Chapter 3

Performing I/O

C* provides several methods of performing I/O:

- Parallel versions of the Standard C functions **`fread`** and **`fwrite`**. See Section 3.2.
- Functions that let you create an NMO buffer and transfer data between parallel variables and it. These routines let you do parallel I/O directly through CMOST system calls like **`CM_read`** and **`CM_write`**, and manipulate the parallel NMO buffers used to make these calls. See Section 3.3.
- An interface to functions in the CMFS library. See Section 3.4.
- CMFS functions for performing physical I/O. See Section 3.5.

For information on I/O on the CM-5, see the *CM-5 I/O System Programming Guide*.

3.1 Which Method to Use

Here are some guidelines as to which I/O method to use:

- **`fread`** and **`fwrite`** are the standard method for performing I/O in C*.
- Use the CMFS routines only if you are performing I/O to the DataVault. CMFS I/O to the SDA is available only for back-compatibility.

Copyright © 1990–1995 Thinking Machines Corporation

- Use the CMFS physical I/O functions to perform physical I/O to the DataVault or SDA.
- Create your own NMO buffer and use CMOST system calls in preference to **fread** and **fwrite** under one of these circumstances:
  - If you want more control over buffering than **fread**/**fwrite** will give you. For example, you may want to combine data from several small parallel variables into one NMO buffer and then do a single write operation, so that you don't pay the penalty of the fixed overhead of the system calls.
  - If you are doing I/O to a device for which streaming I/O is not appropriate — for example, I/O over a UDP socket, or I/O over CM5-HIPPI using a raw protocol. The **fread** and **fwrite** functions assume that you can do streaming I/O; they are not appropriate, for example, if a read operation may read less data than requested.
  - If you want to use parallel I/O system calls other than **CM_read** and **CM_write**, such as **CM_readv** or **CM_sendto**.
  - If you want to perform physical I/O without using the CMFS interface.

3.2 Using Parallel Versions of fread and fwrite

CM-5 C* provides overloadings of the Standard C **fread** and **fwrite** interfaces to support parallel variables.

These overloaded versions work much like the ordinary scalar **fread** and **fwrite** functions (part of the "streams" interface that **<stdio.h>** defines), except that they deal with parallel instead of scalar data. You can use any ordinary stream with these functions, such as **stdin**, **stdout**, and **stderr**, or streams obtained by calling **fopen**. (You can also use **fdopen** to create a stream from a file descriptor.) If the stream is associated with an SDA file or a parallel socket, the functions perform parallel I/O; otherwise, the operation is performed as scalar I/O.

With these functions it is possible, for example, to write a program that writes its output to **stdout** and at run time pipe the output to a parallel file,

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

another program, or a scalar file. The mechanism uses parallel I/O in cases where this is possible, and scalar I/O otherwise.

The functions have these prototypes:

```
#include <stdio.h>

overload fread, fwrite;

size_t fread(void:void *dest, size_t size,
             size_t nitems, FILE *stream);

size_t fwrite(const void:void *source, size_t size,
             size_t nitems, FILE *stream);
```

where

dest (for **fread**) is a pointer to the parallel variable to which the data from **stream** is to be read.

source (for **fwrite**) is a pointer to the parallel variable from which data is to be written to **stream**.

size specifies the number of bytes of data per position to be read or written.

nitems specifies the total number of parallel values of each **size** bytes to be written or read. See below for more information.

stream is a pointer to the I/O stream.

The value returned by the parallel **fread** and **fwrite** functions is the total number of items read or written — that is, the total number of bytes read or written divided by **size**. (Note that this is different from the behavior of **CMFS_read_file_always** and **CMFS_write_file_always**, in which the number of bytes per position is returned; see Section 3.4.) If the read or write operation is only partially successful, the return value may not exactly represent the number of bytes read or written. (This is also the case with the scalar **fread** and **fwrite** functions.)

The parallel **fread** and **fwrite** functions work regardless of current shape or context; that is, they read and write all positions.

Copyright © 1990–1995 Thinking Machines Corporation

For socket I/O, the parallel **fread** tries to read until all the data is read or there is an error; this is different from the behavior of the scalar **fread.**

These functions interact correctly with calls to other functions in the streams library. Thus, a sequence of I/O operations with parallel and scalar data works correctly.

The functions impose a file ordering on the data in multidimensional parallel variables; the lowest-numbered axis is always most contiguous in the file. Thus, if we have a shape

```
shape [2][3][2]s;
```

the positions are written out by **fwrite** in the following order:

```
[0][0][0]
[1][0][0]
[0][1][0]
[1][1][0]
[0][2][0]
[1][2][0]
[0][0][1]
[1][0][1]
[0][1][1]
[1][1][1]
[0][2][1]
[1][2][1]
```

This is the same ordering used by the CMFS functions. It is not the ordering used by the **read_from_pvar** and **write_to_pvar** functions.

As mentioned above, the **nitems** argument specifies the total number of parallel values of each **size** bytes to be written or read. This works as if **nitems fread** or **fwrite** operations were performed in sequence. For example, if we have

```
FILE *f;
int:current data[2];
```

then the operation

```
fwrite(data, sizeof(int:current), 2, f);
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

is equivalent to

```
fwrite(data, sizeof(int:current), 1, f);
fwrite(data + 1, sizeof(int:current), 1, f);
```

Both examples write the first four bytes from each position and then the next four bytes from each position. This is *not* equivalent to

```
fwrite(data, 2 * sizeof(int:current), 1, f);
```

This example writes eight bytes from each position, with those eight bytes contiguous in the file.

With the scalar **`fread`** and **`fwrite`** functions, in cases analogous to those above, there is no such difference in semantics.

3.2.1 Performance

For general information about I/O performance, see the *CM-5 I/O System Programming Guide*.

Parallel Files

When doing I/O operations involving parallel (that is, SDA) files, there are two overheads to be concerned with:

- the overhead of doing the "NMO reordering" (that is, reordering the data so that it is in NMO order); see the *CM-5 I/O System Programming Guide* for a discussion of NMO order (*NMO* stands for *New Mexico order*, because the shape of the buffer resembles that of the state of New Mexico)
- the overhead of doing the actual I/O system call

The NMO overhead depends upon

- the element size (the **`size`** argument)
- whether the file position is on a four-byte boundary
- the number of PNs in the partition
- the total amount of data being read or written

Copyright © 1990–1995 Thinking Machines Corporation

When the element size is not a multiple of four bytes, or when the file position is not on a four-byte boundary, this overhead increases by as much as a factor of four. The most efficient NMO transfers happen with element sizes that are multiples of 16 bytes.

The I/O system call overhead can be characterized in terms of

- A fixed cost per system call. The current implementation of **`fread`** and **`fwrite`** performs one **`CM_read`** or **`CM_write`** system call per item in **`nitems`**, and never does any buffering of parallel I/O operations. Thus, it is easy to run up against the high I/O fixed cost.
- A cost that scales with the amount of data. This cost, for SDA files, depends upon the number of disks in the file system.

Parallel Sockets

The considerations described in the previous section also apply when doing I/O operations that involve parallel sockets, with two exceptions:

- The file position does not affect performance (since there is no file position with sockets).
- The fixed overhead of doing parallel socket I/O is substantially less than that of doing parallel SDA I/O.

I/O Involving the Partition Manager

Parallel I/O operations are supported on all types of files and sockets, but performance is severely limited in the case where data must flow through the partition manager. The bandwidth of the interface between the partition manager and the nodes peaks at a few megabytes/second, and the transfer rate may be further limited by the speed of the file system or partition manager network interface involved. Parallel writes to files on serial NFS file servers, like all writes to NFS file servers, will be particularly slow.

Copyright © 1990–1995 Thinking Machines Corporation

3.2.2 Limitations

Please note these limitations on the **fread**/**fwrite** interface:

- As mentioned above, the **fread** and **fwrite** functions do not work with DataVault files. You can access those files only through the CMFS mechanism; see Section 3.4.
- The functions do not work with nodal C* (that is, C* programs compiled with **-node**).
- These functions work with CMOST Version 7.2 or earlier, but they improperly set **errno**. A successful parallel **fread** may incorrectly set **errno** to 22, and a successful parallel **fwrite** may set **errno** to 23. If the function's return value indicates that a complete operation was performed, you should ignore **errno** and assume that the operation was successful.
- These functions are not appropriate for use with sockets that do not support streaming I/O (both parallel and scalar). Thus, they work with TCP sockets but not with UDP sockets.
- When used with parallel sockets, the I/O works properly only when **size * positionsof(*dest)** is a multiple of 4 bytes.

3.2.3 Examples

The examples in this section are shipped with the C* compiler. Check with the system administrator for the location of the C* examples directory at your site.

Parallel Reads and Writes

Here is a simple program that uses **fwrite**:

```
/*
 *  This program demonstrates use of the parallel fwrite
 *  function to perform parallel write operations.
 */
```


Copyright © 1990–1995 Thinking Machines Corporation

```
#include <stdio.h>

#define DIM 100000

int main(int argc, char *argv[])
{
    shape [DIM]S;
    float:S my_data;
    FILE *outfile;

    outfile = fopen(argv[1],"w");
    if(outfile == NULL) {
         printf("fopen() failed\n");
         exit(1);
    }

    with(S)
    {
         my_data = pcoord(0);
    }

    fwrite(&my_data, sizeof(my_data), 1, outfile);

    fclose(outfile);
    exit(0);
}
```

And here is a simple program that uses **fread**:

```
/*
 * This program demonstrates use of the parallel fread
 * function to perform parallel read operations.
 */

#include <stdio.h>

#define DIM 100000

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


Copyright © 1990–1995 Thinking Machines Corporation

```
        shape [DIM]S;
        float:S my_data;
        FILE *infile;

        infile = fopen(argv[1], "r");
        if(infile == NULL) {
             printf("fopen() failed\n");
             exit(1);
        }

        fread(&my_data, sizeof(my_data), 1, infile);

        with(S)
        {
             printf("Data sum: %f\n", += my_data);
        }

        fclose(infile);
        exit(0);
    }
```

Socket I/O

The examples in this section perform parallel I/O to and from a socket. The first two programs show the use of **fread** and **fwrite** to perform the I/O. Supporting routines are included after the programs.

This program creates a socket and writes parallel data to it:

```
/*
 * This program demonstrates the producer side of a client/server
 * network connection.  The program creates a socket and attempts
 * to bind to a port number (where there should be a consumer
 * program listening). When the connection is accepted, the
 * program writes a series of simple images as PARALLEL data to
 * the socket.
 */

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


Copyright © 1990–1995 Thinking Machines Corporation

```
#define  WIDTH     256       /* default dimensions for X grid */
#define  HEIGHT    256
#define  DEPTH     8

#define NAPTIME    4

#define NFRAMES    16        /* default # of movie frames     */

#define BLACK (pixel_t:current)X_black_pixel
#define WHITE (pixel_t:current)X_white_pixel
extern  int   X_black_pixel, X_white_pixel;

typedef char pixel_t;

pixel_t:current checker_board(int:current x, int:current y,
                              int i);

main(int argc, char *argv[])
{
    char *s, *getenv();
    int port = ((s = getenv("CM_READER_PORT")) ? atoi(s) : 6666);

    shape [WIDTH][HEIGHT]frame;

    with (frame)
    {
         pixel_t:current pixels;
         int:current x = pcoord(0), y = pcoord(1);
         int s, i, nbytes;
         FILE *fd;

         if ((s = My_Socket_to_Server(port)) < 0) exit(1);
         fd = fdopen(s, "w");

         for (i = 0; i <= NFRAMES; i++)
         {
              pixels = checker_board(x, y, i);

              nbytes = fwrite(&pixels, boolsizeof(pixels),
                              1, fd);
              if (nbytes < 0)
              {
                   perror("fwrite");
                   exit(1);
```


Copyright © 1990–1995 Thinking Machines Corporation

```
            }
            sleep(NAPTIME);
        }
        (void)fclose(fd);
    }
}

pixel_t:current
checker_board(int:current x, int:current y, int i)
{
    return ((x/(i+1) & 0x1) == (y/(i+1) & 0x1)) ? BLACK : WHITE;
}
```

This example uses **`fread`** to read data from the socket:

```
/*
 * This program reads in a series of image frames from a parallel
 * socket and writes them to an X display window.
 */

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

#define  WIDTH     256       /* default dimensions for frames */
#define  HEIGHT    256
#define  DEPTH     8
#define  NFRAMES   16        /* default # of movie frames   */

typedef char pixel_t;

main(int argc, char *argv[])
{
    char *s, *getenv();
    int port = ((s = getenv("CM_READER_PORT")) ? atoi(s) : 6666);
    shape frame = allocate_shape(&frame, 2, WIDTH, HEIGHT);

    X_open_display(DEPTH, WIDTH, HEIGHT);

    with (frame)
    {
```


Copyright © 1990–1995 Thinking Machines Corporation

```
        pixel_t:current pixels;
        int s, nbytes;
        FILE *fd;

        if ((s = My_Socket_to_Client(port)) < 0) exit(1);
        fd = fdopen(s, "r");

        while (nbytes = fread(&pixels, boolsizeof(pixels), 1,
              fd))
        {
             if (nbytes < 0)
             {
                  perror("fread");
                  exit(1);
             }
             X_write_to_display(pixels);
        }
        (void)fclose(fd);
    }
}
```

Here are the X graphics routines used in this program:

```
/*
 *  X Graphics Routines
 *
 *  These routines provide a simple, uniform interface
 *  for displaying parallel variables from C* programs
 *  on (monochrome) X servers.
 */

int X_white_pixel =  0,
    X_black_pixel = -1;

#include <cm/cmxdisplay.h>
#include <cm/cmx.h>
#define  NCOLORS        256
#define  FONTWIDTH      11

CMX_display_t cmxdisplay;
Display *display;
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
Window window;
GC gc;
int Width, Height;

void
X_open_display(int depth, int width, int height)
{
   cmxdisplay = CMXCreateSimpleDisplay(NCOLORS, width, height);
   display = CMXGetDisplay(cmxdisplay);
   window = CMXGetDrawable(cmxdisplay);
   gc = CMXGetGC(cmxdisplay);
   Width = width;
   Height = height;
   CMXClearDisplay(display, window);
   X_white_pixel = XWhitePixel(display, 0);
   X_black_pixel = XBlackPixel(display, 0);
}

void
X_write_to_display(int:current display_pvar)
{
    CMXPutImage(display, window, gc, display_pvar,
                CMXGetDepth(CMXGetWidget(cmxdisplay)), 0, 0,
                0, 0, Width, Height);
}

void
X_display_string(char *string, int x, int y)
{
    int x0 = x - (strlen(string) * (FONTWIDTH/2));

    XDrawImageString(display, window, gc, (x0 >? 0), y, string,
                     strlen(string));
    XFlush(display);
}
```

And here are the socket routines used in the programs:

```
/*
 * sockets.c  --   C*-callable functions for creating sockets
 *                 and binding/connecting them to ports
```


Copyright © 1990–1995 Thinking Machines Corporation

```
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <ctype.h>
#include <netinet/in.h>
#include <netdb.h>

#include <cm/cm_file.h>
#include <cm/cm_ioctl.h>
#include <cm/cm_errno.h>

#define  PAR_BUF_SIZE   4*1024*1024

int
My_Socket_to_Server(int port)
{
    int s;
    struct sockaddr_in server_addr;
    struct hostent *host, *gethostbyname();
    int name_length, pbufsize = PAR_BUF_SIZE;
    char *ev, *getenv(char *);
    char *server = ((ev = getenv("SERVER_READER"))
                    ? ev : "localhost");

    /* Create a Stream socket in the Internet address family */

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
         perror("socket(AF_INET, SOCK_STREAM, 0)");
         return -1;
    }
    /* Get the (binary) Internet address of the server host */

    if (!(host = gethostbyname(server)))
    {
         fprintf(stderr, "%s: unknown host\n", server);
         return -1;
    }

    server_addr.sin_family = AF_INET;
    bcopy(host->h_addr, &server_addr.sin_addr, host->h_length);
    server_addr.sin_port = port;
    name_length = sizeof(server_addr);
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
    if (connect(s, (struct sockaddr *)&server_addr,
                name_length) < 0)
    {
         perror("connect(s, &server_addr, &name_length)");
         return -1;
    }
    if (setsockopt(s, SOL_SOCKET, SO_PAR_SNDBUF,
                  (char *)&pbufsize, sizeof(pbufsize)) <0)
    {
         perror("setsockopt SO_PAR_SNDBUF");
         return -1;
    }
    return (s);
}

int
My_Socket_to_Client(int port)
{
    int s, client_s;
    struct sockaddr_in addr, client_addr;
    struct hostent *host, *gethostbyname();
    int name_length, pbufsize = PAR_BUF_SIZE;

    /* Create a Stream socket in the Internet address family */

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
         perror("socket(AF_INET, SOCK_STREAM, 0)");
         return -1;
    }

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = port;
    name_length = sizeof(addr);

    if (bind(s, (struct sockaddr *)&addr, name_length) < 0)
    {
         perror("bind(s, &addr, name_length)");
         return -1;
    }
    listen(s, 5);

    client_s = accept(s, (struct sockaddr *)&client_addr,
```


Copyright © 1990–1995 Thinking Machines Corporation

```
                    &name_length);
    if (client_s < 0)
    {
         perror("accept(s, &client_addr, &name_length)");
         return -1;
    }
    if (setsockopt(client_s, SOL_SOCKET, SO_PAR_RCVBUF,
                 (char *)&pbufsize, sizeof(pbufsize)) <0)
    {
         perror("setsockopt SO_PAR_RCVBUF");
         return -1;
    }
    return (client_s);
}
```

3.3 Creating and Using an NMO Buffer

C* provides a group of functions that let you make system calls such as **CM_read** and **CM_write** to perform parallel I/O operations. To use any of the functions described in this section, you must include the file **<cmsys/csio.h>**.

These functions use an NMO buffer. An NMO buffer, conceptually, is parallel memory that is used to store data for the purposes of doing I/O. All the CM parallel I/O system calls use this buffer. The buffer has a special format, so there is a mapping between the I/O data stream (that is, the sequence of bytes in a file) and the data in an NMO buffer.

See the *CM-5 I/O System Programming Guide* for a discussion of **CM_read**, **CM_write**, and NMO buffers.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

3.3.1 Allocating and Freeing an NMO Buffer

Use the functions **CMC_alloc_io_buffer** and **CMC_free_io_buffer** to allocate and free memory, respectively, for an NMO buffer. Here are their prototypes:

```
#include <cmsys/csio.h>

void *CMC_alloc_io_buffer(size_t total_length,
                          int blocking_factor);

void CMC_free_io_buffer(void *address,
                        size_t total_length,
                        int blocking_factor);
```

where

address (for **CMC_free_io_buffer**) is the address on the PNs of the beginning of the buffer. This address is returned by **CMC_alloc_io_buffer**; the data in the buffer cannot be directly accessed on the partition manager.

total_length
is the total number of bytes the buffer must be able to contain. This must be large enough to accommodate the *notch*, described below. The value must be the same for **CMC_alloc_io_buffer** and the corresponding **CMC_free_io_buffer**.

blocking_factor
is the NMO buffer's blocking factor, which determines the number of consecutive bytes of data that are stored in each PN in the NMO format. The blocking factor for SDA files depends upon the number of disks in the file system. For other files, and for socket I/O, it is 16 bytes. The value must be the same for **CMC_alloc_io_buffer** and the corresponding **CMC_free_io_buffer**.

Some parallel I/O operations require the data to be offset a certain number of bytes from the beginning of the NMO buffer; this offset is called a *notch*. The notch is required for SDA files; it is the file's position modulo the blocking factor. The NMO buffer must be large enough to accommodate this

Copyright © 1990–1995 Thinking Machines Corporation

notch and the amount of data, so you must take the notch into account when allocating the NMO buffer.

CMC_alloc_io_buffer returns a pointer to the NMO buffer. You can use this pointer as the value for the **cma_addr.cma_base** member of the **cmaddr_t** structure when making system calls like **CM_read** and **CM_write**.

3.3.2 Writing to and Reading from an NMO Buffer

Parallel Data

Use the functions **CMC_send_to_io_buffer** and **CMC_get_from_io_buffer** to copy parallel data to and from an NMO buffer. Their prototypes are

```
#include <cmsys/csio.h>

void CMC_send_to_io_buffer(void *io_buffer,
                           size_t offset,
                           void:void *source,
                           size_t element_size,
                           int blocking_factor,
                           int *axis_sequence);

void CMC_get_from_io_buffer(void:void *dest,
                            void *io_buffer,
                            size_t offset,
                            size_t element_size,
                            int blocking_factor,
                            int *axis_sequence);
```

where

dest (for **CMC_get_from_io_buffer**) is a pointer to the parallel variable that is to get the data from the NMO buffer.

io_buffer is a pointer to the NMO buffer allocated by **CMC_alloc_io_buffer**.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

offset specifies a byte offset into the NMO buffer's data; use this argument when a notch is necessary, or when assembling several pieces of data into a single NMO buffer.

source (for **CMC_send_to_io_buffer**) is a pointer to the parallel variable from which the data is to be sent to the NMO buffer.

element_size
specifies the number of bytes transferred to or from each position in the parallel variable.

blocking_factor
specifies the NMO buffer's blocking factor, as described in Section 3.3.1.

axis_sequence
is a pointer to an array of integers, each corresponding to one axis. The axis with sequence 0 will have its elements most contiguous in the NMO stream, the axis with sequence 1 will be next-most contiguous, and so on. See below for more information.

The **axis_sequence** argument lets you control the way positions of multidimensional parallel variables are arranged in the stream represented by the NMO buffer. For example, if we have a shape declared

```
shape [2][3][2]s;
```

and we specify this axis sequence:

```
int axis_sequence[3];
axis_sequence[0] = 2;
axis_sequence[1] = 0;
axis_sequence[2] = 1;
```

then the ordering in the NMO buffer of the positions of this shape would be

```
[0][0][0]
[0][1][0]
[0][2][0]
[0][0][1]
[0][1][1]
[0][2][1]
```


Copyright © 1990–1995 Thinking Machines Corporation

```
[1][0][0]
[1][1][0]
[1][2][0]
[1][0][1]
[1][1][1]
[1][2][1]
```

The **axis_sequence** argument may be **NULL**, in which case the default ordering of 0, 1, 2 . . . is used.

CMC_send_to_io_buffer is synchronous; that is, it doesn't return until all the data is in place in the NMO buffer. This means that you don't need to explicitly synchronize after the function before making a system call that uses the data.

Both these functions have roughly the performance of a send operation. They are significantly slower (by up to a factor of four) when the element size or offset is not a multiple of four bytes, or when the parallel data is not aligned on a four-byte boundary.

Scalar Data

The functions **CMC_write_to_io_buffer** and **CMC_read_from_io_buffer** copy data between scalar variables and NMO buffers. The speed of these functions is limited by the connection between the partition manager and the nodes; typically, these functions are useful when transferring small amounts of scalar data that may be part of a buffer or a communication header. Their prototypes are

```
#include <cmsys/csio.h>

void CMC_write_to_io_buffer(void *io_buffer,
                            size_t offset,
                            void *source,
                            size_t total_length,
                            int blocking_factor);

void CMC_read_from_io_buffer(void *dest,
                             void *io_buffer,
                             size_t offset,
                             size_t total_length,
                             int blocking_factor);
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

The **total_length** argument specifies the total number of bytes to be copied. The other arguments are the same as those of **CMC_send_to_io_buffer** and **CMC_get_from_io_buffer**, except that **source** and **dest** refer to scalar data.

Like **CMC_send_to_io_buffer**, **CMC_write_to_io_buffer** is synchronous; it doesn't return until all the data is in place in the NMO buffer. This means that you don't need to explicitly synchronize after the function before making a system call that uses the data.

3.3.3 Examples

The examples in this section are shipped with the C* compiler.

This example uses **CM_write** to write data to an SDA file:

```
/*
 * This program demonstrates parallel writes from C*
 * using the CM_write system call.
 *
 * This program does I/O to an SDA, but the procedure
 * for sockets is similar.
 */

#include <stdio.h>
#include <cmsys/cmio.h>
#include <cmsys/csio.h>
#include <sys/file.h>
#include <sys/fcntlcom.h>
#include <sys/vfs.h>

#define DIM 100000

int main(int argc, char *argv[])
{
    shape [DIM]S;
    float:S my_data;
    int outfile;
    void *nmo_buffer;
    int blocking_factor, notch;
    struct statfs stfs_struct;
```


Copyright © 1990–1995 Thinking Machines Corporation

```
cmaddr_t cmaddr;
int iolen, io_nmo_size;

outfile = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
if(outfile == -1) {
     printf("open() failed\n");
     exit(1);
}

if (fstatfs(outfile, &stfs_struct)) {
     printf("could not fstatfs()\n");
     exit(1);
}

if(stfs_struct.f_width <= 0) {
     printf("output file is not an sda file\n");
exit(1);
}

/*
 * For sockets, the blocking factor will always be
 * 16 bytes. For SDA files this depends on the
 * number of disks.
 */
blocking_factor = stfs_struct.f_width * CMIOLEN;

/*
 * For SDA files, we require a notch that depends on
 * the file position. (Here the file position is
 * zero, of course.)
 */
notch = lseek(outfile, 0, L_INCR) % blocking_factor;

iolen = positionsof(S) * sizeof(my_data);

/* The nmo buffer is large enough to handle a notch
 * if necessary.
 */
io_nmo_size = iolen + blocking_factor * sizeof(my_data);
nmo_buffer = CMC_alloc_io_buffer(io_nmo_size,
                                 blocking_factor);

/* Initialize the data */
with(S)
{
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
        my_data = pcoord(0);
    }

    /* Transfer the contents of the pvar to nmo buffer */
    CMC_send_to_io_buffer(nmo_buffer, notch, &my_data,
                sizeof(my_data), blocking_factor, NULL);

    /* Set up the cmaddr structure */
    cmaddr.cma_addr.cma_base = (CM_memaddr_t) nmo_buffer;
    cmaddr.cma_offset = 0;

    /* Write the data to the disk */
    CM_write(outfile, &cmaddr, iolen);

    CMC_free_io_buffer(nmo_buffer, io_nmo_size, blocking_factor);
    close(outfile);
    exit(0);
}
```

And here is a program that uses **CM_read** to read from an SDA file:

```
/*
 * This program demonstrates parallel reads from C* using the
 * CM_read system call.
 *
 * This program does I/O from an SDA, but the procedure for
 * sockets is similar.
 */

#include <stdio.h>
#include <cmsys/cmio.h>
#include <cmsys/csio.h>
#include <sys/file.h>
#include <sys/vfs.h>

#define DIM 100000

int main(int argc, char *argv[])
{
    shape [DIM]S;
    float:S my_data;
```


Copyright © 1990–1995 Thinking Machines Corporation

```
int infile;
void *nmo_buffer;
int blocking_factor, notch;
struct statfs stfs_struct;
cmaddr_t cmaddr;
int iolen, io_nmo_size;

infile = open(argv[1], O_RDONLY);
if(infile == -1) {
     printf("open() failed\n");
     exit(1);
}

if (fstatfs(infile, &stfs_struct)) {
     printf("could not fstatfs()\n");
     exit(1);
}

if(stfs_struct.f_width <= 0) {
     printf("input file is not an sda file\n");
     exit(1);
}

/*
 * For sockets, the blocking factor will always be 16 bytes.
 * For SDA files this depends on the number of disks.
 */
blocking_factor =  stfs_struct.f_width * CMIOLEN;

/*
 * For SDA files, we require a notch that depends on the
 * file position.  (Here the file position is zero, of
 * course.)
 */
notch = lseek(infile, 0, L_INCR) % blocking_factor;

iolen = positionsof(S) * sizeof(my_data);

/* The nmo buffer is large enough to handle a notch if
 * necessary
 */
io_nmo_size = iolen + blocking_factor * sizeof(my_data);
nmo_buffer = CMC_alloc_io_buffer(io_nmo_size,
                                 blocking_factor);
```


Copyright © 1990–1995 Thinking Machines Corporation

```
    /* Set up the cmaddr structure */
    cmaddr.cma_addr.cma_base = (CM_memaddr_t) nmo_buffer;
    cmaddr.cma_offset = 0;

    /* Read the data from the disk */
    CM_read(infile, &cmaddr, iolen);

    /* Transfer the contents of the nmo buffer to the pvar */
    CMC_get_from_io_buffer(&my_data, nmo_buffer, notch,
                           sizeof(my_data), blocking_factor, NULL);

    /* Examine the results */
    with(S)
    {
         printf("Data sum: %f\n", += my_data);
    }

    CMC_free_io_buffer(nmo_buffer, sizeof(my_data),
                       blocking_factor);

    close(infile);
    exit(0);
}
```

3.4 Using the CMFS Library

CM-5 C* provides synchronous parallel I/O support to the DataVault and SDA via an interface to functions in the CMFS library. The interface is the same for CM-5s with and without vector units. For complete information on CMFS functions, see the *CM-5 I/O System Programming Guide*. **NOTE:** The CMFS interface to the SDA is provided for back-compatibility. The parallel versions of **`fread`** and **`fwrite`** are the preferred method of performing SDA I/O; see Section 3.2.

The CMFS I/O library is not available if you compile with the **`-cmsim`** option to run on a Sun-4.

Note these points in using the **`CMFS`** interface:

- Include **`<cm/cmfs.h>`**.

Copyright © 1990–1995 Thinking Machines Corporation

- Link with **-lcmfs_cs -lcmfs_cm5**, in that order. Or simply specify **+lcmfs**, as described in Table 3.

- These calls are specific to the CM-2 and are not supported in the CM-5 CMFS library:

```
CMFS_cm_to_standard_byte_order
CMFS_convert_vax_to_ieee_float
CMFS_convert_ieee_to_vax_float
CMFS_partial_read_file_always
CMFS_partial_write_file_always
CMFS_transpose_always
CMFS_transpose_record_always
CMFS_file_geometry
CMFS_twuffle_to_serial_order_always_1L
CMFS_twuffle_from_serial_order_always_1L
```

- There are C*-specific versions of **CMFS_read_file_always**, **CMFS_read_file**, **CMFS_write_file_always**, and **CMFS_write_file**. The declarations (from **<cm/cmfs.h>**) are

```
overload CMFS_read_file, CMFS_read_file_always;
overload CMFS_write_file, CMFS_write_file_always;

int CMFS_read_file (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_read_file_always (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_write_file (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_write_file_always (int fd, void:void *dest,
                    int bytes_per_position);
```

  These interfaces provide basic compatibility with CM-200 C* code that calls CMFS.

  The functions are called with pointers to parallel variables. A pointer to a parallel variable of any type may be used. The specified length may be any number of bytes, but performance is significantly diminished when the length is not a multiple of four bytes. See below for a further discussion of I/O performance.

- The lengths passed to and returned by these functions are always in bytes. For the C* interface, they indicate the number of bytes read or written in each position of the parallel variable. Note that on the CM-2/200 the **CMFS_read_file** and **CMFS_write_file** functions

Copyright © 1990–1995 Thinking Machines Corporation

take bit lengths, and that in either case **boolsizeof** should be used to specify the length; this will make the program portable.

- There is currently no difference between the regular and the "always" versions of these functions. Users should only use the "always" versions until this restriction is lifted.
- Streaming and buffered I/O are not supported.
- You can use standard UNIX I/O routines to do serial I/O on CMFS files if the CMFS file system is NFS-mounted. See the *CM-5 I/O System Programming Guide* for more information.
- The total size of a file on a CMFS file system (that is, on the DataVault) will always be rounded up to be a multiple of 512 bytes.
- I/O performance may be significantly diminished if any of the following is true:
  - The size specified to the CMFS functions is not a multiple of 4 bytes.
  - The total amount of data being read or written is not a multiple of 16 bytes on the SDA, or 512 bytes on the DataVault.
  - The file position is not on a 4-byte boundary.
  - The parallel data passed to the CMFS function is an address that is not on a 4-byte boundary (for example, when the pointer points to a member of a parallel structure).
- The **CMFS_lseek** routine when called from C* seeks into a file the number of bytes you specify multiplied by the number of positions in the current shape. The routine **CMFS_serial_lseek** seeks an absolute number of bytes into a file.

Copyright © 1990–1995 Thinking Machines Corporation

3.5 Performing Physical I/O

C* provides routines for doing physical I/O.

The physical I/O routines are **CMFS_write_file_physical** and **CMFS_read_file_physical**. They have the same interfaces as **CMFS_write_file_always** and **CMFS_read_file_always**, namely,

```
int CMFS_write_file_physical(int fd,
                             void:void* source,
                             int bytes_per_position);

int CMFS_read_file_physical(int fd,
                            void:void* destination,
                            int bytes_per_position);
```

These routines are faster than the corresponding standard CMFS I/O routines and use less memory. However, note these points about their use:

- They are not portable. If you use the **CMFS_write_file_physical** routine to write data to an external storage device, you can recover the data only by using the **CMFS_read_file_physical** routine to read the data onto the same set of physical processors, and into a parallel variable with the same memory layout as the parallel variable from which the original write occurred; you must also use the same file system. Essentially, you must use the same partition size, file system, and shape for both writing and reading; if you do a physical write, you must do a physical read.

- These routines return the total number of bytes read or written, rather than the number of bytes read or written per-position.

- The I/O data read or written by these routines is padded. When the file pointer is not located at an alignment boundary appropriate to the I/O device where the file referenced through the **fd** argument resides, the file pointer is adjusted upward to such a boundary before the transfer of the data between the CM and the I/O device begins. Furthermore, garbage data is written to pad the length of the I/O transaction to the same alignment. For DataVault I/O, the alignment padding used is 512 bytes; for SDA I/O, it is 16 bytes times the number of disks across which the SDA stripes.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

To anticipate the impact of padding before doing a read or write, use this routine:

```
int CMFS_physical_transfer_length(int fd,
                                  void:void* target,
                                  int bytes);
```

CMFS_physical_transfer_length takes the arguments of a **CMFS_read_file_physical** or **CMFS_write_file_physical** function call and returns the number of bytes that such a call will transfer. In other words, it returns the same value that a successfully completed call to the read or write function would return.

Because of the complications of padding, we recommend using a physical read only when the file pointer is in the same position it was in when the physical write was performed.

Copyright © 1990–1995 Thinking Machines Corporation