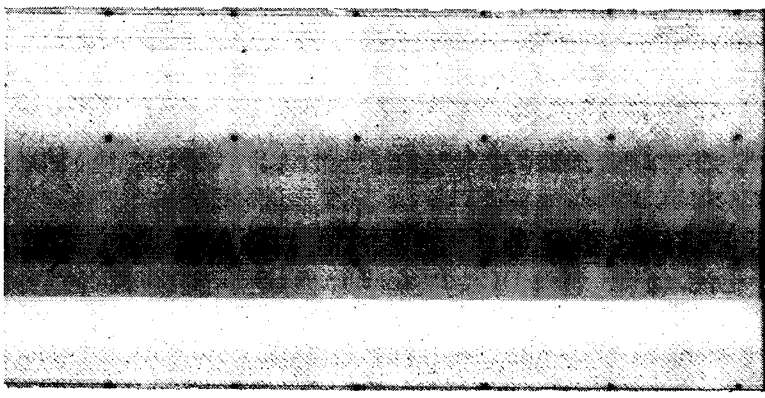
AMES GRANT / IN/ 61 - CR

85723

17 P.



(NASA-CR-180988) TWO DEMONSTRATORS AND A SIMULATOR FOR A SPARSE, DISTRIBUTED MEMORY (NASA) 17 p Avail: NTIS HC A02/MF A01 CSCL 09B

N87-26567

Unclas G3/61 0085723

RIACS

Research Institute for Advanced Computer Science

Two Demonstrators and a Simulator for a Sparse, Distributed Memory

Robert L. Brown

June 1987

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 87.17

Two Demonstrators and a Simulator for a Sparse, Distributed Memory

Robert L. Brown

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-87.17
June, 1987

This paper describes two programs demonstrating different aspects of Kanerva's Sparse, Distributed Memory (SDM). These programs run on Sun 3 workstations, one using color, and have straightfoward graphically oriented user interfaces and graphical output. Presented herein are descriptions of the programs, how to use them, and what they show. Additionally, this paper descibes the software simulator behind each program. The author assumes the reader has a working knowledge of the SDM.

Work reported herein was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

Two Demonstrators and a Simulator for a Sparse, Distributed Memory

Robert L. Brown
Research Institute for Advanced Computer Science
rlb@riacs.edu

1. Introduction

The operation of the Sparse, Distributed Memory (SDM), described by Kanerva [Kanerva86] and Denning [Denning88], is conceptually simple once understood, but reaching that point of understanding requires a moderate amount of effort. To solve this problem, it was decided to develop a set of graphical demonstrators for the SDM, each being able to show its operation at a particular level of abstraction or show a particular feature. To this end, two such demonstrators were developed at RIACS by the author.

The first program, *sdmview*, works at a low level of abstraction, demonstrating at a nearly hardware level how data are stored in and retrieved from the memory. The second program, *sdmdemo*, works at a higher abstract level and shows a simple application for SDM, specifically, saving and retrieving simple patterns. A third program was desired, also, to show at a very high level of abstraction how items stored in the memory "flood" a region of a high-dimensional hypercube, but the problem of displaying, say, 1000 dimensions on a two-dimensional CRT screen has stalled this project.

Each program was developed for a Sun 3/160C workstation [Sun86a] running Sun OS (UNIX) version 3.2. Only one program, *sdmview*, makes use of the color capabilities of the model 3/160C. The programs make extensive use of Sun's SunView user interface development package [Sun86b] and thus incorporate control panels, display windows, and a mouse-oriented interface.

The structure of the two programs is very similar, comprising three main components: user interface, graphical backend, and an SDM simulator. The operation of each program is described by describing its interface, and the simulator is discussed separately.

2. Simulator Overview

Behind each of the demonstrators is a minimum-function SDM simulator. Each

SDM operation requested by the user is turned into one or more requests on the simulator. The SDM is divided into six major pieces for the purpose of simulation. These are as follows:

Input address register
Address bank
Select array
Input datum register
Datum bank
Output datum registers

As in Kanerva's descriptions of SDM, the number of dimensions is called **N** and the number of storage locations is **M**. Hence, the address bank is **N** bits wide and holds **M** randomly generated addresses. The datum bank is **N** counters wide and **M** locations long. The input address register holds the address being used in a select operation. The input datum register holds a value to be written. There are two output datum registers, one unthresholded and one thresholded. The select array contains one bit for each of the **M** locations such that when a read or write is performed, only those locations with enabled select bits are involved.

Using this model for SDM, a write operation involves placing a value in the input address register, performing a select, then placing a value in the input datum register, and then adding that value into all selected locations. A read operation involves loading an address into the input address register, performing a select, then adding together all selected locations and placing the result in the raw output datum register. Optionally, a threshold may be applied to the result in order to generate a thresholded result in the thresholded output datum register.

A section on the structure and operation of the simulator appears later in this paper. The cursory description here is necessary in order to understand the demonstrators.

3. Sdmview

The first of the two programs is entitled *sdmview*. Its purpose is to give the user an insight into the inner workings of the SDM by showing an abstract digital storage array representing an implementation of the memory. Figure 1 shows the layout presented to the user on a monochrome workstation. On a color screen, colors are used instead of the dithering patterns across the bottom. In the bottom window, the six parts of the SDM are presented. The row in the upper left is the input address register, the two-dimensional array on the left side is the address bank, the row in the upper right is the input datum register, the array on the right is the datum bank, and the two rows on the lower right are the raw and thresholded output datum registers respectively. The grid along the bottom of the display shows the patterns (monochrome screen) or colors (color screen) that are used to represent the counter values in the datum-bank part of the display. The colors range from bright blue on the left darkening towards the center, denoting negative values, then dark red

brightening towards the right end, denoting positive values.

The display reflects the values of the simulated registers and memory inside the simulator. Each time *sdmview* calls on a simulator function, it also calls a display function that makes the image on the screen consistent with the memory. Looking at it this way, *sdmview* can be thought of as an exerciser for the simulator, and the display merely as a visualization of the internal data structures in the simulator.

3.1. Overview

The user operates *sdmview* through the control panel at the top of the window frame. The center one-line window is for status and error messages from *sdmview*, and the bottom subwindow shows the state of the simulated memory.

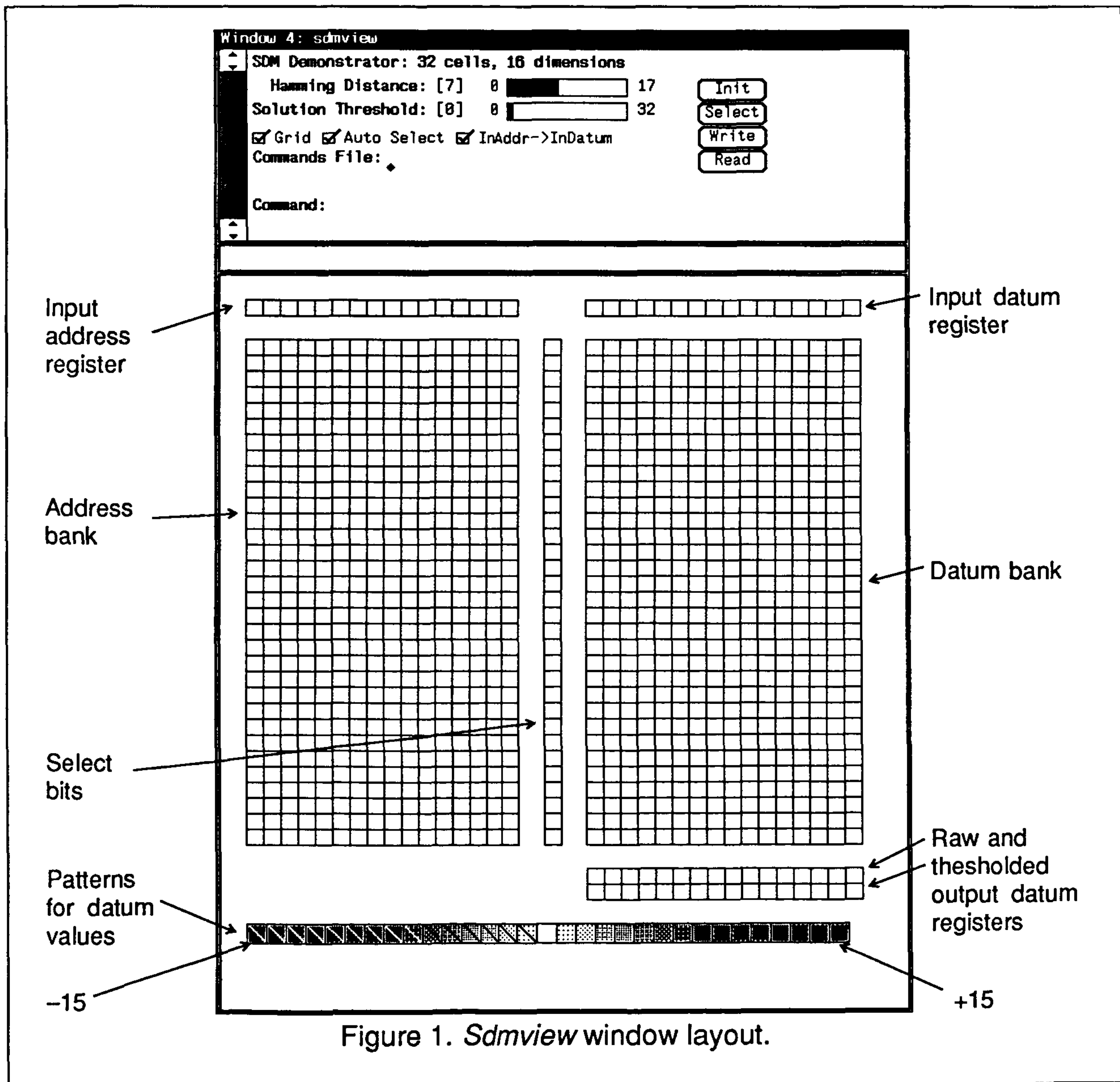


Figure 1. *Sdmview* window layout.

The user controls the operation of the program through commands and screen buttons in the control panel. Anything that can be done by a button can also be invoked using a typed command. This feature allows previously prepared demonstrations to be written as a file of commands and then fed into *sdmview* by clicking on a screen button.

The basic usage scenario is as follows:

```
set InAddr->InDatum
set Auto Select
intialize the memory
do several times:
    load input address register
    write
do several times:
    load input address register
    read
```

3.2. Operation

If *sdmview* is started with no arguments, it creates a memory with 32 locations (**M**) and 16 dimensions (**N**). This can be changed by passing two arguments, the first for **M** and the second for **N** (memory trick: alphabetical order). Though there is no limit to the size of the memory that can be simulated (except for the UNIX maximum process size limit), the display cannot show memories of greater than **M**=256 and **N**=128 because the cells become too small on the screen.

Initially, the user is presented with an empty memory; not even the random addresses are assigned. The first thing to do is to set the program parameters by adjusting the sliders and checking boxes. Sliders can be adjusted by pointing to the black bar inside the slider box, depressing the left mouse button, sliding the bar right or left, and releasing the button. The current value in the square brackets changes as you move the slider. The option boxes can be checked and unchecked by pointing to them and clicking the left mouse button. The meaning of each item in the control panel is as follows:

Hamming Distance - sets the maximum Hamming radius used in select operations.

Solution Threshold - sets the threshold used when converting a raw result of a read operation to a thresholded, or binary result.

Grid - toggles the grid in the display window on and off. For large memories where the square cell size is small, it is easier to visualize when the grid is turned off.

Auto Select - if checked, causes a select operation to happen whenever an address is loaded into the input address register.

InAddr->InDatum - if checked, causes values loaded into the input address register to be automatically copied into the input datum register.

Once the operational parameters are set up, the user can start invoking operations on the memory. The very first should be to load the random addresses into the address bank with the **init** command. This can be shortcut by clicking (left mouse button) on the button labelled **Init**. From that point on, commands can be typed at the **Command** prompt, and each time **Return** is struck, the command is performed. Note that once a command is completed, it is not erased from the screen. The user must type control-U to erase the command or control-W to erase just the last word (this is useful if typing **address** commands). Some commands can take arguments formed like UNIX shell command arguments. The commands are documented in brief below.

init - loads random addresses into the address bank and clears the datum bank. Normally this is only done once and is not done automatically at startup so that the user has time to explain why it must be done. The **Init** screen button performs the same function.

address - loads the address provided by the user as the first argument into the input address register. Address values may be specified in either binary (ASCII ones and zeros) or hexadecimal if prefixed with "0x ." If the **Inaddr->InDatum** box is checked, the value is also loaded into the input datum register. If **Auto Select** is checked, a select operation takes place. A sample command is "address 0xfe00."

select - performs a select operation, setting the select bits for all SDM locations within the specified (in the slider) Hamming radius. The **Select** button in the control panel performs the same function.

datum - loads the value given as the first argument into the input datum register. The rules for parsing datum values are the same as for addresses. This command has an alias **data**.

write - causes the value in the input datum register to be written into all selected locations. An alias for this command is **store.** The **Write** button in the control panel performs the same function.

read - causes a read operation using the address in the input address register. When invoked as a command, reads are performed, copying the result from the output datum register to the input address register, until either a cycle is detected or the read converges. If this operation is invoked by clicking the **Read** screen button, the user can control the read repetitions by clicking on the **Step** button that pops up next to the **Read** button. Each time the **Step** button is pressed, the contents of the thresholded output datum register are copied to the input address register and a read operation is re-initiated.

stop - stops the reading of commands and displays its first argument in the message window. This only makes sense if the input is a file and *sdmview* is being run in continuous mode (see below). Because only the first argument is displayed, a

message containing white space must be enclosed in quotation marks.

hamming - sets the Hamming distance slider to the value given as the first argument. Causes a select as a side effect.

message - displays its first argument in the message window or clears the window if no argument is given.

set select - sets the **Auto Select** feature.

set load - sets the **InAddr->Datum** option.

set grid - turns on the grid.

unset - can be followed by any of the arguments to set described above and causes that feature or option to be turned off.

Commands may come from either the **Command** prompt in the control window or from a file. If the user types a file name at the **Commands File** prompt (after selecting that prompt by clicking the left mouse button on it) and hits **Return**, *sdmview* opens that file and pops up a cycle labelled **Read Mode** and a button labelled **Go**. By default, the read mode is single-stepping, and each time the user clicks the **Go** button, another command is read and invoked. If the user changes the read mode to **continuous**, then clicking the **Go** button causes commands to be read from the file until either an end of file is reached or a **stop** command is read.

A typical commands file might contain the following commands:

```
init
hamming 6
set select
set load
address 0xff00
write
address 0x0123
write
address 0xdead
write
stop "Change to Single Step and step through the read operations"
address 0xff00
read
address 0x4321
read
```

This sample would load the memory with three values and then instruct the user to switch the operation of the program to single-step mode and manually step through the read operations. Figure 2 shows the state of the *sdmview* window after completing the first ot the two read operations in the sample.

ORIGINAL PAGE IS
OF POOR QUALITY

4. Sdmdemo

The second demonstrator is entitled *sdmdemo*. Its purpose is to show the user a simple application of the SDM by storing an retrieving small images. Figure 3 shows the layout as presented to the user. In that figure, three of the large subwindows are labelled with their names. Of the remaining subwindows, the upper leftmost is the main control panel, below it is the messages window, and in the upper right is the pad control panel.

4.1. Overview

The basic operational scenario for *sdmdemo* is very much like that for *sdmview*.

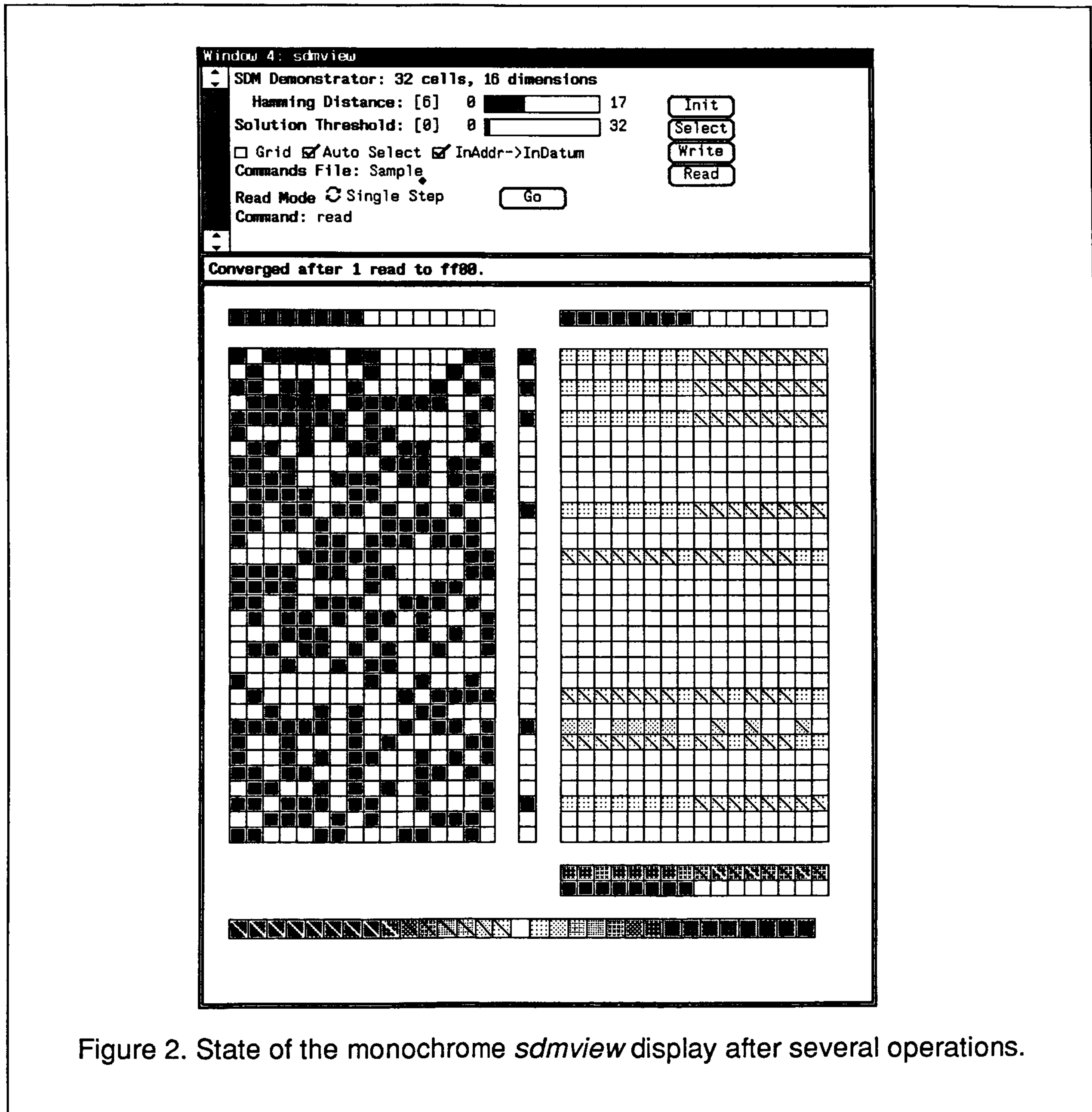


Figure 2. State of the monochrome *sdmview* display after several operations.

When the program starts, it creates a SDM with 2048 locations of 256 bits each. The number of locations can be changed by providing a single numeric argument when the program is invoked. Throughout the execution of the program, the 256-bit locations are represented as 16 by 16 bit squares with the first bit in the upper left and the last in the lower right. This representation allows the user to imagine each SDM cell as a 16-by-16 bit image.

4.2. Operation

Unlike *sdmview*, *sdmdemo* automatically assigns random addresses to the SDM. Hence, once it starts running, the user need only store and retrieve data. All operations are initiated from one of the two control panels; there is no command mode. Read and write operations use the pad as the source of the value for the input address and datum registers. Hence, the cycle of usage begins by creating a value there and then performing one of the memory operations.

The pad window is where the user enters images to store into the memory or to use as keys for reading. Initially the pad is blank, denoting all zeros. The user can toggle bits by clicking on the cells in the pad. Images can be predefined and stored in files by using Sun's *iconedit* program in cursor mode. Additionally, images can be loaded

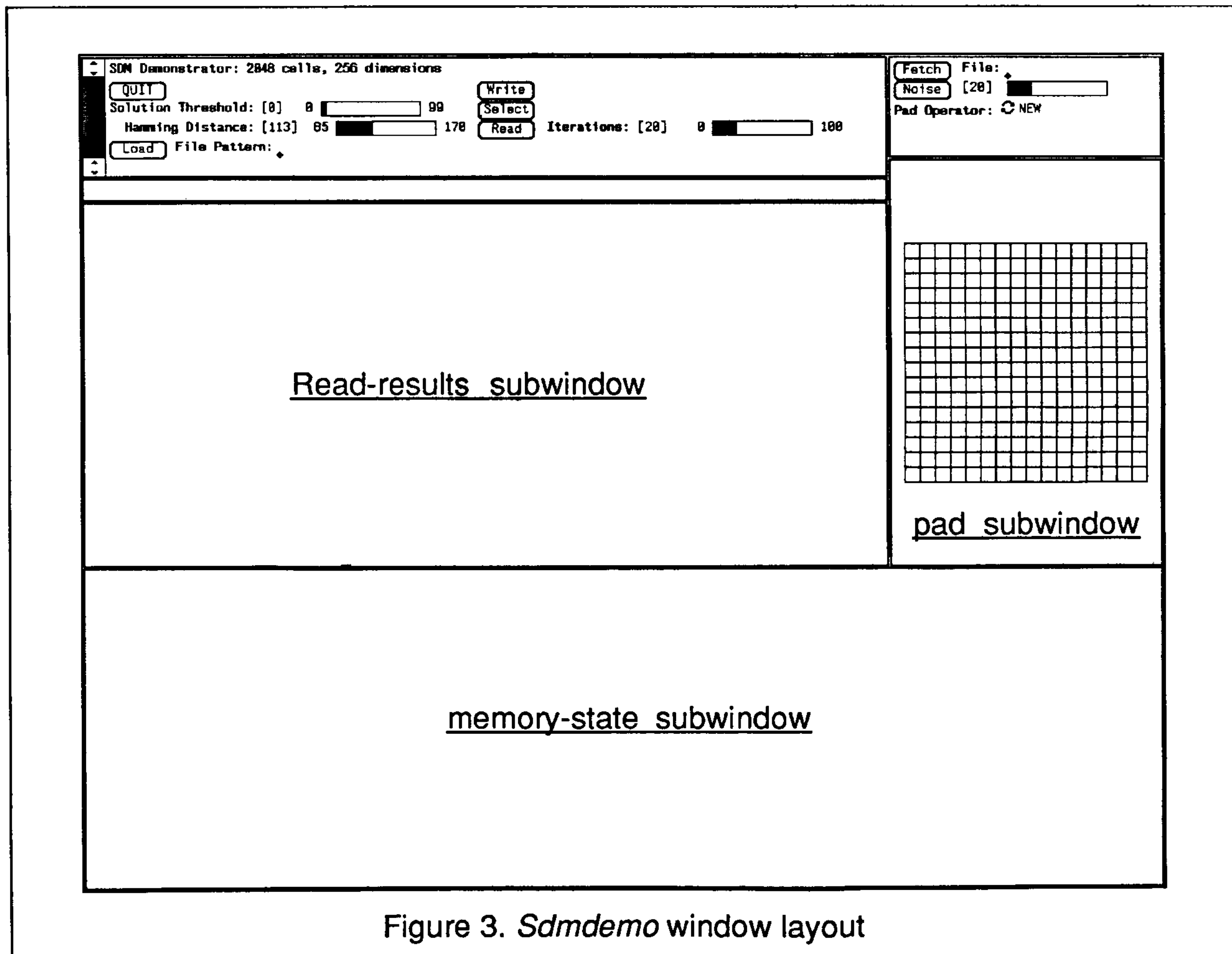


Figure 3. *Sdmdemo* window layout

into the pad from the SDM by clicking the mouse on the desired image in the memory-state window. The items in the pad control panel are as follows:

Fetch/File - when the **Fetch** button is pressed, the contents of the file given after the **File** prompt are loaded into the pad. The given file should have been created with Sun's *iconedit.*

Noise - when pressed, noise is added to the value in the pad. The slider specifies what percentage of bits, at random, should be inverted.

Pad Operator - specifies the boolean operator to use when loading values into the pad.

Items on the main control panel are as follows:

Solution Threshold - same as in *sdmview.* This is the threshold value for converting raw read results to thresholded results. Only the thresholded results are displayed in this program.

Hamming Distance - same as in *sdmview.* This is used whenever a select operation is performed on the memory.

Write - this button copies the image in the pad into the input address and datum registers, performs a select, and then performs a write operation. In other words, it writes the pad contents into all locations whose addresses are within the displayed Hamming distance of its value. This operation is more extensive than the corresponding operation in *sdmview* in that it loads the registers and performs the select as well as the write. Once the operation has completed, the value is added to the memory-state window.

Select - loads the pad contents into the input address register and performs a select operation on the SDM. No other side effects take place. This is useful only for seeing how many locations are selected, which is displayed in the messages window.

Read - copies the pad contents into the input address register, performs a select, and initiates a read operation. The thresholded result is then used as the new value of the input address register and the read cycle starts again. This cycle is repeated until either two reads in a row yield the same result (the read converges) or a cycle is detected, or the number of reads given in the **Iterations** slider is performed.

Iterations - states how many reads to perform while seeking convergence before giving up.

Load/File Patterns - if a file pattern is entered here (using Shell metacharacters) and the **Load** button is pressed, *sdmdemo* reads each file matching the pattern, loads it into the pad, and then performs a **Write** operation. This is a fast way of loading several patterns into the memory.

Figure 4 shows the state of the display after eight previously generated images

have been stored in the SDM and a query using a noisy pattern has been executed. Notice that the read operation converged in very few attempts.

5. Simulator Details

The interface to the simulator is through a library of procedures, each performing a specific operation. The library uses a simple design philosophy: present a simple, consistent interface and remain independent of the application using it. Hence, once the pattern of procedure names is learned, the programmer can accurately guess the name of a procedure implementing a particular operation. Additionally, the library does not contain any code that is specific to any application that may use it.

There are routines to allocate and initialize the SDM, to set and get register values, to set select bits, to read and write, to allocate and copy addresses and data, and to perform input and output conversions. Three externally visible data types are defined, one for holding addresses and two for holding data. These types are pointers to structures used to store these quantities, so that the programmer must allocate storage for them explicitly. The types and allocation procedures are as follows:

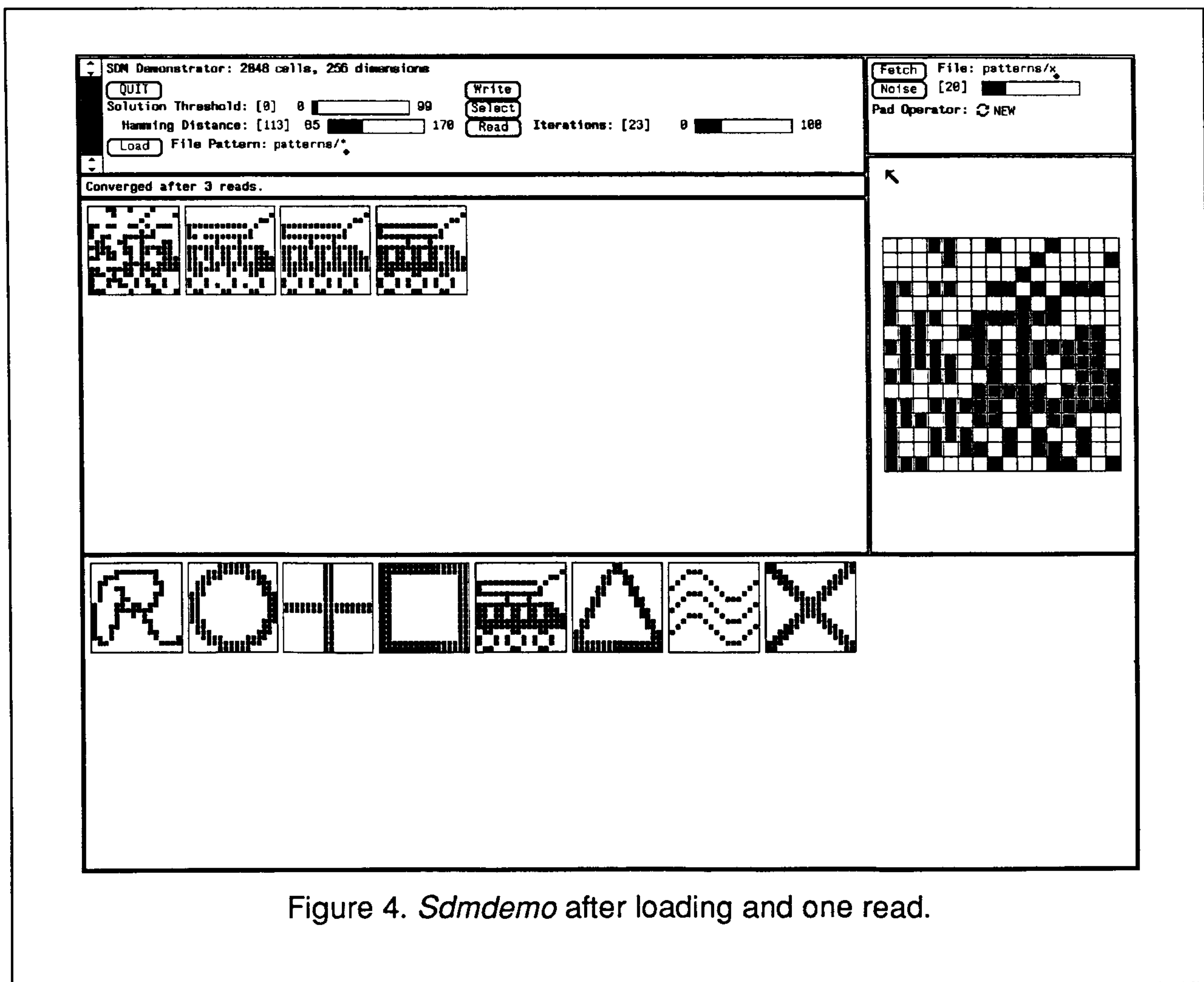


Figure 4. *Sdmdemo* after loading and one read.

```
addr_t *addrNew();
datum_t *datumNew();
bigdatum_t *bigdatumNew();
```

Normally, the user need not use the type *bigdatum_t*. It is only used for the unthresholded output datum register. One internal data type is defined for holding all the information necessary for one SDM location, as follows:

```
typedef struct {
      addr_t    *addr;
      datum_t   *datum;
      bool      select;
} cell_t;
```

Addresses and data may be copied from one location to another using the copy routines as follows:

addrCopy(*dst, src*)
addr_t **dst*, **src*;

datumCopy(*dst, src*)
datum_t **dst*, **src*;

In implementation, SDM addresses are stored as an array of binary integers, currently 16 bits each (though changeable), where each bit in the integer corresponds to one address bit. The number of elements in the addr_t array is a function of **N**, the number of dimensions in the simulated SDM. SDM data are stored as an array of binary integers, also, though there is one element for each of the **N** counters in each location. In the current implementation, the datum counters are 8-bit integers, though it is a one-line compile-time binding (we have never simulated a memory holding as many as 127 patterns using this simulator).

Before using the simulator routines, the program must allocate memory for the SDM and load the random addresses. This is performed with two procedure calls as follows:

sdmInit(*M, N***);**
sdmLoadAddresses();

The dynamic allocation of SDM memory make writing demonstrators and exercisers that configure their memory sizes at runtime possible and convenient. The amount of virtual memory required by the simulated memory, not counting the

registers around the memory, is roughly

$$\mathbf{M} \times \left(\text{sizeof(cell_t)} + \frac{\mathbf{N} + \text{sizeof (addr_t)} \times 8 - 1}{\text{sizeof (addr_t)} \times 8} + \mathbf{N} \times \text{sizeof (datum_t)} \right)$$

For example, a memory comprising 2048 locations and 256 dimensions (this is the default size for the SDM in *sdmdemo*) requires just over 800 kilobytes of memory. Because both demonstrators described herein use relatively small memories, performance degradation due to paging activity is not a problem.

The body of the simulator is a collection of routines for copying data into and out of the registers, for setting select bits, and for reading and writing. Each of the important routines is described briefly here. In these descriptions, the abstract types of the arguments are given as an argument list rather than dummy parameters. Whenever a procedure returns a result in a location specified by a parameter, it assumes that the caller has previously allocated storage for the value by calling **addrNew()**, **datumNew()**, or **bigdatumNew()**. The following detailed interface description may be skipped with little loss for the reader.

sdmSetInaddr(addr_t *) - copies the given address into the input address register. No other side effects (such as setting select bits) occur.

sdmGetInaddr(addr_t *) - copies the contents of the input address register into the (previously allocated) address variable.

sdmSetIndatum(datum_t *) - copies the given datum value into the input datum register. No other side effects (such as writing) occur.

sdmGetIndatum(datum_t *) - copies the contents of the input datum register into the given datum variable.

sdmGetOutdatum(datum_t *) - copies the contents of the thresholded output datum register into the named datum variable.

sdmGetOutdatum_raw(bigdatum_t *) - copies the contents of the unthresholded output datum register into the named bigdatum variable.

sdmGetAddr(addr_t *, int) - copies the address portion of the given SDM location into the given address variable.

sdmGetDatum(datum_t *, int) - copies the datum portion of the given SDM location into the given datum variable.

bool sdmGetSelect(int) - returns the value of the named SDM location select bit.

sdmSelect(int) - sets the select bits for those locations whose addresses are within the given Hamming distance of the address in the input address register.

sdmWrite() - writes (adds) the contents of the input datum register into all selected locations.

sdmRead() - sums all the selected locations and saves the result in the unthresholded output register.

sdmThresh(int) - applies the given threshold to the raw output datum register and saves the result (as ones and zeros) in the thresholded output datum register. Any counter whose value is greater than the threshold is assigned a one and all the other are assigned zeros.

sdmAddrFromDatum(addr_t *, datum_t *) - copies and converts the given datum variable into the given address variable. Both arguments must have been previously allocated. A threshold of zero is hard-wired.

sdmDatumFromAddr(datum_t *, addr_t *) - copies and converts the given address variable into the given datum variable.

bool datumEqual(datum_t *, datum_t *) - returns TRUE if the two given data variables are precisely equal. No thresholding is applied.

int sdmSelectedCount() - returns the number of selected locations in the memory.

bool addrCvt(addr_t *, char *) - converts the given ASCII string into address format and stores the result in the given address variable. The string must be all ones and zeros, or it must start with "0x" and be all hexadecimal digits. The result is padded on the right with zeros. If the convsion succeeds, TRUE is returned; otherwise FALSE is returned.

bool datumCvt(datum_t *, char *) - converts the given ASCII string into datum format and stores the result in the given datum variable. The string must be all ones and zeros, or it must start with "0x" and be all hexadecimal digits. The result is padded on the right with zeros. If the convsion succeeds, TRUE is returned; otherwise FALSE is returned.

char *addrToHex(addr_t *) - converts the given address variable into ASCII in hexadecimal and returns a pointer to a statically allocated buffer.

char *datumToHex(addr_t *) - converts the given datum variable into ASCII in hexadecimal and returns a pointer to a statically allocated buffer.

5.1. Organization

The simulator is packaged in a way that isolates the operations that work directly on the simulated SDM address and datum bank. This way, both serial and parallel versions may be built. Constructing a parallel version for the Sequent Balance 21000 multiprocessor of the simulator was accomplished in less than one afternoon using Sequent's microtasking package [Osterhaug86]. The binding of the particular version

to the application program is accomplished by linking with a library named either **sdm_serial.a** or **sdm_microtask.a**. Naturally, the parallel version only makes sense for the Sequent multiprocessor, and so both demonstrators described in this paper use the serial version on a Sun Microsystems workstation.

The parallel implementation of the library changes only those operations that affect the select bits, the address bank, or the datum bank. The **M** locations are divided into **M/P**-sized chunks, where **P** is the number of processors being used by the simulator. Each processor is then responsible for performing the operation on its chunk. The only complication is the read operation, which requires the summing up of the results. For this, a global array of **P bigdatum_t** objects is allocated and each processor writes its (partial) result in an element of that array. The partial results are then added serially and the overall result is placed in the raw output datum register.

6. Conclusion and Future Work

The two demonstrators have been used to describe the operation of the SDM to visitors at RIACS, and has been generally well-received. The simulator has proven sufficiently powerful and flexible to support these projects. The SDM research group at RIACS has taken the simulator and the first demonstrator, *sdmview*, and adapted it to use for a video movie describing the operation and use of SDM. This work is not described herein.

An extension to the simulator is under development. It will allow the SDM operations to be executed on a remote machine by use of a custom remote-procedure-call (RPC) interface. The goal of this effort is to be able to run graphical displays of the SDM on a graphics workstation while managing the simulation on a high-speed or parallel processor.

7. References

[Kanerva86]
Kanerva, Pentti, “Parallel Structures in Human and Computer Memory,” RIACS TR 86.2, Research Institute for Advanced Computer Science (January 1986).

[Denning86]
Denning, Peter J., “A View of Kanerva's Sparse Distributed Memory,” RIACS TR 86.14, Research Institute for Advanced Computer Science (June 1986).

[Sun86a]
Sun Microsystems, “Sun System Overview,” Sun Part No. 800-1300-02, Revision A (February 1986).

[Sun86b]
Sun Microsystems, “Sun View Programmer's Guide,”, Sun Part No. 800-1345-10, Revision A (September 1986).

[Osterhaug86]
"Guide to Parallel Programming on Sequent Computer Systems,” Sequent Computer Systems (1985).