CMMD Version 3.2
Release Notes

1 Introduction

1.1 Product Description

The CMMD library provides facilities for programming the CM-5 in a MIMD style. It supports the following operations:

- sending and receiving messages between nodes
- global operations: scan, reduce, broadcast, concatenate, synchronization
- timing functions
- node-level I/O (both independent and cooperative)
- active messages and rport operations
- I/O functions support Scalable Disk Arrays (SDA)

1.2 Dependencies

CMMD runs on both the CM-5 and the CM-5E. CMMD Version 3.2 requires CMOST Version 7.3 Beta 3.2 or later.

CMMD can be called from programs written in C, C++, C*, FORTRAN 77, or CM Fortran. Specifically,

- CM Fortran Version 2.1 final (and following versions) are supported. C* Version 7.1 is supported.

Copyright © 1994 Thinking Machines Corporation

- Sun F77 versions 2.0 and earlier are supported. To debug Version 2.0, pndbx Version 1.2-final-patch6 (or later), or Prism Version 2.0 (or later) is required.
- All Sun bundled C compilers are supported. Versions of **`acc`** (unbundled Sun C) prior to and including Version 2.0 are supported. To debug Version 2.0, pndbx Version 1.2-final-patch6 or later is required.
- GNU C Version 2.3.3 is supported.
- The Sun CFront compiler Version 1.0 and the GNU G++ compiler Version 2.3.3 and earlier (both C++ compilers) are supported for compilation and linking. Currently neither the the pndbx nor Prism debuggers support C++ debugging.

C and C* programs using CMMD must include the include file **`cm/cmmd.h`**. Other standard include files, such as **`stdio.h`**, **`fcntl.h`**, and **`sys/types.h`**, may also be needed, depending on the particular program.

Fortran programs must include **`cm/cmmd_fort.h`**, in addition to whatever files they would normally include.

1.3 Compiling, Linking, and Debugging

Applications should be recompiled and relinked when moving from earlier versions of CMMD to 3.2.

1.3.1 General Linking Notes

Don't Link With the `-g` Option

Linking with the **`-g`** option causes your program to link with the debug versions of the CMMD libraries; this will cause the debugger to try to find the CMMD library sources, and you will see potentially confusing error messages such as "**`couldn't read /proj/cmmd_release`**. . ."

The debug version of the libraries are useful only for debugging CMMD sources; they are of no particular use to user applications.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Link Lines Must Be Less than 4Kbytes Long

The **cmmd-ld** linker requires that all lines be less than approximately 4 Kbytes long. If your list of **.o** files exceeds that length, you should collapse a group of **.o** files into a single **.o** file via

```
ld -r -o <output>.o <list of input .o's>
```

Linking F77 Modules with Nodal CMF Code

The CM Fortran 2.1.1-2 driver supports linking of **f77** modules with nodal CM Fortran code. Users who have difficulty linking F77 and CM Fortran modules should contact Thinking Machines Customer Support.

1.3.2 FORTRAN 77 Compilation Notes

Use −Nx Option to Increase Symbol Table Limit

FORTRAN 77 programmers find that the default F77 symbol table size is not large enough to contain the symbols used in a CMMD program. You can tell this is the case if you see the following error when you try to compile a program:

```
f77 program.pn.F ... Compiler error: Too many external
symbols.
```

A simple workaround is to specify the **-Nx** option to increase the size of the symbol table. For example,

```
f77 program.pn.F ... -Nx500
```

Use the −f or −dalign Option to Align Double Precision Data Correctly

It is possible for an F77 program to contain double-precision data that is not double aligned (F77 does not force double alignment by default). The F77 versions of certain CMMD routines will get a bus error and crash the program if they are called on such data. (This is because CMMD routines always attempt double word loads when dereferencing double pointers.) To avoid this problem, compile with the **-f** option, or with the **-dalign** option.

Copyright © 1994 Thinking Machines Corporation

1.3.3 Optimization and the "Volatile" Keyword

When programming in CMMD, variables may be set by a handler invoked asynchronously (as an argument to, e.g., **CMMD_receive_async** or **CMAML_rpc**) and polled in some fashion by the main thread of computation. When interrupts are enabled, this polling often takes the form of

```
while (!polling_variable)
  ;
```

Optimizing compilers such as Sun's CC and GCC will optimize away the test of the polling variable, turning this into an infinite loop and causing the program to hang. To avoid this problem,

- Using Sun's CC, avoid specifying optimization levels greater than two (i.e., only use **-O2** or less aggressive optimization). This prevents the compiler from optimizing these kinds of variable references away.
- Using ANSI C compilers (such as GCC), you may use the "volatile" keyword to mark variables which may change without action of the main thread of computation. If these variables are so marked, references to them will not be optimized away.

1.3.4 Debugging with Node Prism

Prism for MIMD programming (Node Prism) is available in Prism Version 2.0. For complete information on it, see the *Prism User's Guide*, Version 2.0, or Prism's on-line help system.

1.4 Reproducibility

Message-passing programming has the disadvantage that the program behavior (specifically bugs) may be non-repeatable if the code pathways are affected by the timing of the arrival of messages. This may be avoided by coding applications so that execution correctness is not so dependent. Specifically, if the following steps are taken, the reproducibility of your program should be increased:

- Avoid the use of handlers (e.g., in **CMMD_receive_async**). The effect of these is to execute code when a message arrives instead of at a determined point in your program. Instead, when you want to use the

Copyright © 1994 Thinking Machines Corporation

CMMD asynchronous primitives to overlap communication and computation, use **CMMD_{send,receive}_async** coupled with **CMMD_msg_wait**. This does not make your programs control flow dependent on message arrival time.

- Avoid using **CMMD_msg_pending** and **CMMD_mcb_pending**. These specifically allow code branches based on whether a message is ready to be received or not.
- Avoid using the active message primitives; again, when the remote handler is executed will be dependent on when the message reaches the remote node.
- Do not use the wildcards **CMMD_ANY_NODE** and **CMMD_ANY_TAG** in a receive call; these make your control flow dependent on which node's message arrives first.
- Avoid using the **CMMD_{scan,reduce,sync}_done()** functions to test for completion of split-phase global operations; instead block for completion with the **CMMD_{scan,reduce,sync}_finish()** functions.

These guidelines represent a tradeoff of functionality (and potentially performance) vs. ease of coding and debugging; for some applications this tradeoff will be appropriate, while for others it will not be.

2 New and Changed Features

Features added in CMMD 3.2 since CMMD 3.2-Beta:

- Fortran libraries targetted for superscalar SPARC work correctly.
- Setting the I/O mode from Fortran no longer flushes I/O buffers.
- **CMMD_write_channel** polls.
- The examples can work with non-standard installations.
- User access to CMMD version information has been expanded.
- All linker flags are passed through to the linking compiler.

Copyright © 1994 Thinking Machines Corporation

Features added in CMMD 3.2-Beta since CMMD 3.1-Final72:

- extensions to I/O system functionality
- configuration-independent support for higher-performance nodes
- extensions to CMMD "global" (scan, reduce, etc.) functionality
- minor extensions and changes to the CMMD error reporting system
- improved error checking and use of **`/tmp`** for temp space in the **`cmmd-ld`** link script
- support of third-party link time tools
- support for Apogee compilers
- correction of **`CMAML_allocate_this_tag`** semantics

Features added in CMMD 3.1-Final72 since CMMD 3.1-Alpha72B2:

- additional functions returning the number of pending send and receive "slots" available
- change in the meaning of the argument to the **`cmmd-ld`** flag **`-cmmd_root`**

Features added in CMMD 3.1-Alpha72B2 since CMMD 3.0.2:

- support for vector unit I/O
- support for the Global/Local programming model
- improvement in performance of the CMMD "global" functions (scan, reduce, etc.)

These changes are described in more detail below.

Fortran Libraries Targetted for Superscalar SPARC Work Correctly

Some Fortran compilers (e.g., Sun's F77) have run-time libraries that are targetted at superscalar SPARC platforms. These libraries return an error at run time if the value returned by **`gethostid`** does not indicate that the underlying hardware is a superscalar SPARC. As of CMMD 3.2, the value returned by a nodal **`gethostid`** correctly reflects whether or not the node hardware is superscalar.

Copyright © 1994 Thinking Machines Corporation

Setting the I/O Mode from Fortran No Longer Flushes I/O Buffers

The CMMD libraries no longer call the Fortran utility routine "**flush**" when the user requests that the IO mode of a Fortran unit is changed. This allows linkage with the Apogee-FORTRAN 77 libraries, which do not provide this utility routine.

This means that the user now has the responsibility to assure that Fortran does not have any data in its internal I/O buffers when switching I/O modes. Failure to satisfy this condition may result in mangled output or program failure.

CMMD_write_channel Polls

The routine **CMMD_write_channel** now polls the network if it is called with interrupts disabled. This allows a programming idiom such as

```
while(ret = CMMD_write_channel()) {
  if (ret == CMMD_ERRVAL)
    abort();
}
```

(i.e., spin until **CMMD_write_channel** returns zero *without* calling **CMMD_is_channel_writable**) to suceed when interrupts are disabled.

The Examples Can Work with Non-Standard Installations

The example program makefiles no longer specify hardcoded paths for compilers; the **PATH** environmental variable is searched for them instead. This allows users to specify (by manipulation of **PATH**) non-standard locations for these compilers.

User Access to CMMD Version Information Has Been Expanded

Invoking **cmmd-ld** with the **-V** switch will return version information on the installed CMMD. Example output is

```
$ cmmd-ld -V CMMD Version: cmmd_3_2 $
```

In addition, there is now a new example program (**hostnode/c/version**) that may be invoked for more detailed version information. The level of

Copyright © 1994 Thinking Machines Corporation

detail is controlled by its one argument, ranging from an argument of "**0**" for least detailed to an argument of "**2**" for most detailed.

All Linker Flags Are Passed Through to the Linking Compiler

Users should now use the appropriate compiler flags during the **cmmd-ld** link phase. In releases previous to 3.2, the **cmmd-ld** link script manipulated the output of the native compiler link phase. This prevented the use of some native link flags. The behavior has changed so that all link flags are passed to the native compiler. Users who try optimizing their executables may see improved performance.

Unfortunately, this means that the **-g** flag to **cmmd-ld** now has two meanings, both passing the **-g** flag to the underlying linker and linking with the CMMD debug libraries (See "Don't link with the **-g** option," above). In Version 3.3, these two meanings will be separated; users who wish to link with the CMMD debug libraries will use **-G** instead of **-g**.

Extensions to I/O System Functionality

CMMD 3.2 supports four new socket routines: **sendmsg()**, **recvmsg()**, **getsockopt()**, and **setsockopt()**. Further, all socket routines now support HIPPI domain sockets. Please see Section 12.7 of the *CMMD Reference Manual* (Version 3.0) for a brief overview of CMMD network programming, and the Connection Machine HIPPI documentation for a description of the HIPPI domain.

Previous versions of CMMD line-buffered all UNIX files under standard I/O. As of Version 3.2, CMMD uses standard UNIX default buffering, retaining line-buffering by default only for tty's.

With this last change comes a reminder that programs should always call "**fflush**" before changing the mode of a **stdio** or stream file pointer; that is, the program should make sure that all pending data has been written before changing the CMMD I/O mode for standard I/O during the course of a program. The change in **stdio** buffering, described above, increases the likelihood of trouble if the buffer is not explicitly flushed before the mode is changed.

Copyright © 1994 Thinking Machines Corporation

Configuration-Independent Support for Higher Performance Nodes

Version 3.2 has been restructured to remove library dependence on the hardware platform. There is now a single CMMD library, which can be used to run programs on any CM-5 configuration. For instance, users are no longer required to specify, at link time, whether their code requires vector unit support. In 3.2, this change allows users to take advantage of higher performance (SPARC superscalar) nodes where they are available. Future configuration dependencies will similarly be included in this manner. Platform independence is gained by nearly transparent run-time configuration determination and code path selection. Users running on existing nodes (as opposed to high performance nodes) will see a decrease of approximately 0.5 Mbytes per second in maximum achievable bandwidth.

Because the underlying hardware configuration is now determined at run time, the **`-vu`** switch is no longer necessary. Use of this switch will produce a warning message in this release (3.2) and may not be supported in subsequent releases.

Please note that nodal C* does not currently work on SPARC superscalar nodes. It will work correctly as of C* Version 7.1.1.

New CMMD Global Functions and Improved Performance

There are two kinds of new functions: 64-bit integer scans and reductions, and split-phase scans and reductions.

- Functions for 64-bit integers

  There are six new combiners (**`CMMD_combiner_ladd`**, **`_lmin`**, **`_lmax`**, **`_luadd`**, **`_lumin`**, and **`_lumax`**) and eight new functions:

  ```
  CMMD_scan_lint
  CMMD_scan_luint
  CMMD_reduce_lint
  CMMD_reduce_luint
  CMMD_reduce_to_host_lint
  CMMD_reduce_to_host_luint
  CMMD_reduce_from_nodes_lint
  CMMD_reduce_from_nodes_luint
  ```

  The **`_lint`** functions and the **`_l`** combiners perform 64-bit signed integer scans and reductions; the **`_luint`** functions and the **`_lu`** combiners perform 64-bit unsigned integer scans and reductions. The

Copyright © 1994 Thinking Machines Corporation

64-bit combiners may also be used with **CMMD_scan_v** and **CMMD_reduce_v**.

The C standard does not specify the word order for 64-bit integers. CMMD assumes that the high-order word comes first, as is the case, for example, with code compiled by **gcc** or CM Fortran.

- Split-phase functions

  For each **{type}** (**int**, **uint**, **float**, **double**, **lint**, **luint**), there is a pair of **_start** and **_finish** scan and reduction functions:

```
void CMMD_scan_{type}_start
   ({type} value, combiner, direction, smode, bit, inclusion)
   {type}

CMMD_scan_done()

CMMD_scan_{type}_finish
   (int value, combiner, direction, smode, sbit, inclusion)
   int

void CMMD_reduce_{type}_start ({type} value,
   combiner) {type}
CMMD_reduce_{type}_finish ({type} value, combiner)
   int CMMD_reduce_done()
```

  The **_start** function initiates a scan or reduction, but does not block pending completion. The **_done** function is a predicate that returns 1 or 0 according to whether the scan or reduction is or is not ready to finish. The **_finish** function blocks until the scan or reduction finishes, and returns its value.

  Similarly, there is a set of functions for split-phase synchronization:

```
void CMMD_sync_with_nodes_start()
void CMMD_sync_with_nodes_finish()
int CMMD_sync_done()
```

- Performance enhancements

  The performance of single-precision floating-point strided scans and reductions has improved (as double-precision performance improved in CMMD 3.1). Timings shown here are for additive combining; scans are unsegmented and exclusive.

Copyright © 1994 Thinking Machines Corporation

Function, data type	3.0 peak bw	3.2 peak bw
CMMD_reduce_v, float	0.156 Mb/s	1.250Mb/s
CMMD_scan_v, float	0.104 Mb/s	1.470Mb/s
CMMD_scan_v, double	0.301 Mb/s	1.938 Mb/s

The function **CMMD_scan_double_v** is now deprecated. Instead, the performance of **CMMD_scan_v** with a double-precision combiner has been enhanced (without **CMMD_scan_double_v**'s restriction as to other pending messages).

Changes to the CMMD Error Reporting System

The name for one of the error codes returned by **CMMD_get_errno**, **CMMD_ERR_NO_CHANNELS**, has been changed to **CMMD_ERR_NO_RPORTS**, to more closely reflect the resource dearth that is being reported.

Also, a new function, **CMMD_error_string**, has been added. This function takes a value to which errno has been set by the CMMD IO routines, and returns a string which describes the error in question. The C prototype for this function is

```
char *CMMD_error_string(int errno)
```

and the Fortran declaration is:

```
      external cmmd_error_string
character*(512) cmmd_error_string
```

Improved Error Checking and Use of /tmp in cmmd−ld

There are several minor improvements that have been made to the **cmmd-ld** linking script:

- Temporary files are now put in **/tmp**. There is a new switch to **cmmd-ld**, **-temp <tmpdir>**, to use to specify a different location for temporary files. This may be necessary if, for example, you do not have enough space in **/tmp**.
- Temporary files are cleaned up in several cases in which they had previously failed to be.
- Several new error checks for incompatible arguments are done.

Copyright © 1994 Thinking Machines Corporation

Support of Third-Party Link Time Tools

CMMD now supports the use of third-party link time tools. As an example, PureLink, from Pure Software, can be used to reduce the link time required by CMMD programs. Use of such tools is controlled by the shell environment variable, **CMMD_LD_LINKTOOL**. If the variable is set to the name of the tool, **cmmd-ld** will invoke the tool as part of each link.

Examples:

```
setenv CMMD_LD_LINKTOOL purelink
```

or, in a makefile:

```
$(target):    $(dependencies)
    CMMD_LD_LINKTOOL=purelink; export CMMD_LD_LINKTOOL;
\
    $(CMMD_LD) -comp $(CC) $(CFLAGS) $(dependencies) \
         -o $(target) ...
```

Either form will result in

```
purelink ld -e cmmd_start ...
```

Note that any flags or paths to be passed to the linktool should be included in the assignment to the environment variable. For example,

```
setenv CMMD_LD_LINKTOOL "/usr/local/bin/purelink -banner=0"
```

Support for Apogee Compilers

The **cmmd_ld** script now allows the use of the Apogee-C and Apogee-FORTRAN 77 compilers (products of Apogee Software, Inc.)

Correction of CMAML_allocate_this_tag Semantics

The *CMMD Reference Manual* currently states that the function

```
int CMAML_allocate_this_tag(int i)
```

returns **TRUE** if it succeeds in allocating tag i, otherwise it returns **FALSE**. The current implementation of **CMAML_allocate_this_tag()** has a bug: it returns **TRUE** upon success, otherwise it returns **CMAML_ERROR**, which is different from any tag, but nevertheless logically equivalent to **TRUE**. This is

Copyright © 1994 Thinking Machines Corporation

fixed in CMMD 3.2. However, some users may have already noticed it and coded around it (by checking the return value for equality with **CMAML_ERROR** rather than for logical equivalence to **TRUE** or **FALSE**); they must change their code to conform to the new, correct functionality. The following is an example of how to do this (a function that attempts to allocate the same tag on all nodes):

```
#include <stdio.h>
#include <cm/cmmd.h>

int allocate_same_tag_on_all_nodes()
{
  int i;

  for( i = 0; i < CMAML_NUSER_HWTAGS; i++ ) {
    if( !CMMD_reduce_int( !CMAML_allocate_this_tag( i ),
                  CMMD_combiner_ior)) {
      return i;
    }
  }

  if ( i == CMAML_NUSER_HWTAGS )
    return -1;
}
```

Send and Receive Slots

There are now two functions that return the number of pending sends and receives that a node is allowed to initiate. This corresponds to the communications resources available for sending or receiving at the time the function is called on the node on which it is called. The functions are

```
int CMMD_available_sends();
int CMMD_available_receives();
```

For CMAML hackers, note that **CMMD_available_receives() == CMAML_number_of_free_rports()**.

Change to Meaning of −cmmd_root

Previously, the argument of the **cmmd_root** flag to **cmmd-ld** corresponded to the root directory for a default installation of **cmmd** (i.e., **-cmmd_root**

Copyright © 1994 Thinking Machines Corporation

/foo told **cmmd-ld** to look for its libraries in **/foo/usr/lib**). As of this release of CMMD, the argument to the **cmmd_root** flag will correspond to the **/usr** directory in a default installation of cmmd, so that specifying **-cmmd_root /foo** will tell **cmmd-ld** to look for its libraries in **/foo/lib**.

This is of special interest to users of nodal CM Fortran, as that program also provides a **-cmmd_root** flag. Version 2.1.1-2 of CM Fortran understands the above sense of **-cmmd_root**, but versions of CM Fortran previous to this one do not.

Vector Unit I/O

The **read()** and **write()** system calls now support vector unit (VU) addresses as buffers. The address passed may be data or instruction on any of the VUs, but it must be word-aligned. The buffer data is assumed to start on VU 0. The length must be a multiple of 16. The distribution of bytes across the 4 VUs is:

VU 0: bytes 0, . . . , 15, 64, . . . , 79, . . .
VU 1: bytes 16, . . . , 31, 80, . . . , 95, . . .
VU 2: bytes 32, . . . , 47, 96, . . . , 111, . . .
VU 3: bytes 48, . . . , 63, 112, . . . , 127, . . .

I/O performance on VU data can be up to two times slower than the performance on SPARC data. There are plans to improve this in a later version.

Global/Local Support

This version of CMMD supports global/local CM Fortran. Global/local programming allows global CM Fortran programs to take advantage of message-passing programming techniques. Thus, it allows the unification of the global and local (or nodal) views of the CM within a single program. A global/local application begins with a global **MAIN** routine, written in CM Fortran, executing in data parallel fashion: that is, laying out its arrays across the VUs of an entire partition and operating on those arrays in a global, synchronous fashion, with the compiler and run-time system taking care of communication and synchronization.

When the application wishes to take explicit control of the nodes, it calls a "local routine." Invoking the local routine temporarily transforms the

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

application into a nodal program, executing in message-passing style. During local routines, each node operates on its own "subarrays" of global parallel arrays; nodes use CMMD calls for synchronization and communication.

At this preliminary release, the global portions of global/local applications must be written in CM Fortran; the local portions may be written either in CM Fortran or in C.

Global/local programs can be run only on Connection Machine CM-5 systems equipped with vector units (VUs).

For further information, see the *CM Fortran User's Guide,* Version 2.1, and the *CM Fortran Programming Guide,* Version 2.1.

Improved CMMD Global Function Performance

The peak performance of the CMMD global functions has been markedly improved, in some cases by as much as a factor of five.

3 Documentation

The reference manual and user's guide for Version 3.2 are the same as for Version 3.0. The usual on-line manual pages are also available.

These release notes are provided on-line, in the file

```
/usr/doc/cmmd-3.2.releasenotes
```

Information on known and fixed bugs is also provided on-line; see the file

```
/usr/doc/cmmd-3.2.bugupdate
```

(The pathnames may be different at your site; contact your system administrator if you cannot find the files at these default locations.)

Copyright © 1994 Thinking Machines Corporation

4 Known Problems and Workarounds

This section describes some problems and non-intuitive issues that exist in CMMD. Also see the CMMD Bug Update (refer to "Documentation" above) for other known bugs.

4.1 Using Hardware Tag Handlers

Note that hardware tag handlers on Cypress platforms must be no longer than 64 machine instructions. If this restriction is violated the results are unpredictable and most likely will result in a segmentation fault or a bus error. If you wish to register a hardware tag handler that is longer than 64 instructions, you can register a short stub handler that simply calls the desired handler. This restriction does not apply on SPARC superscalar platforms.

Also note that the interface to data router tag handlers is subject to change without notice from one release to another. Programmers coding at this level should be prepared to modify their code in future releases.

4.2 CMMD's Emulation of the UNIX sendto System Call

The **`sendto()`** system call is restricted to a *len* argument which is a multiple of the word size. In other words, the value must be a multiple of four. A *len* that does not meet this restriction will result in **`sendto()`** returning **`-1`** and *errno* being set to **`EINVAL`**.

5 Writing Version-Independent Code

There is a C preprocessor symbol, **`CMMD_VERSION`**, that you can use to make your code compatible with both Version 2.0 and Version 3.1-Final73B1 of CMMD. Version 2.0 defines it as

```
#define CMMD_VERSION 20
```


Copyright © 1994 Thinking Machines Corporation

Version 3.0 through Version 3.1 define it as

```
#define CMMD_VERSION 30
```

and Version 3.2 defines it as

```
#define CMMD_VERSION 32
```

The definition of **CMMD_VERSION** as 30 in Version 3.1 of CMMD was an oversight.

This definition allows you to include conditional compilation constructs such as the following in your code:

```
#include <cm/cmmd.h>
#if CMMD_VERSION == 20
#include <cm/cmmd-io.h>
#endif
```


Copyright © 1994 Thinking Machines Corporation