KTH

Royal Institute of Technology
Department of Teleinformatics

An Evaluation of the CMMD Library and an Efficient Parallel Implementation of the RTRL Algorithm on the CM-5

Ulf Johansson Gunnar Franzén

940226

IT 94/13

An Evaluation of the CMMD Library and an Efficient Parallel Implementation of the RTRL Algorithm on the CM-5

Ulf Johansson Gunnar Franzén

February 26, 1994

Abstract

The use of parallel computers becomes more widely spread in order to speedup software applications involving heavy computations. Algorithms used for Artificial Neural Networks (ANNs) are often characterized by a high degree of parallelism and are therefore suitable for implementation on parallel computers. Ideally, the speedup should be linear when the number of processors is increased. This, however, demands that the parallelism of the algorithm can be preserved in the implementation. A major drawback is that the communication latency is generally much higher than the computation latency on a parallel computer. Therefore, for implementations with a high degree of communication among processors, the speedup will decline when the number of processors increases.

In this master's thesis the possibilities for efficient communication on a CM-5 parallel computer are investigated. This is done by evaluating CMMD, the CM-5 message-passing library functions. Especially the possibilities to overlap communication and computation are investigated. Moreover, a parallel implementation of an ANN algorithm, the Real Time Recurrent Learning algorithm (RTRL), is designed. This implementation is written in the C programming language and uses CMMD for communication among the processors. To evaluate the performance of the implementation it is finally compared to an existing implementation written in the C* programming language.

The results show that it is not possible to achieve overlapped communication and computation on the CM-5. This is due to limitations in the computer hardware. However, the communication latency can be decreased considerably compared to the C* message-passing features. This results in a more efficient implementation of the RTRL-algorithm and a more linear speedup when the number of processors is increased.

Acknowledgements

We would like to thank all people involved in this master's thesis for their support, helpfulness and encouragement.

Especially we want to thank Mr Igor Milosavljevićh, Dr Marwan Jabri, Mr Barry Flower and Mrs Kim Murphy, all working at the Department of Electrical Engineering, University of Sydney, Australia, Dr Björn Lisper and Mr Per Hammarlund at the Royal Institute of Technology, Stockholm, Sweden, and Mr Lindsay Hood at Thinking Machines Corporation, Sydney, Australia.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Objective of the Thesis . . . 1
- 1.3 Report Layout . . . 1

2 Neural Networks **2**
- 2.1 Artificial Neural Networks . . . 2
  - 2.1.1 The Biological Neuron . . . 2
  - 2.1.2 Adaption, Learning and Information Processing . . . 3
  - 2.1.3 The Artificial Neuron . . . 3
  - 2.1.4 ANN Architectures . . . 3

3 Mapping of the RTRL-algorithm **5**
- 3.1 Systolic arrays . . . 5
- 3.2 Mapping of algorithms on systolic arrays . . . 5
- 3.3 The RTRL Algorithm . . . 5
  - 3.3.1 Basic Properties of the Algorithm . . . 5
  - 3.3.2 Algorithm Definition . . . 5
  - 3.3.3 Mapping of the Algorithm . . . 7
  - 3.3.4 Analysis of the Mapping . . . 9

4 The Environment **11**
- 4.1 The CM-5 Parallel Computer . . . 11
  - 4.1.1 Global Architecture . . . 11
  - 4.1.2 Local Architecture . . . 12
  - 4.1.3 The Network Interface . . . 12
- 4.2 The C* Programming Language . . . 13
  - 4.2.1 Using Shapes . . . 13
  - 4.2.2 Virtual Processors . . . 13
- 4.3 The CM Message Passing Library . . . 13
  - 4.3.1 CMMD Features . . . 13
  - 4.3.2 CMAML . . . 14

5 Evaluation of Communication Features **15**
- 5.1 Communication in C* . . . 15
- 5.2 Synchronous Communication with CMMD . . . 16
- 5.3 Asynchronous communication with CMMD . . . 16
- 5.4 Using Virtual Channels with CMMD . . . 18
- 5.5 Communication with CMAML . . . 19

6 Results of Evaluations **21**
6.1 Communication in C* 21
6.2 Synchronous communication with CMMD 22
6.3 Asynchronous Communication with CMMD 23
6.4 Virtual Channels with CMMD 24
6.5 CMAML Communication with CMMD 25
6.6 Analysing the results 26

7 The Implementation **28**
7.1 Description of the implementation 28
7.1.1 Calculations 28
7.1.2 Virtual processors 29
7.2 Tests of performance 30
7.2.1 Speedup 30
7.2.2 Comparison with the C* implementation 31

8 Conclusions **33**

A Code **34**
A.1 Evaluations 34
A.1.1 Cstar_comm.tex 34
A.1.2 CMMD_sync.c 38
A.1.3 CMMD_async.c 42
A.1.4 CMMD_virtual.c 47
A.1.5 CMAML_comm.c 52
A.2 Implementation 56
A.2.1 rtrl_serial.c 56
A.2.2 rtrl_parallel.c 60

1 Introduction

1.1 Background

Research and applications of Artificial Neural Networks (ANN) become more and more widely spread in a variety of different areas. ANN algorithms often involve very heavy computational load and therefore cannot be implemented efficiently on conventional serial computers. However, as the degree of parallelism in the algorithms is usually high, great efforts to implement ANN algorithms on parallel computers have been made. This means a significant enhancement of performance, but the task to utilize the parallelism is quite a difficult one as parallel programming differs considerably from conventional serial programming.

The communication latency on a parallel computer is typically much greater than the computation latency. This means that the amount of communication among processors has to be kept on a low level to preserve the parallelism in the algorithm. A problem arises as the degree of interconnectivity on a parallel computer is usually far less than in an ANN, implying that the amount of communication will be higher than desirable if precautions are not taken. An important task is to transform the algorithm to a low-level form in order to match the hardware of the computer. In other words, a mapping of the algorithm that exploits the actual parallel computer in an efficient way has to be found. Moreover, the performance can be enhanced considerably if the communication latency for a particular communication operation can be decreased, or even better if the communication can be overlapped with the computations.

1.2 Objective of the Thesis

In this thesis the possibilities to communicate efficiently on a CM-5 parallel computer are investigated. This is done by evaluating the message passing features of the CM Message passing Library (CMMD), supplied by Thinking Machines Corporation [12]. Especially the possibilities to overlap communication/computation are evaluated. Then a ANN algorithm, the Real Time Recurrent Learning (RTRL) algorithm, is implemented on the CM-5. This implementation is written in the C language and the CMMD features are used for message passing among the processors. To evaluate the efficiency of this implementation it is then compared to an existing implementation of the algorithm, written in the C* programming language. A mapping technique described in [9] is used to map the RTRL algorithm in an efficient way. This is the same mapping technique used for the C*-implementation.

1.3 Report Layout

The structure of the report is as follows. First, an introduction to Artificial Neural Networks is presented, followed by a description of the RTRL algorithm. The mapping technique used is then described, as well as an analysis of the communication requirements resulting from it. The CM-5 and the message passing features are then described in the next chapter. The main part of the report then deals with the evaluation of communication features and the corresponding results. Finally the RTRL-implementation is described and evaluated.

Figure 1: The biological neuron.

2 Neural Networks

2.1 Artificial Neural Networks

Artificial neural net (ANN) models have been studied for many years aiming to achieve human-like performance in a variety of fields. Some common examples are speech recognition, different kinds of forecasting, e.g., financial forecasting etc. A significant advantage of using ANNs to solve a problem is that no well-defined process for algorithmically converting an input to an output is needed. Rather, all that is needed for most networks is a collection of representative examples of the desired translation. The ANN then adapts itself to reproduce the desired outputs when presented with the example inputs. In addition, an ANN is robust in the sense that it will respond with an output even when presented with inputs it has never seen before, such as patterns containing noise. The ability to deal with noisy patterns is a significant advantage over a traditional algorithm solution.

2.1.1 The Biological Neuron

The basic building block of the neural network is the neuron. The artificial neuron is inspired by the nerve cell which is the basic building block of the brain. In human beings there are about 10^{12} neurons which can be grouped into at least 50 types of cells. The complexity of the brain is not limited to the vast number of neurons, there are an even larger number of connections between them. One estimate is that there are a thousand connections per neuron on average, resulting in a total of 10^{15} connections in the brain.

The components of a neuron are shown in Fig.1. There are a cell body, a number of dendrites (input), and an axon (output). The axon splits and connects to other neurons (or muscles, etc.). The length of an axon varies from less than a millimeter to more than a meter. The function of the connections are similar to a sort of chemical resistor and are called synapses. The input comes to the neuron through the synapses on the dendrites. If there are enough stimuli on the inputs there will be an activation (impulse) through the axon which

Figure 2: The artificial neuron.

connects to other cells via synapses.

2.1.2 Adaption, Learning and Information Processing

The most interesting thing about the brain is its ability to learn new things. There are basically two ways in which the adaption can take place, by changing the structure and by changing the synapses. The first often takes place only in the first part of an animals life. The second is a more continuos process that takes place throughout the animals entire life. The information, which is represented as events called nerve impulses, is processed in most types of neurons as a summation or integration. The impulse rates may vary from one event every few seconds to about 1000 events per second in extreme cases. The speed of the impulses along the axon is around 0.1 m/s to 10 m/s.

2.1.3 The Artificial Neuron

One model that is frequently used, is the model in which information is contained as levels/values corresponding to the impulse frequencies. Then the integration of pulses is done as a summation. The synapses are represented as weights, w_j, multiplied by inputs i_j. To make the model more realistic a function, f, is applied to the sum, and the result, $o = f(\sum w_j i_j)$, is sent to the neurons connected to it (Fig.2). By changing the structure of the connections and adaption rules[1] it is possible to change the behaviour of the network.

2.1.4 ANN Architectures

There have been many suggestions of how to connect neurons to create an ANN. All the suggested models are parallel in nature and therefore suitable for implementation on parallel computers. The models are characterized by their network topology, node characteristics and training rules. Some frequently used models are:

[1] The adaption rules specify an initial set of weights and indicate how weights should be adapted during use to improve performance.

Figure 3: A six node feedback network.

- *Multilayer feedforward networks with supervised learning by error backpropagation*: This model can recognize and separate different features or patterns presented to its inputs.
- *Feedback networks (recurrent networks)*: This model can be used as, e.g., an autoassociative memory. This means that whenever a portion or a disordered version of a pattern is presented, the remainder of the pattern is filled in or the pattern is corrected.
- *Sparse distributed memory (SDM)*: Can be used for statistical predictions.

The network used in this thesis is of the feedback type. A feedback network consists of a single set of N nodes that are completely interconnected (Fig.3). All nodes can serve as both input and output nodes. Each node computes a weighted sum of all its inputs (Fig.2), and then applies a nonlinear activation function to the sum, resulting in an activation value or output of the node. This value is treated as input to the network in the next time step. Training or learning can be done, as in the implementation of the RTRL-algorithm described later in this report, while the network runs.

3 Mapping of the RTRL-algorithm

3.1 Systolic arrays

A systolic array consists of a set of interconnected cells. Every cell is considered to be able to perform a simple operation. Cells in a systolic array are interconnected in a regular pattern. The cells communicate only with their neighbours, accordingly there are no long communication links in the system. In a systolic system data flows in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. One thing that is specific for a systolic system is that both inputs and outputs flow in the system. In a classical pipelined system only the results flow.

3.2 Mapping of algorithms on systolic arrays

Because ANN algorithms are CPU bound[2] by nature, they should map well to systolic arrays. A method for mapping a whole range of ANN algorithms to systolic arrays is presented in [2]. In [3], an algorithm transformation called time-space mapping is proposed for mapping an algorithm in the nested loop form, e.g., the RTRL algorithm, to a systolic array. The time-space mapping method is based on the fact that the order of execution of loop body instances can be changed. This can be done if the statement that generates a variable is executed before the statement using this variable. This requirement is called the data-flow dependence. The idea is to change the execution sequence by grouping together instances of the loop body which are data-flow independent, and can therefore be executed simultaneously on an array of processors.

3.3 The RTRL Algorithm

3.3.1 Basic Properties of the Algorithm

Real Time Recurrent Learning (RTRL) is a general training algorithm for recurrent neural networks. RTRL is a global learning algorithm, which means that every weight in the network affects the way every other weight changes during the learning process. Compared to it's more widely used counterpart, the Back-Propagation Through Time algorithm, its main advantage is that the training process is integrated in the network model. No external control is needed to switch operation between retrieving and learning mode, and the training data set may be of infinite length. Its disadvantage is that the computational complexity of the algorithm makes it impractical to use for training of networks with a large number of neurons.

3.3.2 Algorithm Definition

This section presents an overview of the RTRL-algorithm definition. A complete definition can be found in [1]. Let the network have n units, with m external input lines. The n-tuple of unit outputs at time t are denoted by the vector $\mathbf{y}(t)$ and the m-tuple of external inputs by the vector $\mathbf{x}(t)$. We concatenate $\mathbf{y}(t)$ and $\mathbf{x}(t)$ to form the $(m+n)$-tuple $\mathbf{z}(t) = (\mathbf{y}(t)^t \mathbf{x}(t)^t)^t$. We also define two sets of indices, I and U, so that

[2] The time spent on computation is greater than the time spent on I/O operations.

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \tag{1}$$

Let $\mathbf{W}$ denote the weight matrix for the network, with a unique weight between every pair of units and also from each input line to each unit. This matrix has dimensions $n \times (m+n)$ and the weight connecting the source z_j to unit i is denoted w_{ij}.

Let

$$s_k(t) = \sum_{l \in U \cup I} w_{kl} z_l(t) \tag{2}$$

denote the net input to the kth unit at time t, for $k \in U$, with its output at the next time step being

$$y_k(t+1) = f_k(s_k(t)) \tag{3}$$

where f_k is the squashing function of the unit.

These two equations, (2) and (3), where k ranges over U, constitute the entire dynamics of the network.

The next step is to derive an algorithm to train the network in a supervised mode, meaning that certain of the outputs of the units are to match specified target values at specified times. In order to express this mathematically we let $T(t)$ denote the set of indices $k \in U$ for which there exists a specified target value $d_k(t)$ that the output of the kth unit should match at time t. Then define a time-varying n-tuple $\mathbf{e}$ by

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The discrepancies between $d(t)$ and $y(t)$ on the units will form an overall network error at each time step t. We denote this error

$$J(t) = 1/2 \sum_{k \in U} [e_k(t)]^2 \tag{5}$$

The goal is to minimize this network error. This is done by a gradient descent procedure, adjusting $\mathbf{W}$ along the negative of $\nabla_{\mathbf{W}} J(t)$. The weight change for any particular weight w_{ij} can thus be written as

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} \tag{6}$$

where α is some fixed positive learning rate. This expression can be further evaluated as

$$-\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} \tag{7}$$

Here $\partial y_k(t)/\partial w_{ij}$ can be computed by differentiating the network dynamics, given by equations (2) and (3), yielding

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f_k{}'(s_k(t)) \left[\sum_{l \in U} w_{kl} \frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t) \right] \tag{8}$$

where δ_{ik} denotes the Kronecker delta. To compute this expression we need the initial state of the network. This state is given by

$$\frac{\partial y_k(0)}{\partial w_{ij}} = 0 \tag{9}$$

because it is assumed that the initial state of the network has no functional dependence on the weights. These equations hold for all $k \in U$, $i \in U$, and $j \in U \cup I$. For simplicity we denote $\partial y_k(t)/\partial w_{ij}$ with $p_{ij}^k(t)$.

Thus, a dynamical system with variables $p_{ij}^k(t)$ has been created, where the dynamics of the system is given by

$$p_{ij}^k(t+1) = f_k{}'(s_k(t)) \left[\sum_{l \in U} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t) \right], \tag{10}$$

with initial conditions

$$p_{ij}^k(0) = 0. \tag{11}$$

When all variables $p_{ij}^k(t)$ have been computed at time step t, the final formula for computing the weight changes is then given by

$$\Delta w_{ij}(t) = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t) \tag{12}$$

In conclusion, the precise algorithm consists of, at each time step t, computing the quantities $p_{ij}^k(t)$, using equations (10) and (11), and then using the discrepancies $e_k(t)$ between the desired and actual outputs to compute the weight changes $\Delta w_{ij}(t)$.

3.3.3 Mapping of the Algorithm

When mapping the RTRL algorithm, a circular time-space mapping transformation technique is used. This technique is derived from the linear time-space mapping for systolic algorithm synthesis [3]. The nonlinearity is introduced by using the modulo operator. The technique aims to map an algorithm into a 1-D array, in this case, a ring structure.

The standard serial algorithm corresponding to Eq.10 is a nested loop of depth 4, implying a 4-D index space. However, we can consider the expression on the right side of the equation as an atomic operation, which reduces the index space to a 3-D space depicted in Fig.4. The order of atomic operations within so defined index points is not imposed by the data dependencies.

If rings[3] of index points $\Re_{tu}$ are defined, it is possible to express the mapping as

$$\Re_{tu}(s) = s\mathbf{v} \oplus_n (u\mathbf{b}_1 + t\mathbf{b}_2). \tag{13}$$

[3] A ring is a set of index points , e.g., $\mathbf{p}_{00}\mathbf{w}_0, \mathbf{p}_{11}\mathbf{w}_1, \mathbf{p}_{22}\mathbf{w}_2$ in Fig.4 forms a ring.

Figure 4: The index space for computing the partial derivative in Eq.10. The factor $f_k{}'(s_k(t))$ is omitted to simplify the drawing.

where $\Re_{tu}(s)$ denotes the bijective mapping $(s,t,u) \longmapsto (i,j,k)$. In this expression $\oplus_n$ denote the "modulo n" operation , i.e., $a \oplus_n b$ is the same as $(a+b) modulo\ n$. By assigning the vectors $\mathbf{v}$, $\mathbf{b}_1$, $\mathbf{b}_2$ in a proper way it is possible to achieve independence among points lying on the same ring, make the $\mathbf{p}_{ij}$-matrices stationary and map all $\mathbf{p}_{ij}$-matrices with the same i to the same processor. In [9] it is shown that, for an example with 3 nodes, i.e., $n = 3$, if

$$\mathbf{v} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ and } \mathbf{b}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

it is possible to create rings of index points that are independent and can therefore be executed in parallel. Hence, the mapping of the partial derivative computation given by Eq.10 is expressed as a series of rings sweeping the index space shown in Fig.4. The execution of a particular index point will take place on a processor pointed out by the variable s at a time step pointed out by a combination of n, u and t. This combination is given by the expression

$$Timestep = n * u + t. \tag{14}$$

The following example shows that the mapping works in a proper way.

Example:

Assume that $\begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$, $n = 3$ and rewrite Eq.13 as:

$i = s$
$j = s \oplus_n t$
$k = s \oplus_n u$

then $s = 1, t = 1$ and $u = 2$. This means that this index point will be executed on processor 1. Eq.14 yields that the time step, when the execution will take place, will be 7.

3.3.4 Analysis of the Mapping

In this section we will analyze the RTRL algorithm and the mapping of it described in sections 3.3 and 3.3.3. This analysis is done in order to determine what can be gained by decreasing the communication latency or, if possible, by overlapping communication/computation.

Consider Eq. 10. This equation describes the main part of the algorithm, and it is here most of the computations are performed. The time consuming part of this equation is to compute the multiply and ackumulate (multiply&add) operations described by the sum

$$\sum_{l \in U} w_{kl} p_{ij}^{l}(t) \tag{15}$$

With n neurons in the network, the number of such multiply&add operations are n for each $p_{ij}^{k}(t+1)$ value to compute. Here the indices i, j, k ranges from 1 to n, implying that the number of $p_{ij}^{k}(t+1)$ values to compute is n^3. In conclusion, n^4 multiply&add operations are performed each time the weights are to be updated.

If we denote the multiply&add latency t_{comp}, the total computation time can be written as

$$T_{comp} = \frac{n^4}{p} \cdot t_{comp} \tag{16}$$

Here p denotes the number of physical processors on the parallel computer used.

However, due to the mapping, $\mathbf{w}$, f_k' and z variables are not stationary but shifted among the processors. This implies that communication between processors is involved. With n neurons in the network, $\mathbf{w}$, f_k' and z respectively have to be shifted n times, and each shift is due to a communication operation.

If we denote the communication latency t_{comm}, then the total communication time when updating the weights once is equal to

$$T_{comm} = n \cdot t_{comm}(n) + n \cdot t_{comm}(1) + n \cdot t_{comm}(1) \tag{17}$$

Here the values within the parenthesis denote the number of elements to be shifted in a particular communication operation. The first term on the righthand side of Eq. 17 corresponds to the shift of $\mathbf{w}$ as each $\mathbf{w}$ vector contain n elements. The second and the third term correspond to the shift of f_k' and z respectively. To simplify Eq. 17, we assume that the time needed to shift n elements almost equals the time needed to shift one element. This is true in most cases, as the communication latency is generally not due to the actual transfer of data, but rather to startup overhead. With this simplification in mind, Eq. 17 can be written as

$$T_{comm} = 3 \cdot n \cdot t_{comm} \tag{18}$$

The total time to update the weights once is equal to the sum of T_{comp} and T_{comm}. Thus, the total time can be written as

$$T = \frac{n^4}{p} \cdot t_{comp} + 3 \cdot n \cdot t_{comm} \tag{19}$$

According to Eq. 19, the total time is approximately proportional to n^4, while the communication time is proportional to n. This means that the gain that can be obtained by decreasing the communication latency is hardly noticeable for large networks (i.e., $n^4 \gg n$). However, the communication latency is typically much greater than the computation latency for a parallel computer. This means that the communication is still the most time consuming part of the algorithm for smaller networks. Hence, decreasing the communication latency should enhance the performance significantly in this case.

Figure 5: Global architecture of the CM-5.

4 The Environment

4.1 The CM-5 Parallel Computer

4.1.1 Global Architecture

The CM-5 is a parallel computer from Thinking Machines Corporation. A CM-5 system contains a number of parallel processing nodes, where the number can be anything from just a few up to several thousands. The CM-5 used in this thesis has 32 processing nodes. Each node has its own memory. Nodes can fetch from the same address in their respective memories to execute the same (SIMD-style) instruction, or from individually chosen addresses to execute independent (MIMD-style) instructions.

The processing nodes are supervised by a control processor, which runs an enhanced version of the UNIX operating system. Program loading begins on the control processor. It broadcasts blocks of instructions to the parallel processing nodes and then initiates execution. To maximize system usefulness it is possible to divide the parallel processing nodes into groups, known as partitions. There is a separate control processor, known as a partition manager, for each partition and each user process executes on a single partition. Other control processors manage the systems I/O devices and interfaces. (see Fig. 5).

Every control processor and parallel processing node is connected to two interprocessor communication networks. The first one, the Control Network, is used for operations that involve all the nodes at once, such as synchronization operations and broadcasting. The second one, the Data Network, is used for data transfers where each item transmitted has a single source and destination. A third network, the Diagnostics Network, is visible only to the system administrator. It keeps tabs on the physical well-being of the system.

Figure 6: Local architecture of the CM-5.

4.1.2 Local Architecture

A processing node consists of a standard RISC microprocessor (currently a SPARC) with four vector units (VU). Each VU has its own memory bank and acts like a memory controller for that bank. (see Fig. 6). A memory address is mapped into the right memory bank by the controllers, making the address space appear contigous to the microprocessor. The CM-5 used in this thesis has 32 Mbytes of memory on each node. Each processing node also includes a CM-5 Network Interface, providing access to the Control Network and Data Network.

4.1.3 The Network Interface

The CM-5 Network Interface provides a memory-mapped control-register interface to a processor memory bus. All network operations are initiated by writing data to specific addresses in the bus address space. Different addresses may be mapped to the same control register, implying that information about the network operation is encoded in the address bits. The logical interface is divided into a number of functional units, where one functional unit is associated with the Data Network and several are associated with the Control Network. (see fig 3.) Each functional unit presents two FIFO interfaces, one for outgoing data and one for incoming data.

Once a message has been accepted by the Data Network functional unit, this functional unit takes on all responsibility for its eventual delivery. The sending processor can then perform other computations while the message is in transit. However, the functional unit can not handle more than five 32-bit chunks of data each time it is invoked. This, as we will see later, is a constraint which limits the possibilities to implement overlapped computation and communication on the CM-5.

4.2 The C* Programming Language

C* is an extension of the C programming language designed for the Connection Machine data parallel computing system. C* implements the ANSI standard C language, which means that programs written in C compile and run correctly under C*. In addition, C* provides new features that make it possible to perform data parallel computing.

4.2.1 Using Shapes

The central feature of C* is the use of shapes. A shape is a way of logically configuring parallel data on the CM-5. C* includes a new construct called left indexing that is used in declaring a shape. The left index specifies the number of dimensions (or axes) in the shape and the number of positions in each dimension. Positions correspond to processors. For example a declaration like

$$shape[32][16]s;$$

declares s to be laid out as a 32 x 16 grid on the processing nodes. It is also possible to declare arrays of shapes. Once a shape has been specified, one can declare parallel variables of that shape. When manipulating a parallel variable, all participating processors on the CM-5 manipulates its own position of the parallel variable. C* also supports features for communication among parallel variables.

4.2.2 Virtual Processors

The size of a shape is not limited to the number of physical processors. If a shape is declared to be of a size larger than the partition size, the CM-5 divides the memory associated with each physical processor and creates virtual processors. The time required to operate on a shape increases as the number of virtual processors increases.

4.3 The CM Message Passing Library

The CM Message Passing Library (CMMD) is a software library for interprocessor communication on the CM-5. Programs written in C, C++ or Fortran77 can call routines in this library when communication among the processing nodes is to be performed. If the CM-5 has vector units, programs can be written in C* or CM Fortran on each node and CMMD calls can be used to handle the communication among the nodes.

4.3.1 CMMD Features

CMMD allows the processing nodes to communicate in a number of different ways. Some of the features are synchronous message-passing, asynchronous message-passing and the use of virtual channels when the communication is determined by some predefined pattern. CMMD also support I/O routines to achieve efficient and flexible I/O for the processing nodes. These routines extends the I/O features supported by UNIX.

4.3.2 CMAML

The CMMD Active Message Layer (CMAML) is an internal layer of the CMMD software. It is the protocol-less transport layer upon which the higher level CMMD routines are built. CMAML is made available to programmers and can be used to define custom network protocols and to perform specialized low-level communication.

5 Evaluation of Communication Features

This chapter describes the different communication features supported by CMMD and how the operation latency of each feature is evaluated. First the C* communication feature is described, followed by the different features of CMMD. These features are

- Synchronous communication
- Asynchronous communication
- Virtual channels
- CMAML

All programs designed for the evaluation configure the processors in a ring structure and the communication operations are initiated at the same time on every processor. This is in accordance with the mapping of the algorithm described in section 3.3.3. The results of these evaluations are presented in the next chapter.

5.1 Communication in C*

The C* programming language provides for programming in a pure SIMD-style. The parallel data, structured as shapes (see section 4.2), are distributed among the physical processors. To perform an operation on a parallel variable, the operating instruction is broadcasted to all active processors. The active processors then use this instruction to operate on their respective share of the parallel variable at the same time. Communication operations are performed in the same manner, which implies that the communication is synchronous.

To communicate in C* a feature called parallel left indexing is used. The left indices are used to choose which elements of some parallel variables are to participate in the operation. For example, the statement

$$[10]dest = [5]source$$

means that the tenth element of the parallel variable *dest* will be replaced by the fifth element of the parallel variable *source*. As different elements of the parallel variables are located on different processors, communication among the processors are expressed implicitly in C*.

To be able to exploit *grid communication*, i.e., when parallel variables of the same shape communicate in regular patterns, parallel variables can be used as indices with the parallel left indexing feature. To shift elements among neighbours in a ring of processors, the C*-function *pcoord()* can be used to get the right indices. A call to *pcoord()* creates a parallel variable of the *current shape*[4] and each element in this variable is initialized to its coordinate along the axis specified in the call.

Suppose there is n processors connected in a ring structure and that *dest* and *source* are one dimensional parallel variables of the same shape. Then the statement

$$dest = [pcoord(0)\%n]source \tag{20}$$

[4] in C* a shape has to be specified as the current shape before it can be operated on.

where % is the *modulo* operator, shifts the elements of variable *source* one step and assigns this shifted variable to the variable *dest*.

To be able to evaluate the latency of this feature a program was designed, where the central part perform the operation (20) several times and then computes the mean value and the standard deviation of the latency. The name of this program is *cstar_comm.cs* and can be found in Appendix. The results are presented in section 6.1.

5.2 Synchronous Communication with CMMD

CMMD support separate functions for sending and receiving messages synchronously as well as it supports for the possibility to send and receive simultaneously. Due to the mapping of the algorithm one processor will always receive data from its first neighbour and send data to its second neighbour. Moreover, these two steps can be performed at the same time without influencing the correctness of the algorithm. Therefore the latter function is used by the testprogram.

The name of this function is *CMMD_send_and_receive()* and can be called by each node in the ring structure. The following arguments must be supplied

- *source_node* : an integer specifying which node to receive a message from.
- *source_tag* : an integer labeling the message to be received.
- **inbuffer* : the area in memory that will contain the message to be received.
- *inbuf_desc* : the length, in bytes, of the *inbuffer*.
- *dest_node* : an integer specifying which node to send a message to.
- *dest_tag* : an integer labeling the the message to be sent.
- **outbuffer* : the area in memory holding the message to be sent.
- *outbuf_desc* : the length, in bytes, of the *outbuffer*.

To evaluate the latency of this function a program was designed. This program performs the *CMMD_send_and_receive()* operation several times and then computes the mean value and the standard deviation of the operation latency. Moreover, the operation latency is evaluated for different sizes of messages. The name of the program is *CMMD_sync.c* and can be found in Appendix.The results are presented in section 6.2.

5.3 Asynchronous communication with CMMD

CMMD support functions for sending and receiving messages asynchronously, i.e., the sending and receiving nodes do not have to synchronize to transmit a message. These functions allow one node to signal readiness to transmit a message, then continue other computations until the second node honors the request. When both nodes are ready, the ongoing computation is interrupted. At this point, the message is sent and received.

The behaviour described above can be useful and save time in some applications. For example, consider an application where the processing nodes are fetching jobs from a common job-pool and where the goal is to empty this pool as fast as possible. We also assume that the processing nodes have to synchronize due to communication at certain points in the program. In this case, if asynchronous communication is not supported, a node that for some reason is faster than the others will have to wait when communication is to take place. The program would be more efficient if the faster nodes could signal its readiness to communicate and then continue to fetch jobs from the pool until the slower nodes are ready to communicate.

However, the systolic nature of the implementation of the RTRL algorithm implies that this feature cannot be utilized. In this implementation each node has the same amount of work to perform between each communication step. The total time to run the program is therefore determined by the slowest processing node. For comparison with the other features, however, the asynchronous communication functions were evaluated as well.

The name of the sending function is *CMMD_send_async()* and the following arguments must be supplied

- *dest_node* : an integer specifying which node to send a message to.
- *tag* : an integer labeling the the message to be sent.
- **buffer* : the area in memory holding the message to be sent.
- *buffer_desc* : the length, in bytes, of the *buffer*.
- **handler* : a function (or NULL) which is invoked upon completion.
- *handler_args* : arguments to the handler function.

The name of the receiving function is *CMMD_receive_async()* and the following arguments must be supplied

- *source_node* : an integer specifying which node to receive a message from.
- *tag* : an integer labeling the the message to be received.
- **buffer* : the area in memory that will hold the received message.
- *buffer_desc* : the length, in bytes, of the *buffer*.
- **handler* : a function (or NULL) which is invoked upon completion.
- *handler_args* : arguments to the handler function.

To evaluate the performance of the asynchronous functions a program was designed. This program measures the time to perform one asynchronous send and one asynchronous receive operation. Moreover, a varying amount of multiply&add operations are inserted between the calls to the communication functions. This is done in order to determine whether overlapped communication/computation can be achieved using this communication feature. Different lengths of the transmitted messages are also tried in order to determine its impact on communication latency. The results are presented as the mean value and the standard deviation. The name of the program is *CMMD_async.c* and can be found in Appendix. The results are presented in section 5.3.

5.4 Using Virtual Channels with CMMD

CMMD support functions for the use of virtual channels between communicating nodes. A virtual channel is a communication link that is set up once and can then be used several times with no handshaking involved. This is particularly useful when the pattern of communication between nodes is repeatedly used and relatively static in form. This is certainly the case with the implementation of the RTRL algorithm. In this implementation each processing node sends data to one of its neighbours all the time. In the same way it also receives data from its other neighbour all the time. Therefore the use of virtual channels should be efficient for this implementation.

There is a specific protocol for establishing and using a virtual channel:

- The function *CMMD_open_send_channel()* is called by the source node, and a send channel is opened if there are any channels available. Arguments supplied are *dest_node*, *tag*, *buffer*, and *buffer_desc*. For an explanation of these arguments see section 5.3.

- The destination node opens a receive channel by calling the function *CMMD_open_receive_channel*. Arguments supplied are *send_node*, *tag*, **buffer*, and *buffer_desc*. For an explanation of these arguments see section 5.3. The send and receive channels represent the two ends of the channel pipeline.

- When the source node needs to send a message, it writes data to the buffer associated with the send channel and then performs a *write* operation. This is done by calling the function *CMMD_write_channel*. One argument is supplied, which specifies the channel to be used. A call to this function delivers the data to the destination node, and marks the channel as *readable*.

- By calling the function *CMMD_receive_channel_status()*, the destination node can determine whether a message has arrived or not. If a message has arrived, it copies the received data and then calls the function *CMMD_reset_channel()*. This call marks the channel as writable again. For both of these functions an argument specifying the channel is supplied.

- When the source node is finished sending data, it marks the send channel as *closed* by calling the function *CMMD_close_send_channel()*. Likewise, when the receiving node is finished receiving data, it marks the receive channel as *closed* by calling the function *CMMD_close_receive_channel()*. For both of these functions an argument specifying the channel is supplied.

To evaluate the performance when using virtual channels a program was designed. This program measures the time to perform one *write* operation and then wait for the receive channel to become readable. Moreover, a varying amount of multiply&add operations are inserted while waiting for a receive channel to become readable. This is done in order to determine whether overlapped communication/computation can be achieved using this communication feature. Different lengths of the transmitted messages are also tried in order to determine its impact on communication latency. The program presents the results as the

mean value and the standard deviation of the communication latency. The name of the program is *CMMD_virtual.c* and can be found in Appendix. The results are presented in section 6.4.

5.5 Communication with CMAML

CMAML (CMMD Active Message Layer) provides transport functions for two types of message transfer :

- *Active Messages* are one packet long. Their purpose is to invoke a function upon the receiving node and to supply arguments to be passed to the function upon invokation.
- *Block data transfers* transfer a single block of data from one or more source nodes to a single receiving node.

When implementing the RTRL algorithm, the *Block data transfer* feature is sufficient.

As described earlier, CMAML is the protocol-less transport layer upon which the higher-level CMMD functions are built. CMAML functions therefore assume that some higher level of software is providing any protocol needed. For example, that a receiving node knows what to do with any data sent to it. Users must ensure that their applications provide such control, therefore, CMAML also provides functions that allow polling of the networks, manipulating interrupt state, and preserving register contents.

Before any datablocks can be tranmitted over the network, the participating nodes have to agree on the parameters for the transfer. To do this, the receiving nodes allocate receive ports, and the parameters supplied to these ports determine the characteristics of the coming data transfers. Before the receive ports are set up, interrupts must be disabled. This is done by calling the function *CMAML_disable_interrupts()*. Then the following functions are called:

- *CMAML_allocate_this_rport(port_nr)* allocates a receive port if there are any available. This port will later be referred to as *port_nr*, which is an integer.
- *CMAML_set_rport_address(port_nr, *buffer)* associates a pointer to an area in memory, **buffer*, with the receive port *port_nr*. The polling function invoked when data has arrived, will then store the arrived data on this location in memory.
- *CMAML_set_rport_byte_counter(port_nr, nbytes)* sets the *byte counter* on the receive port *port_nr* to *nbytes*.This value is the expected number of bytes to receive when a data block arrives. For each byte of data that actually arrives, the byte counter is decremented.
- *CMAML_set_rport_handler(port_nr, *function)* sets the user supplied handler-function *function* to be invoked when a data block has arrived. Handler-functions are optional and *function* is set to NULL if they are not used.
- *CMAML_set_rport_handler_arg2(port_nr, arg2* sets an argument that will be used by the optional *handler function*.

- *CMAML_set_rport_total_nbytes(port_nr, nbytes)* sets the number of bytes to which the bytecounter will be reset after a data block has arrived.

When receive ports are allocated and all associated parameters are initiated, interrupts are enabled by calling the function *CMAML_enable_interrupts()*.

At this stage the receive ports can be used to receive data blocks transmitted over the network. To detect when data has arrived to a receive port, the application program has to check the value of the *byte counter* associated with the port. This is done by calling the function *CMAML_get_rport_byte_counter(port_nr)*. This function returns the value of the byte counter associated with *port_nr*. When data has arrived the application program then has to store the received data at the memory location specified by *buffer*. To do this the function *CMAML_poll()* is called, and this must be done before the buffering capacity of the *network interface* is exceeded. Otherwise the network will clog. Interrupts have to be disabled during the procedure described above.

To send data over the network, the functions *CMAML_scopy()* or *CMAML_pcopy()* are used. The former is used for sending serial arrays of data, and the latter is used for sending parallel arrays of data. The following arguments must be supplied:

- *dest_node* : an integer specifying the destination node number.
- *dest_rportno* : the destination receive port number.
- *dest_align* : the alignment of the destination array.
- *dest_offset* : the byte offset into the destination array (0 = no offset).
- *source_addr* : the address in memory where the data to be sent are located.
- *nbytes* : the number of bytes to send.
- *handler* : handler function called after last byte is transferred.
- *arg* : handler functions argument.

The use of handler functions are optional and if not used, the arguments *handler* and *arg* are set to NULL.

To evaluate the performance when using the CMAML communication features a program was designed. This program measures the communication latency when different sizes of messages are transmitted over the network. Moreover, varying amounts of multiply&add operations are inserted between the send and receive operations. This is done in order to determine whether overlapped communication/computation can be achieved using this feature. The results are presented as the mean value and the standard deviation. The name of the program is *CMAML_comm.c* and can be found in Appendix. The results are presented in section 6.5.

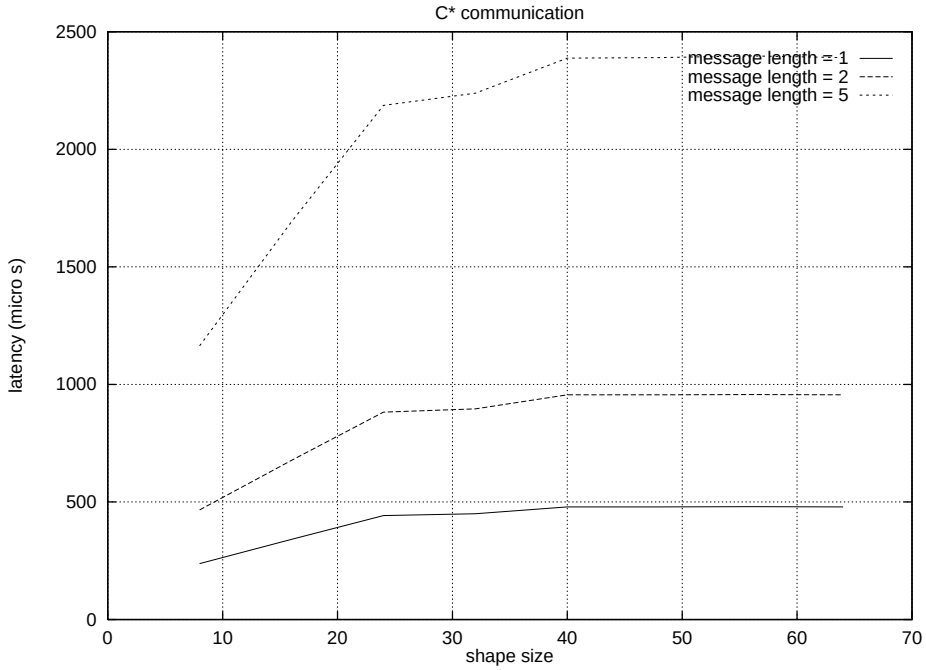

Figure 7: Evaluations of communication in C*.

6 Results of Evaluations

In this chapter the results of the evaluations described in chapter 5 are presented.

6.1 Communication in C*

In Fig.7 the results of the evaluations of communication in C* are presented. The latency shown is the time needed to perform a communication operation using the *parallel left indexing* feature. The latency is plotted as a function of the *shapesize*. As shown in the figure, the latency increases when the shapesize is increased. However, when the shapesize reaches a certain size the latency becomes stable. Moreover, the execution time as a function of the message transmitted were evaluated. Fig.7 shows that the latency increases linearly when the message-length is increased. The reason for this is that the elements were transmitted one at a time in the test program. The reason for this, in turn, is that this is the way in which the communication is performed in the current C*-implementation of the RTRL-algorithm. It is maybe possible to transmit more than one element at a time in C*, but it is not obvious. For example, it is not possible to declare a shape of arrays in this language. One way to do this could be to declare a shape of structures, and let each structure contain an array of elements to be transmitted. Then, when an array is to be transmitted, the corresponding structure is transmitted by using the *parallel left indexing* feature. However, this possibility has not been evaluated.

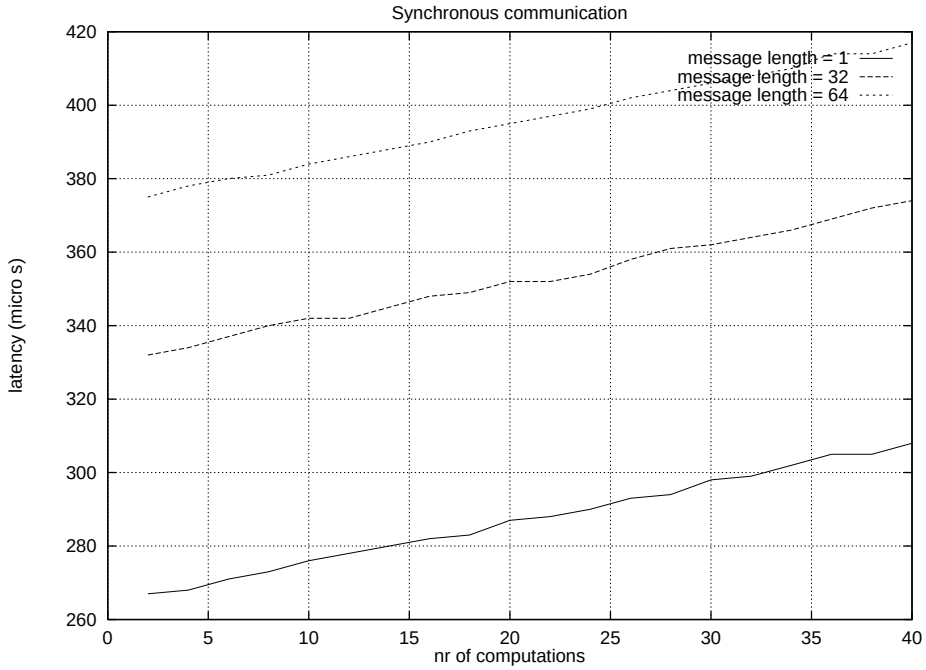

Figure 8: Evaluations of the CMMD Synchronous Communication feature.

6.2 Synchronous communication with CMMD

In Fig.8 the results of the CMMD synchronous communication evaluations are presented. The latency is the time for one node to perform one *CMMD_send_and_receive()* operation, followed by a varying amount of multiply&add operations. The fact that the latency increases linearly when the amount of multiply&add operations are increased, shows that no overlapped communication/computation is possible when using this feature. This is quite obvious in this case, but for comparison with other features these evaluations were performed anyway. Moreover, the evaluations were performed for different lengths of the transmitted message. Fig.8 shows that the latency increases when the length of the message is increased. However, in the range studied here, the major time is used for the setup of communication links and handshaking between participating nodes. This can be seen as the increase of the latency is just 25% when the message-length is increased from 1 to 32 elements.

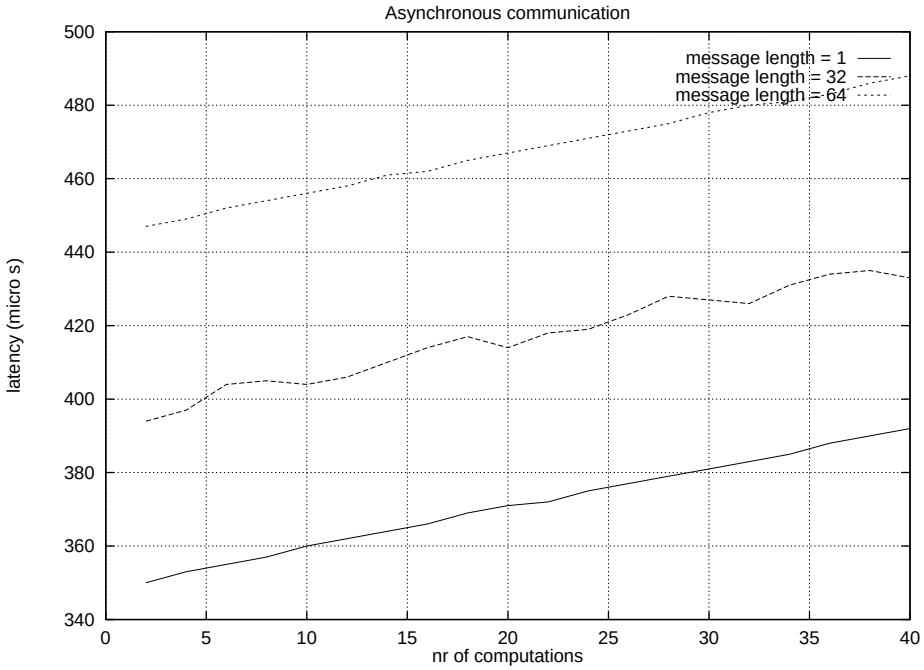

Figure 9: Evaluations of the CMMD Asynchronous Communication feature.

6.3 Asynchronous Communication with CMMD

In Fig.9 the results of the CMMD asynchronous communication evaluations are presented. The latency is the time for one node to perform one *CMMD_send_async()* operation, then a varying amount of multiply&add operations, and finally one *CMMD_receive_async()* operation. The fact that the latency increases linearly when the number of multiply&add operations are increased, shows that there is no possibility to overlap communication and computation when using this feature. Moreover, different lengths of the message transmitted were evaluated. Fig.9 shows that the latency increases when the message-length is increased. However, as in the case with synchronous communication, the major time is used for the setup of communication links and for handshaking between participating nodes. This can be seen as the increase in execution time is just 13% when the message-length is increased from 1 to 32 elements.

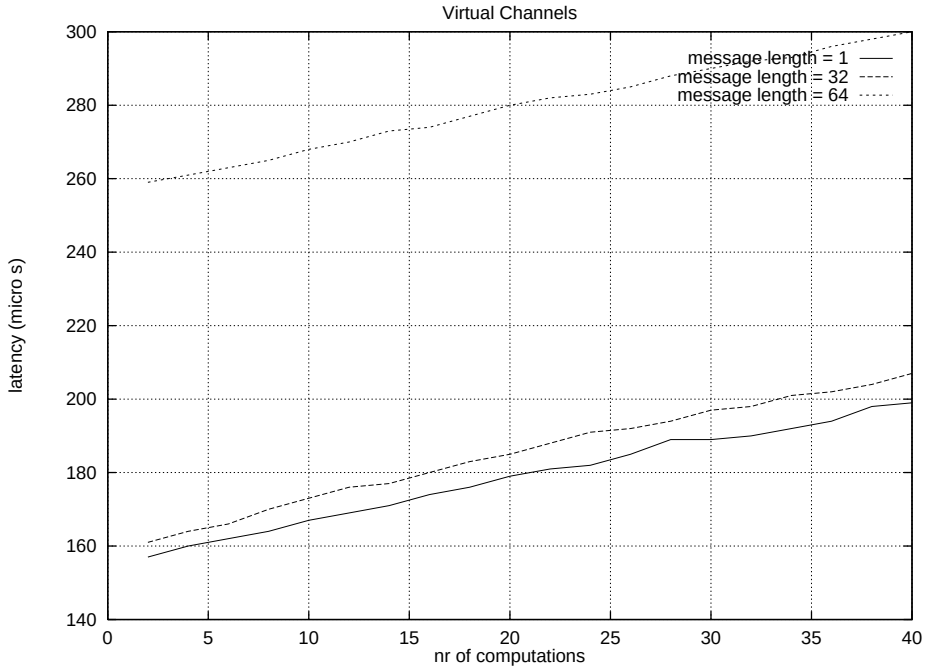

Figure 10: Evaluations of the CMMD Virtual Channels Communication feature.

6.4 Virtual Channels with CMMD

In Fig.10 the results when using virtual channels are presented. The latency shown is the time for one node to perform one *CMMD_write_channel()* operation, then perform a varying amount of multiply&add operations, and finally wait for its own *receive channel* to become readable by calling the function *CMMD_is_channel_readable()*. The fact that the latency increases linearly when the number of multiply&add operations are increased, shows that no overlapped communication/computation is possible when using this feature. Moreover, different lengths of the message transmitted were evaluated. Fig.10 shows that the latency increases when the message-length is increased. However, a noticeable increase doesn't occur until the transmitted message is quite long (64 elements in this case).

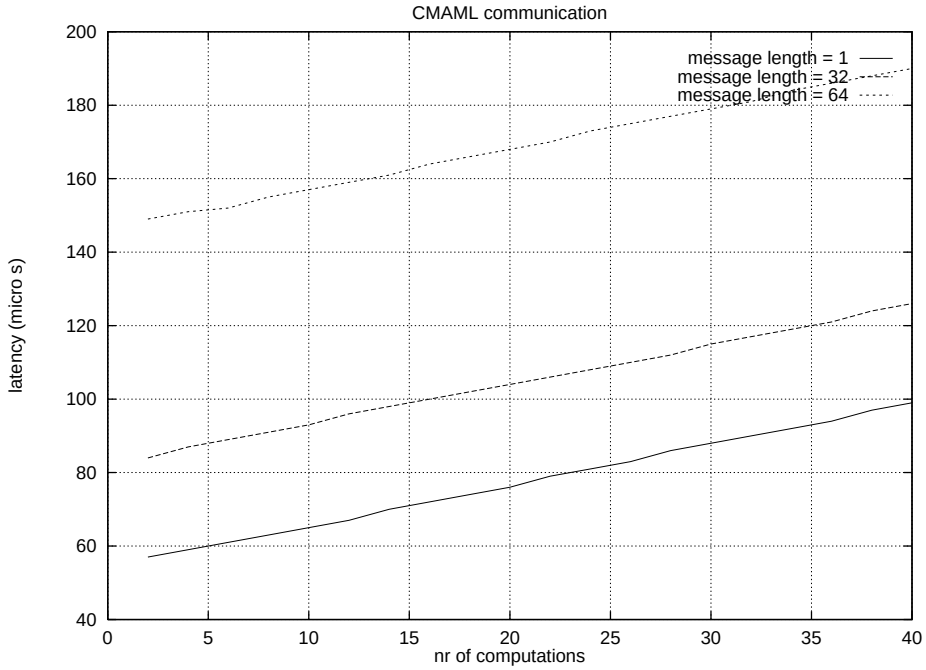

Figure 11: Evaluations of the CMAML Communication feature.

6.5 CMAML Communication with CMMD

In Fig.11 the results from the evaluations of the CMAML communication feature are presented. The latency shown is the time for one node to disable interrupts, perform one *CMAML_scopy()* operation, perform a varying amount of multiply&addd operations, poll the *Network Interface* until the *byte counter* is set to zero by repeatedly calling the function *CMAML_poll()*, then resetting the *byte counter*, and finally enable interrupts again. The fact that the latency increases when the number of multiply&add operations are increased, shows that no overlapped communication/computation is possible when using this feature. Moreover, different lengths of the message transmitted were evaluated. Fig.11 shows that the latency increases when the message-length is increased. However, the increase in latency isn't as remarkable when increasing the message-length from 1 to 32 elements as when increasing it from 32 to 64 elements. This fact indicates, as in the case with *Virtual Channels*, that the CM-5 is more efficient when handling smaller messages.

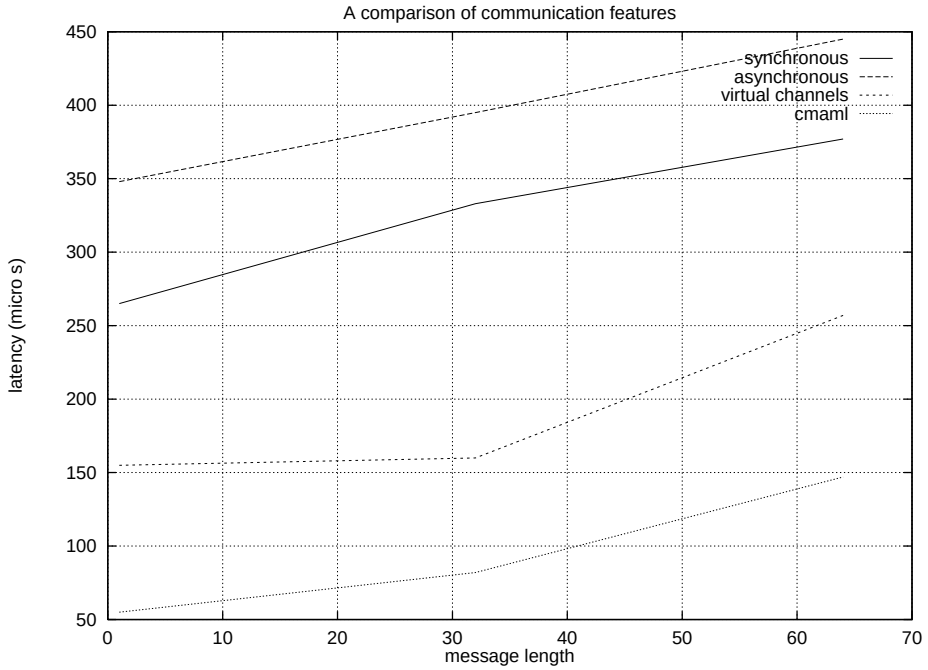

Figure 12: A comparison of the CMMD communication features.

6.6 Analysing the results

The results in the previous sections show that the goal to achieve overlapped communication/computation can not be satisfied when using the CMMD communication features. This is not due to any limitations in the CMMD software, but rather to the CM-5 architecture. As mentioned before, the possibility to buffer messages in the *Network Interface* is strictly limited on the CM-5. Moreover, the CM-5 hardware doesn't support DMA. This means that the CPU is interrupted very frequently when messages arrive at the Network Interface, and each time it is interrupted it has to transfer data between this Network Interface and the memory. However, compared to the C* communication feature, the communication latency can be decreased significantly when using CMMD.

In Fig.12 the different CMMD communication features are compared in the same diagram. The C* communication feature was ignored because the latency for long messages was to long to fit in to the diagram. As can be seen in this figure, the most efficient feature is CMAML. This feature is approximately ten times faster than the C* *parallel left indexing* feature for one element messages (when the shapesize is over 30 in C*). For longer messages the difference is even greater.

Fig.12 shows that the Asynchronous Communication feature is the one with the worst latency. This is probably due to more operating system overhead when using this feature. For example, if one node isn't ready to receive a message, the operating system must buffer the message and keep track of the sender and the receiver to be able to transmit the message later.

Figure 13: The main part of the calculations of Eq.10. The $\mathbf{w}$ vectors are multiplied with the corresponding $\mathbf{p}$ vector. When the multiplication is done the $\mathbf{w}$ vectors are shifted to the next processor.

7 The Implementation

7.1 Description of the implementation

Due to the global learning rule in the RTRL-algorithm, the communication requirements will be high. One important task to be considered is to decide which of the variables shall be stationary and which shall be shifted among the processors. Due to the mapping used in this implementation the $\mathbf{p}$ matrices are stationary and the z and $f_k{}'$ variables are shifted. It is probably easier to understand the parallel implementation on the CM-5 if the reader begins with a quick look at the serial program in Appendix. The name of the parallel implementation is *RTRL_parallel.c* and the name of the serial program is *RTRL_serial.c*.

7.1.1 Calculations

The key part of the calculations is the calculations of the partial derivatives given by Eq.10. This is the part of the RTRL-algorithm that has been implemented in this thesis. The main part of Eq.10 is the sum of the $\mathbf{pw}$ dot products given by (15), so from now on we will concentrate on this part of the algorithm.

The order of execution is given by the mapping and in order to make it easier to understand what actually happens, Fig.13 might be of help. This figure shows that, if the $\mathbf{w}$ vectors are rotated and multiplied with the corresponding stationary $\mathbf{p}$ vectors, the behaviour of rings sweeping through the index space shown in Fig.4 are achieved. E.g., at time step 0 the calculations $\mathbf{p}_{00}\mathbf{w}_0, \mathbf{p}_{11}\mathbf{w}_1, \mathbf{p}_{22}\mathbf{w}_2$ are performed on node 0, 1, 2 respectively and at time step 4 the calculations $\mathbf{p}_{01}\mathbf{w}_1, \mathbf{p}_{12}\mathbf{w}_2, \mathbf{p}_{20}\mathbf{w}_0$ are performed on node 0, 1, 2 respectively. In this case, with 3 nodes, each $\mathbf{p}$ and $\mathbf{w}$ vector contains 3 elements each, e.g., $\mathbf{p}_{00}$ contains $p^0_{00}, p^1_{00}, p^2_{00}$ and $\mathbf{w}_0$ contains w_{00}, w_{01}, w_{02}. The nonstationary variables in Eq.10 are shifted

Timestep	Node 0	Node 1
1	$\mathbf{p}_{00}\mathbf{w}_0$	$\mathbf{p}_{22}\mathbf{w}_2$
	$\mathbf{p}_{11}\mathbf{w}_1$	$\mathbf{p}_{33}\mathbf{w}_3$
2	$\mathbf{p}_{01}\mathbf{w}_0$	$\mathbf{p}_{23}\mathbf{w}_2$
	$\mathbf{p}_{12}\mathbf{w}_1$	$\mathbf{p}_{30}\mathbf{w}_3$
3	$\mathbf{p}_{02}\mathbf{w}_0$	$\mathbf{p}_{20}\mathbf{w}_2$
	$\mathbf{p}_{13}\mathbf{w}_1$	$\mathbf{p}_{31}\mathbf{w}_3$
4	$\mathbf{p}_{03}\mathbf{w}_0$	$\mathbf{p}_{21}\mathbf{w}_2$
	$\mathbf{p}_{10}\mathbf{w}_1$	$\mathbf{p}_{32}\mathbf{w}_3$
shift of weight vectors		
5	$\mathbf{p}_{00}\mathbf{w}_1$	$\mathbf{p}_{22}\mathbf{w}_3$
	$\mathbf{p}_{11}\mathbf{w}_2$	$\mathbf{p}_{33}\mathbf{w}_0$
6	$\mathbf{p}_{01}\mathbf{w}_1$	$\mathbf{p}_{23}\mathbf{w}_3$
	$\mathbf{p}_{12}\mathbf{w}_2$	$\mathbf{p}_{30}\mathbf{w}_0$

Table 1: Example of execution using virtual processors.

in the same manner as $\mathbf{w}$, but at different timesteps.

7.1.2 Virtual processors

When the number of neurons is greater than the number of physical processors, virtual processors have to be used. The idea is to make it look like the number of physical processors equals the number of neurons. This can be achieved by mapping more than one neuron to each processor. This can be done by breaking the rings that contains independent index points and map more than one index point to each processor in each time step. In an example with four neurons and two physical processors, each processor will, at each time step, contain two p matrices and two w vectors. During the computations the w vectors are shifted both internally within one processor and, as described in 7.1.1, externally between different processors. In Table 1 the first six time steps of the execution, of the example with four neurons and two physical processors, are shown. Vector $\mathbf{p}_{00}$ contains the elements $p^0_{00}, p^1_{00}, p^2_{00}$ and p^3_{00} and vector $\mathbf{w}_0$ contains the elements w_{00}, w_{01}, w_{02} and w_{03}. Between timestep four and five the w vectors have to be shifted. When using virtual processors the amount of work in each timestep increases and therefore the time steps are longer than without virtual processors.
If the number of neurons is not a multiple of the number of physical processors, the processing nodes will contain different number of neurons. E.g., if we have an example with 7 neurons and 3 physical processors, processor 1 will contain 3 neurons and the other two will contain 2 neurons each. The execution time in this case is determined by the processors containing the highest number of neurons.

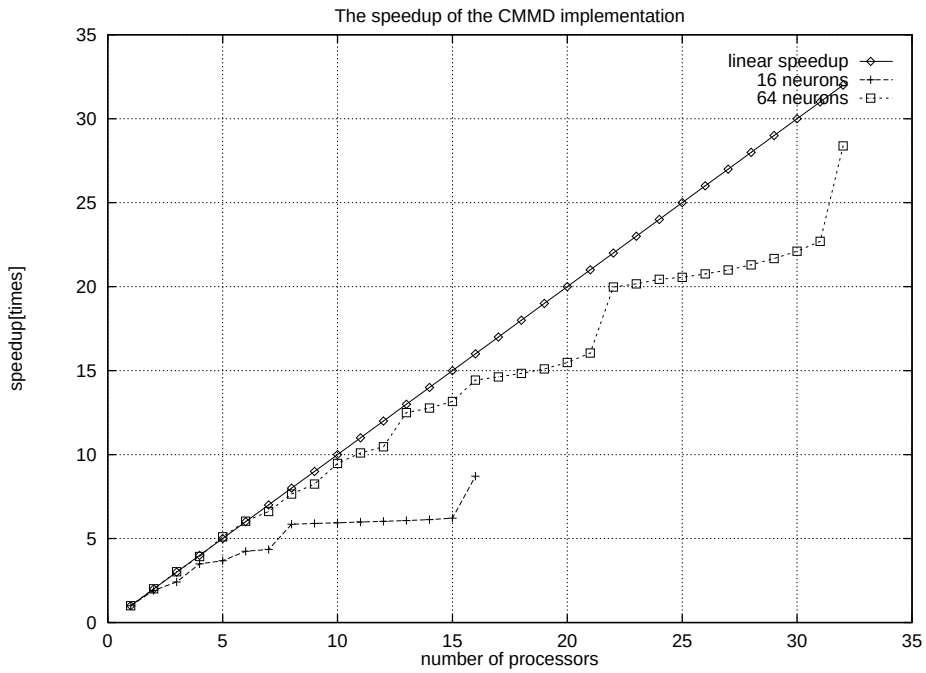

Figure 14: Speedup with 16 and 64 neurons compared to linear speedup using 32 processors.

7.2 Tests of performance

In this subsection the performance of the implementation is presented. The execution time used below is the time used by the program to perform one update of the p values.

7.2.1 Speedup

It is interesting to measure the speedup of the program when the number of processors are increased. The speedup is given by the equation

$$speedup_x = \frac{executiontime_1}{executiontime_x} \tag{21}$$

where $speedup_x$ denotes the speedup of the execution time when x processors is used compared to the execution time when one processor is used. There are some areas in Fig.14 where the speedup remains the same when the number of processors is increased. E.g., for 16 neurons the speedup is constant when the number of processors is increased from 8 to 15. This occurs because the maximum number of neurons per processor is constant in this area (in this case 2). When the number of processors are increased to 16, each node contains 1 neuron and the speedup is increased.

In Fig.14 it is shown that the speedup is not linear for small networks, e.g., for 16 neurons, as the number of processors are doubled. This is due to the fact that the number of computations on each processor decreases, while the number of communication operations

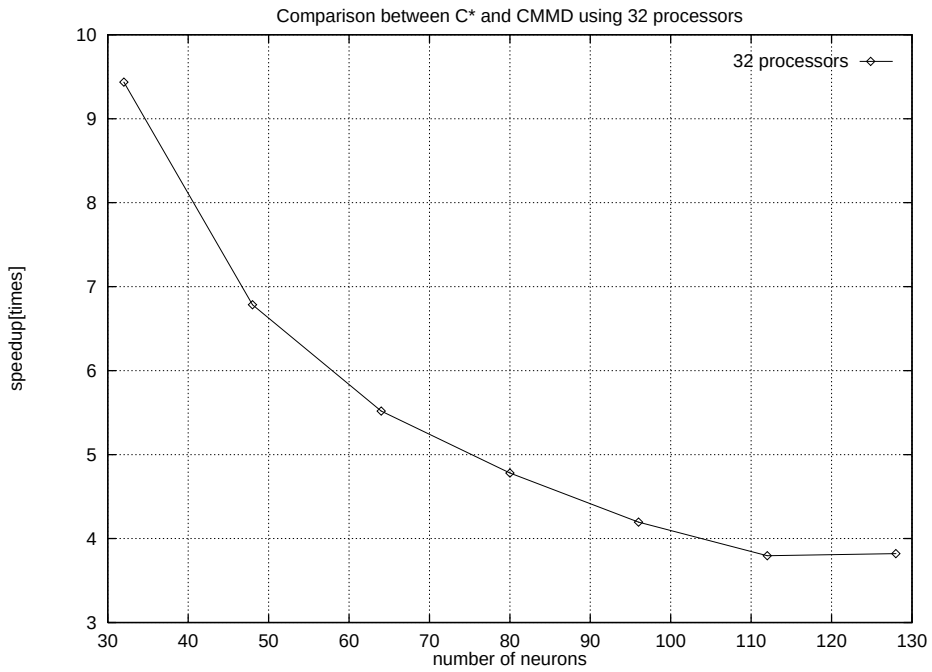

Figure 15: Comparison of the execution time of the CMMD implementation and the C* implementation. E.g., for 32 neurons the figure shows that the CMMD implementation runs approximately 9.5 times faster than the C* implementation.

is constant. For small networks the communication time soon becomes a significant part of the total execution time and the speedup curve declines. However, for larger networks the communication time remains a minor part and the speedup curve declines later.

7.2.2 Comparison with the C* implementation

The purpose of this thesis was to implement the major part of the RTRL algorithm using the CMMD routines. This was done in order to achieve a program that runs faster than the existing C* implementation. The results from chapter 6 show that the fastest communication is achieved when the CMAML facilities are used and therefore this feature is used in the implementation done in this thesis.
According to chapter 3.3.4 it is only possible to achieve a considerable increase in performance for small networks. This is because the number of computations increase much faster than the number of communications when the size of the network increases. Therefore the most obvious difference between the CMMD and the C* implementation occurs for reasonably small networks. This is verified in Fig.15, which shows that the CMMD implementation runs approximately 9.5 times faster than the C* implementation for a 32 neuron network. When the size of the network increases the difference in execution time between the two implementations decreases. E.g., for 96 neurons the CMMD implementation is approximately 4 times faster. When the size of the network increase even more the difference between the two implementations is constant. The reason for this is probably that it is easier to optimize

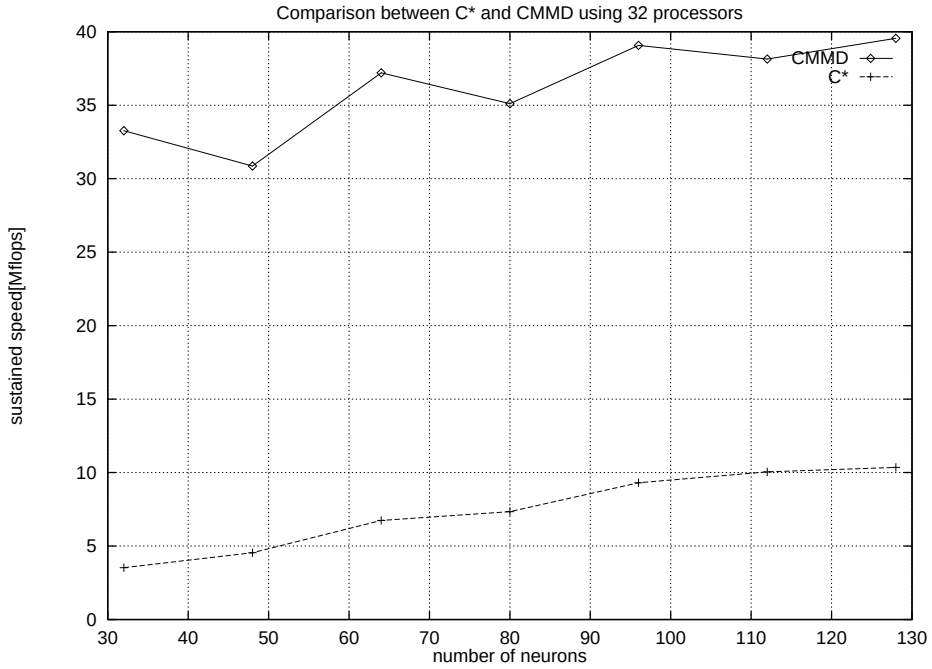

Figure 16: The sustained speed for the CMMD and the C* implementation for 32 processors.

a C program than it is to optimize a C* program.

In Fig.16 the sustained speed is plotted as a function of the networksize. The sustained speed is given by the equation

$$S_x = \frac{2 * n^4}{executiontime_x} \tag{22}$$

where S_x denote the sustained speed for a network of size x and $executiontime_x$ denotes the execution time for updating the p values once in a network of size x. The factor $2 * n^4$ is used because this is approximately the number of floating point operations performed when updating the p values once. In Fig.16 it is shown that the CMMD implementation utilize approximately 25 percent of the peak performance of the CM-5, which is 160 Megaflops[5]. It is probably possible to increase the sustained speed of the CMMD implementation if the vector units are used.

[5]32 SPARC is used and each has a peak performance of 5 Megaflops.

8 Conclusions

In this masters thesis the CMMD communication features on a CM-5 parallel computer were evaluated in order to implement the RTRL algorithm in an efficient way. The evaluations show that it is not possible to achieve overlapped communication/computation on this computer. This is due to limitations in the CM-5 hardware. Its capacity to buffer messages transmitted over the network is strictly limited. Moreover, the CM-5 hardware doesn't support DMA, resulting in that the CPU is interrupted when messages are to be stored in memory. With these limitations the gain from overlapped communication/computation is not noticeable. However, the evaluations show that the CMMD communication latency is remarkably less than the corresponding C* facilities. Using CMMD should therefore significantly enhance performance for programs with a high degree of communication. This is the case with the RTRL algorithm when smaller networks are used.

An implementation of the RTRL algorithm was designed in this thesis. It was written in the C programming language and the most efficient CMMD feature was used for communication among the processors. The evaluations of this implementation show that the performance is far better than the corresponding C* implementation for smaller networks. Even for large networks, where communication is a minor part of the execution time, an enhancement of performance was observed. This result shows that programs written in the C language are easier to optimize and runs faster on the CM-5 compared to corresponding C* programs.

Appendix

A Code

A.1 Evaluations

A.1.1 Cstar_comm.tex

```
/*************************************************************************
*                                                                        *
*  File:    Cstar_comm.cs                                               *
*                                                                        *
*  Authors: Gunnar Franzen & Ulf Johansson,                              *
*           guests at SEDAL, University of Sydney.                       *
*                                                                        *
*  Date:    17/12/93                                                    *
*                                                                        *
*  Description:                                                          *
*                                                                        *
*  Measure the communication and computation latency for a C* program.   *
*  The results are presented as the time needed to perform a parallel    *
*  left indexing communication operation, and an ackumulated multi-      *
*  plication respectively. The shapes are one-dimensional ringstructures.*
*                                                                        *
*  The following command line arguments are read by the program:         *
*                                                                        *
*   -r : the range, i.e., the maximum shape-size to be evaluated.        *
*   -s : the step in shape-size between consecutive measurements.        *
*   -b : the shape-size to begin with.                                   *
*                                                                        *
*************************************************************************/


#include <stdio.h>
#include <cm/timers.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{

shape []current_config;
int range, step, begin, array_size, i, k, c;
extern char* optarg;
double *ma_time, *cgc_time, *ma_first, *cgc_first;
```

```
/* Set default values */

range = 100000;
step  = 5000;
begin = 5000;
niter = 1000;

/* Parse command line arguments */

while ((c = getopt(argc, argv, "r:s:b:")) != -1)
   switch(c)
   {
      case 'r' : range = atoi(optarg); break;
      case 's' : step  = atoi(optarg); break;
      case 'b' : begin = atoi(optarg); break;
   }

/* Compute required memory to store results */

array_size = (range - begin + step)/step;


/* Allocate memory */

ma_time = calloc(array_size, sizeof(double));
ma_first = ma_time;
cgc_time = calloc(array_size, sizeof(double));
cgc_first = cgc_time;



/* Measuring begins */

for(i = begin; i < (range + 1); i += step)
{
   current_config = allocate_shape(&current_config, 1, i);
   {
      double:current_config x,y,z;
      with(current_config)
      {
 x = 3.141598234;
 y = 8.556679987;
 z = 0.0;
```

```
 CM_timer_clear(0);
 CM_timer_clear(1);

         /* Start timer 0 */

         CM_timer_start(0);

         /* Perform some arithmetic operations */

 for(k = 0; k < niter; k++)
    z += x*y;

         /* Stop timer 0 */

         CM_timer_stop(0);

         /* Start timer 1 */

         CM_timer_start(1);

         /* Communicate */

 for(k = 0; k < niter; k++)
    x = [(pcoord(0) - 1)%%i]y;

         /* Stop timer 1 */

         CM_timer_stop(1);

         /* Store results and convert to micro-seconds */

 *ma_time = ((CM_timer_read_cm_busy(0))/niter)*1000000;
  ma_time++;
 *cgc_time = ((CM_timer_read_cm_busy(1))/niter)*1000000;
  cgc_time++;
      }
   }
}

/* Present results */

ma_time = ma_first;
cgc_time = cgc_first;
```

```
printf("\n\n");
printf("  size   mult. & ack.    circ. shift  (micro s / op)\n\n");
for(i = begin; i < (range + 1); i += step)
{
    printf("%7d    %5.0f           ", i, *ma_time);
    printf("%5.0f\n", *cgc_time);
    ma_time++;
    cgc_time++;
}
}
```

A.1.2 CMMD_sync.c

```
/*****************************************************************************
*                                                                            *
*  File:    CMMD_sync.c                                                      *
*                                                                            *
*  Authors: Gunnar Franzen & Ulf Johansson,                                  *
*           guests at SEDAL, University of Sydney.                           *
*                                                                            *
*  Date:    17/12/93                                                         *
*                                                                            *
*  Description:                                                              *
*                                                                            *
*  Measure the time needed for communication and computation for nodes       *
*  connected in a 16-node-ringstructure. The result is presented as the      *
*  time needed to perform different numbers of arithmetic operations         *
*  followed by one synchronous send&receive-operation, where each node       *
*  communicates with its closest neighbours in the ring. The standard-       *
*  deviation for each measurement is also given.                             *
*                                                                            *
*  The following command line arguments are read by the program:             *
*                                                                            *
*   -m : the maximum number of arithmetic operations.                        *
*   -s : the step in arithmetic operations between                           *
*        consecutive measurements.                                           *
*   -a : the length of the message transmitted.                              *
*                                                                            *
*****************************************************************************/


#include <stdlib.h>
#include <stdio.h>
#include <cm/cmmd.h>
#include <cm/timers.h>
#include <math.h>

main(argc, argv)
int argc;
char** argv;
{
int num_nodes, my_address, niter, i, j, r, u, c;
int max_ops, step, loop=0, array_length;
```

```
int source, dest;
double ack_sum=1.00012, *msg_in, *msg_out, *mean_time, *time_value;
double variance, *std_dev;
extern char* optarg;

/* Set default values */

max_ops = 50;
step = 5;
array_length = 1;
niter = 10000;

/* Parse command line arguments */

while ((c = getopt(argc, argv, "m:s:a:")) != -1)
    switch (c) {
      case 'm' : max_ops = atoi(optarg); break;
      case 's' : step = atoi(optarg); break;
      case 'a' : array_length = atoi(optarg); break;
    }

/* Check environment and set corresponding parameters */

num_nodes = CMMD_partition_size();
my_address = CMMD_self_address();

/* Allocate memory */

mean_time    = (double *) calloc((max_ops/step), sizeof(double));
msg_in  = (double *) calloc(array_length, sizeof(double));
msg_out = (double *) calloc(array_length, sizeof(double));
time_value  = (double *) calloc(niter, sizeof(double));
std_dev = (double *) calloc((max_ops/step), sizeof(double));

/* Initialize message buffers */

for(u = 0; u < array_length; u ++)
{
  msg_in[u] = u;
  msg_out[u] = u;
}

/* Compute source and destination addresses */
```

```
source = my_address + 1;
dest   = my_address - 1;
if (dest < 0) dest += num_nodes;
if (source > (num_nodes - 1)) source -= num_nodes;

/* Measuring begins */

for(j = step; j < (max_ops + 1); j += step)
{
  CMMD_sync_with_nodes();

  /* Repeat many times to get reliable results */

  for(i = 0; i < niter; i++)
  {
  CMMD_sync_with_nodes();

  CMMD_node_timer_clear(0);

  /* Start timer */

  CMMD_node_timer_start(0);

     /* Communicate */

     CMMD_send_and_receive(source, CMMD_DEFAULT_TAG, msg_in,
                           array_length*sizeof(double), dest,
           CMMD_DEFAULT_TAG, msg_out,
                           array_length*sizeof(double));

     /* Perform some arithmetic operations */

     for(r = 0; r < j; r++)
       ack_sum += ack_sum*ack_sum;

  /* Stop timer */

  CMMD_node_timer_stop(0);

  /* Save mesured time */

  time_value[i] = CMMD_node_timer_busy(0);
  }
```

```
/* Compute the mean time */

mean_time[loop] = 0;
for(i = 10; i < niter; i++)
  mean_time[loop] += time_value[i];
mean_time[loop] /= (niter - 10);

/* Compute the standard deviation */

variance = 0;
for(i = 10; i < niter; i++)
  variance += pow(fabs(time_value[i] - mean_time[loop]), 2.00);
variance /= (niter - 10);
std_dev[loop]  = sqrt(variance);

/* Convert the results to micro-seconds */

mean_time[loop] *= 1000000;
std_dev[loop] *= 1000000;
loop++;
}

/* Set I/O-mode for presentation of the results */

CMMD_fset_io_mode(stdout, CMMD_sync_bc);

/* Present the results */

loop = 0;
for(j = step; j < (max_ops + 1); j += step)
{
  printf("%3d     %.0f    %.0f\n", j, mean_time[loop], std_dev[loop]);
  loop++;
}
}
```

A.1.3 CMMD_async.c

```
/*****************************************************************************
*                                                                            *
*  File:    CMMD_async.c                                                     *
*                                                                            *
*  Authors: Gunnar Franzen & Ulf Johansson,                                  *
*           guests at SEDAL, University of Sydney.                           *
*                                                                            *
*  Date:    17/12/93.                                                        *
*                                                                            *
*  Description:                                                              *
*                                                                            *
*  Measure the time needed for communication and computation for nodes       *
*  connected in a 16-node ringstructure. The result is presented as the      *
*  time ( plus standard deviation ) needed to perform different amounts      *
*  of arithmetic operations followed by one asynchronous send operation      *
*  and one asynchronous receive operation .                                  *
*                                                                            *
*  The following command line arguments are read by the program :            *
*                                                                            *
*   -m : the maximum number of arithmetic operations.                        *
*   -s : the step in arithmetic operations between                           *
*        consecutive measurements.                                           *
*   -a : the length of the message transmitted.                              *
*                                                                            *
*****************************************************************************/


#include <stdlib.h>
#include <stdio.h>
#include <cm/cmmd.h>
#include <cm/timers.h>
#include <math.h>

main(argc, argv)
int argc;
char** argv;
{
int num_nodes, my_address, tag = 0, niter, i, j, c, l, u;
int k, array_length;
int source, dest, max_ops, step, loop=0;
int r,s;
```

```
double ack_sum = 1.000012, *msg_in, *msg_out, *time;
double variance, *std_dev, *mean_time, *time_value;
extern char* optarg;
CMMD_mcb send_mcb, receive_mcb;

/* Set default values */

max_ops = 50;
step = 5;
array_length = 1;
niter = 10000;

/* Parse command line arguments */

while((c = getopt(argc, argv, "m:s:a:")) != -1)
    switch (c) {
      case 'm' : max_ops = atoi(optarg); break;
      case 's' : step = atoi(optarg); break;
      case 'a' : array_length = atoi(optarg); break;
    }

/* Check environment and set corresponding parameters */

num_nodes = CMMD_partition_size();
my_address = CMMD_self_address();

/* Allocate memory */

mean_time = (double *) calloc((max_ops/step), sizeof(double));
msg_in = (double *) calloc(array_length, sizeof(double));
msg_out = (double *) calloc(array_length, sizeof(double));
time_value = (double *) calloc(niter, sizeof(double));
std_dev = (double *) calloc((max_ops/step), sizeof(double));

/* Initialize message buffers */

for(u =0; u < array_length; u ++)
{
  msg_in[u] = u;
  msg_out[u] = u;
}

/* Compute source and destination addresses */
```

```
source = my_address + 1;
dest   = my_address - 1;
if (dest < 0) dest += num_nodes;
if (source > (num_nodes - 1)) source -= num_nodes;

/* Measuring begins */

for(j = step; j < (max_ops + 1); j += step)
{
  CMMD_sync_with_nodes();

  /* Repeat many times to get reliable results */

  for(i = 0; i < niter; i++)
  {
     CMMD_sync_with_nodes();

     /* Start timer */

     CMMD_node_timer_clear(0);
     CMMD_node_timer_start(0);

     /* Communicate */

     send_mcb = CMMD_send_async(dest, tag, msg_out,
                                array_length*sizeof(double), NULL, 0);
     receive_mcb = CMMD_receive_async(source, tag, msg_in,
                                      array_length*sizeof(double), NULL,0);

     /* Perform some arithmetic operations */

     for(r = 0; r < j; r++)
        ack_sum += ack_sum*ack_sum;

     /* Wait until messages are sent and received */

     CMMD_msg_wait(send_mcb);
     CMMD_msg_wait(receive_mcb);

     /* Free message control blocks */

     CMMD_free_mcb(send_mcb);
     CMMD_free_mcb(receive_mcb);
```

```
      /* Stop timer */

      CMMD_node_timer_stop(0);

      /* Save measured time */

      time_value[i] = CMMD_node_timer_busy(0);
   }

   /* Compute the mean time */

   mean_time[loop] = 0;
   for(i = 10; i < niter; i++)
      mean_time[loop] += time_value[i];
   mean_time[loop] /= (niter - 10);

   /* Compute the standard deviation */

   variance = 0;
   for(i = 10; i < niter; i++)
      variance += pow(fabs(time_value[i] - mean_time[loop]), 2.00);
   variance /= (niter - 10);

   std_dev[loop] = sqrt(variance);

   /* Convert results to micro-seconds */

   mean_time[loop] *= 1000000;
   std_dev[loop] *= 1000000;

   loop++;
}

/* Set I/O-mode for presentation of the results */

CMMD_fset_io_mode(stdout, CMMD_sync_bc);
loop = 0;

/* Present the results */

for(j = step; j < (max_ops + 1); j += step)
{
   printf("%3d    %.0f   %.0f\n", j, mean_time[loop], std_dev[loop]);
   loop++;
```

```
}
}
```

A.1.4 CMMD_virtual.c

```
/*****************************************************************************
*                                                                            *
*  File:    CMMD_virtual.c                                                   *
*                                                                            *
*  Authors: Gunnar Franzen & Ulf Johansson,                                  *
*           guests at SEDAL, University of Sydney.                           *
*                                                                            *
*  Date:    17/12/93.                                                        *
*                                                                            *
*  Description:                                                              *
*                                                                            *
*  Measure the time to communicate with the use of virtual channels.         *
*  Measurements are done with different amounts of arithmetic oper-          *
*  ations performed between each pair of send and receive operations.        *
*  The result is presented as the time (plus standard-deviation)             *
*  needed to perform a send operation followed by a specified amount         *
*  of arithmetic operations and a receive operation.                         *
*                                                                            *
*  The following switches are read by the program :                          *
*                                                                            *
*   -m : the maximum number of arithmetic operations.                        *
*   -s : the step i arithmetic operations between                            *
*        consecutive measurements.                                           *
*   -a : the length of the message transmitted.                              *
*                                                                            *
*****************************************************************************/


#include <stdlib.h>
#include <stdio.h>
#include <cm/cmmd.h>
#include <cm/timers.h>
#include <math.h>

main(argc, argv)
int argc;
char** argv;
{
int num_nodes, my_address, tag = 0, niter, i, j, c, u, loop=0;
int array_length, step, scd, rcd;
int source, dest;
```

```
int max_ops;
double ack_sum=1.0000012, *msg_in, *msg_out, *mean_time, *time_value;
double variance, *std_dev;
extern char* optarg;
CMMD_mcb send_mcb, receive_mcb;

/* Set default values */

max_ops = 50;
step = 5;
array_length = 10;
niter = 10000;

/* Parse command line arguments */

while ((c = getopt(argc, argv, "m:s:a:")) != -1)
   switch(c)
   {
      case 'm' : max_ops = atoi(optarg); break;
      case 's' : step = atoi(optarg); break;
      case 'a' : array_length = atoi(optarg); break;
   }

/* Check environment and set corresponding parameters */

num_nodes = CMMD_partition_size();
my_address = CMMD_self_address();

/* Allocate memory */

time_value = (double *) calloc(niter, sizeof(double));
mean_time = (double *) calloc((max_ops/step), sizeof(double));
msg_in  = (double *) calloc(array_length, sizeof(double));
msg_out = (double *) calloc(array_length, sizeof(double));
std_dev = (double *) calloc((max_ops/step), sizeof(double));

/* Initialize message buffers */

for(u = 0; u < array_length; u++)
{
  msg_in[u] = u;
  msg_out[u] = u + j;
}
```

```
/* Compute source and destination addresses */

source = my_address + 1;
dest   = my_address - 1;
if (dest < 0) dest += num_nodes;
if (source > (num_nodes - 1)) source -= num_nodes;

/* Open communication channels */

scd = CMMD_open_send_channel(dest, tag, msg_out, array_length*sizeof(double));
rcd = CMMD_open_receive_channel(source, tag, msg_in,
                                array_length*sizeof(double));

/* Measuring begins */

for(j = step; j < (max_ops + 1); j += step)
{
  CMMD_sync_with_nodes();

  /* Repeat many times to get reliable values */

  for(i = 0; i < niter; i++)
  {
      CMMD_sync_with_nodes();

      /* Start timer */

      CMMD_node_timer_clear(0);
      CMMD_node_timer_start(0);

      /* Wait until channel is writable */

      while(CMMD_is_channel_writable(scd) != 1);

      /* Send message */

      CMMD_write_channel(scd);

      /* Perform some arithmetic operations */

      for(u = 0; u < j; u++)
         ack_sum += ack_sum*ack_sum;

       /* Wait for message to arrive */
```

```
        while(CMMD_is_channel_readable(rcd) != 1);

        /* Prepare for next message to arrive */

        CMMD_reset_channel(rcd);

        /* Stop timer */

        CMMD_node_timer_stop(0);

        /* Save measured time */

        time_value[i] = CMMD_node_timer_busy(0);
    }

    /* Compute the mean time */

    mean_time[loop] = 0;
    for(i =10; i < niter; i++)
       mean_time[loop] += time_value[i];
    mean_time[loop] /= (niter - 10);

    /* Compute the standard deviation */

    variance = 0;
    for(i = 10; i < niter; i++)
       variance += pow(fabs(time_value[i] - mean_time[loop]), 2.00);
    variance /= (niter -10);

    std_dev[loop] = sqrt(variance);

    mean_time[loop] *= 1000000;
    std_dev[loop] *= 1000000;

    loop++;
}

/* Set I/O-mode for presentation of the results */

CMMD_fset_io_mode(stdout, CMMD_sync_bc);

/* Present results */
```

```
loop = 0;
for(j = step; j < (max_ops + 1); j += step)
{
   printf("%3d    %.0f   %.0f\n", j, mean_time[loop], std_dev[loop]);
   loop++;
}
}
```

A.1.5 CMAML_comm.c

```
/*************************************************************************
*                                                                        *
*  File:    CMAML_comm.c                                                 *
*                                                                        *
*  Authors: Gunnar Franzen & Ulf Johansson,                              *
*           guests at SEDAL, University of Sydney.                       *
*                                                                        *
*  Date:    17/12/93.                                                    *
*                                                                        *
*  Description:                                                          *
*                                                                        *
*  Measure the time needed for communication and computation for nodes   *
*  connected in a 16-node ringstructure when the CMAML functions are     *
*  used. The result is presented as the time (plus standard-deviation)   *
*  needed to perform a send operation followed by a specified number of  *
*  arithmetic operations and a receive operation.                        *
*                                                                        *
*  The following command line arguments are read by the program :        *
*                                                                        *
*   -m : the maximum number of arithmetic operations.                    *
*   -s : the step in arithmetic operations between                       *
*        consecutive measurements.                                       *
*   -a : the length of the message transmitted.                          *
*                                                                        *
*************************************************************************/


#include <stdlib.h>
#include <stdio.h>
#include <cm/cmmd.h>
#include <cm/timers.h>
#include <math.h>

main(argc, argv)
int argc;
char** argv;
{
int num_nodes, my_address, tag = 0, niter, i, j, c;
int array_length;
int source, dest, max_ops, step, loop=0;
int r,s, receive_wait, count;
```

```
double ack_sum = 1.000012, *msg_in, *msg_out, *time_value, *mean_time;
double variance, *std_dev;
extern char* optarg;
int  send_port, receive_port = 1;

/* Set default values */

max_ops = 50;
step = 5;
array_length = 1;
niter = 10000;

/* Parse command line arguments */

while((c = getopt(argc, argv, "m:s:a:")) != -1)
    switch (c) {
      case 'm' : max_ops = atoi(optarg); break;
      case 's' : step = atoi(optarg); break;
      case 'a' : array_length = atoi(optarg); break;
    }

/* Check environment and set corresponding parameters */

num_nodes = CMMD_partition_size();
my_address = CMMD_self_address();

/* Allocate memory */

time_value = (double *) calloc(niter, sizeof(double));
mean_time = (double *) calloc((max_ops/step), sizeof(double));
std_dev = (double *) calloc((max_ops/step), sizeof(double));
msg_in  = (double *) calloc(array_length, sizeof(double));
msg_out = (double *) calloc(array_length, sizeof(double));

/* Initialize message buffers */

for(i = 0; i < array_length; i ++)
{
  msg_in[i] = i;
  msg_out[i] = i + array_length;
}

/* Compute source and destination addresses */
```

```
source = my_address + 1;
dest   = my_address - 1;
if (dest < 0) dest += num_nodes;
if (source > (num_nodes - 1)) source -= num_nodes;

/* Set I/O-mode for error messages */

CMMD_fset_io_mode(stdout, CMMD_independent);
CMMD_fset_io_mode(stderr, CMMD_independent);

/* Set CMAML parameters */

CMAML_disable_interrupts();
if(!CMAML_allocate_this_rport(receive_port))
   printf(" processor %d failed to allocate port\n", my_address);
CMAML_set_rport_addr(receive_port, msg_in);
CMAML_set_rport_byte_counter(receive_port, array_length*sizeof(double));
CMAML_set_rport_handler(receive_port, NULL);
CMAML_set_rport_handler_arg2(receive_port, NULL);
CMAML_set_rport_total_nbytes(receive_port, array_length*sizeof(double));
CMAML_enable_interrupts();

/* Measuring begins */

for(j = step; j < (max_ops + 1); j += step)
{
  CMMD_sync_with_nodes();

  /* Repeat many times to get reliable results */

  for(i = 0; i < niter; i++)
  {
    CMMD_sync_with_nodes();

    /* Start timer */

    CMMD_node_timer_clear(0);
    CMMD_node_timer_start(0);

    /* Send message */

    CMAML_disable_interrupts();
    CMAML_scopy(dest, receive_port, 0, 0, msg_out,
              array_length*sizeof(double), NULL, NULL);
```

```
      /* Perform some arithmetic operations */

      for(r = 0; r < j; r++)
         ack_sum += ack_sum*ack_sum;

      /* Poll for the arriving message */

      while(CMAML_get_rport_byte_counter(receive_port) != 0)
         CMAML_poll();

      /* Reset byte-counter for the next message */

      CMAML_set_rport_byte_counter(receive_port, array_length*sizeof(double));
      CMAML_enable_interrupts();

      /* Stop timer */

      CMMD_node_timer_stop(0);

      /* Save measured time */

      time_value[i] = CMMD_node_timer_busy(0);
   }

/* Compute the mean time */

mean_time[loop] = 0;
for(i = 10; i < niter; i++)
   mean_time[loop] += time_value[i];
mean_time[loop] /= (niter - 10);

/* Compute the standard-deviation */

variance = 0;
for(i = 10; i < niter; i++)
    variance += pow(fabs(time_value[i] - mean_time[loop]), 2.00);
variance /= (niter - 10);
std_dev[loop] = sqrt(variance);
mean_time[loop] *= 1000000;
std_dev[loop] *= 1000000;
loop++;
}
```

```
/* Set I/O-mode for presentation of the results */

CMMD_fset_io_mode(stdout, CMMD_sync_bc);

/* Present the results */

loop = 0;
for(j = step; j < (max_ops + 1); j += step)
{
  printf("%3d    %.0f   %.0f\n", j, mean_time[loop], std_dev[loop]);
  loop++;
}
}
```

A.2 Implementation

A.2.1 rtrl_serial.c

```
/*************************************************************************
*                                                                        *
*  File:    rtrl_serial.c                                               *
*                                                                        *
*  Authors: Gunnar Franzen & Ulf Johansson,                              *
*           guests at SEDAL, University of Sydney.                       *
*                                                                        *
*  Date:    17/12/93                                                    *
*                                                                        *
*  Description:                                                          *
*                                                                        *
*  This is a serial implementation of the main part of the RTRL-         *
*  algorithm. The part implemented corresponds to Eq. (10) described     *
*  in the report.                                                        *
*                                                                        *
*  The following command line arguments are read by the program:         *
*                                                                        *
*   <num_neurons> : the desired number of neurons in the neural          *
*                   network.                                             *
*   <file>        : the file containing the input data.                  *
*                                                                        *
*************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <cm/cmmd.h>
#include <cm/timers.h>

main(argc, argv)
int argc;
char **argv;
{
  double *pk, **pj,  ***p, *pk_new, **pj_new, ***p_new;
  double *wj, **w;
  double *y;
  double *fd;
  double time;
  int i, j, k, l;
  int n;
  char *datafile;
  FILE *fptr;


  /* Parse command line arguments */

  if(argc != 3)
  {
    fprintf(stderr, "USAGE: %s <num_nodes> <datafile>\n", argv[0]);
    exit (1);
  }

  n = atoi(argv[1]);
  datafile = argv[2];


  /* Allocate memory */

  pk = (double *)malloc(n*n*n*sizeof(double));
  pj = (double **)malloc(n*n*sizeof(double *));
  p  = (double ***)malloc(n*sizeof(double **));

  pk_new = (double *)malloc(n*n*n*sizeof(double));
  pj_new = (double **)malloc(n*n*sizeof(double *));
  p_new  = (double ***)malloc(n*sizeof(double **));

  wj = (double *)malloc(n*n*sizeof(double));
  w  = (double **)malloc(n*sizeof(double *));

  y  = (double *)malloc(n*sizeof(double));
```

```
fd = (double *)malloc(n*sizeof(double));


/* Read input data from datafile */

fptr = fopen(datafile, "r");

for(i = 0; i < n*n; i++)
  fscanf(fptr, "%lf", &wj[i]);

for(i = 0; i < n*n*n; i++)
  fscanf(fptr, "%lf", &pk[i]);

for(i = 0; i < n; i++)
  fscanf(fptr, "%lf", &y[i]);

for(i = 0; i < n; i++)
  fscanf(fptr, "%lf", &fd[i]);

fclose(fptr);


/* Initialize pointers to data-matrices */

for(i = 0; i < n*n; i++)
{
   pj[i] = &(pk[i*n]);
   pj_new[i] = &(pk_new[i*n]);
}

for(i = 0; i < n; i++)
{
   p[i] = &(pj[i*n]);
   p_new[i] = &(pj_new[i*n]);
}

for(i = 0; i < n; i++)
   w[i] = &(wj[i*n]);


/* Compute new p-values */

for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
```

```
      for(k = 0; k < n; k++)
      {
        for(l = 0; l < n; l++)
          p_new[i][j][k] += (w[k][l])*(p[i][j][l]);

        if(i == k)
          p_new[i][j][k] += y[j];

        p_new[i][j][k] *= fd[k];
      }


  /* Print new p-values */

  for(i = 0; i < n; i++)
  {
    printf("\n");
    for(j = 0; j < n; j++)
    {
      printf("\n");
      for(k = 0; k < n; k++)
        printf("%f  ", p_new[i][j][k]);
    }
  }
  printf("\n");
}
```

A.2.2 rtrl_parallel.c

```
/*************************************************************************
*                                                                        *
*  File:    rtrl_parallel.c                                              *
*                                                                        *
*  Authors: Gunnar Franzen & Ulf Johansson,                              *
*           guests at SEDAL, University of Sydney.                       *
*                                                                        *
*  Date:    17/12/93                                                     *
*                                                                        *
*  Description:                                                          *
*                                                                        *
*  This is a parallel implementation of the main part of the RTRL-       *
*  algorithm. The part implemented corresponds to Eq. (10) described     *
*  in the report.                                                        *
*                                                                        *
*  The following command line arguments are read by the program:         *
*                                                                        *
*  <num_neurons> : the desired number of neurons in the neural           *
*                  network.                                              *
*  <part_size>   : the number of physical processors to be               *
*                  used.                                                 *
*  <input_file>  : the name of the file containing the input             *
*                  data.                                                 *
*  <bin_file>    : the desired name of the file that will                *
*                  contain the input data when it is converted           *
*                  by the program.                                       *
*                                                                        *
*************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <cm/cmmd.h>
#include <cm/timers.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>


int my_address, ma;
```

```
int i, j, k, l, s, r, u;
int source, dest, n = 2;
int part_size, c, vp, vppp, file_desc, t, tag =0;
int w_receive_port, y_receive_port, fd_receive_port;
int comm_counter = 0;
char *input_file, *bin_file;
extern char* optarg;
double *wmsg_in, *wmsg_out, *ymsg_in, *ymsg_out;
double *fdmsg_in, *fdmsg_out;
double **p, **p_new, **w, **w_tmp;
double *p1, *p1_new, *w1, *w1_tmp;
double *y, *fd, *input_data;
double time, max_time, comm_time, max_comm_time, cp_time, max_cp_time;


/* Specifications of functions */

void allocate();
void convert_input();
void read_binary_file();
void source_and_dest();
void set_up_ports();
void compute_p();
void print_result();



/***** Function main() *****/

main (argc, argv)
int argc;
char **argv;
{

/* Parse command line arguments */

if(argc != 4)
{
  CMMD_fset_io_mode(stderr, CMMD_sync_bc);
  fprintf(stderr, "USAGE: %s <num_neurons> <num_proc.> <input_file>\n",
          argv[0]);
  exit (1);
}
```

```
n          = atoi(argv[1]);
part_size  = atoi(argv[2]);
input_file = argv[3];
bin_file   = argv[4];


/* Set global and local parameters */

CMMD_reset_partition_size(part_size);
file_desc  = CMMD_global_open(bin_file, O_RDWR | O_CREAT | O_TRUNC, 0666);
my_address = CMMD_self_address();
ma = my_address;


/* Detemine the number of virtual processors on this node */

if( n > part_size )
{
   vppp = n / part_size;
   if(n % part_size > ma)
       vppp = vppp + 1;
}
else vppp = 1;


/* Set I/O-modes */

CMMD_fset_io_mode(stdin, CMMD_independent);
CMMD_fset_io_mode(stdout, CMMD_independent);
CMMD_fset_io_mode(stderr, CMMD_independent);



allocate();

/* Node 0 converts input file to a suitable format */

if ( ma == 0)
  convert_input();

CMMD_sync_with_nodes();

read_binary_file();
```

```
source_and_dest();

set_up_ports();

CMMD_node_timer_clear(1);

/* Start timer */

CMMD_node_timer_start(1);

compute_p();

/* Stop timer */

CMMD_node_timer_stop(1);

print_result();

time = CMMD_node_timer_elapsed(1);

/* Find the slowest node */

max_time = CMMD_reduce_double(time, CMMD_combiner_dmax);

CMMD_fset_io_mode(stdout, CMMD_sync_bc);

printf("Processors  = %d\n", part_size);
printf("Time  = %f\n", max_time);

} /* End main() */



/***** Function allocate() *****/

void allocate()
{

  /* Write info */

  if(my_address == 0)
  {
    printf("function allocate() running...\n");
    fflush(stdout);
```

```
  }


  /* Allocate memory for all variables */

  p1 = (double *)calloc(vppp*n*n, sizeof(double));
  p = (double **)calloc(vppp, sizeof(double *));
  p1_new  = (double *)calloc(vppp*n*n, sizeof(double));
  p_new  = (double **)calloc(vppp, sizeof(double *));
  w1 = (double *)calloc(vppp*n, sizeof(double));
  w = (double **)calloc(vppp, sizeof(double *));
  w1_tmp = (double *)calloc(vppp*n, sizeof(double));
  w_tmp = (double **)calloc(vppp, sizeof(double *));
  y = (double *)malloc(vppp*sizeof(double));
  fd = (double *)malloc(vppp*sizeof(double));

  wmsg_in  = (double *) malloc(n*sizeof(double));
  wmsg_out = (double *) malloc(n*sizeof(double));
  ymsg_in  = (double *) malloc(sizeof(double));
  ymsg_out = (double *) malloc(sizeof(double));
  fdmsg_in  = (double *) malloc(sizeof(double));
  fdmsg_out = (double *) malloc(sizeof(double));

  /* Initialize row-pointers to matrices */

  for ( i = 0; i < vppp ; i++ )
    p[i] = &(p1[i*n*n]);

  for ( i = 0; i < vppp ; i++ )
  p_new[i] = &(p1_new[i*n*n]);

  for ( i = 0; i < vppp ; i++ )
    w[i] = &(w1[i*n]);

  for ( i = 0; i < vppp ; i++ )
    w_tmp[i] = &(w1_tmp[i*n]);

} /* End allocate() */



/***** Function convert_input() *****/

void convert_input()
```

```
{
  int i, array_length;
  FILE *fptr;

  /* Write info */

  if(my_address == 0)
  {
    printf("function convert_input() running...\n");
    fflush(stdout);
  }

  /* Compute the total number of elements (p,w,z, and fd) */

  array_length = n*n*n + n*n + 2*n;

  /* Allocate memory for all elements */

  input_data = (double *)malloc(array_length*sizeof(double));

  /* Read from input-file */

  fptr = fopen(input_file, "r");

  for(i = 0; i < array_length; i++)
     fscanf(fptr, "%lf", &input_data[i]);

  /* Write to binary-file */

  write(file_desc, input_data, array_length*sizeof(double));

  fclose(fptr);
  free(input_data);

} /* End convert_input() */



/***** Function read_binary_file() *****/

void read_binary_file()
{

  /* Write info */
```

```
  if(my_address == 0)
  {
    printf("function read_binary_file() running...\n");
    fflush(stdout);
  }

  /* Set I/O-mode for binary-file */

  CMMD_set_io_mode(file_desc, CMMD_sync_seq);

  /* Reset file-pointer */

  lseek(file_desc, 0L, SEEK_SET);

  /* Read sequentially from the binary-file */

  read(file_desc, w1, n*vppp*sizeof(double));
  read(file_desc, p1, n*n*vppp*sizeof(double));
  read(file_desc, y, vppp*sizeof(double));
  read(file_desc, fd, vppp*sizeof(double));

} /* End read_binary_file() */



/***** Function source_and_dest() *****/

void source_and_dest()
{

  /* Write info */

  if(my_address == 0)
  {
    printf("function source_and_dest() running...\n");
    fflush(stdout);
  }

  /* Compute source and destination addresses */

  source = my_address + 1;
  dest   = my_address - 1;
  if (dest < 0) dest += part_size;
```

```
  if (source > (part_size - 1)) source -= part_size;

} /* End source_and_dest() */



/***** Function set_up_ports() *****/

void set_up_ports()
{

  /* Write info */

  if(my_address == 0)
  {
    printf("function set_up_ports() running...\n");
    fflush(stdout);
  }

  /* Label CMAML receive-ports */

  w_receive_port = 1;
  y_receive_port = 2;
  fd_receive_port = 3;

  /* Disable interrupts while setting port-parameters */

  CMAML_disable_interrupts();

  /* Set parameters related to the w_receive_port */

  if(!CMAML_allocate_this_rport(w_receive_port))
    printf("processor %d failed allocating w_receive_port ...\n");
  CMAML_set_rport_addr(w_receive_port, wmsg_in);
  CMAML_set_rport_byte_counter(w_receive_port, n*sizeof(double));
  CMAML_set_rport_handler(w_receive_port, NULL);
  CMAML_set_rport_handler_arg2(w_receive_port, NULL);
  CMAML_set_rport_total_nbytes(w_receive_port, n*sizeof(double));

  /* Set parameters related to the y_receive_port */

  if(!CMAML_allocate_this_rport(y_receive_port))
    printf("processor %d failed allocating y_receive_port ...\n");
  CMAML_set_rport_addr(y_receive_port, ymsg_in);
```

```
  CMAML_set_rport_byte_counter(y_receive_port, sizeof(double));
  CMAML_set_rport_handler(y_receive_port, NULL);
  CMAML_set_rport_handler_arg2(y_receive_port, NULL);
  CMAML_set_rport_total_nbytes(y_receive_port, sizeof(double));

  /* Set parameters related to the fd_receive_port */

  if(!CMAML_allocate_this_rport(fd_receive_port))
    printf("processor %d failed allocating fd_receive_port ...\n");
  CMAML_set_rport_addr(fd_receive_port, fdmsg_in);
  CMAML_set_rport_byte_counter(fd_receive_port, sizeof(double));
  CMAML_set_rport_handler(fd_receive_port, NULL);
  CMAML_set_rport_handler_arg2(fd_receive_port, NULL);
  CMAML_set_rport_total_nbytes(fd_receive_port, sizeof(double));

  /* Enable interrupts again */

  CMAML_enable_interrupts();

  /* Synchronize */

  CMMD_sync_with_nodes();

} /* End set_up_ports() */



/***** Function compute_p() *****/

void compute_p()
{
  int p_new_index, p_index_r, p_index_s, w_index;
  int vpnn, mavppp, vpn, nn;
  int start, stop, i,counter=0;
  double *p_ptr, *w_ptr;

  /* Compute ma*vppp to simplify subsequent computations */

  if ( ma <= (n % part_size))
    mavppp = ma*((n/part_size)+1);
  else
    mavppp = (n%part_size)*((n/part_size)+1)+(ma-(n%part_size))*(n/part_size);

  /* Computation begins */
```

```
 for ( s = 0; s < n; s++ )
  {
   for ( r = 0; r < n; r++)
     {
      for ( vp = 0; vp < vppp; vp++)
 {

           /* Compute indices for matrices */

           p_index_r = vp*n*n + (r+vp+mavppp)*n%(n*n);

           p_index_s = (s+vp+mavppp)%n;

           p_new_index = p_index_r + p_index_s;

           w_index = vp*n;

           start = p_index_r;
           stop = start + n;

           p_ptr = &p1[start];
           w_ptr = &w1[w_index];

           /* Initialize p1_new */

   p1_new[p_new_index] = 0;

           /* Compute the dot-product p*w */

           for(i = start; i < stop; i++, p_ptr++, w_ptr++)
              p1_new[p_new_index] += (*p_ptr)*(*w_ptr);

           if(s == 0)
      {
                /* Add corresponding z-value (y) if necessary */

                p1_new[p_new_index] += y[vp];

                /* Shift z-values if necessary */

if ( vp == vppp - 1 )
  {
           *ymsg_out = y[0];
```

```
  for( i = 0; i < vppp; i++ )
      y[i] = y[(i+1) % vppp];

                  CMMD_sync_with_nodes();
                  CMAML_disable_interrupts();
                  CMAML_scopy(dest, y_receive_port, 0, 0, ymsg_out,
                              sizeof(double), NULL, NULL);
                  while(CMAML_get_rport_byte_counter(y_receive_port) != 0)
                    CMAML_poll();
                  CMAML_set_rport_byte_counter(y_receive_port,
                                               sizeof(double));
                  CMAML_enable_interrupts();

          y[vppp-1] = *ymsg_in;

} /* if vp */
      } /* if s */

          /* Multiply with corresponding fd-value */

          p1_new[p_new_index] *= fd[vp];

 } /* for vp */

         CMMD_sync_with_nodes();

      } /* for r*/

      CMMD_sync_with_nodes();

     /* Shift fd-values */

     *fdmsg_out = fd[0];
     for ( i = 0; i < vppp; i++)
fd[i] = fd[(i+1)%vppp];

     CMMD_sync_with_nodes();
     CMAML_disable_interrupts();
     CMAML_scopy(dest, fd_receive_port, 0, 0, fdmsg_out,
                              sizeof(double), NULL, NULL);
     while(CMAML_get_rport_byte_counter(fd_receive_port) != 0)
                    CMAML_poll();
     CMAML_set_rport_byte_counter(fd_receive_port, sizeof(double));
     CMAML_enable_interrupts();
```

```
       fd[vppp-1] = *fdmsg_in;

       /* Shift w-vectors */

       for ( i = 0; i < n; i++)
             wmsg_out[i] = w[0][i];

       for ( vp = 0; vp < vppp; vp++)
 for ( i = 0; i < n; i++)
             w[vp][i] = w[(vp+1)%vppp][i];

       CMMD_sync_with_nodes();
       CMAML_disable_interrupts();
       CMAML_scopy(dest, w_receive_port, 0, 0, wmsg_out,
                                n*sizeof(double), NULL, NULL);
       while(CMAML_get_rport_byte_counter(w_receive_port) != 0)
                     CMAML_poll();
       CMAML_set_rport_byte_counter(w_receive_port, n*sizeof(double));
       CMAML_enable_interrupts();

       for ( i = 0; i < n; i++)
             w[vppp-1][i] = wmsg_in[i];

      CMMD_sync_with_nodes();

   } /* for s */

} /* End compute_p() */



/***** Function print_result() *****/

void print_result()
{
 for( i = 0; i < n ; i++)
   {

     /* Every node prints its updated p-values sequentially */

     if ( i == my_address )
      {
        printf("\n");
```

```
        for ( vp = 0; vp < vppp; vp++)
          {
            for ( r = 0; r < n; r++ )
      {
for ( l = 0; l < n; l++)
  {
    printf("%-.6f ", p_new[vp][r*n+l]);
  }
        printf("\n");
      }
    printf("\n");
          }
      }
     CMMD_sync_with_nodes();
   }
} /* End print_result() */
```

References

[1] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computing*, 1:270-280, 1989.

[2] Jenq-Neng Hwang and Sun-Yuan Kung. Parallel algorithms/ architectures for neural networks. *Journal of VLSI Signal Processing*, 1:221-251, 1989.

[3] Dan I. Moldovan and Jose A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, Volume C-35, No. 1:1-12, January 1986.

[4] Richard P. Lippmann. An introduction to Computing with Neural Nets, *IEEE ASSP Magazine*: 4-22, April 1987.

[5] Tomas Nordström and Bertil Svensson. Using and Designing Massively Computers for Artificial Neural Networks. *Journal of Parallel and Distributed Computing*, 14: 260-285, 1992

[6] H. T. Kung. Why systolic architechtures? *Computer*, 15: 37-46, January. 1982.

[7] Jose A. B. Fortes and Benjamin W. Wah. Systolic arrays, from concept to implementation *Computer*, 7: 12-17, July. 1987.

[8] John Hertz, Anders Krogh and Richard G. Palmer. *Introduction to the Theory of Neural Computing*, p 184-186.

[9] Igor Z. Milosavljević and Marwan A. Jabri. Mappings of Real Time Recurrent Learning to SIMD Processor Arrays. *Proceedings of the Fifth Australian Conference on Neural Networks*, Januari-Februari 1994.

[10] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, January 1992.

[11] Thinking Machines Corporation. *C* Programming Guide*, May 1993.

[12] Thinking Machines Corporation. *CMMD Reference Manual*, May 1993.

[13] Thinking Machines Corporation. *CMMD User's Guide*, May 1993.