Chapter 10

Miscellaneous Performance Issues

10.1 The CMSSL Library

The CMSSL library contains many efficient implementations of parallel arithmetic and communication operators.

To use the CMSSL library, you must put the following **INCLUDE** statement in each routine that calls it:

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

To link with CMSSL, specify it on the **cmf** command line as follows:

```
% cmf -cm5 -vu program.fcm -lcmssl
```

10.1.1 CMSSL Communication Routines

The CMSSL library provides these types of communication operators:

- polyshift (efficient CSHIFT-like operations on multiple arrays)
- all-to-all rotation, broadcast and reduction
- matrix transpose
- sparse gather and scatter
- vector move (extract and deposit)
- array permutations along an axis
- general communication compiler

Copyright © 1994 Thinking Machines Corporation

Most of the CMSSL communication routines operate in a standard pattern. For a given *cmssl_function*, the following steps are used:

- An initial call is made to a function named *cmssl_function*`_setup`, which records information about the array layouts and sizes to be used in subsequent function calls. This information is saved in a data structure called a *trace*, so that it does not need to be recomputed.
- One or more calls are made to the actual *cmssl_function*, which takes the trace data structure as an argument. Having the precomputed trace information about the arrays being used and the communication patterns required means that *cmssl_function* can execute much more efficiently.
- Finally, a function call to `deallocate_`*cmssl_function*`_setup` is used to free the memory space taken up by the trace data structure.

The extra overhead required in computing and saving the trace data structure is distributed across the number of calls made to *cmssl_function*; the more calls to *cmssl_function* that are made using the same trace, the better the performance compared to (for example) a CMRTS function that does not store and use traces in this fashion. However, for single calls to *cmssl_function*, the overhead of the trace data structure outweighs the extra efficiency of the function. Thus, CMSSL functions are primarily useful for repeatedly moving data between large arrays.

The performance characteristics of the various CMSSL operation categories are described below. For more information, consult the manual *CMSSL for CM Fortran*.

Polyshift Operation

The CMSSL operator

```
pshift
```

effectively does a `CSHIFT` or `EOSHIFT` of the contents of a set of similarly shaped arrays, with each array being shifted a user-specified distance along a specified axis. The `pshift` function can be used to efficiently implement *n*-way stencils.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

All-to-All Operations

The CMSSL operators

all_to_all (rotation)
all_to_all_reduce
all_to_all_broadcast

perform essentially the same function as the CM Fortran intrinsics for global reduction and spreading of data (**SUM**, **PRODUCT**, **COUNT**, **SPREAD**, etc.). The CMSSL routines give better performance results for large arrays than their CM Fortran intrinsic counterparts.

Matrix Transpose

The CMSSL operator

gen_matrix_transpose

exchanges two axes of a multidimensional array, and can be used in place of a **FORALL** statement or the intrinsic function **transpose**. The axis lengths of the source and destination arrays are kept identical except for the two exchanged axes. Currently, this routine gives enhanced performance only when the axes to be exchanged are a serial and a non-serial axis.

The **gen_matrix_transpose** routine can improve performance when you carry out specially optimized local routines such as FFTs (that is, routines that operate best on serial axes). You can transpose each non-serial axis of an array into a serial axis, perform the local routine (FFT, etc.) on it, and then transpose the axis back in place.

Sparse Gather/Scatter

The CMSSL operators

sparse_util_[vec_]gather/scatter
part_[vector_]gather/scatter
block_gather/scatter

move values between arrays of differing rank (for example, vectors and 2D arrays). The **sparse** and **part** (partitioned) functions use an algorithm that is efficient for cases where only a few elements of the larger-ranked array

Copyright © 1994 Thinking Machines Corporation

send or receive values (sparse matrixes, for example). The **block** functions are efficient for moving elements along serial axes.

These routines can help speed up **send** and **get** general communication by distributing data values so that there are fewer collisions between elements, thus reducing network congestion. (In particular, these routines can speed up general communication operations with a "fan out" of 500 or 1000 sends per array element.)

Vector Move (Extract and Deposit)

The CMSSL operator

vector_move

moves a specified vector from each subgrid of a source array into a specified vector location in each subgrid of the destination array. The two arrays must have the same rank and physical distribution over processors, but need not be the same shape. The **vector_move** function can be used to extract or replace edges or faces of the local portion of an array, to isolate elements involved in interprocessor communication (as in a stencil operation).

This routine can help you to improve performance by separating the in-processor and interprocessor parts of a communication operation that would otherwise require inefficient general communication calls.

Array Permutations

The CMSSL operator

permute_cm_matrix_axis_from_fe

efficiently permutes the rows or columns of one or more matrices, using a permutation that is specified by a front-end array.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

Communication Compiler

The CMSSL operators

```
comm_get
comm_send
```

allow you to establish a trace of an arbitrary general communication operation. Repeated use of this same general communication pattern is made much more efficient than the use of ordinary **send** and **get** operations.

10.1.2 Additional CMSSL Optimizations

The compiler treats most utility library procedures as external procedures, generating the appropriate calls to RTS communication routines. Certain procedures are specially optimized if you link in the CMSSL software library:

MATMUL (on floating-point data only)
CMF_RANDOM **CMF_SORT** **CMF_ORDER** **CMF_RANK**

Beginning with CMSSL Version 3.2, **CMF_RANDOM_LONG_S_INTEGER** is also specially optimized with CMSSL is linked in. Note that 64-bit integers are supported in the VU model only.

10.2 Alternate Programming Models

This section describes the alternate programming models that you can use in compiling and running a CM Fortran program. These are

- The *nodal programming model*, described in the *CM Fortran Programming Guide*.
- The *global/local programming model*, described in the *CM Fortran Libraries Reference Manual* and the *CM Fortran Programming Guide*.

Summaries of the effects of each programming model, as well as their general performance impacts, are presented below.

Copyright © 1994 Thinking Machines Corporation

10.2.1 Nodal Programming

When a CM Fortran program is compiled using the nodal model, the program executes in multiple copies, one for each PN in the CM. Each PN acts as a partition manager in its own right, with its four associated VUs acting as PEs. Each processing node executes the entire CM Fortran program independently of the other nodes; each node also operates only on the parallel data accessible through its four VUs.

Compiling a program under the nodal model automatically links in the CMMD message-passing library; individual nodes can use CMMD message-passing calls to exchange data with other nodes. A full discussion of the CMMD library's use and performance characteristics is beyond the scope of this manual; for more information, see the *CMMD User's Guide* and *CMMD Reference Manual*.

10.2.2 Global/Local Programming

When a CM Fortran program is compiled using the global/local model, specific subroutines are specially compiled as *local* subroutines. The bulk of the program executes normally from the PM, allocating parallel arrays across the memory of the PEs and performing parallel computation and communication as described in this manual.

The local subroutines, however, execute much like the nodal programming model described above; when a local subroutine is called, each PN becomes (for the duration of the subroutine call) a PM in its own right, with its four associated VUs acting as PEs. Each processing node executes the subroutine independently of the others, and operates only on the parallel data accessible through its four VUs.

A "global" parallel array, spanning the entire CM, is seen by a local subroutine to be only that part of the array that is local to each processing node. In essence, calling a local subroutine transparently divides a global array into node-sized subarrays, which can be accessed as usual.

Local routines can also define "local" parallel arrays that exist only within the memory of the four VUs on each node. These local arrays are distinct from each other, and do not automatically occupy the same memory region on each node, as do global parallel arrays.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

Local routines can also make calls to the CMMD message-passing library to exchange messages; see the *CM Fortran Programming Guide* and the CMMD documentation for more information.

10.2.3 Performance Effects

In general, using nodal or global/local programming is just a change of programming style. The performance information and advice given in this manual applies unchanged to the global portion of a global/local program. It also generally applies to nodal programs and local subroutines, if you remember that each processing node operates independently, can define its own arrays, and in general executes like a four-node CM-5.

One notable performance effect of the nodal and global/local programming models is that all execution takes place on the PNs. Thus, the startup time of a PE code block is much lower, because there is no need for a CMRTS function call to send information from the PM to the nodes.

For example, the PE code block timing program of Section 7.1.4 produces the following output when compiled and run under the nodal programming model:

```
PE startup time: approx.   9.9844718375207E-002 usecs.
PE loop time: approx.   13.49552567431133 usecs.
Per element: approx.   1.349552567431133E-002 usecs.
```

Compare these times with the results from running the program under the normal global model:

```
PE startup time: approx.  0.3794040531803398 usecs.
PE loop time: approx.   13.76751340314921 usecs.
Per element: approx.   1.376751340314921E-002 usecs.
```

The PE loop time and the time per element are virtually the same, but the PE startup time is about one-fourth as long.

Copyright © 1994 Thinking Machines Corporation