C* 7.1.1 Release Notes

1 About Version 7.1.1

1.1 Introduction

Version 7.1.1 is a maintenance release of the CM-5 C* compiler. It provides bug fixes, performance enhancements, and new I/O capabilities, as well as other new features.

These release notes supersede all previous release notes for CM-5 C*.

1.1.1 Hardware Required

Version 7.1.1 runs on the CM-5 and CM-5E Connection Machine systems, with or without vector units, and on Sun-4 workstations.

1.1.2 Software Required

Version 7.1.1 requires CMOST Version 7.2 or higher. Use of the new routine **`CMC_call_pn_function`** requires Version 8.0.1 or higher of the run-time system (CMRTS).

1.2 Porting Information

Code developed under previous releases must be recompiled and relinked to execute under Version 7.1.1.

Copyright © 1994 Thinking Machines Corporation

1.3 New Features

Version 7.1.1 contains these new features:

- expanded I/O support; see Section 2
- a function that lets you call PN functions from C*; see Section 3.1
- variable-length versions of certain communication-library functions; see Section 3.2
- new command-line options:
  - **`+l`**, which simplifies linking with standard CM software libraries; see Section 3.3.1
  - **`-pedantic`**, which prints warnings in certain cases when your code doesn't exactly conform to Standard C; see Section 3.3.2
  - **`-E`** and **`-P`**, which create preprocessor output only; see Section 3.3.3
- improved support for collection of performance data in Prism Version 2.0 using the **`-cmprofile`** option; see Section 3.4

Section 4 discusses performance enhancements. Section 5 provides other information. Appendix A repeats information from the addendum to the *CM-5 C* User's Guide*.

1.4 Documentation for This Release

Documentation for this release, in addition to these release notes, is the same as for Version 7.1, namely:

- *C* Programming Guide*, May 1993
- *CM-5 C* User's Guide,* Version 7.1
- *CM-5 C* Performance Guide*, Version 7.1
- *Getting Started in C**, May 1993

These manuals are on-line in CMview, collection CM5-1 or higher. ASCII versions of the manuals are available in commands-only Prism, Version 2.0 Beta or higher.

Copyright © 1994 Thinking Machines Corporation

Sample programs are available in **/usr/cstar-7.1.1/examples/cs** by default. If they aren't there, check with your system administrator for their location at your site.

1.5 Errors, Feedback, and Assistance

1.5.1 Bug-Update Files

To learn about restrictions in this release, see the on-line bug-update file, which by default is in **/usr/doc/cstar-7.1.1.bugupdate**. If this file doesn't exist on your system, check with your system administrator.

1.5.2 Request for Feedback

We encourage users to report errors in this product or its documentation, and to suggest improvements in them. If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machine's home office customer support staff at **customer-support@think.com**.

2 New I/O Support

C* Version 7.1.1 provides two new I/O interfaces:

- Parallel versions of the Standard C functions **fread** and **fwrite**. See Section 2.1.
- Routines 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 the CMOST system calls like **CM_read** and **CM_write**, and manipulate the parallel NMO buffers used to make these calls. See Section 2.2.

Both interfaces are suitable for use with all devices except the DataVault. To perform I/O with a DataVault, use the CMFS interface described in the *CM-5 C* User's Guide*.

Copyright © 1994 Thinking Machines Corporation

2.1 Support for Parallel Versions of fread and fwrite

The 7.1.1 release 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, 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.

Copyright © 1994 Thinking Machines Corporation

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.) 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.

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 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.

Copyright © 1994 Thinking Machines Corporation

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);
```

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.

2.1.1 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)

Copyright © 1994 Thinking Machines Corporation

- 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

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

Copyright © 1994 Thinking Machines Corporation

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.

2.1.2 Limitations

Please note these limitations on this 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.
- 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.

2.1.3 Examples

The examples in this section are also shipped with the C* Version 7.1.1 compiler. By default they are installed in **`/usr/cstar-7.1.1/examples/cs`**; check with your system administrator if they aren't there.

Copyright © 1994 Thinking Machines Corporation

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.
 */

#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);
}
```


Copyright © 1994 Thinking Machines Corporation

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[])
{
    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.

Version 7.1.1, May 1994
Copyright © 1994 Thinking Machines Corporation

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>

#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;
```


Copyright © 1994 Thinking Machines Corporation

```
        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);
             }
             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
```


Copyright © 1994 Thinking Machines Corporation

```
#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)
    {
         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);
    }
}
```


Copyright © 1994 Thinking Machines Corporation

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;
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,
```


Copyright © 1994 Thinking Machines Corporation

```
                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
 */

#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 *);
```


Copyright © 1994 Thinking Machines Corporation

```
    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);

    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;
```


Copyright © 1994 Thinking Machines Corporation

```
    /* 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,
                      &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);
}
```


Copyright © 1994 Thinking Machines Corporation

2.2 Support for Creating and Using an NMO Buffer

C* also provides a new 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.

2.2.1 Allocating and Freeing the 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

Copyright © 1994 Thinking Machines Corporation

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 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**.

2.2.2 Writing to and Reading from the 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,
```


Copyright © 1994 Thinking Machines Corporation

```
                                    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**.

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 2.2.1.

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;
```


Copyright © 1994 Thinking Machines Corporation

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]
[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 transfer-

Copyright © 1994 Thinking Machines Corporation

ring 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);
```

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.

2.2.3 Examples

The examples in this section are also shipped with the C* Version 7.1.1 compiler. By default they are installed in **/usr/cstar-7.1.1/examples/cs**; check with your system administrator if they aren't there.

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.
 */
```


Version 7.1.1, May 1994
Copyright © 1994 Thinking Machines Corporation

```
#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;
    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;

    /*
```


Copyright © 1994 Thinking Machines Corporation

```
     * 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)
    {
         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
```


Copyright © 1994 Thinking Machines Corporation

```
 * 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;
    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.
     */
```


Copyright © 1994 Thinking Machines Corporation

```
    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);

    /* 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);
}
```


Copyright © 1994 Thinking Machines Corporation

3 Other New Features

3.1 Calling PN Functions from C*

C* Version 7.1.1 contains a new routine, **CMC_call_pn_function**, that calls PN functions from C*. Using this function requires Version 8.0.1 or later of the run-time system (CMRTS).

The syntax is

```
#include <csshape.h>

void CMC_call_pn_function(void (*function)(),
                          void *data, size_t len);
```

where

function is a pointer to a PN function that is to be called from the partition manager. The name of this function must begin with **CMPE_** because the linker (**cmld**) makes only these names visible to partition-manager code.

data is a pointer to scalar data that is to be broadcast to the PN function. Typically, you declare a structure containing the data to be broadcast, and pass a pointer to this structure.

len is the number of bytes of data to be broadcast.

CMC_call_pn_function causes the specified function to be called on every node. The function is called with two arguments. The first is a pointer to the copy of the data that was broadcast to the PNs; the second is **CMC_call_pn_function**'s **len** argument. For example, your PN function might be declared as

```
void CMPE_my_pnfunction (void *data, size_t len);
```

CMC_call_pn_function returns as soon as it has broadcast all the required information. It does not wait for the PN function to return.

In practice, **CMC_call_pn_function** is useful for calling PN code that you have written. This code may be written in C, DPEAC, CDPEAC, and so on, and may involve the CMMD library for doing communication.

Copyright © 1994 Thinking Machines Corporation

You can pass parallel data to PN routines by identifying the data's memory address using the **CMC_pointer_mem_addr** function and passing this as part of the data broadcast to the PN function. Other information may also be required for generality, particularly the data's stride (obtained through **CMC_pointer_mem_stride**) and information about the data's layout (obtained through **CMC_shape_subgrid_size** and other functions). When doing this, you should understand how parallel variables are laid out in memory, as described in appendixes B and C of the *C* Programming Guide*.

3.2 Variable-Length Versions of Communication-Library Routines

The CM-5 C* communication library now contains variable-length versions of these functions:

- **scan**
- **copy_spread**
- **copy_multispread**
- **read_from_position**
- **write_to_position**

Note that using the variable-length versions of these functions will make your program nonportable between the CM-2 and the CM-5.

The definition of the variable-length version of **scan** is

```
void scan (
    void:current *destp,
    void:current *sourcep,
    unsigned length,
    int axis,
    CMC_combiner_t combiner,
    CMC_communication_direction_t direction,
    CMC_segment_mode_t smode,
    bool:current *sbitp,
    CMC_scan_inclusion_t inclusion);
```

where

destp is a pointer to the location where the results of the scan will be placed.

Copyright © 1994 Thinking Machines Corporation

sourcep is a pointer to the beginning of the data to be used in the operation.

length is the length, in **bool**s, of the data.

axis is the axis along which the scan class or classes are to be created.

combiner specifies the type of operation that **scan** is to carry out. Possible values are **CMC_combiner_logxor**, **CMC_combiner_copy**, **CMC_combiner_logior**, and **CMC_combiner_logand.**

direction specifies the direction of the operation. Possible values are **CMC_upward** and **CMC_downward**.

smode specifies whether the sbit is a segment bit or a start bit. Possible values are **CMC_start_bit**, **CMC_segment_bit**, and **CMC_none**. Specify **CMC_none** if there is no sbit.

sbitp is a scalar pointer to a **bool**-sized parallel variable of the current shape. This parallel variable is the sbit, which creates scan sets for the operation. Specify **CMC_no_field** if there is no sbit.

inclusion specifies whether the operation is exclusive or inclusive. Possible values are **CMC_exclusive** and **CMC_inclusive**.

See Section 13.3 of the *C* Programming Guide* for more information about **scan**.

The definition of the variable-length version of **copy_spread** is

```
void copy_spread (
    void:current *destp,
    void:current *sourcep,
    unsigned length,
    int axis,
    int coordinate);
```

where

destp is a pointer to the location where the results of the copy will be placed.

Copyright © 1994 Thinking Machines Corporation

sourcep is a pointer to the beginning of the data to be used in the operation.

length is the length, in **bool**s, of the data.

axis is the axis along which the scan class or classes are to be created.

coordinate is the coordinate along **axis** that specifies the element whose value is to be copied.

See Section 13.5.2 of the *C* Programming Guide* for more information about the **copy_spread** function.

The definition of the variable-length version of **copy_multispread** is

```
void copy_multispread (
    void:current *destp,
    void:current *sourcep,
    unsigned int length,
    unsigned int axis_mask,
    CMC_multicoord_t multi_coord);
```

where

destp is a pointer to the location where the results of the copy will be placed.

sourcep is a pointer to the beginning of the data to be used in the operation.

length is the length, in **bool**s, of the data.

axis_mask is a bit mask that specifies the axis or axes along which the positions in a hyperplane are allowed to differ.

multi_coord specifies the coordinates that determine the elements of the source data from which values are to be copied.

For complete information on **copy_multispread**, see Section 14.5 of the *C* Programming Guide.*

The definition of the variable-length version of **read_from_position** is

```
void read_from_position (
    CMC_sendaddr_t send_address,
    void *destp,
```


Copyright © 1994 Thinking Machines Corporation

```
    void:void *sourcep,
    int length);
```

where

send_address
: is the send address of a position from which a value is to be read.

destp
: is a pointer to the scalar location to which the parallel data is to be written.

sourcep
: is a pointer to the beginning of the parallel data to be read.

length
: is the length, in **bool**s, of the data.

For complete information on **read_from_position**, see Section 14.4.1 of the *C* Programming Guide.*

The definition of the variable-length version of **write_to_position** is

```
void write_to_position (
    CMC_sendaddr_t send_address,
    void:void *destp,
    void *sourcep,
    int length);
```

where

send_address
: is the send address of a position from which a value is to be read.

destp
: is a pointer to the parallel location to which scalar data is to be written.

sourcep
: is a pointer to the beginning of the scalar data to be written.

length
: is the length, in **bool**s, of the data.

For complete information on **write_to_position**, see Section 14.4.2 of the *C* Programming Guide*.

Copyright © 1994 Thinking Machines Corporation

3.3 New Compiler Options

3.3.1 Simplifying Linking with Other CM Software Libraries: The +l Option

C* Version 7.1.1 adds a new option to the **cs** command called **+l**; this option simplifies the process of linking C* code with other CM software libraries. Instead of specifying specific libraries to link with, you can use this option to simply specify the software component that needs to be linked in. The **+l** option can be used with the forms listed in the table below. In all cases, the previous methods of linking will continue to work.

Form	Use
+lcmfs	Link in the CMFS C* interface, instead of using **-lcmfs_cs** and **-lcmfs_cm5** as described in the *CM-5 C* User's Guide*.
+lcmssl	Link with the CMSSL libraries, replacing the steps described in the manual *CMSSL for C**.
+lcmf	Link in the necessary CM Fortran libraries for a C* program that calls CM Fortran. (This means that you can use **cs** to do the link, and it does everything that Step 3 of Section 3.6 in the *CM-5 C* User's Guide* does.)
+lf77	Link in the **f77** libraries. This is necessary when linking **f77** objects into a C* program. The **f77** libraries must be installed on the machine on which you are linking.
+lcmx11	Link with the CMX11 libraries, instead of following the steps in the *CMX11 Reference Manual*.
+lXm **+lXt**	Bring in the Motif and Toolkit libraries when linking with X. (This does what **-lXm** and **-lXt** would do.) Some CMX11 programs need to add these.

Copyright © 1994 Thinking Machines Corporation

+lcmavsflow **+lcmavssim** **+lavsflow** **+lavssim** **+lavscommon**	Link in the CM/AVS libraries, as appropriate.
+lcmmd	Link in the CMMD libraries.

For advanced users: The **+l** option is controlled by a configuration file read by the C* compiler driver. The file is normally called **cstar-config** and resides in the *lib_dir* directory, the location of which you can obtain by issuing the **cs** command with the **-dirs** option. This configuration file is set up when the compiler is installed, and is maintained by the system administrator. It defines exactly which libraries are needed by the various **+l** options and where these libraries are to be found; it also defines some default compiler options. (The **-L** option does not affect where libraries that are specified via the **+l** option are found.)

You can override the entire configuration file with another file by specifying the option **-csconfig** *filename* to the **cs** command.

You can override the location of a specific set of libraries by using the syntax **+l***component***=***path*, where *component* is the specific **+l** option being overridden, and *path* is the directory in which libraries for this component should be found. For example,

```
% cs +lcmf=/u/bob/cmf/lib
```

uses the CM Fortran libraries found in **/u/bob/cmf/lib**.

It is also possible to override the location and name of a library, with the same syntax as above, but where *path* is the name of the library itself. If the library has scalar and nodal components, *path* should specify the scalar component.

3.3.2 Displaying Warnings for Noncompliant Code: The −pedantic Option

A **-pedantic** option is now available for the **cs** command. When you specify this option, the compiler prints warnings in a few cases in which the source code is not exactly compliant with Standard C.

Copyright © 1994 Thinking Machines Corporation

3.3.3 Creating Preprocessor Output: The −E and −P Options

The **-E** and **-P** options are now available for the **cs** command. These options are comparable to **cc**'s **-E** and **-P** options. Specifically,

- The **-E** option sends the source file through the C* preprocessor only. The resulting output contains line numbers. The output is sent to the standard output, or is placed in a file you specify via the **-o** option.
- The **-P** option also causes the source file to be sent through the C* preprocessor only. The resulting output does not contain line numbers, and is placed in a file with the **.i** suffix. For example, if the source file is named **test.cs**, the preprocessor output is placed in the file **test.i**.

3.4 Support for New Prism Performance Analysis Feature

C* Version 7.1.1 supports the improved performance feature, available in Prism Version 2.0 Beta or later, in which, among other things, Prism measures the time spent on the nodes by routines that weren't compiled with the **-cmprofile** option; Prism allocates this time to the **Node not profiled** resource for data parallel programs, and to the **Scalar cpu** resource for message-passing programs.

Note these points:

- Time spent by functions in the C* communication library is allocated to the **Node not profiled** or **Scalar cpu** resource; the times are accurate, but Prism does not provide information about the particular communication operations being performed.
- Prism performance analysis works only if the C* program contains a C* **main** routine (or a CM Fortran **MAIN** routine) that was compiled with the **-cmprofile** option. If you have a combined C*/CM Fortran program, and you are linking with **cmf** (as described in Section 3.6 of the *C* User's Guide*), you must have a CM Fortran **MAIN** routine, compiled with **-cmprofile**, call your C* program. (Note also that you can now link with **cs** if you compile with the **+lcmf** option, as described in Section 3.3.1 of these release notes.)

If you do not have Prism Version 2.0 Beta or later, time spent in routines not compiled with **-cmprofile** is not measured.

Copyright © 1994 Thinking Machines Corporation

For complete information on Prism performance analysis, see the *Prism User's Guide*.

4 Performance Improvements

C* Version 7.1.1 improves performance in these areas:

- Grid communication functions are now much faster than in the previous release. In many cases **from_grid**, **from_grid_dim**, **to_grid**, and **to_grid_dim** are twice as fast or more, and their performance is often more in line with the torus functions. The greatest speedups occur when operating on large shapes. The **from_grid** function remains faster than the **to_grid** function; **from_grid** is approximately 10−25% faster than **to_grid** when all positions are active, and twice as fast when operating under context.
- The performance of **where** operations has been improved in some cases; **where** statements no longer always break code blocks.
- **CMC_initialize_shared_table** and the arbitrary-length overloading of **CMC_lookup_shared_table** are faster.
- The performance of parallel I/O in cases where the element size is not a multiple of 4 bytes, or where the file position is not on a 4-byte boundary, is improved; performance is still significantly worse than when data is aligned, however.

5 Other Information

5.1 Return Value Conventions for Functions That Return floats

The return value conventions for functions that return **float**s are incompatible between CM-5 C* and Sun's **/bin/cc**. In C*, an unprototyped function declared to return a scalar **float** returns the value as if it were a **float**. This is consistent with the Sun ANSI C compiler and **gcc**. However, Sun's **/bin/cc** compiler returns such values as if they were **double**s.

Copyright © 1994 Thinking Machines Corporation

In C* code, when calling C functions compiled by **/bin/cc** that return **float** values, declare these in the C* code to return **double** values. In C code compiled by **/bin/cc**, when calling C* functions that return **float** values, rewrite the C* functions to return **double** values instead.

Version 7.1.1, May 1994
Copyright © 1994 Thinking Machines Corporation