INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

A Symbolic Complexity Analysis of Connectionist Algorithms for Distributed-Memory Machines

Jonathan Bachrach

TR-92-043

July 1992

Abstract

This paper attempts to rigorously determine the computation and communication requirements of connectionist algorithms running on a distributed-memory machine. The strategy involves (1) specifying key connectionist algorithms in a high-level object-oriented language, (2) extracting their running times as polynomials, and (3) analyzing these polynomials to determine the algorithms' space and time complexity. Results are presented for various implementations of the back-propagation algorithm [4].

Name	Binding	Description
n		Number of rows in MATRIX or elements in VECTOR
m	n	Number of columns in MATRIX
d	$\frac{\sqrt{n}}{n}$	Density of SPARSE-MATRIX, $d \in [0,1]$
p		Number of processors

Table 1: The variables used in time polynomials. Also shown are the bindings to these variables used for the analysis.

1 Purpose

The purpose of this analysis is to rigorously determine the computation and communication requirements of connectionist algorithms running on a distributed-memory machine. Furthermore, the desired results should be architecture independent, that is, they should be independent of the exact topology of the communication network and the computer architecture of the processor. Of course this type of analysis will limit the scope and detail of the results, but the hope is that it will guide the choice of various subsystems and will serve as the basis for more refined analysis.

2 Strategy

The strategy involves specifying key connectionist algorithms in a high-level object-oriented language and then extracting their running times as polynomials. The polynomials are composed of variables and constants, where the variables specify the size and connectivity of the network and number of processors, and the constants specify the average time to complete each primitive routine such as the time to read a word from memory. The variables are listed in Table 1 and the constants are shown in Table 2. As an example, the average time to complete a VECTOR addition of length n would be $n(2r + w + a)$. The Sather code for this example is shown in Table 3.

The algorithms are coded in an object-oriented programming language called Sather (Omohundro, [3]). There are objects for all the connectionist data structures such as VECTOR's, MATRIX's, LAYER's, CONNECT's, and NETWORK's. Each object has a set of routines that define the object's allowable routines. Objects are composed out of other objects in the usual manner.

Unlike usual object-oriented programming, the characteristics of all data structures (e.g., sizes, density, and number of processors to be spread over) are specified both symbolically and numerically as illustrated in Table 4. The POLYNOMIAL class and its subclasses are show in Figure 1. Every POLYNOMIAL class has a numeric `value`, can evaluate itself with `eval`, can output itself (using `print`) in a suitable form for inputting to Mathematic (Wolfram, [5]), and can build compound POLYNOMIAL's us-

Name	Binding	Description
a	1	Primitive arithmetic routine
r	1	Single word memory read
w	1	Single word memory write
i	8	Network routine initiation
x	8	Network transfer time / word

Table 2: The constants used in time polynomials, representing the average times to perform the given routine. Also shown are the bindings to these constants used for the analysis.

```
add(vector:$ANY_VECTOR) : $ANY_VECTOR is
   i:INT := 0;
   until (i>=asize) loop
      self[i] := self[i] + vector[i];
      i       := i + 1;
   end;
   res := self;
end;
time_add : $POLYNOMIAL is
  res := length.mul(S::two.mul(S::r).add(S::w).add(S::a));
end;
```

Table 3: A simple symbolic timing routine. This table shows the Sather code for adding two `VECTOR`'s and its corresponding timing routine, `time-add`. Note that the `S` class contains all the constants and variables, and `length` is the size of the vector as a polynomial.

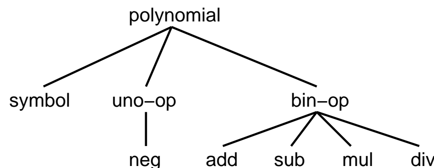

Figure 1: POLYNOMIAL's class hierarchy.

ing one of the routines named after the compound POLYNOMIAL classes (such as neg or add). SYMBOL is a special POLYNOMIAL that has a name and is required to have a value. All POLYNOMIAL's are constructed using SYMBOL's as the basic building blocks. UNO-OP's and BIN-OP's are one and two argument compound POLYNOMIAL's, respectively.

In order to build the time polynomials there are time routines for all the regular routines. For example the VECTOR object's add routine has a corresponding time-add routine. The time routines [1] return a polynomial giving the average time to complete the routine (as shown in Table 3). These time routines create polynomials using the symbolic characteristics of the objects (e.g., the size of the VECTOR). Polynomials are created compositionally just as objects are organized and routines are invoked. The polynomials created in this way are input to Mathematica to produce the analysis results reported in this paper.

The polynomials are created by hand counting the routines in a given routine. A few simple rules determine how this is accomplished:

1. loops incur zero overhead, including no overhead for updating any loop counters,
2. there is no overlap between memory access and arithmetic routines nor any overlap between communication and computation,
3. MATRIX and VECTOR access incurs no extra overhead,
4. a send or receive takes on average $i + nx$ time, where n is the number of words transferred, and
5. it is assumed that there exists a conditional move instruction which counts the same as two primitive arithmetic routines: one to set the condition and one to perform the conditional move.

The first results that can be produced from this analysis are the raw amounts of various routines and their ratios. This paper reports the ratio between memory and

[1] For the rest of this paper, unless otherwise stated, it is assumed that time is the same as average time.

```
class POLYNOMIAL is
   value:INT;
   ...
end;

class ANY_VECTOR is
   length:$POLYNOMIAL;
   ...
end;

class VECTOR is
   ANY_VECTOR;
   ARRAY{REAL};

   create(size:$POLYNOMIAL) : VECTOR is
      res        := new(size.eval);
      res.length := size;
   end;
   ...
end;
```

Table 4: How data-structures are created both symbolically and numerically. This table shows relavent excerpts from the `POLYNOMIAL`, `ANY-VECTOR`, and `VECTOR` classes. The `length` of a vector is defined to be a `$POLYNOMIAL` which contains both a symbolic component as well as a numeric attribute, `value`. The characteristics of all data structures (e.g., sizes, density, and number of processors to be spread over) are specified both symbolically and numerically. This example shows how a `VECTOR` is created. `Create` is called with a `POLYNOMIAL` specifying the symbolic and numeric size of the `VECTOR`. `Eval` is a `POLYNOMIAL` which computes the actual value of the `POLYNOMIAL` from its constituent components. After creation, `length` can be used for timing routines.

```
scaled_outer_product_add
      (scale:REAL; cv:$ANY_VECTOR; rv:$ANY_VECTOR)
         : $ANY_MATRIX is
   row := rv.scale_into(scale,row);
   res := outer_product_add(cv,rv);
end;
time_scaled_outer_product_add : $POLYNOMIAL is
   res := row.time_scale_into.add(time_outer_product_add);
end;
```

Table 5: An example of building the time routines up compositionally. This table shows the `MATRIX` routine for adding the result of a scaled outer-product of two `VECTOR`'s into itself. Also included is the timing routine, `time-scaled-outer-product-add`. The time to run this routine is equal to the time to scale the row `VECTOR` plus the time to compute and add in an outer-product to the `self` `MATRIX`.

arithmetic routines (M/A), the ratio between memory and network routines (M/N), and the ratio between computation ($M+A$) and network routines (C/N). To ground this analysis, this paper also reports the raw number of network transfers (X). This analysis gives a rough picture of the subsystem demands required by the various algorithms.

The next results concern the execution time of these algorithms. In order to produce these questions, there must exist (1) accurate time polynomials and (2) accurate estimates for the constants of Table 2. The main problem with the accuracy of the time polynomials produced in this paper are that they neglect the overlap of memory/arithmetic/network routines. In order to circumvent this problem, it is assumed that both memory and network access are much larger than arithmetic and that the overlap between routines can be well approximated by decreasing the average times for memory and network routines. Values for the constants of Table 2 are chosen using reasonable values from likely architectures.

Given the time polynomials, a number of questions can be asked.

- How sensitive are the algorithms to the values chosen for the constants?
- At what density is it faster to use a `SPARSE-MATRIX`?

The answers to further questions can elucidate the impact of parallel processing on the time to perform these algorithms:

- How do the algorithms scale with n and p?
- What is the size of the problem (n_2) for which it is faster to run the algorithm on two processors instead of one?

- What is the time optimal number of processors (p^*) for running this algorithm?
- What is the efficiency of the algorithm, or in other words, what percentage of the parallelism is leveraged:

$$\frac{T(n,1)}{p * T(n,p)},$$

where $T(n,1)$ is the time to run the algorithm on a $n \times n$ sized network on one processor and $T(n,p)$ is the time to run the algorithm on a $n \times n$ sized network on p processors?

3 Algorithms and Data-Structures

The focus of this paper is on the standard back-propagation algorithm (Rumelhart et. al, [4]). The overall design of the data-structures are loosely based on data-structures found in a connectionist simulator called CLONES (Kohn, [1]). For the purposes of this paper, a very simple network with one input `LAYER` and one output `LAYER` and a `CONNECT` between them is considered. Each `LAYER` contains a `VECTOR` of outputs, errors, and biases and a `LOOKUP-TABLE` for the transfer function and its derivative. A `CONNECT` contains a `MATRIX` of weights and pointers to the input and output `LAYER`'s.

Whereas `VECTOR` objects come in one main version, `MATRIX` objects come in four distinct versions. These versions correspond to the cross product of two orthogonal dimensions. The first dimension is whether the `MATRIX` is sparse or dense. Figure 2 shows the `SPARSE-MATRIX` representation. In this representation, there is a one-dimensional array, called `rows`, containing one-dimensional arrays of `SPARSE-ENTRY`'s. Each `SPARSE-ENTRY` contains a column index and the corresponding `MATRIX` value. The second dimension is whether the `MATRIX` includes both itself and its transpose, a `MATRIX` type called a `DOUBLE-MATRIX`. This type of `MATRIX` is useful for the `backward-propagate` routine used in back-propagation. A `DOUBLE-MATRIX` doubles the needed storage and doubles the computation necessary to update the `MATRIX` (since two updates are required), but decreases certain communication and other computation requirements.

It is informative to look at the `MATRIX` routines in more detail. The three needed routines are: `vector-multiply` (shown in Figure 3), `transposed-vector-multiply` (shown in Figure 4), and `outer-product-add` (shown in Figure 5). For the `sparse` case, the `vector-multiply` routine constructs the resulting `VECTOR` one element (column) at time by sequentially stepping down the rows of the `SPARSE-MATRIX`. For each row of the `SPARSE-MATRIX`, the corresponding column of the resulting `VECTOR` is successively incremented by amounts specified by multiplying the non-zero values of the `MATRIX` by the corresponding values (specified by the column attributes) of the `VECTOR`.

For the `DOUBLE-MATRIX` objects, the `transposed-vector-multiply` routine is the same as the `vector-multiply` routine except that it uses the transposed `MATRIX`

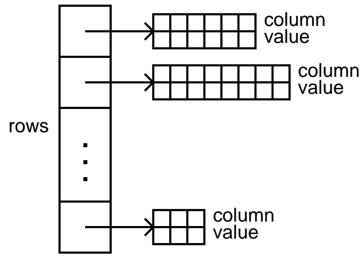

Figure 2: The representation for a `SPARSE-MATRIX`.

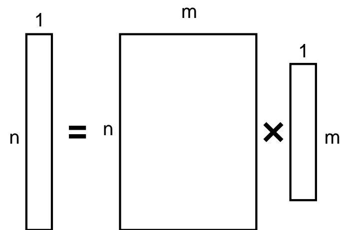

Figure 3: The `vector-multiply` `MATRIX` routine.

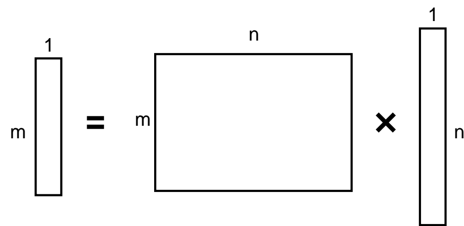

Figure 4: The `transposed-vector-multiply` `MATRIX` routine.

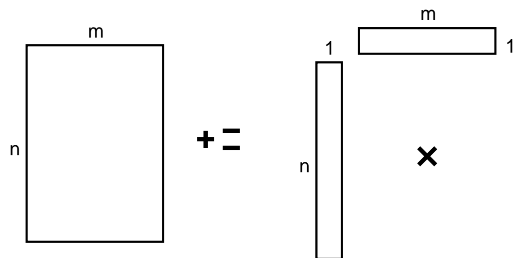

Figure 5: The `outer-product` `MATRIX` routine.

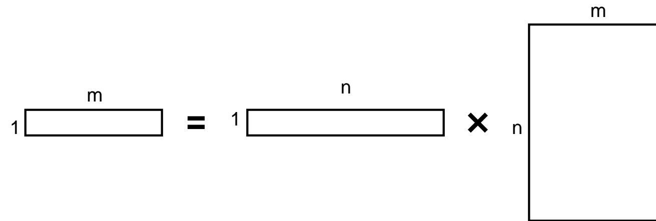

Figure 6: An alternative method for computing the `transposed-vector-multiply` `MATRIX` routine.

for the computation. For the non-`DOUBLE-MATRIX` objects, it is useful to view the `transposed-vector-multiply` routine as a `vector-multiply` routine as shown in Figure 6. For the `SPARSE-MATRIX` case, the `vector-multiply` routine constructs the resulting `VECTOR` by succesively adding in intermediate results to values of the resulting `VECTOR`. It proceeds by successively stepping down the `rows` of the `MATRIX`, each time multiplying the corresponding element of the `VECTOR` (corresponding to the row index of the `MATRIX`) by the non-zero elements of that row and adding these results to the elements of the resulting `VECTOR` specified by the column attributes of the `SPARSE-ENTRY`'s.

The `outer-product-add` routine is performed differently for each different `MATRIX` type. It is trivial for the `single/dense` case. For the sparse cases, the values of the two `VECTOR`'s are chosen by the indices of the non-zero elements of the `SPARSE-MATRIX`, where the row index corresponds to the index into the `rows` and the column index corresponds to the value of the `SPARSE-ENTRY`'s column attribute. Finally, for the double cases, both the `MATRIX` and its transpose must be updated.

Almost all the objects come in `local` and `distributed` variations. The local versions run on a single processor and require only local storage, whereas the distributed versions run in systolic mode on a ring of processors and require storage across the memories of these processors. In all cases the distributed versions of objects are built out of their local versions. `DISTRIBUTED-VECTOR`'s maintain both a local and global copy of its values on all the processors, where the length of each local vector is n/p. If n is not an exact multiple of p then slight variations can be employed. It is beyond the scope of this paper to discuss these issues. Distributed routines proceed along local parts of the `DISTRIBUTED-VECTOR` until all processors need the full set of values. At this time, the global values are updated using an explicit `replicate` routine which broadcasts each processor's local values to every other processor.

In contrast, `DISTRIBUTED-MATRIX`'s maintain *only* a local copy, where the local copy is n/p rows of the `MATRIX`. Figure 7 shows how an $n \times m$ `MATRIX` is divided up among p processors. All `MATRIX` routines assume that each processor contains up to date values for all the elements of the `DISTRIBUTED-VECTOR`. Given this as-

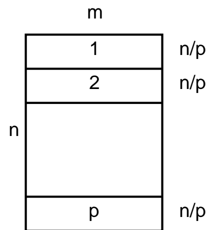

Figure 7: The rows of a $n \times m$ DISTRIBUTED-MATRIX are divided up among p processors.

sumption, the `vector-multiply` routine requires only the local `vector-multiply` routine to compute the local portion of the resulting DISTRIBUTED-VECTOR. The `outer-product-add` routine can also run without interprocessor communication.

For the most part, the DISTRIBUTED-MATRIX routines are independent of the underlying local representation of the local submatrices. The exception is the DISTRIBUTED-DOUBLE-MATRIX which requires that the local submatrices be DOUBLE-MATRIX's. This is because the `transposed-vector-multiply` routine can be performed without coordinating local computations, by using only a local `vector-multiply` routine. For `transposed-vector-multiply` routines for non-DOUBLE-MATRIX's, the intermediate results of local computations must be merged. This merge calculation can be performed efficiently on a ring of processors as discussed in Morgan et. al, [2].

Using these VECTOR and MATRIX objects as building blocks, there are eight possible combinations of networks as shown below:

$$\begin{array}{c}\texttt{local}\\ \texttt{distributed}\end{array} \times \begin{array}{c}\texttt{dense}\\ \texttt{sparse}\end{array} \times \begin{array}{c}\texttt{single}\\ \texttt{double}\end{array}.$$

The `local/distributed` distinction applies to all objects, while the `dense/sparse` and `single/double` distinctions apply only to MATRIX objects. The list of all concrete MATRIX and VECTOR classes is: VECTOR, DISTRIBUTED-VECTOR, MATRIX, SPARSE-MATRIX, DOUBLE-MATRIX, DOUBLE-SPARSE-MATRIX, DISTRIBUTED-MATRIX, DISTRIBUTED-SPARSE-MATRIX, DISTRIBUTED-DOUBLE-MATRIX, DISTRIBUTED-DOUBLE-SPARSE-MATRIX. Notice that the naming convention is that `local`, `dense`, `single` are the defaults.

4 Analysis

The analysis was performed on two major routines of the back-propagation algorithm: `forward-propagation` and `backward-propagate` corresponding to the retrieval and learning phases of back-propagation. The `run` routine is the combination of both of these routines. For the purposes of simplifying the analysis, it is assumed that $n = m$.

Routine	M/A	M/N	C/N	N
Forward	$\frac{11+2n+2p}{8+2n}, 1$	$1+\frac{11}{2p}+\frac{n}{p}, \frac{n}{p}$	$1+\frac{19}{2p}+\frac{2n}{p}, \frac{2n}{p}$	$2n$
Backward	$\frac{11+6n+10p}{9+4n+2p}, \frac{3}{2}$	$\frac{11+6n+10p}{4p}, \frac{3n}{2p}$	$3+\frac{5}{p}+\frac{5n}{2p}, \frac{5n}{2p}$	$4n$
Run	$\frac{22+8n+12p}{17+6n+2p}, \frac{4}{3}$	$\frac{11+4n+6p}{3p}, \frac{4n}{3p}$	$\frac{39+14n+14p}{6p}, \frac{7n}{3p}$	$6n$

Table 6: Rough and asymptotic ($n \gg p$ and $n \gg 1$) measurements for the `DISTRIBUTED-NETWORK`.

4.1 Rough Measurements

Tables 6, 7, 8, 9 give the raw and asymptotic amounts of various routines and their ratios. In these measurements for a given time polynomial M is the sum of the coefficients of r and w, A is the coefficient of a, N is the coefficient of x, and C is $M+A$. For the purposes of this analysis it is useful to assume that $n \gg p$ and $n \gg 1$. Given these assumptions, the asymptotic values for the various ratios can be ascertained. For the M/A measurements, the n (or dn for the `sparse` case) terms dominate and the effective result is the ratio of the coefficients of n (or dn) in the numerator and denominator. For example, the effective M/A for `DISTRIBUTED-NETWORK`'s `forward-propagate` routine is 1, for `backward-propagate` routine it is 3/2, and for `run` routine it is 4/3. For the M/N and C/N measurements, the n/p (or dn/p for the `sparse` case) terms dominate and determine the effective M/N and C/N, respectively.

A number of conclusions can be drawn from this analysis. First, the memory bandwidth requirements exceed the arithmetic requirements, that is, $M/A > 1$, for all but `DISTRIBUTED-NETWORK` and `DISTRIBUTED-DOUBLE-NETWORK`'s `forward-propagate` routines. Second, the `dense` routines are computation bound ($C/N > 1$). Third, the `sparse` routines are computation bound when n is sufficiently bigger than p. Assuming communication routines take eight times as long to execute as computation routines (and $d = \frac{\sqrt{n}}{n}$), then `DISTRIBUTED-SPARSE-NETWORK`'s `run` routine is communication bound when $p > \frac{43+18\sqrt{n}}{36}$. Figure 8 shows this curve for various values of n. Fourth, the routines for `DOUBLE-MATRIX` objects decrease the network traffic for `backward-propagate` by $\frac{3}{2}$ but increase computation by $\frac{10}{7}$.

4.2 Average Execution Times

Tables 10, 11, 12, 13, 14, 15, 16, 17 show the polynomials for the average time to execute the various `NETWORK` routines. These polynomials are used for the rest of the analyses. Values for the constants of Table 2 are needed for some of the remaining analysis. Values were chosen to match our expectations of potential hardware. The set used in this paper is: $r = w = a = 1$ and $i = x = 8$. Network costs are expected to be a factor of eight slower than memory access or arithmetic. Furthermore, the

Routine	M/A	M/N	C/N	N
Forward	$\frac{11+2n+2p}{8+2n}, 1$	$1+\frac{11}{2p}+\frac{n}{p}, \frac{n}{p}$	$1+\frac{19}{2p}+\frac{2n}{p}, \frac{2n}{p}$	$2n$
Backward	$\frac{15+10n+2p}{11+6n}, \frac{5}{3}$	$1+\frac{15}{2p}+\frac{5n}{p}, \frac{5n}{p}$	$\frac{13+8n+p}{p}, \frac{8n}{p}$	$2n$
Run	$\frac{26+12n+4p}{19+8n}, \frac{3}{2}$	$1+\frac{13}{2p}+\frac{3n}{p}, \frac{3n}{p}$	$1+\frac{45}{4p}+\frac{5n}{p}, \frac{5n}{p}$	$4n$

Table 7: Rough and asymptotic ($n \gg p$ and $n \gg 1$) measurements for the `DISTRIBUTED-DOUBLE-NETWORK`.

Routine	M/A	M/N	C/N	N
Forward	$\frac{12+3dn+2p}{8+2dn}, 1$	$1+\frac{6}{p}+\frac{3dn}{2p}, \frac{3dn}{2p}$	$1+\frac{10}{p}+\frac{5dn}{2p}, \frac{5dn}{2p}$	$2n$
Backward	$\frac{14+9dn+8p}{9+4dn+2p}, \frac{9}{4}$	$\frac{14+9dn+8p}{4p}, \frac{9dn}{4p}$	$\frac{23+13dn+10p}{4p}, \frac{13dn}{4p}$	$4n$
Run	$\frac{26+12dn+10p}{17+6dn+2p}, 2$	$\frac{13+6dn+5p}{3p}, \frac{2dn}{p}$	$2+\frac{43}{6p}+\frac{3dn}{p}, \frac{3dn}{p}$	$6n$

Table 8: Rough and asymptotic ($n \gg p$ and $n \gg 1$) measurements for the `DISTRIBUTED-SPARSE-NETWORK`.

Routine	M/A	M/N	C/N	N
Forward	$\frac{12+3dn+2p}{8+2dn}, \frac{3}{2}$	$1+\frac{6}{p}+\frac{3dn}{2p}, \frac{3dn}{2p}$	$1+\frac{10}{p}+\frac{5dn}{2p}, \frac{5dn}{2p}$	$2n$
Backward	$\frac{18+13dn+2p}{11+6dn}, \frac{13}{6}$	$1+\frac{9}{p}+\frac{13dn}{2p}, \frac{13dn}{2p}$	$\frac{29+19dn+2p}{2p}, \frac{19dn}{2p}$	$2n$
Run	$\frac{30+16dn+4p}{19+8dn}, 2$	$1+\frac{15}{2p}+\frac{4dn}{p}, \frac{4dn}{p}$	$1+\frac{49}{4p}+\frac{6dn}{p}, \frac{6dn}{p}$	$4n$

Table 9: Rough and asymptotic ($n \gg p$ and $n \gg 1$) measurements for the `DISTRIBUTED-SPARSE-DOUBLE-NETWORK`.

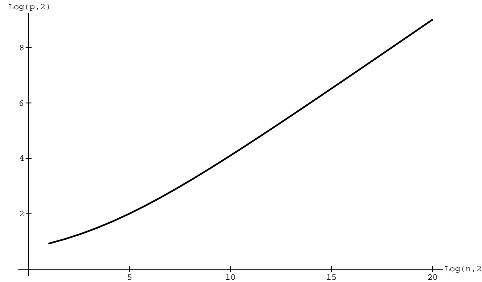

Figure 8: Maximum number of processors before the `SPARSE-MATRIX`'s `run` routine is communication bound.

Routine	Average Time
forward	$n^2(2a+2r)+n(8a+5r+4w)$
backward	$n^2(4a+5r+w)+n(10a+6r+5w)$
run	$n^2(6a+7r+w)+n(18a+11r+9w)$

Table 10: Average times for `NETWORK` routines.

extra time needed for memory access is expected to overlap arithmetic execution. Density was chosen to be equal $\frac{\sqrt{n}}{n}$, which means that a network with a million units has a fan-in of a thousand. Unless otherwise mentioned, average execution times for `DISTRIBUTED-NETWORK` and `DISTRIBUTED-SPARSE-NETWORK`'s `run` routines will be used for the calculations.

4.3 Density

The first interesting question to ask is at what density is it faster to use a `SPARSE-MATRIX`? This can be answered by solving for the density at which the time

Routine	Average Time
forward	$n^2(2a+2r)+n(8a+5r+4w)$
backward	$n^2(6a+8r+2w)+n(10a+6r+5w)$
run	$n^2(8a+10r+2w)+n(18a+11r+9w)$

Table 11: Average times for `DOUBLE-NETWORK` routines.

Routine	Average Time
forward	$n^2 d\,(2a + 3r) + n\,(8a + 6r + 4w)$
backward	$n^2 d\,(4a + 7r + 2w) + n\,(10a + 8r + 4w)$
run	$n^2 d\,(6a + 10r + 2w) + n\,(18a + 14r + 8w)$

Table 12: Average times for **SPARSE-NETWORK** routines.

Routine	Average Time
forward	$n^2 d\,(2a + 3r) + n\,(8a + 6r + 4w)$
backward	$n^2 d\,(6a + 11r + 2w) + n\,(10a + 9r + 5w)$
run	$n^2 d\,(8a + 14r + 2w) + n\,(18a + 15r + 9w)$

Table 13: Average times for **SPARSE-DOUBLE-NETWORK** routines.

Routine	Average Time
forward	$\frac{n^2}{p}\,(2a + 2r) + n\,(r + w + 2x) + \frac{n}{p}\,(8a + 6r + 5w) + p\,(2i + 2)$
backward	$\frac{n^2}{p}\,(4a + 5r + w) + n\,(2a + 5r + 5w + 4x) + \frac{n}{p}\,(9a + 6r + 5w) + p\,(4i + 4) - 2$
run	$\frac{n^2}{p}\,(6a + 7r + w) + n\,(2a + 6r + 6w + 6x) + \frac{n}{p}\,(17a + 12r + 10w) + p\,(6i + 6) - 2$

Table 14: Average times for **DISTRIBUTED-NETWORK** routines.

Routine	Average Time
forward	$\frac{n^2}{p}\,(2a + 2r) + n\,(r + w + 2x) + \frac{n}{p}\,(8a + 6r + 5w) + p\,(2i + 2)$
backward	$\frac{n^2}{p}\,(6a + 8r + 2w) + n\,(r + w + 2x) + \frac{n}{p}\,(11a + 8r + 7w) + p\,(2i + 2)$
run	$\frac{n^2}{p}\,(8a + 10r + 2w) + n\,(2r + 2w + 4x) + \frac{n}{p}\,(19a + 14r + 12w) + p\,(4i + 4)$

Table 15: Average times for **DISTRIBUTED-DOUBLE-NETWORK** routines.

Routine	Average Time
forward	$\frac{n^2 d}{p}\,(2a + 3r) + n\,(r + w + 2x) + \frac{n}{p}\,(8a + 7r + 5w) + p\,(2i + 2)$
backward	$\frac{n^2 d}{p}\,(4a + 7r + 2w) + n\,(2a + 4r + 4w + 4x) + \frac{n}{p}\,(9a + 9r + 5w) + p\,(4i + 4) - 2$
run	$\frac{n^2 d}{p}\,(6a + 10r + 2w) + n\,(2a + 5r + 5w + 6x) + \frac{n}{p}\,(17a + 16r + 10w) + p\,(6i + 6) - 2$

Table 16: Average times for **DISTRIBUTED-SPARSE-NETWORK** routines.

Routine	Average Time
forward	$\frac{n^2 d}{p}\,(2a + 3r) + n\,(r + w + 2x) + \frac{n}{p}\,(8a + 7r + 5w) + p\,(2i + 2)$
backward	$\frac{n^2 d}{p}\,(6a + 11r + 2w) + n\,(r + w + 2x) + \frac{n}{p}\,(11a + 11r + 7w) + p\,(2i + 2)$
run	$\frac{n^2 d}{p}\,(8a + 14r + 2w) + n\,(2r + 2w + 4x) + \frac{n}{p}\,(19a + 18r + 12w) + p\,(4i + 4)$

Table 17: Average times for **DISTRIBUTED-SPARSE-DOUBLE-NETWORK** routines.

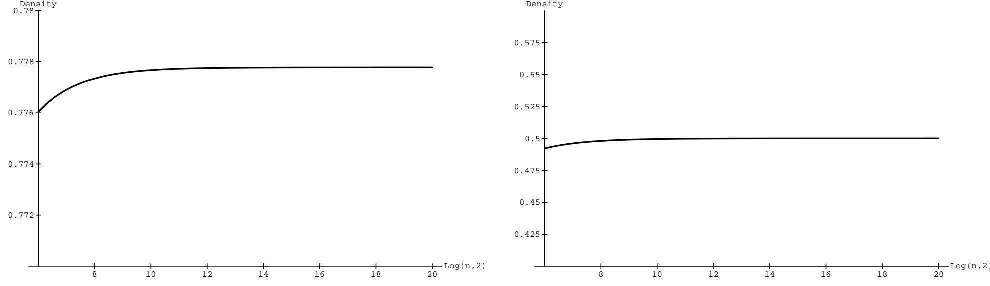

Figure 9: SPARSE-MATRIX density values at which it is better to use a SPARSE-MATRIX over a DENSE-MATRIX. The left graph shows the SPARSE-MATRIX density at which the time to execute the **run** routine takes the same amount of time for both the SPARSE-MATRIX and DENSE-MATRIX. The right graph shows the SPARSE-MATRIX density at which the memory usage is the same for both the SPARSE-MATRIX and DENSE-MATRIX.

taken to execute the DENSE-MATRIX and SPARSE-MATRIX **run** routines is equal. The symbolic form of the answer is

$$d = \frac{n^2(6a + 7r + w) + n(3r + w)}{n^2(6a + 10r + 2w)}. \tag{1}$$

The left graph of Figure 9 shows this numeric density value over various values of n. For the chosen constants, Equation 1 converges to 7/9 as n goes to ∞.

It is also interesting to compare the memory usage of the DENSE-MATRIX and SPARSE-MATRIX. The amount of memory consumed by a DENSE-MATRIX is n^2 words. Assuming pointers occupy the same amount of memory as data values, then a SPARSE-MATRIX consumes $n + 2n^2d$ words. The two representations have the same memory usage when $d = \frac{n-1}{2n}$. The right graph of Figure 9 shows this numeric density value over various values of n and shows that the density converges to $\frac{1}{2}$ as n goes to ∞.

For the CNS it is more likely that pointers will occupy twice the amount of memory as data values. This means that a SPARSE-MATRIX will consume $2n + 3n^2d$ words. The two representations would then have the same memory usage when $d = \frac{n-2}{3n}$, and the asymptotic density is $\frac{1}{3}$.

4.4 Scaling

How do the algorithms scale with n and p? Figure 10 shows the connections per cycle (CPC) over various values of n and discrete choices for p. To interpret the results as connections per second (CPS) merely multiply the CPC value by the cycles per second for a given processor. Beware that the y-axes of the two graphs are scaled differently. Some standard conclusions can be made from inspecting this figure. First, using too

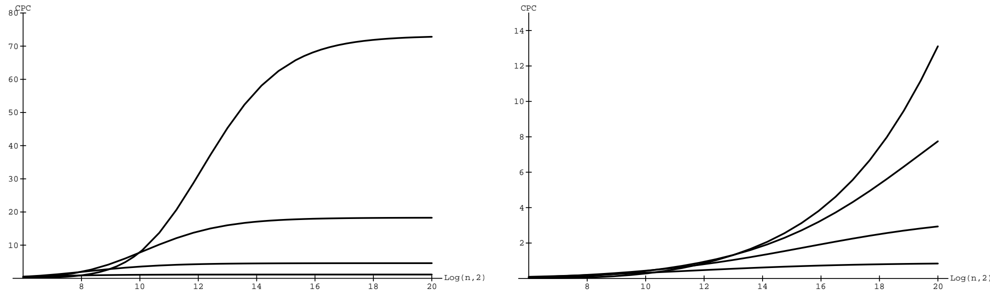

Figure 10: Connections per cycle over various values of n and discrete choices for p. The curves are from top to bottom $p = 1024$, $p = 256$, $p = 64$, and $p = 16$. The left graph is for the `dense` case and the right graph is for the `sparse` case ($d = \frac{\sqrt{n}}{n}$).

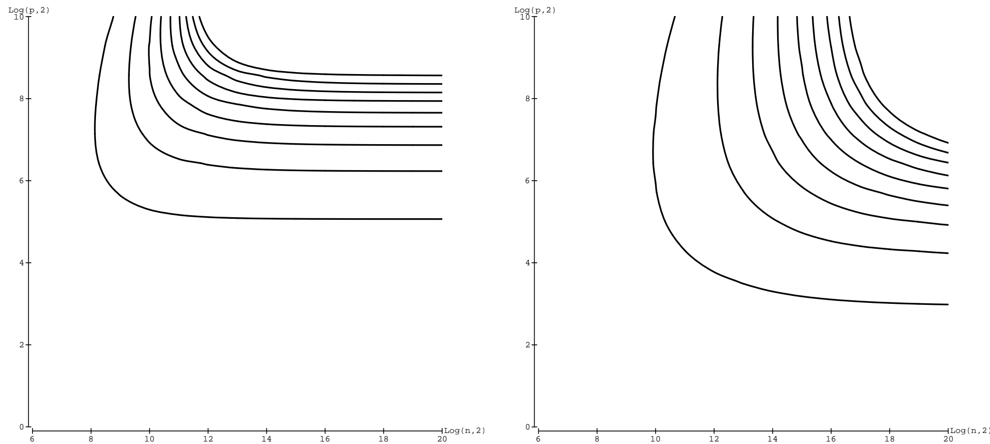

Figure 11: Level connections per cycle curves over n and p. The left contour plot is for the `dense` case and the right contour plot is for the `sparse` case ($d = \frac{\sqrt{n}}{n}$).

many processors on a problem of a given size is detrimental. This can be seen in the `dense` case of Figure 10 where the upper curve ($p = 1024$) dips below the next curve ($p = 256$). Second, the connections per cycle for a given number of processors levels off as the size of the problem is increased. This is also true of the `sparse` case but only as n grows even larger than shown in Figure 10. The reason for the differences between the two graphs of Figure 10 is that the `dense` case is computation bound and the `sparse` case is communication bound. Figure 11 is a contour graph showing equal connections per cycles over the range of n and p.

4.5 n_2

What is the size of the problem (n_2) for which it is faster to run the algorithm on two processors instead of one? The solution to this problem can be found by solving for what value of n makes $T(n,1) = T(n,2)$. The symbolic answer is for the dense case is

$$n_2 = \frac{-5 + \frac{39r+13w+24x}{6a+7r+w} + \sqrt{\frac{64(5+6i)}{6a+7r+w} + (5 + \frac{39r+13w+24x}{6a+7r+w})^2}}{4}. \quad (2)$$

Plugging in the values for the constants, $n_2 = 8.1$. The answer for the sparse case is much messier. The numeric answer is $n_2 = 25.7$.

4.6 p^*

A more general question is what is the time-optimal value of p for a given problem of size n. The solution to this problem can be found by differentiating the time polynomial with respect to p and then solving for p such that this derivative equals zero. The symbolic solution to this problem for the dense case is

$$p_d^* = \frac{\sqrt{n}\sqrt{17a + 12r + 10w + n(6a + 7r + w)}}{\sqrt{6}\sqrt{1+i}}, \quad (3)$$

and for the sparse case is

$$p_s^* = \frac{\sqrt{n}\sqrt{17a + 12r + 10w + \sqrt{n}(6a + 10r + 2w)}}{\sqrt{6}\sqrt{1+i}}. \quad (4)$$

Figure 12 shows p^* for various values of n. Beware that the y-axes of the two graphs are scaled differently. For the chosen constants and for a large majority of the useful range of n, p_d^* is approximately equal to $n/2$. Therefore, p_d^* is unreasonable for large n. In fact, p_d^* exceeds the expected number of processors ($2^{10} = 1024$) when $n > 2^{11} = 2048$. In contrast, p_s^* is much lower than p_d^*. For example, $p_s^* > 2^{10}$ only when $n > 2^{17}$.

4.7 Efficiency

What is the efficiency of the algorithm, or what percentage of the parallelism is leveraged:

$$\frac{T(n,1)}{p * T(n,p)},$$

where $T(n,1)$ is the time to run the algorithm on a $n \times n$ sized network on one processor and $T(n,p)$ is the time to run the algorithm on a $n \times n$ sized network on p processors? Figure 13 shows various curves of equal efficiency over n and p. The results show that the dense case is extremely efficient. In contrast, the results show

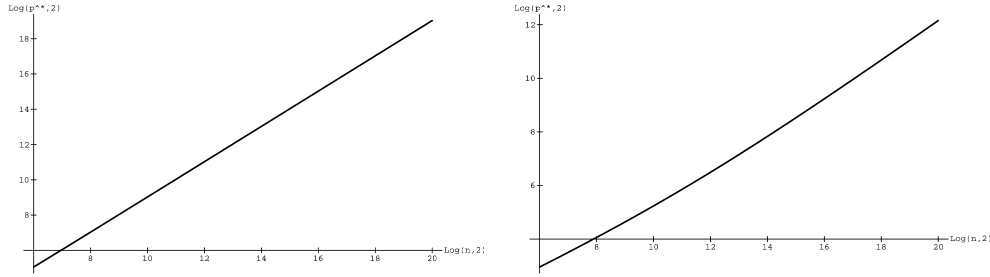

Figure 12: Optimal number of processors, p^* for various values of n. The left graph is for the `dense` case and the right graph is for the `sparse` case ($d = \frac{\sqrt{n}}{n}$).

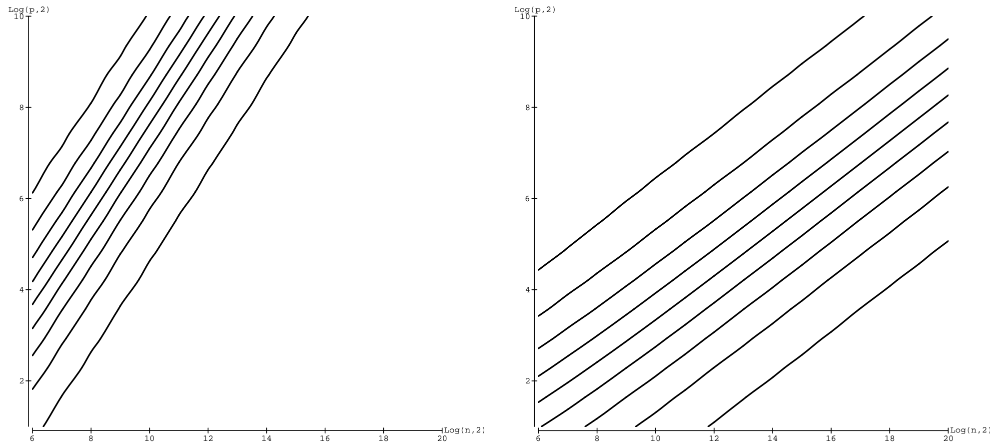

Figure 13: Level efficiency curves over n and p. The curves are at efficiency values of $0.1k$ for $k \in \{1, \ldots, 9\}$. The left contour plot is for the `dense` case and the right contour plot is for the `sparse` case ($d = \frac{\sqrt{n}}{n}$).

that for the `sparse` case in order to achieve better than 90% efficiency on a given number of processors, $n > 2^{10}p^2$, or equivalently, there must be at least $p2^{10}$ units per processor. (This result was obtained by fitting a line to the 90% curve.) For example, for the case with two processors, the network must have more than 4096 units (with 2148 units per processor), and for $p = 4$, n must be larger than 16384 (with 4096 units per processor), and for $p = 32$, n must be larger than $2^{20} = 1048576$ (with $2^{15} = 32768$ units per processor).

5 Future Research

The major direction of future research is in making this analysis more realistic. There are two main branches: hardware realism and application realism. In this paper, the

hardware is modeled at a very coarse level of granularity to permit some amount of hardware independence. Unfortunately, this glosses over hardware features that can greatly impact performance. A key example feature is vector processing support, where different memory access patterns now have different performance. For example, it is usually very efficient to access a row of a `MATRIX` organized in row-major format, while, unless there is hardware stride support, it is more expensive to access a column of that same `MATRIX` (as in `transposed-vector-multiply` for `SINGLE-MATRIX`'s). Furthermore, it is even more expensive still to access `SPARSE-MATRIX` elements.

Another hardware area in need of more detailed modeling is the hardware network. Even though the routines compute in systolic mode, they incur a penalty for both the `send` and `receive`. Also, there is no modeling of network traffic and distance between nodes. In the future, these aspects will play a more important role in modeling efforts.

Of equal importance is the need to more accurately model the actual neural network applications. This paper considers only very simple networks. Future research must consider more complicated modular architectures as well as a variety of retrieval and learning routines. One prime direction is the consideration of sparser networks, because (1) with a million units, the only realistic networks are sparse, and (2) sparse networks place a much higher demand on the underlying hardware network. The `SPARSE-NETWORK`'s considered in this paper are unrealistic for the majority of neural network applications. More realistic network architectures involve a large number of dense submatrices (clumps in the global connectivity matrix). One future direction is to consider networks composed of many `dense` subnetworks.

References

[1] Phil Kohn. Connectionist layered object-oriented network simulator (clones): User's manual. Technical Report TR-91-073, International Computer Science Institute, 1947 Center Street, Suite 600; Berkeley, California 94704-1105, 1992.

[2] Nelson Morgan, James Beck, Phil Kohn, Jeff Bilmes, Eric Allman, and Joachim Beer. The ring array processor: A multiprocessing peripheral for connectionist applications. *Journal of Parallel and Distributed Computing*, 14(3):248–259, March 1992.

[3] Stephen M. Omohundro. The sather language. Unpublished Manual, International Computer Science Institute, 1947 Center Street, Suite 600; Berkeley, California 94704-1105, June 1991.

[4] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations*. Bradford Books/MIT Press, Cambridge, MA, 1986.

[5] Stephen Wolfram. *Mathematica*. Addison-Wesley, 1988.