A User's Manual and Guide to an SDM Simulator on a Connection Machine

Jim Marshall
Lisa Meeden
Computer Science Department
Indiana University
Bloomington, Indiana

11 December 1989

Abstract

In this document we briefly describe the Sparse Distributed Memory model and a menu-based system which allows a user to gain hands-on experience with this model. The system is designed to aid beginners by providing stored exercises and examples. It also provides experienced users with options for configuring the memory and running their own learning experiments. The system is implemented in the *Lisp language on a Connection Machine.

1 Background

A sparse distributed memory (SDM) is a model of human long-term memory developed by Pennti Kanerva [3], which offers another approach to the problem of machine learning. The model is constructed from simple, neuron-like components which makes it ideal for implementation on a parallel computer.

1.1 The Basic Features of SDM

The SDM model is based on the properties of the space of n-bit binary strings, where n is large (in the range 100 to 10,000). Both the addresses to the memory and the data stored in the memory are n-bit binary strings. This implies an enormously large address space. As Kanerva points out:

> There is no way to construct a random-access memory that has, say 2^{1000} storage locations. Even 2^{100} locations would be too many to fit into the human brain, as 2^{100} molecules of water would more than fill it (the number of neurons in the nervous system is "only" about 2^{36}) ([3], p 53).

Therefore in the actual model, only a very sparse subset of all possible memory addresses have physical storage locations, termed *hard locations*, associated with them. These hard locations are chosen at random from the 2^n total possible memory addresses.

The *distance* between two memory addresses is defined as the Hamming distance between the addresses, or the number of bits in which the addresses differ. Approximate memory addressing is accomplished by associating a *response region* with each hard location address. A hard location is activated during a read or write operation whenever an address is used that falls within its response region, defined to be the set of addresses within a specified Hamming distance of the hard location address. Thus there is a many to many mapping of addresses to physical storage locations because each hard location responds to many addresses and each address activates many hard locations. Memory read and write operations are therefore distributed over many hard locations.

Writing a pattern to memory stores a copy of the pattern in each activated hard location. Data are not overwritten as in conventional computer memories; instead, the new data pattern is simply added to the existing set of data stored in the active hard location. Reading the memory at a particular address retrieves all of the data patterns stored in all of the hard locations activated by the read operation. These data patterns are combined into a single statistical representative of all of the patterns, which is then returned as the result of the read operation.

1.2 Writing and Reading Sequences

> Fundamental to the [SDM] model are the storage and retrieval of sequences, that is, memory items for events that occur one after the another in time. A person's life record is one long sequence that grows while the person lives. Acquiring skills, in particular, involves the learning of sequences ([3], p 3).

A sequence of patterns can be stored in the memory as a simple linked-list data structure, whereby any given pattern is stored at the address corresponding to the pattern preceding it in the sequence. An important concept that arises as a result of the properties of the memory is that of *convergent* and *divergent sequences.* Two sequences S_1 and S_2 are said to converge if reading S_1 with pattern p_1 and S_2 with pattern p_2 in general yields two patterns whose distance is smaller than the distance between p_1 and p_2. Otherwise the sequences are said to diverge.

Whether two sequences converge or diverge depends on the distance between the initial two patterns used to read the sequences. If the initial patterns are beyond a certain distance apart, called the *critical distance*, the resulting read sequences will in general diverge. The critical distance depends on several factors, such as the size of the stored bit patterns, the number of hard locations, and the size of a hard location response region. Note that since the behavior of the memory is statistical, the critical distance is only a probabilistic indication as to whether or not two sequences will eventually converge: beyond the critical distance, divergence becomes more likely than convergence. A related phenomenon is *chance convergence*, whereby two initially divergent sequences can by chance converge to a common point or cycle of random noise. However, chance convergence tends to occur only for small memories and non-uniform data sets.

1.3 Multiple-Fold SDM

The technique of storing the successor of a pattern at the address corresponding to the pattern itself makes it easy to implement a method of predicting patterns that depend on only one previous pattern. However, such a method breaks down when the next pattern in a sequence depends not only on the previous pattern, but also on patterns two or more steps back in the sequence. This problem is remedied by extending the model to support several *folds.* A fold is a component of the memory that stores sequence transitions of a particular length. The memory as described above is simply a one-fold memory, since successive patterns depend only on patterns one step back in a sequence. A k-fold memory allows prediction of successive patterns based on the previous k patterns of the sequence.

1.4 SDM-Based Learning

A learning system based on a sparse distributed memory can be designed as follows. Sensory input to the system is encoded as long patterns of bits, corresponding to the presence or absence of certain abstract "features" of the perceived environment. This sensory data feeds into a central region of the system called the *focus*, which serves as the interface to and from the memory. Intuitively, a sequence of patterns in the focus can be interpreted as the subjective experience of the system over time. At any given moment, the current pattern in the focus will trigger similar patterns stored in the memory, which in turn feed back into the focus. A sequence of patterns coming in from the environment will recall similar sequences which the system has previously experienced (if any), and this allows the system to predict the outcome of its current situation. The real outcome (as perceived by the senses) can then be compared to the predicted outcome; a good match will strengthen the sequence stored in memory, whereas a poor match will cause the information stored in memory to be revised to more accurately reflect the perceived outcome.

For a more detailed description of the sparse distributed memory model, the reader is referred to [3]. The remainder of this paper describes our implementation of a k-fold sparse distributed memory system, and the menu-based system built to accompany it.

2 Implementation

Our system is implemented on a Connection Machine (CM) parallel computer in the *Lisp language, an extension of Common Lisp providing parallel data structures and data operations. The CM consists of an array of up to (depending on the physical size of the machine) 65,536 parallel processors configurable by the user. The machine we used consisted of 32,768 physical processors, although we generally used no more than 8,192 processors for our memory simulations.

The user may specify how many physical and virtual processors are needed for the duration of the program. Once these processors have been allocated to the user, he or she may restrict the set of active processors during execution of the program to be various subsets of the total allocated processors.

Parallel operations on data are accomplished by storing the data in the active processor set (one data item per processor), and then performing a single (parallel) data operation on all of the active processors simultaneously.

2.1 Hard Locations

Out of the 2^n possible memory addresses, the system chooses a random subset equivalent in size to the number of virtual processors. So configuring the number of virtual processors is equivalent to choosing how many hard locations will be used during a run. Each hard location contains two arrays of length n: a bit-vector representing the hard location's address; and an array of integer counters, representing the set of all data stored in the hard location so far. The hard location address vectors and counter arrays are CM parallel variables, and essentially comprise the entire sparse distributed memory data structure.

2.2 Read and Write Operations

To perform a write operation with approximate addressing in parallel, the write address is loaded into each processor and simultaneously compared to each internal hard location address. If the Hamming distance between the two addresses is within a predefined *access radius*, that hard location is included in the set of hard locations active during the write operation. All active hard locations store a copy of the pattern being written to memory.

It is not possible to explicitly store each data pattern in a hard location. Instead a running average is maintained in an array of *bit-counters*, which store the relative numbers of 0-bits and 1-bits occurring in each bit position of the location, over the set of all data words stored in the location so far. A positive value $+v$ in counter j indicates that v more 1's have occurred in bit position j than 0's, over all the data stored in that location. A negative value indicates the amount of excess 0-bits occurring in bit position j, over all the data stored in that location. The routine which performs this operation in our system given in Figure reffig:write.

A short description of this code will reveal the flavor of parallel programming on the CM and our implementation of the model. There are two basic types of variables in *Lisp: scalers and parallel variables (called *pvars*). Functions whose names end in "!!" generally operate on and return pvars. For instance the function *or!!* compares two pvars of boolean values and returns a pvar of boolean values, and the function *!!* converts a scaler into a pvar. The **when* function reduces the set of active processors to only those that contain true for the boolean pvar which is its single argument.

Our function *sdm-write*, takes as parameters an address at which to perform the write (an n-bit vector), the data to write (another n-bit vector),

```
(defun sdm-write (address data fold)
  (if (< fold k)
      (*when (activate-fold!! fold)
        (*when (select!! address)
          (*map #'(lambda (c b)
            (*if (or!! (and!!
                          (=!! b (!! 0))
                          (>!! c (!! lower-bound)))
                       (and!!
                          (=!! b (!! 1))
                          (<!! c (!! upper-bound))))
                 (*set c (+!! c (if!! (=!! b (!! 0))
                                      (!! -1)
                                      (!! +1)))))))
                counters
                (!! data))))
      (say "Invalid fold number!")))
```

Figure 1: *Lisp Code for the SDM Write Operation

and the memory fold to write in (an integer from 0 to $k-1$, where k is the global variable for the total number of folds). The first call to **when* reduces the set of active virtual processors to only those in the appropriate fold. The second call to **when* reduces this new set of active virtual processors to only those within the access radius of the given address. Finally, **map* is called over the arrays of bit counters in the virtual processors which still remain active, and the pvar of the data vector. It increments or decrements the counters as appropriate. The **if* ensures that the counter values do not exceed their designated bounds.

To perform a parallel read operation, a set of active hard locations sufficiently similar to the read address is computed in a way analogous to the write operation. To compute a representative pattern from the data stored in all of the active hard locations, the bit-counters of all active locations are summed, bit-wise, to produce an array containing, for each bit position, the total excess 0-bits or 1-bits over all of the data from all of the active locations. These values are then thresholded at zero, to produce the final representative string. A positive bit-sum value corresponds to a 1-bit, a negative value corresponds to a 0-bit, and a zero value is randomly chosen to be either a 0-bit or a 1-bit. The variance of the bit-sums (i.e. the average of the magnitudes of their absolute values) gives an indication of the relative strength of the data being read, where large variances correspond to stored data and small variances correspond to random noise. The variance values provide an internal check on whether or not stored data is being read.

Since the read and write operations are performed in parallel across all hard locations, their execution time is essentially independent of the number of hard locations. (Although some increase in execution time will be noted when a large number of virtual processors are used.) Therefore, the model scales up well.

2.3 The Focus

As mentioned previously, the center of the learning system is the focus. It provides the interface between the environment, represented as an input file containing patterns, and the other major component of the system, a k-fold sparse distributed memory, essentially just a collection of one-fold memories. The (k-fold) focus is implemented as an array of k bit-vectors, each of length n. Each fold associates patterns of a sequence with patterns encountered several steps back in the sequence, the number of previous steps depends on the fold number. A k-fold memory allows predictions to be made based on all the information made available during the last k steps of the sequence.

At any particular time t, the k-fold memory is used to predict the outcome of the current situation, that is, the expected environment pattern at time $t+1$. Each fold of the memory is addressed with the corresponding fold of the focus. The k resulting patterns are then averaged into a final pattern (using the thresholding technique described for reading above), which becomes the *prediction* of the memory at time $t+1$. The prediction is then combined with the input pattern read from the environment (called the *perception*), and the resulting pattern (called the *composite*) is stored back into each memory fold at each respective fold address. Finally, the composite replaces the most recent pattern of the focus (stored in fold 0); the remaining focus patterns are shifted over one position; and the oldest focus pattern (stored in fold $k-1$) is discarded. There are a variety of possible methods to produce the composite pattern from the prediction and the perception:

1. weight the prediction according to a confidence measure (such as the variance of the read data);
2. weight both the prediction and the perception equally;
3. have a fixed bias towards the perception (such as 75%);
4. ignore the prediction completely.

The current default in our system is to ignore the prediction and "correct" the errant "belief" by writing the actual perception back to the memory.

2.4 The User-Interface

Our menu-driven simulator system is hierarchically structured. Each menu has a unique title to help orient the user in the hierarchy. Each menu is defined as a separate *Lisp function, where the function name indicates the path taken to invoke the menu. For example, the top level menu (called from the welcome menu) is named *menu0*. The responses to *menu0* which lead to further menus are *a* and *b*. These responses invoke the functions *menu0a* and *menu0b* respectively. All of the top level menu functions take no arguments, so an experienced user may skip the welcome screen and other top level menus and jump directly to a particular screen (assuming he or she already knows the appropriate path). As each menu appears on the screen it is separated from the previous menu or previous computation by a line of

```
-------------------------------------------------------------
Top

A. Execute stored exercises
B. Set up and execute your own task
C. Quit

Enter your selection>
-------------------------------------------------------------
```

Figure 2: Menu for Top Level

dashes. Each menu also offers, as its last two choices, the option to return to the previous menu or to quit.

There are essentially two major paths in the menu system: one for beginning users and one for experienced users. This is evident from the top menu shown in Figure 2.

3 Running the System

Our system is stored on the Connection Machine that can be reached by the following address: *cmns-sun.think.com* The relevant files can be found at: */cmns/jm/sdm.lisp* and */cmns/jm/menus.lisp*.

Once you have logged into the CM front-end, type "starlisp" at the Unix prompt. At the starlisp prompt, type "(in-package '*LISP)". To actually attach to the CM, type "(*cold-boot)". If this command produces an error message, then there are no processors available. This command assigns 8192 physical/virtual processors to your session. To configure the CM for more than one virtual processor per physical processor use the following syntax:

(*cold-boot :initial-dimensions '(*virtual-proc-num*))

The *virtual-proc-num* must be an integer power of two that specifies the total number of virtual processors (i.e. hard-locations) for this session. Then you must load both of the lisp files by typing "(load *filename*)" for each. To enter the menu-based system, simply type "(welcome)".

If you want to keep a record of your session with the SDM Simulator, you must start a Unix script before entering the starlisp environment.

```
....@@@@@@@@@...    .@@@@@@@@@@@....    @@@@@......@@@@@
..@@@@.....@@@..    .@@@.......@@@..    @@@@@@....@@@@@@
..@@@@@@........    .@@@........@@@.    @@@.@@@..@@@.@@@
....@@@@@@@@....    .@@@........@@@.    @@@..@@@@@@..@@@
........@@@@@@..    .@@@........@@@.    @@@...@@@@...@@@
..@@@.....@@@@..    .@@@.......@@@..    @@@....@@....@@@
...@@@@@@@@@....    .@@@@@@@@@@@....    @@@..........@@@
```

Figure 3: Example memory address/data patterns

4 Overview of Exercises

4.1 Memory Patterns

In order to allow easy visualization of the operation of the memory during the exercises, we chose to display data patterns as 7×16 grids of characters, with '@' standing for a 1-bit and '.' standing for a 0-bit. Thus, our patterns were 112 bits long (where row one contains bits 1-16, row two contains bits 17-32, etc.). Each pattern was chosen to resemble a visual depiction of a letter of the alphabet, as shown in Figure 3. Note that the addresses themselves are not meant to correspond to actual representations of visual perception data, but were merely chosen as an aid to observing the behavior of the memory. All 26 letters of the alphabet were represented in this way.

4.2 Exercise 1: Generalization

To test the SDM's ability to generalize, we write fuzzy versions of a pattern at the address described by that same fuzzy pattern. We then read from the memory with different fuzzy versions of the same pattern and compare the result to the true pattern. The closer (in Hamming distance) the result is to the true pattern, the better the generalization. The fuzziness is measured by the amount of noise which is introduced into the original pattern. For bit strings of size 126, a pattern containing 10% noise will have 13 of its bits randomly chosen and switched from their original values.

In our experiments with the SDM Simulator, we have found that the quality of the generalization improves when the number of noisy patterns written increases and the amount of noise read with and written with de-

```
--------------------------------------------------------------
Exercises

A. Generalizations of 'letter' patterns
B. Predictions of 'letter' sequences
C. Go back to Top menu
D. Quit

Enter your selection> a
--------------------------------------------------------------
```

Figure 4: Menu for Exercises

creases. All of these tendencies seem intuitively correct, but are also explained by the properties of the model. As the number of patterns written increases, the chances that the written patterns will overlap increases and thus improves the generalization. Similarly as the amount of noise in a read or write operation decreases, the closer to the center of the developing generalization the operation will occur. And thus improve the generalization.

To explore these properties of the memory yourself, select the "Execute stored exercises" option from the top menu. When the "Exercises" menu appears (see Figure 4), select option *a*.

After the SDM has been configured for the exercise, you will be asked to choose a letter on which to perform the generalization. Choose the letter *z*. Instruct the system to write 5 versions of *z* with 20% each. After the system has completed writing the 5 patterns, you will be presented with the "Generalizations options" menu, shown in Figure 5. Select option *a*.

You will be prompted to select the amount of noise with which to read from memory. Enter 0 (for no noise). Although only z-patterns of 20% noise were written, the memory returns a pattern only 5 bits away from a true z-pattern. Continue to read from the memory and enter 20 for the amount of noise of the next read. This time the memory returns a pattern 14 bits away from a true z-pattern. As the noise we read with increased, the quality of the generalization decreased.

When you are done exploring the SDM's generalization capabilities return to the "Exercises" menu.

```
----------------------------------------------------------------
Generalization options

A. Read from memory
B. Write more noisy patterns of the same letter
C. Choose a different letter
D. Re-initialize the memory
E. Go back to Exercises menu
F. Quit

Enter your selection> a
----------------------------------------------------------------
```

Figure 5: Menu of Generalization Options

4.3 Exercise 2: Simple Prediction

At the "Exercises" menu select option *b*: "Predictions of 'letter' sequences". The resulting menu is shown in Figure 6. Select option *a*.

The memory will be reconfigured and the alphabetic sequence A through M will be stored into the memory. This write operation takes a little bit of time so don't worry if the system is unresponsive for a few minutes. Recall that to store this sequence, we write the B-pattern at the address of the A-pattern, and the C-pattern at the address of the B-pattern, etc. By storing the bit strings in this manner, we have built up a pointer chain from the A-pattern up through the M-pattern.

After the system completes the writing of the sequence, you will be asked to select a pattern and the amount of noise from which to begin a memory read. Select a j-pattern with 20% noise. After a single read the system converges on the stored sequence, because a true k-pattern is returned. Continue to read this sequence for several steps. When reading with the n-pattern the system returns the pattern shown in Figure 7.

This pattern is not one of our letter patterns, but appears to be a conglomerate of several of the them. This is an example of one of the problems which can occur when using a relatively small memory ($n = 112$) where the stored patterns are quite close in Hamming distance. Some of our letter patterns are as close as 24 bits. Continue reading with this pattern and ob-

```
--------------------------------------------------------------
Predictions

A. Simple predictions of next letter in alphabet
B. Predictions of next letter in similar words
C. Go back to Exercises menu
D. Quit

Enter your selection> a
--------------------------------------------------------------
```

Figure 6: Menu for Predictions

```
.@@@@@@@@@@@@...
.@@@@...........
.@@@............
.@@@@.@@@@......
.@@@............
.@@@........@@@.
.@@@@@@@@@@@@@@.
```

Figure 7: Result of read with N-pattern in Simple Prediction

serve the results. It appears that a closed cycle ensues which consists only of these strange letter conglomerates.

Another problem which you might encounter when exploring the memory's predictive capabilities is chance convergence. If you read the memory with a very noisy version of one of the letters (try $40-50\%$) you may see an example of this.

5 Conclusion

We have only described a small subset of all of the features of our Simulator system. We also plan to extend and modify the system as our needs evolve. Still in its earliest version, this menu-based system has proved very useful to us in exploring the SDM model, and we hope that it will prove as useful to others interested in SDMs.

References

[1] W. Daniel Hillis. *The Connection Machine.* ACM Distinguished Dissertation. MIT Press, Cambridge, MA, 1985.

[2] W. Daniel Hillis. The connection machine. *Scientific American*, pages 108–115, August 1987.

[3] Pentti Kanerva. *Sparse Distributed Memory.* MIT Press, Cambridge, MA, 1988.