechanisms and policies that support High Throughput Computing (HTC) on large collections of distributively owned computing resources. University of Wisconsin, See [http://www.cs.wisc.edu/condor/](http://www.cs.wisc.edu/condor/)

[DQS] The Distributed Queueing System (DQS) is designed as a management tool to aid in computational resource distribution across a network. Florida State University. See [http://www.scri.fsu.edu/~pasko/dqs.html](http://www.scri.fsu.edu/~pasko/dqs.html)


[JOVE] JOVE web page maintained by and is a product of Instantiations Inc., Tualatin, Oregon. See http://www.instantiations.com/jove/jove.htm

[jPVM] jPVM’s web page maintained by the Center for Human-Machine Systems Research at the Georgia Institute of Technology, Atlanta. See http://www.isye.gatech.edu/chmsr/jPVM/.


[LSF] Load Sharing Facility (LSF) is a workload-manager distributed load sharing and sophisticated job scheduling for heterogeneous UNIX and NT computing environments. The product is offered by Platform Computing Corp. (Toronto, Canada). See http://www.platform.com/


[PRM] The Prospero Resource Manager (PRM) supports the allocation of processing resources in large distributed systems, enabling users to run sequential and parallel applications on processors connected by local or wide-area networks. A project of University of Southern California. See http://nii-server.isi.edu:80/gost-group/products/prm/


[TSPLIB] TSPLIB web site maintained by the Discrete Optimization group of the Institut für Angewandte Mathematik at the University of Heidelberg, Germany. See http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.


Bibliography

Books:


Papers:


Others:

RMC course notes for "EE:575, Introduction to Neural Network.", winter term 1998.
Appendix A - Web Sites

This appendix provides a list of relevant web sites that provide information on distributed & parallel computing, Java, and neural networks. These sites also give access to many papers. Approximately a hundred of these papers were consulted. The web sites are not sorted in a particular order except the first six which have been the source of abundant interesting papers.

Northeast Parallel Architectures Center, Syracuse University, focuses on research and development on modern computer science for both large-scale information and simulation applications. See http://www.npac.syr.edu/


Berkeley Network of Workstations (NOW) project seeks to harness the power of clustered machines connected via high-speed switched networks. See http://now.CS.Berkeley.EDU/.

Scandal Project from Carnegie Mellon University. Its primary research interest is the development of a portable, interactive environment for programming a wide range of supercomputers. http://www.cs.cmu.edu/~scandal/


Concurrent Systems Architecture Group (CSAG) of University of California, San Diego focuses on hardware and software architecture issues in parallel and distributed computer systems and present their High Performance Virtual Machines (HPVM). See http://www-csag.ucsd.edu/
Simulation, Object Oriented Languages and Parallelism Group from University of Nice (France). SLOOP's main objective is to develop methods and tools for the efficient use of multi-processor machines in discrete event systems simulation. See http://www.inria.fr/Equipes/SLOOP-eng.html.
http://www-sop.inria.fr/sloop/personnel/Denis.Caramel/

RAIN project. Redundant Array of Inexpensive workstations for Neurocomputing (RAIN) focuses on the development of two technology demonstrators in the field of neurocomputing on a cluster of workstations. The applications are selected among the most representative in the industrial and medical field. See http://www.esng.dibe.unige.it/RAIN/.


Parallel Applications Development Environment (PADE) has for mission to facilitate development of parallel applications for heterogeneous networked computers. PAPE is from the National Institute of Standards and Technology which is an agency of the U.S. Department of Commerce's Technology Administration. See http://www.nist.gov/itd/div895/pade/

Scientific Computing Associates, Inc offers products like Java-Paradise that is a concept of virtual shared memory, which allows applications to share objects and process data across distributed networks incorporating LANs, WANs, and even wireless devices. See http://www.sca.com/

Parallel and Distributed Systems Research Group (PADS) from the University of Waterloo. See http://www.pads.uwaterloo.ca/

GAMMA Project: Genoa Active Message Machine from Universita' di Genova, Italy. GAMMA is an efficient communication software for message passing parallel processing on clusters of Personal computers based on Fast Ethernet that supports SPMD/MIMD parallel processing, possibly in a multiprogrammed environment. See http://www.disi.unige.it/project/gamma/


International Parallel Processing Symposium sponsored by IEEE Computer Society Technical Committee on Parallel Processing in cooperation with ACM SIGARCH. See http://www.ippsxx.org/
First International Workshop on Web Engineering WWW7 Conference, Brisbane, Australia, 14 April 1998. See

Extreme! Computing Group from Indiana University. See
http://www.extreme.indiana.edu/

Arcade project from Old Dominion University, Virginia. See
http://www.cs.odu.edu/~ppvm/

Distributed.net projects builds up a network of computers over the Internet all
coordinating on various tasks such as finding public encryption keys. See
http://www.distributed.net/

ICE-T. Interprise Computing Environment Toolkit (ICE-T) enables building,
extending and deploying client-server applications for the Web. See
http://www.nada.kth.se/systemgruppen/docs/IceT/intro.doc.html#3633

Charlotte Research Project from New York University focusses on researches
and investigations on how to utilize the World Wide Web as a metacomputing
platform. See http://www.cs.nyu.edu/milan/charlotte/index.html

Ninf: A Network based Information Library for Global World-Wide Computing
Infrastructure from the Computer Systems Division Electrotechnical Laboratory,
Japan. See http://ninf.etl.go.jp/

Quorum a Group from US Department of Defence that is to provide seamless
interoperability, distribution over multiple nodes and the sharing of information in
support of rapidly organized joint and coalition missions. See

Web Computer Project from Buxton House Research is a virtual supercomputer
for the masses. See http://www.wizzo.demon.co.uk/research.html

Swiss Center for Scientific Computing provides computing resources with the
leading parallel vector and scalable scalar computing platforms. See
http://www.cscs.ch/Official/Services/TechProfile.html

Globus project is developing basic software infrastructure for computations that
integrate geographically distributed computational and information resources.
Globus is a joint project of Argonne National Laboratory and the University of
Southern California. See http://www-fp.globus.org/
Legion Project from University of Virginia is World Wide virtual computer. See http://www.cs.virginia.edu/~legion/

TPVM from University of Virginia. Threads-oriented PVM (TPVM) is an experimental subsystem for PVM which supports the use of light-weight processes or "threads" as the basic unit of scheduling and parallelism. See http://www.cs.virginia.edu/~ajf2j/tpvm.html

Institute for Program Structures and Data Organization (IPD) from University of Karlsruhe, Germany. See http://wwwipd.ira.uka.de/Tichy/. The sites links to ParaStation Project which is to build a scalable and efficient parallel processor from off-the-shelf workstations. See http://wwwipd.ira.uka.de/ParaStation/.
JavaParty is an extension of Java that provides Transparent Remote Objects and Object Mobility. See http://wwwipd.ira.uka.de/JavaParty/

Neural Networks Research Centre at Helsinki University of Technology, Finland, investigates the theory and applications of the Self-Organizing Map (SOM) and Learning Vector Quantization (LVQ) algorithms of prof. Kohonen. See http://wwwipd.ira.uka.de/Tichy/

Internet Parallel Computing Archive (IPCA) hosted by University of Kent at Canterbury, UK. See http://www.hensa.ac.uk/parallel/


8th Heterogeneous Computing Workshop (HCW '99) Puerto Rico April 1999 sponsored by the IEEE Computer Society. See http://garage.ecn.purdue.edu/~maheswar/hcw/about.html


1999 Advanced Simulation Technologies Conference organized by The Society for Computer Simulation. See http://www.informatik.uni-rostock.de/HPC99/


Paderborn Center for Parallel Computing, Germany. See http://www.uni-paderborn.de/pc2/index.htm
Sets of papers written by W.W.Y. Liang, Ph.D. that relates to Distributed Shared Memory for Network of Workstations. See http://arch11.ee.ntu.edu.tw/~wyliang/papers/

Supercomputing and Parallel Computing Conferences and Journals. See http://www.cs.cmu.edu/~scandal/conferences.html

Parallel and Distributed Computing Practices Journals (PDCP) published by NOVA Science Publishers. See http://orca.st.usm.edu/pdcp/

Upcoming Compiler and Parallel Computing Conferences provided by G. Roth from Gonzaga University, Spokane, WA. See http://www.cs.rice.edu/~roth/conferences.html

A list of links related to parallel computing provided by N.C. Schaller Professor, Computer Science Department Rochester Institute of Technology. See http://www.cs.rit.edu/~ncs/parallel.html
Appendix B - File Samples

******************* Self-Organizing Map Neural Network ************
This file is to be used with Disom.
This file is not to be edited.
For more information refers to the manual.
*******************

 rl5915.hst
 16 Oct 1998

10

su1
su2
su3
su4
su5
su6
su7
su8
su9
su10

Sample of a host file.
***********************************************************************
Self-Organizing Map Neural Network **********
This file is to be used with Disom.
This file is not be edited.
For more information refers to the manual.
***********************************************************************

rl5915.ipt
17 Dec 1998
5915
2

1
1.81920e+04
8.95400e+03

2
1.81920e+04
9.85600e+03

3
1.81920e+04
1.13190e+04

4
1.82720e+04
8.62400e+03

5
1.84160e+04
7.34800e+03

{etc.}

Sample of input and weight files.
This file is to be used with DISOM
This file is to be edited manually.
For more information on its format refers to the DISOM manual.

<table>
<thead>
<tr>
<th>******* File Names ******* (6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r15915.cfg</td>
</tr>
<tr>
<td>r15915.hst</td>
</tr>
<tr>
<td>r15915.ipt</td>
</tr>
<tr>
<td>r15915.wgt</td>
</tr>
<tr>
<td>&lt;results file&gt;</td>
</tr>
<tr>
<td>&lt;sys&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>******* Neurons ******* (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5915</td>
</tr>
<tr>
<td>WorkerCircular</td>
</tr>
<tr>
<td>Uniform</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>******* Operating Mode ******* (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>** Load Balancing ** (7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>******* Communication ******* (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
</tr>
<tr>
<td>5432</td>
</tr>
<tr>
<td>auto</td>
</tr>
</tbody>
</table>

Sample of a configuration file.
Appendix C - Experimental Measurements

This appendix provides a table with all the experimental measurements. Time is expressed in millisecond.

<table>
<thead>
<tr>
<th>nbs of workstations</th>
<th>pcb1173 P-90</th>
<th>Ultra 1 case 1</th>
<th>Ultra 30 case 1</th>
<th>PII-266 case 1</th>
<th>pcb3038 case 1</th>
<th>case 3</th>
<th>pcb5915 case 1</th>
<th>case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93300</td>
<td>63771</td>
<td>44625</td>
<td>24606</td>
<td>309190*</td>
<td></td>
<td></td>
<td>1173376</td>
</tr>
<tr>
<td>2</td>
<td>46969</td>
<td>32853</td>
<td>21994</td>
<td>12453</td>
<td>152549</td>
<td>156472</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>22049</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>14220</td>
<td>10135</td>
<td>5628</td>
<td>63247</td>
<td>64171</td>
<td>237700</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>20328</td>
<td>10885</td>
<td></td>
<td>4234</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>21298</td>
<td>9802</td>
<td>6936</td>
<td>3863</td>
<td>36799</td>
<td>35356</td>
<td>128620</td>
<td>123000</td>
</tr>
<tr>
<td>11</td>
<td>22223</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6793</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>8455</td>
<td>6742</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>8183</td>
<td>6787</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>8332</td>
<td>6900</td>
<td>30365</td>
<td>27739</td>
<td>99436</td>
<td>90000</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>7006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>8360</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>8467</td>
<td>7406</td>
<td>28600</td>
<td>24809</td>
<td>88694</td>
<td>78000</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>28400</td>
<td>24481</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>28344</td>
<td>24081</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>28340</td>
<td>23748</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>10503</td>
<td>8432</td>
<td>28693</td>
<td>23160</td>
<td>85508</td>
<td>71000</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td>29179</td>
<td>22802</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>29600</td>
<td>22571</td>
<td>85318</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>29800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td>30216</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* communication overhead measured was 687 ms.
Appendix D - CD-ROM

Accompanying this thesis is a CD-ROM that contains a copy of the following items:

1. this thesis;
2. the source code;
3. the compiler used to develop this application;
4. inputs files
5. outputs files;
6. configuration files
7. weights files;
8. numerous papers consulted; and
9. divers other things.