CM Fortran Release Notes

1 About Version 2.1

Version 2.1 enhances both the functionality and the performance of CM Fortran.

This document provides full information on all new features and optimizations (pending the completion of the Version 2.1 documentation set). See the on-line ASCII release notes for information on current documentation, on porting code from previous versions, and on restrictions imposed and removed.

1.1 Release Highlights

- Special compiler optimizations on serial dimensions, which avoid communication and otherwise speed up many local operations.
- Features that give further program control over array layout, including facilities for aliasing arrays of different shapes or layouts, and compiler switches that inhibit axis reordering and local vector-length padding. These features enable a program to get the full benefit of the serial optimizations.
- Two new execution models — nodal and global/local — which move program control to the node level and use CMMD message passing for explicit communication and synchronization. The global/local model provides the means to call node-local subroutines from within a global program.

Copyright © 1993 Thinking Machines Corporation

- Parallel I/O to the Scalable Disk Array and DataVault via the **READ** and **WRITE** statements, and many other enhancements to CM Fortran I/O (including the keyword **PHYSICAL** for the fastest possible parallel I/O).
- Optimizations of **FORALL**, of single-precision memory accesses, of the global optimizer **-O**, and of operations on dynamically allocated arrays.
- Fortran 90 array pointers (though not pointer assignment) for dynamic management of global storage.
- The 64-bit integer data type and the Fortran 90 **KIND** mechanism for managing the "kind," meaning length in bits, of all numeric types.

1.2 CM Platforms and Execution Models

Version 2.1 supports all CM platforms and offers two new execution models that permit node-local programming with message passing. The supported execution models are:

- CM-5 release of the compiler
  - global vector-units model (**-vu**)
  - global SPARC-nodes model (**-sparc**)
  - nodal CM Fortran (**-vu -node ...**)
  - global/local CM Fortran (**-vu ... -local**)
  - CM Fortran simulator (**-cmsim**)
- CM-2/200 release of the compiler
  - slicewise model (**-slicewise**)
  - CM Fortran simulator (**-cmsim**)

Version 2.1 does not support the Paris execution model on CM-2/200 (**-paris**).

Copyright © 1993 Thinking Machines Corporation

1.3 New Features by Execution Model

This section lists all the new features and major compiler optimizations provided in Version 2.1. All these items are available on the CM-5 global vector-units model (**-vu**). Some features do not support all execution models, as noted below.

NOTE: The nodal and global/local models are variants of the global vector units model and share all its features except I/O. For I/O from nodal CM Fortran, use CMMD I/O. Global/local programs (using CM Fortran Version 2.1 and CMMD Versions 3.1 or 3.2) require that all I/O be done from a global program unit, using any of the CM Fortran I/O facilities; no I/O is permitted from local subroutines.

Key: yes = feature is supported
– = feature has no effect
no = feature is not accepted (program fails)

Features	CM-5		CM-2	Sun
	–vu	**–sparc**	**–slice**	**–cmsim**
▪ optimizations on serial dimensions - no-motion detection - nested subgrid loop, perhaps flattened - context handling in serial sections - simple dependence analysis - contiguous sections passed in place - gathers/scatters with **FORALL** - spreads with **FORALL** or **SPREAD**	yes	yes	–	–
▪ user control over array layout				
- array aliasing package (14 procedures)	yes	yes	yes	–
- controllable axis sequence, **-[no]axis**	yes	yes	no	no
- controllable local padding, **-[no]pad**	yes	–	no	no
▪ node-level programming with message passing - the nodal execution model - the global/local execution model - the CMGL library procedures	yes	no	no	no

Copyright © 1993 Thinking Machines Corporation

Features	CM-5		CM-2	Sun
	−vu	−sparc	−slice	−cmsim
▪ optimizations				
- global optimizer improvements	yes	yes	yes	yes
- geometry lookups in subprograms	yes	yes	yes	–
- run-time check of dynamic array geoms	yes	yes	yes	–
- compile-time check of dynamic array shapes	yes	yes	yes	–
- **FORALL** scans parallelized	yes	yes	–	–
- single-precision **REAL** stores	yes	–	–	–
- single-precision **COMPLEX** loads/stores	yes	–	–	–
▪ I/O enhancements in CM Fortran language				
- parallel **READ/WRITE** to SDA	yes	yes	no	–
- **READ/WRITE** character data to SDA	yes	yes	no	yes
- parallel **READ/WRITE** to DataVault	yes	yes	yes	–
- **READ/WRITE** character data to DataVault	yes	yes	no	–
- **PHYSICAL** keyword for **OPEN**	yes	yes	yes	–
- 4 new specifiers to **OPEN** and **INQUIRE**	yes	yes	yes	yes
- new **NAMELIST** syntax	yes	yes	yes	yes
- **REWIND/BACKSPACE** on redirected output	yes	yes	yes	yes
▪ I/O enhancements in CM Fortran utility lib				
- SDA support via utility lib I/O	yes	yes	no	no
- utility **CMF_FILE_OPEN_READONLY**	yes	yes	yes	no

Copyright © 1993 Thinking Machines Corporation

Features	CM-5		CM-2	Sun
	−vu	−sparc	−slice	−cmsim
▪ array pointers - **POINTER** attribute - **POINTER** statement - enhanced **ALLOCATE** and **DEALLOCATE** - **ASSOCIATED** intrinsic function	yes	yes	yes	yes
▪ the Fortran 90 **KIND** mechanism - **KIND** keyword - kind type parameters (except **_DOUBLE_INT_**) - syntax for typed constants - **KIND** intrinsic function	yes	yes	yes	yes
▪ 64-bit integers - **INTEGER*8** data type - kind type parameter **_DOUBLE_INT_** - utility **CMF_RANDOM_LONG_S_INTEGER** - utility **CMF_FILE_LLSEEK** - utility **CMF_FILE_LTRUNCATE** - arg parameter **CMF_LONG_S_INTEGER** - optional **KIND** arg to **INT** and **NINT** - **INTEGER*8** send-address arrays	yes	no	no	no

Copyright © 1993 Thinking Machines Corporation

1.4 Documentation Corrections

1.4.1 Fortran READ/WRITE and the SO Utilities

This note corrects *Technical Bulletin* CMF 2.1-Beta-002, June 28, 1993.

That bulletin states that the CM Fortran utility library I/O procedures **CMF_CM_ARRAY_TO/FROM_FILE_SO** are compatible with Fortran direct access I/O (**READ/WRITE** with **ACCESS=DIRECT**). That is not correct. Mixing these two I/O mechanisms sometimes causes an end-of-file error and is not supported.

To access files written with Fortran 77 **WRITE**, please use CM Fortran **READ**. CM Fortran **READ/WRITE** are fully compatible with Fortran 77 **READ/WRITE**, as well as now supporting parallel I/O.

1.4.2 Passing an Alias Descriptor to a Subprogram

This note corrects the CM Fortran V2.1 Beta 2.0 *Release Notes: Detailed* and that version's man page for **CMF_ALIAS_DETAILED**.

The discussion of that procedure states that the descriptor of a detailed alias must be passed to a subprogram that declares its layout with an assumed **LAYOUT** directive (all colons). That is not correct. You may pass an alias descriptor to a dummy argument with an explicit **LAYOUT** directive. The directive must, of course, be compatible with the layout information provided in the call to **CMF_ALIAS_DETAILED**.

If the dummy argument is assumed-layout, the behavior is correct but the compiler does not apply the serial optimizations (since the dummy's layout is not known until run time). The compiler does apply the serial optimizations to an array alias as long as the criteria stated below in Section 2.1 of these release notes are met.

Copyright © 1993 Thinking Machines Corporation

1.5 Notice of Possible Future Changes

1.5.1 Phasing Out Support for Foreign Compilers

The CM Fortran compiler accepts the following filename extensions, which it passes on to Sun Microsystem's **f77** or **cc** compiler or to the Connection Machine **cs** (C*) compiler:

```
.f
.F
.c
.cs
```

In later versions of CM Fortran, the compiler will exit with an error when it encounters one of these filename extensions. Please phase out your use of these extensions with the **cmf** compiler. The compiler issues a warning like the following when it encounters them:

```
% cmf myfile.f
cmf: myfile.f: Warning: future versions of cmf will
not process .f files.
```

1.5.2 Phasing Out Linking with Sun f77 Libraries

CM Fortran automatically links with Sun libraries if they are installed on the system. This behavior will be discontinued in later versions of CM Fortran.

When a program that was linked with Sun's dynamically bound libraries does not find them at run time, an error like the following occurs:

```
ld.so: libF77.so.1: not found
```

You can prevent this problem by linking statically by means of the **-Bstatic** switch, which **cmf** passes on to the linker:

```
% cmf myfile.fcm -Bstatic
```

1.5.3 Axis Ordering in Memory

The current default axis ordering, **-axisreorder**, exists in the compiler for historical reasons. The **-noaxisreorder** switch produces the standard For-

Copyright © 1993 Thinking Machines Corporation

tran left-to-right ordering. In the future, the left-to-right ordering may become the default. We encourage its use.

NOTE: **-axisreorder** affects only CM arrays. Front-end array axes are always stored in left-to-right order, regardless of the setting of this switch.

1.5.4 Vector-Length Padding

Similarly, the current default padding of the (non-serial) subgrid to a multiple of vector length also exists for historical reasons. (The compiler did not originally generate clean-up code for loops in which the number of times the loop is executed is not a multiple of the vector length.) In the future, **-nopadding** may replace **-padding** as the default. We encourage its use.

1.5.5 Compatibility

The previous defaults for **-[no]axisreorder** and **-[no]padding** have been retained in this release for back-compatibility. If the defaults change, or if you begin to switch over to the non-default options, consider compatibility issues like the following:

- Suppose you have a library compiled under the current defaults (**-axisreorder** and **-padding**). In the future, when the defaults change, you may try to call this library from a newly compiled procedure (with **-noaxisreorder** and **-nopadding**). The program will fail because all program units in a program must be compiled with the same setting of these switches.
- Suppose you have an array **A** laid out **(:SERIAL,:NEWS)** and you pass **A(1,:)** to a subroutine. With the current default **-axisreorder**, this section is passed in place. In the future, with the **-noaxisreorder** default, the section would first be sent to a temporary location, thus slowing down the program.

Copyright © 1993 Thinking Machines Corporation

2 Optimizations on Serial Dimensions

A *serial* array dimension is guaranteed to be laid out in the memory of a single processor. (*Processor* refers to a node or vector unit, depending on the execution model). The compiler now recognizes many operations on serial dimensions as local and thus generates more efficient code than in releases before 2.1.

This section describes the array dimensions that the compiler recognizes as serial and the ways in which it speeds up operations on them.

- no-motion detection on serial axes
- nested and flattened loops
- better handling of context on serial sections
- simple dependence analysis
- contiguous serial sections passed in place
- gathers/scatters with **`FORALL`**
- spreads with **`FORALL`** or **`SPREAD`**

The new features that give users greater control over array layout — detailed layout syntax, control of vector-length padding, and control of axis sequence in memory, all described below in Section 3 — enable you to tune your program to take maximum advantage of the serial optimizations. See the end of this document for an example of a 5-point stencil program that does this.

Copyright © 1993 Thinking Machines Corporation

2.1 Recognizing Serial Dimensions

The compiler applies the serial optimizations to axes that are declared **:SERIAL** in a **LAYOUT** directive, as long as the array is not deferred-shape (allocatable or pointer).

The compiler also recognizes axes described with :**PROCS=1** or :**PDESC=0** in a detailed **LAYOUT** directive as local for the purpose of applying the serial optimizations, as long as the array is not deferred-shape and the program is compiled under the (non-default) **-noaxisreorder**. Under the default **-axisreorder**, only **:SERIAL** axes are recognized as local. Note that every program unit in a program must be compiled with the same setting of **-[no]axisreorder.**

The optimizations are not applied to deferred-shape arrays in program units where such arrays have the attribute **ALLOCATABLE** or **POINTER**. If a deferred-shape array is passed to a subprogram whose dummy array does not have the **POINTER** attribute and one or more axes are declared **:SERIAL** (or, under **-noaxisreorder**, :**PROCS=1** or **PDESC=0**), the array benefits from serial optimizations in the subprogram.

The compiler also recognizes local axes in arrays laid out with **CMF_ALLOCATE_DETAILED_ARRAY** or with **CMF_ALIAS_DETAILED**, since these arrays must be passed to a subprogram that declares them as CM arrays whose layout is compatible with the descriptor created in the caller. As long as the **LAYOUT** directive in the subprogram meets the above requirements for recognizing local axes, the compiler applies the serial optimizations. If the subprogram uses an assumed **LAYOUT** directive (colons only) or uses detailed **LAYOUT** syntax under **-axisreorder**, the serial optimizations are not applied.

In addition, the compiler applies the serial optimizations to an array that is **ALIGN**'ed with an array whose local dimensions are recognized as such.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

2.2 No-Motion Detection

In previous versions, the compiler treated data motion along serial dimensions in the same way as data motion along parallel dimensions: by generating run-time communication routines. The compiler now recognizes array section assignments on serial dimensions as on-processor, and generates **DO** loops with local memory accesses instead. This change can increase performance by orders of magnitude.

For example, consider these two arrays:

```
      REAL A(100,100),B(100,100)
CMF$  LAYOUT A(:SERIAL,:NEWS),B(:SERIAL,:NEWS)
```

The **LAYOUT** directive ensures that all the elements of the first dimensions of **A** and **B** are co-resident on a processor. Therefore, even if the elements referenced in triplets on this dimension are different, the operation is local:

```
A(1:2,:) = B(3:4,:)
```

In previous versions, this statement generated NEWS communication. Now it generates nested loops, one from 1 to 2 and one from 1 to the on-processor extent of the NEWS axis. This should be several hundred times faster.

Similarly, a stencil expressed as the assignment of on-processor array sections is now much faster:

```
      REAL D(100,100,100),E(100,100,100)
CMF$  LAYOUT D(:SERIAL,:SERIAL,),E(:SERIAL,:SERIAL,)

      FORALL(I=1:N,J=1:M)
     $    D(I,J,:) =
     $    E(I+1,J,:)+E(I,J+1,:)+
     $    E(I-1,J,:)+E(I,J-1,:)+E(I,J,:)
```

A consequence of this enhancement is that it is no longer necessary to write unrolled **DO** loops to express local assignments. You can use the more convenient array notation.

Copyright © 1993 Thinking Machines Corporation

Some forms of data motion on serial dimensions are not optimized in this way. These include:

- References to the intrinsic functions **CSHIFT** and **EOSHIFT**.
- Array transformations, including reductions, whether expressed explicitly via the intrinsic functions or implicitly via **FORALL**.
- Transpositions generated by **FORALL** or by the intrinsic **TRANSPOSE.** For example, suppose **A(N,N)** is **(:SERIAL,:NEWS)** and **B(N,N)** is **(:NEWS,:SERIAL)**. Then,

  ```
  FORALL (I=1:N, J=1:N)  A(I,J) = B(J,I)
  ```

  will continue to generate a run-time transpose routine.

Any off-processor communication in a statement disables the serial optimizations for that statement, since the compiler does not fission its internal loops to isolate the local operations. Examples of off-processor operations include communication along a parallel dimension:

```
      REAL A(100,100),B(100,100)
CMF$  LAYOUT A(:SERIAL,:NEWS),B(:SERIAL,:NEWS)

      A(1:2,1:2) = B(3:4,3:4)
```

and communication between arrays that are not aligned in distributed memory:

```
      REAL A(100,100),C(100,500)
CMF$  LAYOUT A(:SERIAL,:NEWS),C(:SERIAL,:NEWS)

      A(1:50,:) = C(1:50,1:100)
```


Copyright © 1993 Thinking Machines Corporation

2.3 Nested and Flattened Loops

In previous releases, a local code block could only be a single, unit-stride loop over the entire subgrid. In 2.1, the compiler can generate code blocks with strided loops and can generate a sequence of perfectly nested loops.

For example, suppose **A(N,N,M)** is **(:SERIAL,:SERIAL,:NEWS)**. Then, the assignment

```
A(2:6,3:7,:) = 1
```

can be accomplished with one triply nested loop inside a code block. In earlier versions, this was done with a single unit-stride loop traversing all of **A**, or the user could write an explicit doubly nested **DO** loop:

```
DO I = 2,6
  DO J = 3,7
    A(I,J,:) = 1
  END DO
END DO
```

Of course, the user can still write the double **DO** loop, but it turns into a double loop of PN code blocks, and so incurs much higher overhead than the single code block containing a triply nested loop.

The innermost loop of a loop nest is always the one with shortest stride in memory. The compiler flattens the loop nest where possible, but does not yet take advantage of **(:BLOCK=1)** declarations. An example of flattening, assuming **A(:NEWS,:SERIAL), B(:SERIAL,:NEWS)** and the **-noaxisreorder** switch (described below):

`A(:,I:J) = 1` generates a single loop (the double loop is flattened)

`B(I:J,:) = 1` generates a double loop (the double loop is not flattened)

The innermost loop is vectorized under the **-vu** execution model. The innermost loop usually corresponds to the innermost (unit-stride) axis in memory. If the loop is flattened, however, more than one axis is “the” vector axis.

Copyright © 1993 Thinking Machines Corporation

2.4 Better Handling of Context

Assignments of serial sections are also speeded up because the compiler stores only what it computes. For example, consider:

```
      REAL A(100,100)
CMF$  LAYOUT A(:SERIAL,)

      A(1:2,:) = 0
```

In previous versions, this statement looped over all the elements of **A**, masking out stores to elements that were not in the first two rows. Now, it generates nested loops, one from 1 to 2 and one from 1 to the on-processor extent of the **:NEWS** axis. This should be about 50 times faster.

Similarly, strided expressions on serial axes, such as

```
      A(L:U:S,:) = 0
```

now perform the indicated number of operations instead of operating on the whole array.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

2.5 Contiguous Serial Sections Passed in Place

An array section that is specified by scalar indices or unit-stride triplets on one or more serial axes, and which defines a contiguous region of memory, is passed in place. It can also be operated upon with aligned portions of other arrays without communication. For example, with the default **-axisreorder**, the following operations involve no communication:

```
      REAL A(10,100,100), B(100,100), C(3,100,100)
CMF$  LAYOUT A(:SERIAL,,), C(:SERIAL,,)

      A(3,:,:) = B              ! no communication
      A(1:3,:,:) = C            ! no communication
      CALL FOO (A(4,:,:))       ! argument passed in place
```

The first example has always been recognized as not requiring communication, since the two arrays are aligned along their parallel axes. Previous versions required that all serial axes be referenced with scalar subscripts, but that restriction no longer applies as long as the section is contiguous.

The second example, with a triplet on the serial axis, is also recognized by this version of the compiler as a contiguous section.

The subroutine call would previously have required an expensive **SEND** of the array section to a canonical location before it was passed as an argument. This version of the compiler recognizes that the section is a contiguous region of memory, and no **SEND** is generated.

Communication is generated when operating upon or passing non-contiguous array sections.

To determine whether a section is contiguous, you must recognize the axis whose subscripts vary fastest in memory, which axis varies next fastest, and so on. The simplest way to do this is to check the output of **CMF_DESCRIBE_ARRAY**. The field "`subgrid axis increment`" is the speed at which an axis varies: the axis with increment 1 varies fastest, and larger numbers mean slower varying.

You can also determine which axes are faster or slower varying by considering the setting of the **cmf** switch **-[no]axisreorder** (described below in Section 3.1). With **-noaxisreorder**, the leftmost declared axis is fastest varying, the next leftmost next fastest, and so on. Under **-axisreorder** (the default), all **:NEWS** axes vary faster than any **:SERIAL** axis, the rightmost **:NEWS** axis varies faster than other **:NEWS** axes, and the rightmost **:SERIAL** axis varies faster than other **:SERIAL** axes.

Copyright © 1993 Thinking Machines Corporation

Once you know the relative speed with which axis subscripts vary in memory, apply the following rules to determine whether any given array section is contiguous. Complete axes in an array section are those referenced simply as `:`; incomplete axes are those referenced with a scalar or triplet subscript.

A CM array section is contiguous in memory when every incomplete axis is:

1A Local to a processor, meaning described as **`:SERIAL`** or, in detailed **`LAYOUT`** under **`-noaxisreorder`**, as :**`PROCS=1`** or **`:PDESC=0`**.

1B Contiguous on that processor, meaning referenced with either a scalar index or a unit-stride triplet (not a strided triplet).

And in addition:

2 Starting from the axis whose subscripts vary slowest in memory and working toward the fastest varying, there may be an arbitrary number of scalar-indexed axes, followed by at most one incomplete unit-stride triplet axis, followed by all complete axes.

For example, consider the sections referenced in the code shown opposite.

Array	Layout	Switch	Axes Speed	Section	In-Place?	Why
`(10,64,64)`	`(S,N,N)`	`-noaxisreorder`	(1,2,3)	`(2,:,:)`	No	2
`(10,64,64)`	`(S,N,N)`	`-axisreorder`	(3,2,1)	`(2,:,:)`	Yes	
`(64,10,64)`	`(N,S,N)`	`-noaxisreorder`	(1,2,3)	`(:,2,:)`	No	2
`(64,10,64)`	`(N,S,N)`	`-axisreorder`	(2,3,1)	`(:,2,:)`	Yes	
`(64,64,10)`	`(N,N,S)`	`-noaxisreorder`	(1,2,3)	`(:,:,2)`	Yes	
`(64,64,10)`	`(N,N,S)`	`-axisreorder`	(2,1,3)	`(:,:,2)`	Yes	
`(64,64,10)`	`(N,N,S)`	`-axisreorder`	(2,1,3)	`(:,:,1:4:2)`	No	1B
`(64,64,10)`	`(N,N,S)`	`-noaxisreorder`	(1,2,3)	`(:,2,:)`	No	1A
`(4,4,64,64)`	`(S,S,N,N)`	`-noaxisreorder`	(1,2,3,4)	`(1,:,:,:)`	No	2
`(4,4,64,64)`	`(S,S,N,N)`	`-axisreorder`	(4,3,2,1)	`(1,:,:,:)`	Yes	
`(4,4,64,64)`	`(S,S,N,N)`	`-noaxisreorder`	(1,2,3,4)	`(:,1,:,:)`	No	2
`(4,4,64,64)`	`(S,S,N,N)`	`-axisreorder`	(4,3,2,1)	`(:,1,:,:)`	No	2
`(64,4,64,4)`	`(N,S,N,S)`	`-noaxisreorder`	(1,2,3,4)	`(:,1:4,:,3)`	No	2
`(64,4,64,4)`	`(N,S,N,S)`	`-axisreorder`	(2,4,1,3)	`(:,1:4,:,3)`	No	2
`(64,4,64,4)`	`(N,S,N,S)`	`-axisreorder`	(2,4,1,3)	`(:,4,:,3:5)`	Yes	

Copyright © 1993 Thinking Machines Corporation

The program that references those sections is:

```
        SUBROUTINE THREE
        REAL A(10,64,64)
        REAL B(64,10,64)
        REAL C(64,64,10)

CMF$    LAYOUT A(:SERIAL,:NEWS,:NEWS)
CMF$    LAYOUT B(:NEWS,:SERIAL,:NEWS)
CMF$    LAYOUT C(:NEWS,:NEWS,:SERIAL)

        CALL X(A(2,:,:))
        CALL X(B(:,2,:))
        CALL X(C(:,:,2))

        CALL X(C(:,:,1:4:2))
        CALL X(C(:,2:,:))
        END


        SUBROUTINE FOUR
        REAL A(4,4,64,64)
        REAL B(64,4,64,4)

CMF$    LAYOUT A(:SERIAL,:SERIAL,:NEWS,:NEWS)
CMF$    LAYOUT B(:NEWS,:SERIAL,:NEWS,:SERIAL)

        CALL V(A(1,:,:,:))
        CALL V(A(:,1,:,:))

        CALL Z(B(:,1:4,:,3))
        CALL Z(B(:,4,:,3:5))

        END
```


Copyright © 1993 Thinking Machines Corporation

2.6 Simple Dependence Analysis

The compiler now performs some very simple dependence analysis to determine whether it can avoid copying the evaluation of the right-hand side of an assignment to a temporary before performing the assignment. Dependence analysis is also necessary to permit several statements to be merged into a single nested loop, thereby saving memory references.

Suppose **A(M,N)** is **(:SERIAL,:NEWS)**. The analysis is sufficient so that the assignments

```
A(1:M-1,:) = A(2:M,:)
```

```
A(2:M,:) = A(1:M-1,:)
```

can be done with in-place vector moves, the first running forward and the second running backward.

In general, if the left-hand side array is different from all right-hand side arrays, then there is no dependence. (Users of array aliases must be careful.) If the left-hand side array and some array on the right-hand side are the same, then the offset between the references must be a compile-time constant for the compiler to resolve the dependence question. If the offset is not a compile-time constant, then the compiler assumes that there is a dependence and uses a temporary.

For example, the following assignment requires a temporary, since the compiler cannot determine whether **K** is positive or negative.

```
A(K:K+2,:) = A(1:3,:)
```


Copyright © 1993 Thinking Machines Corporation

2.7 Gathers/Scatters with FORALL

When **FORALL** does a gather or scatter on a :**SERIAL** axis, the compiler can determine that the operation is entirely local and uses the fast indirect addressing hardware. See Section 5.1 for detailed information.

NOTE

The preliminary implementation of this feature is disabled by default in the current release. To try it, enable it with the (internal) switch **-indirect_address**.

2.8 Serial Spreads with FORALL or SPREAD

When **FORALL** does a spread operation on a **:SERIAL** axis, the compiler can determine that the operation is entirely local and generates local memory accesses instead of communication routines.

```
      REAL A(10,10), B(10)
CMF$  LAYOUT A(:SERIAL,:NEWS), B(:NEWS)
      ...
      FORALL(I=1:10) A(I,:) = B(:)
```

The intrinsic function **SPREAD** has also been enhanced for serial axes, so that **SPREAD** and **FORALL** give comparable performance for this operation.

NOTE: Spreads expressed either with **SPREAD** or with **FORALL** generate run-time communication when used on **:NEWS** axes or in assignments of unaligned array elements.

NOTE: **SPREAD** is the only non-elemental intrinsic function that has been optimized for serial dimensions. **CSHIFT**, **EOSHIFT**, and all the reduction and transformational intrinsics continue to generate communication routines in all cases.

Copyright © 1993 Thinking Machines Corporation

3 User Control over Array Layout

This section describes **LAYOUT** directive syntax (introduced in Version 2.0 Beta) that allows you to specify array layout in detail. This syntax is supported on the CM-5 and under the slicewise execution model on the CM-2/200.

Versioni 2.1 introduces two new compiler switches that give further control over array layout. These are supported on the CM-5 only:

- **-[no]axisreorder** gives the user added control over the sequence of array axes in memory.
- **-[no]padding** determines whether subgrids — the on-processor portion of arrays — are padded to a multiple of vector length under the CM-5 vector units execution model. Removing the padding constraint gives the user added control over which data elements are co-resident in local memory.

3.1 Controllable Axis Sequence: –[no]axisreorder

The compiler transforms all array section code into one or more **DO** loops (collapsing the loops on more than one array dimension where possible). The dimension whose element indices vary fastest in local memory becomes the innermost loop. This loop is vectorized on a CM-5 with vector units; the vector units process eight "data objects" for each iteration of a loop on the vector axis.

CM Fortran provides the compiler switch **-[no]axisreorder**, which allows you to control which array dimension varies fastest in memory. The negative form **-noaxisreorder** causes the compiler to lay out the dimensions in the order you declare them, with the leftmost axis the fastest varying.

Consider arrays with two serial axes:

```
      REAL A(10,20,100), B(10,20,100)
CMF$  LAYOUT A(:SERIAL,:SERIAL,), B(:SERIAL,:SERIAL,)
      ...
      A(1,:,:) = B(2,:,:)
```

With **-noaxisreorder**, the middle axis has stride 10 in local memory. The double loop that implements this assignment vectorizes along the middle axis.

Copyright © 1993 Thinking Machines Corporation

In previous releases, the compiler always laid out axes in memory in the following way, reordering them from the declared order if necessary.

- All **:NEWS** axes varied faster than any **:SERIAL** axis. (This guaranteed that a section formed by scalar indexing into all serial axes would be a contiguous block of memory.)
- The **:NEWS** axis declared rightmost varied faster than other **:NEWS** axes, the next rightmost **:NEWS** axis varied next fastest, and so on.
- The **:SERIAL** axis declared rightmost varied faster than other **:SERIAL** axes, the next rightmost **:SERIAL** axis varied next fastest, and so on.

For back-compatibility, this axis reordering is still the default. That is, the switch defaults to the positive form **-axisreorder**, and therefore the vector axis is always a **:NEWS** axis.

The current release allows you to choose the axis order in memory. When you supply the negative form **-noaxisreorder**, array dimensions are laid out in column-major order, with the leftmost declared axis the fastest-varying.

Some consequences of controllable axis ordering for performance are:

- A leftmost **:SERIAL** axis, in a code compiled with **-noaxisreorder**, gets the benefit of the new serial optimizations and of vectorization. Similarly, a rightmost **:SERIAL** axis compiled with **-axisreorder** gets the benefit of the new serial optimizations and of vectorization.
- Codes compiled with **-noaxisreorder** are no longer constrained to declare and reference **:SERIAL** axes to the left of **:NEWS** axes for best performance. The compiler no longer generates communication for operations on array sections formed by scalar indexing into one or more serial axes (regardless of their position with respect to parallel axes). Thus, this:

  ```
  A(:) = B(:,1)
  ```

  now gives performance comparable to this:

  ```
  A(:) = B(1,:)
  ```

- See also the discussion of passing serial array sections in place, in Section 2.5 above.

Copyright © 1993 Thinking Machines Corporation

NOTE

All CM Fortran program units of a program must be compiled with the same setting of **`-[no]axisreorder`**.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

3.2 Controllable Vector-Length Padding: –[no]padding

When allocating memory for CM arrays, the CM run-time system always allocates the same memory locations in every processing element (node or vector unit, depending on the execution model). As a result, the size of a *virtual grid* — the block of distributed memory onto which a user array is mapped — is always an integer multiple of **`CMF_NUMBER_OF_PROCESSORS()`**. If the product of an array's non-serial dimensions is not a legal size for a virtual grid, the system allocates the next legal size (and later masks out the unused memory locations). The system then lays out the array's parallel dimensions within the virtual grid so as to minimize the amount of this *global padding*.

In previous versions — and by default in the current version — the run-time system in two execution models met a second constraint in determining the grid size required for parallel dimensions: the on-processor subgrid had to be a multiple of the length of a vector register. As a result, machine geometry sizes (excluding serial dimensions) were integer multiples of:

number-of-processors x 8	CM-5 vector units model
number-of-processors x 4	CM-2/200 slicewise model

The CM-5 SPARC nodes model, lacking vector registers, does not generate this *vector-length padding*.

After the system determines the grid size and layout needed to accommodate the array's parallel dimensions, it extends the per-processor subgrid by the product of the array's serial dimensions. Since serial dimensions are added after the padding has been determined, they are never padded.

Version 2.1 provides a switch **`-nopadding`**, supported on CM-5 only, by which you can specify that the run-time system not pad virtual grids to a multiple of vector length. (*Global padding is not affected by the switch*; you can avoid it by declaring arrays that are evenly divisible by machine size.)

For back-compatibility, padding is enabled by default. Supply **`-nopadding`** explicitly to avoid vector-length padding.

Why is vector padding a problem? There are two reasons.

For very small arrays, it can add significantly to memory usage and to time spent computing on garbage elements. For example, imagine a 16-processor machine and an array declared

```
REAL A(16)
```


Copyright © 1993 Thinking Machines Corporation

No global padding is required, since array size equals machine size. The vector-length constraint, however, means that the minimum machine geometry size is 16 x 8, so 7/8 of the elements contain garbage data.

Further, because the system always uses block layout, the array data is concentrated in the first few processors. In this example, the 16 array elements are all stored in the first two processors, while the other 14 processors do no useful work.

With the **-nopadding** switch, this array is allocated across all 16 processors, one element in each.

A second problem with vector padding is that it can interfere with the user's ability to control array layout. For example, on the same 16-processor machine, imagine that you have laid out a 2-dimensional array like this:

```
      REAL B(100,16)
CMF$  LAYOUT B(:SERIAL, :NEWS)
```

The purpose here is to distribute the second dimension across the machine so that each processor operates on its own 100-element serial subgrid. However, since serial dimensions do not participate in the determination of machine geometry, the machine geometry generated is the same as in the example above: the first two processors have all 16 elements of the parallel dimension and thus all the data. Operations on this array together with another array that is more evenly distributed across the machine would necessarily involve communication.

NOTE

All CM Fortran program units of a program must be compiled with the same setting of **-[no]padding**.

Implementation Note: Consider this subroutine when compiled with **-nopadding**:

```
SUBROUTINE SUB(A,B,C,N)
REAL, ARRAY(N) :: A, B, C
C=A+B
END
```


Copyright © 1993 Thinking Machines Corporation

You still see vectors of length 8 in the **`.pe.dp`** file, even though it's not known at compile time whether **`N`** will leave a multiple-of-8 at each processor. In addition, however, you get wrapper code around the block which runs the loop **`N/8`** times at vector length 8, and if necessary, 1 time at vector length **`MOD(N,8)`**.

Copyright © 1993 Thinking Machines Corporation

3.3 Detailed LAYOUT Directive

As of Version 2.0, CM Fortran provides a form of the **LAYOUT** directive for specifying *detailed CM array layout.* This form is supported on any CM-5, with or without vector units, and in the slicewise execution model on the CM-2/200.

A comparable feature for allocating an array dynamically and specifying its layout in detail is the utility procedure **CMF_ALLOCATE_DETAILED_ARRAY**. This procedure is described in the *CM Fortran Utility Library Reference Manual,* Version 2.0 Beta.

The detailed-layout features are intended for advanced users who are familiar with the architecture and the run-time system of the target CM platform. These features call for the user to specify such values as subgrid lengths and processor masks for array axes. The significance of these terms is not explored here.

3.3.1 Detailed LAYOUT Directive Syntax

The **LAYOUT** directive now accepts all the following forms of axis descriptor. The last two listed pertain to detailed layout.

CMF$ LAYOUT *array-name* (*axis-descriptor*(*s*))

An *axis-descriptor* is one of:

- [*weight*] [**:NEWS**]
- [*weight*] **:SEND**
- **:SERIAL**
- **:**
- **:BLOCK** = *axis-spec* **:PROCS** = *axis-spec*
- **:BLOCK** = *axis-spec* **:PDESC** = *proc-spec*

An *axis-spec* is a decimal integer constant, an integer-valued dummy argument, or an integer variable in **COMMON**. Integer expressions are not supported.

A *proc-spec* is a bit-mask specified as a decimal integer constant, an integer-valued dummy argument, or an integer variable in **COMMON**. Integer expressions are not supported.

Copyright © 1993 Thinking Machines Corporation

The last two axis descriptors provide alternative methods of specifying detailed layout for an array axis. The axis layout can be either:

- the subgrid length (**:BLOCK=**) followed by the number of processors desired for that axis (**:PROCS=**)
- the subgrid length (**:BLOCK=**) followed by a bit-mask indicating which processors are desired (**:PDESC=**)

Depending on the program's execution model, the "processors" are CM-5 vector units, CM-5 nodes, or CM-2/200 nodes. These detailed forms of the **LAYOUT** directive can be used with any array, including dynamically allocated arrays (but see caveat below).

3.3.2 Restrictions on Axis Specifications

- While the subgrid length (**:BLOCK=**) can be any integer; the number of processors (**:PROCS=**) must be a power of 2.
- If one axis of an array is specified with **:BLOCK :PROCS**, then all axes of that array should be either **:BLOCK :PROCS** or **:SERIAL**. If one axis of an array is specified with **:BLOCK :PDESC**, then all axes of that array should be either :**BLOCK :PDESC** or **:SERIAL**.
- The compiler does not accept these detailed forms in an interface block when they use a dummy argument. Use only parameters, literal constants, or variables in **COMMON** in this situation.
- There are also restrictions on array size, as detailed later in this section.
- The compiler recognizes local axes declared with **:BLOCK :PROCS** or **:BLOCK :PDESC** syntax as serial for the purposes of applying the "serial optimizations," but only under the (non-default) switch **-noaxis-reorder**.

3.3.3 Examples of Detailed Layout Directives

- The array **A(16,16,4)** could be laid out on 8 processors as follows:

```
CMF$LAYOUT A(:SERIAL,:BLOCK=8:PROCS=2,:BLOCK=1:PROCS=4)
```


Copyright © 1993 Thinking Machines Corporation

In this mapping, the first axis is totally local (it could also have been specified as **:BLOCK=16:PROCS=1**); the 16 elements of the second axis are divided equally between 2 processors; and the third axis is spread across 4 processors, one element in each.

- The array **B(64,16)** could be laid out on 16 processors, using **:BLOCK :PDESC**, as follows:

```
CMF$  LAYOUT B(:BLOCK=16:PDESC=12,:BLOCK=4:PDESC=3)
```

Since the mask **:PDESC=12** has 2 nonzero bits, it represents a physical axis of length 4. The axis extent (64) equals subgrid length (16) times physical axis length (4). Similarly, the mask of 3 for the second axis has 2 nonzero bits, and 4*4 = 16.

3.3.4 Restrictions on LAYOUT with Deferred-Shape Arrays

Detailed **LAYOUT** directives on dynamic arrays use values upon entry, not at the time of allocation. Use **CMF_ALLOC_DETAILED_ARRAY** if you want to compute layout parameters in the same program unit as the allocation.

The use of **LAYOUT** and **ALIGN** directives with allocatable arrays and array pointers affects their layout only; the compiler does not recognize the dynamic layout (e.g., **:SERIAL** axes) when generating code.

3.3.5 Restrictions on Array Size with Padding Enabled

Finally, you should be aware of restrictions on array size when specifying detailed layout. These restrictions apply only when the program's execution environment pads user arrays such that the number of elements allocated per processor (for non-serial axes) is evenly divisible by the length of the vector registers in the CM-5 vector units or in the CM-2 floating-point accelerator.

These requirements vary by execution model; they apply only if the **cmf** switch **-padding** is enabled.

- For the CM-5 vector units (**-vu**) model with **-nopadding**, there are no restrictions on the product of the **:BLOCK** values.

Copyright © 1993 Thinking Machines Corporation

- For the CM-5 vector units (**-vu**) model with **-padding**, the product of the **:BLOCK** values must be an integer multiple of 8. Axes labeled **:SERIAL** are excluded from the calculation.
- For the CM-5 nodes (**-sparc**) model, there are no restrictions on the product of the **:BLOCK** values.
- For the CM-2/200 slicewise (**-slicewise**) model, the product of the **:BLOCK** values must be an integer multiple of 4. Axes labeled **:SERIAL** are excluded from the calculation. (The switch **-nopadding** is not supported on CM-2/200.)

For example, assume a 32-vector-unit partition on CM-5 and this detailed layout:

```
      REAL B(100,32)
CMF$  LAYOUT B(:SERIAL, :BLOCK=1:PROCS=32)
```

With **-padding** enabled, this layout causes a run-time error:

```
*** RTS-FATAL-CMRTS:
*** Fatal RTS Error ***:
CMRT_intern_detailed_array_geometry: the product of the
specified subgrid lengths, 100, must be a multiple of 8
```

With **-nopadding**, this directive gives the intended layout of one 100-element serial axis per processor.

3.3.6 Note on CM-5 :SEND Ordering

CM Fortran on CM-5 systems accepts the axis specification **:SEND** in **LAYOUT** directives; the compiler passes it on to the run-time system. The run-time system lays out a **:SEND** axis in the same way as a **:NEWS** axis. However, the compiler doesn't understand this and so generates a **SEND** routine for an elemental operation between a **:NEWS**-ordered array and an otherwise identical :**SEND**-ordered array.

Also, some CM-5 libraries do not accept **:SEND** ordering.

Unless your CM-5 program needs to be compatible with the CM-2/200, it is best to avoid **:SEND** ordering.

Copyright © 1993 Thinking Machines Corporation

4 Array Aliasing Utilities

A set of CM Fortran utility library procedures support *subgrid equivalencing*, a form of storage association for CM arrays. These procedures enable you to create an *array alias*, which addresses memory already allocated for another array but treats it as being of a different type, shape, or layout. No data motion occurs; the memory is simply associated with an additional array descriptor.

Some uses of array aliasing:

- You can take advantage of the compiler's optimizations on serial dimensions by creating a 2-dimensional **`(:SERIAL,:NEWS)`** alias for storage that is allocated as a 1-dimensional **`:NEWS`** array. The serial axis of the alias is storage-associated with the on-processor portion of the target array's **`:NEWS`** axis.

- You can write rank-independent subprograms. Pass a 1D actual array argument, but alias it in the subprogram as *n*D with suitable axis extents.

- You can manage memory manually by defining one huge 1D array and then use aliases to choose portions of it for specific tasks.

- You can reduce the compiler's use of temporaries in stencils, etc. Use two aliases into the same data rather than making a temporary copy of the data and shifting it.

This chapter provides a brief introduction to the procedures in the aliasing package. It also provides detailed descriptions of the individual procedures and explanations of the alias package's error codes. Two sample programs appear at the end.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

4.1 Using the Alias Package

4.1.1 Constructing Aliases

Use these procedures to create array aliases:

```
CMF_ALIAS_TYPE( ALIAS,TYPE,ARRAY,RESIZE_MODE,ASTAT )

CMF_ALIAS_DETAILED( ALIAS,RANK,EXTENTS,ORDERS,
   SUBGRIDS,PMASKS,OFFSETS,MASK_CHECK,ARRAY,ASTAT )
```

The **ALIAS** argument is a front-end array that serves as an array descriptor; other arguments fill in the slots of the descriptor. The alias can be used as a CM array when it is passed to a subprogram that declares it as such.

The **ARRAY** argument can be either a CM array (including an alias in the callee scope) *or* a front-end array that is a valid array descriptor. If the latter, the argument could only be an alias or a descriptor generated by a **CMF_ALLOCATE_** routine, since these are the only arrays whose descriptors can be referenced directly.

The alias package has its own run-time safety mechanism. At the nondefault safety level, bounds-checking is performed to ensure that the alias addresses memory within the range addressed by the target **ARRAY**. At the default safety level, the alias may address any allocated memory.

4.1.2 Operating upon Aliases

Use these procedures to identify, mutate, or remove an alias.

```
LOG = CMF_ARRAY_IS_ALIAS( ARRAY,ASTAT )

CMF_SET_ALIAS_LOCATION( ALIAS,LOCATION,ASTAT )

CMF_REMOVE_ALIAS( ALIAS,ASTAT )
```

These procedures may be applied to aliases in either declaration scope (either as front-end arrays in the routine that constructs an alias or as CM arrays in a callee). The second two should be applied *only* to array aliases; at the default safety level, their application to nonaliases is not detected and may lead to incorrect results.

Copyright © 1993 Thinking Machines Corporation

4.1.3 Array Descriptor Inquiries

Use these procedures to derive information about an existing array or alias, usually for the purpose of constructing arguments to an alias constructor or mutator.

```
INT = CMF_GET_GLOBAL_LOCATION( ARRAY,OFFSETS,ASTAT )
INT = CMF_GET_PHYSICAL_LOCATION( ARRAY,OFFSETS,ASTAT )
CMF_GET_AXIS_ORDERS( ARRAY,ORDERS,RANK,ASTAT )
CMF_GET_LOCAL_EXTENTS( ARRAY,EXTENTS,RANK,ASTAT )
CMF_GET_LOCAL_STRIDES( ARRAY,STRIDES,RANK,ASTAT )
CMF_GET_PROCESSOR_MASKS( ARRAY,MASKS,RANK,ASTAT )
```

4.1.4 Execution Environment

Use these procedures to inquire about or set a feature of the execution environment for the aliasing procedures.

```
INT = CMF_ARRAY_SIZE_QUANTUM( )
CMF_SET_ALIAS_SAFETY_LEVEL( SAFETY_LEVEL )
INT = CMF_GET_ALIAS_SAFETY_LEVEL( )
```

The run-time checks performed at the non-default safety level are listed below in the reference descriptions of the two safety-related procedures. The error codes that may be returned in the **ASTAT** argument are described in Section 3.4.

Copyright © 1993 Thinking Machines Corporation

4.2 Restrictions and Caveats

- The aliasing procedures behave incorrectly if the **ARRAY** argument — the target of an alias constructor or the array queried by an inquiry procedure — is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.

- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**. This safety mechanism signals an error if the homes of actual and dummy array arguments do not match.

- Beware of padding in either a target array or its alias. If the target array and the alias have different geometries, the internal garbage-masking software will mistake the location of garbage elements in the alias. The result can be that real data is masked and garbage data is computed upon and stored. Recall that the **cmf** switch **-nopadding** removes vector-length padding only; global padding, if any, is not affected by this switch.

- It is not advisable to pass to a subprogram two (or more) array arguments that address the same or overlapping memory if the subprogram changes either argument. The compiler assumes that argument memory does not overlap and conducts its dependence analysis and global optimizations (under the **-O** switch) on this assumption.

- An example of aliases confounding dependence analysis is seen in the compiler's internal strip-mining. For example, given vectors **A**, **B**, and **A1**, suppose that **A1** is aliased to the location offset by one element from the base of **A**. Then, consider a program executing under the CM-5 vector units model that contains the statements:

```
A = expression_1
B = A1 + expression_2
```

  If both these source lines are included in one local code block, the following happens. The code block turns into a loop where each iteration of the loop works on a vector of length 8. In the first iteration of the loop, elements 1:8 of **A** are updated. Note that these are elements 2:9 of **A1**. So the assignment to **B** is using new values for elements 1:7 of **A1**, and an old value for element 8 of **A1**.

Copyright © 1993 Thinking Machines Corporation

4.3 Procedure Descriptions

Descriptions follow for the aliasing procedures, listed in alphabetical order.

CMF_ALIAS_DETAILED

CMF_ALIAS_TYPE

CMF_ARRAY_IS_ALIAS

CMF_ARRAY_SIZE_QUANTUM

CMF_GET_ALIAS_SAFETY_LEVEL

CMF_GET_AXIS_ORDERS

CMF_GET_GLOBAL_LOCATION

CMF_GET_LOCAL_EXTENTS

CMF_GET_LOCAL_STRIDES

CMF_GET_PHYSICAL_LOCATION

CMF_GET_PROCESSOR_MASKS

CMF_REMOVE_ALIAS

CMF_SET_ALIAS_LOCATION

CMF_SET_ALIAS_SAFETY_LEVEL

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_ALIAS_DETAILED

Creates an array alias with a specified shape and detailed layout.

Syntax

```
 INCLUDE '/usr/include/cm/CMF_defs.h'

 CALL CMF_ALIAS_DETAILED
&    ( ALIAS, RANK, EXTENTS, ORDERS, SUBGRIDS, PMASKS,
&      OFFSETS, MASK_CHECK, ARRAY, ASTAT )
```

Arguments

ALIAS An integer front-end array of size **CMF_SIZEOF_DESCRIPTOR** (a predefined constant). This argument becomes the alias descriptor.

RANK An integer scalar indicating the rank of the alias.

EXTENTS An integer front-end vector that is at least **RANK** in length and contains the dimension extents of the alias.

ORDERS An integer front-end vector that is at least **RANK** in length and contains values specifying axis ordering of the alias. Each value must be one of:

CMF_SERIAL_ORDER
CMF_NEWS_ORDER
CMF_SEND_ORDER

SUBGRIDS An integer front-end vector that is at least **RANK** in length and contains the subgrid length for each axis of the alias.

PMASKS An integer front-end vector that is at least **RANK** in length and contains bit-masks to specify processors. If the **ORDERS** argument contains the value **CMF_SERIAL_ORDER** for any axis, the **PMASKS** argument must contain zero for that axis.

Copyright © 1993 Thinking Machines Corporation

OFFSETS An integer front-end vector that is at least **RANK** in length and contains a subgrid-relative zero-based offset index for each dimension of **ARRAY**. The address of the element of the argument **ARRAY**'s local subgrid that is selected by the values in **OFFSETS** is used as the base address of the array represented by **ALIAS**. The values that correspond to purely inter-processor dimensions (those whose local extent is 1) must be zero.

MASK_CHECK A logical scalar that specifies whether to perform a safety check to ensure that the same garbage mask, if any, is used by the geometries of both the **ARRAY** argument and the generated alias. The error status of the operation is returned in **ASTAT**.

ARRAY An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.

ASTAT An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

If alias safety is nonzero and **OFFSETS** address memory out of the bounds of the argument **ARRAY**, a nonzero value is returned in **ASTAT**.

Result

None.

Description

Fills in the **ALIAS** argument with a detailed array layout so that it may be passed to a subprogram that declares it as a CM array. The alias treats the memory it addresses as having the shape described by **RANK** and **EXTENTS** and the layout described by **ORDERS**, **SUBGRIDS**, and **PMASKS**. The alias addresses memory beginning at the location computed from the base location of the argument **ARRAY**, as offset by the subgrid-relative values in **OFFSETS**.

Copyright © 1993 Thinking Machines Corporation

Restrictions

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.
- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_GET_AXIS_ORDERS
CMF_GET_LOCAL_EXTENTS
CMF_GET_LOCAL_STRIDES
CMF_GET_PROCESSOR_MASKS

Copyright © 1993 Thinking Machines Corporation

CMF_ALIAS_TYPE

Creates an array alias of a specified type.

Syntax

```
 INCLUDE '/usr/include/cm/CMF_defs.h'

 CALL CMF_ALIAS_TYPE
&    ( ALIAS, TYPE, ARRAY, RESIZE_MODE, ASTAT )
```

Arguments

ALIAS An integer front-end array of size **CMF_SIZEOF_DESCRIPTOR** (a predefined constant).

TYPE A predefined integer constant that indicates data type. It must be one of:
CMF_LOGICAL, **CMF_S_INTEGER**,
CMF_FLOAT, **CMF_DOUBLE**,
CMF_COMPLEX, **CMF_DOUBLE_COMPLEX**

ARRAY An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.

RESIZE_MODE A predefined integer constant that indicates which dimension of **ALIAS**, if any, to adjust to accommodate a data type different from **ARRAY**. See below for explanation. It must be one of:
CMF_MUTATE_SELECT
CMF_MUTATE_ANY
CMF_MUTATE_BEST

ASTAT An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

If the **RESIZE_MODE** argument is **CMF_MUTATE_SELECT**, **ASTAT** is used to indicate which alias dimension (beginning from axis 0, not 1) to resize, as explained below. **ASTAT** retains its conventional return semantics.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

Result

None.

Description

Fills in the **ALIAS** argument so that it may be passed to a subprogram that declares it as a CM array. The alias addresses the same data addressed by the argument **ARRAY**, but treats it as the data type represented by **TYPE**. If the new type is not the same element size as the type of **ARRAY**, the **RESIZE_MODE** argument indicates how to adjust the size of the array represented by **ALIAS** with respect to the argument **ARRAY**.

Description of Resizing Modes

The **RESIZE_MODE** argument is ignored if the element size of the **TYPE** argument is the same as the element size of the argument **ARRAY**. If this is not the case, then the length of one of the dimensions of **ALIAS** will differ from the length of the corresponding dimension of **ARRAY**. The **RESIZE_MODE** argument controls which dimension is selected for adjustment.

- **CMF_MUTATE_SELECT** implies that the **ASTAT** argument has been initialized to the axis number that is to be adjusted.
- **CMF_MUTATE_ANY** allows the implementation to select which axis is to be adjusted. In the current implementation, the outermost axis (the one with largest local, or per-processor, stride) is selected if the size of the new data type is less than the size of the old data type. If the size of the new data type is greater than the size of the old data type, preference is given to the outermost axis, if any, of which the length in the original array was a multiple of the ratio of data type size expansion. If, in this case, no axis is a multiple of the expansion ratio, the locally outermost axis will be extended as necessary to ensure that the generated alias accesses at least all of the memory accessible through the original array. Attempts to operate on the outermost elements are not guaranteed success under these circumstances.
- **CMF_MUTATE_BEST** is essentially the same as **CMF_MUTATE_ANY**, unless the data type expansion ratio is greater than one and no dimension of the original array bears a local extent that is a multiple of that ratio, in which case an error is returned. In this mode, all elements of the generated alias array can be accessed successfully.

Copyright © 1993 Thinking Machines Corporation

Restrictions

- The type specified by **CMF_LONG_S_INTEGER** is not supported.
- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.
- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_ARRAY_IS_ALIAS

Reports whether the argument **ARRAY** is a generated array alias.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'

LOGICAL = CMF_ARRAY_IS_ALIAS( ARRAY, ASTAT )
```

Arguments

ARRAY An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.

ASTAT An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates that the **ARRAY** argument is not a valid CM array or array alias.

Result

Logical scalar. Result is **.TRUE.** if the argument **ARRAY** is a generated array alias, **.FALSE.** otherwise.

Restrictions

- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_ALIAS_TYPE
CMF_ALIAS_DETAILED

Copyright © 1993 Thinking Machines Corporation

CMF_ARRAY_SIZE_QUANTUM

Returns the integer quantum by which virtual grid size increases in the current execution environment.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
INTEGER = CMF_ARRAY_SIZE_QUANTUM( )
```

Arguments

None.

Result

Integer scalar:

- Under the CM-5 vector-units model, returns 1 with **`-nopadding`**, or 8 with the default **`-padding`**. The vector register in the CM-5 vector units is of length 8.
- Under the CM-5 SPARC nodes model, returns 1.
- Under the CM-2/200 slicewise model (not yet supported), always returns 4, the length of the vector register in the CM-2/200 64-bit floating-point accelerator. The **`-nopadding`** switch is not supported on the CM-2/200.

Description

The result of this function indicates the vector-length, if any, of which the non-serial subgrid of a virtual grid must be a multiple. This factor, when multiplied by the current number of physical processors, gives the minimum number of elements a user array's non-serial dimensions must have to be stored without padding.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_GET_ALIAS_SAFETY_LEVEL

Returns the current level of run-time alias safety checking.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
INTEGER = CMF_GET_ALIAS_SAFETY_LEVEL( )
```

Arguments

None.

Result

Integer scalar. Zero indicates the default safety level; nonzero indicates extra safety checking.

Description

Run-time alias safety checking may be set to perform either the default checks only or extra checks. Error status of an alias procedure is returned in the argument **ASTAT**.

At the higher safety level, the system checks for:

- attempts to construct an alias that addresses memory outside the range of memory addressed by the alias target
- attempts to dereference pointer elements of the descriptors passed to an alias routine (which is likely to result in a segmentation violation, if the descriptor argument is invalid or erroneous in any way)
- axis descriptor masks of a form inappropriate to the current execution environment

Copyright © 1993 Thinking Machines Corporation

See Also

CMF_SET_ALIAS_SAFETY_LEVEL

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_GET_AXIS_ORDERS

Fills the **ORDERS** argument array with values indicating the ordering (serial, NEWS, or send) of each axis of the argument **ARRAY**.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
CALL CMF_GET_AXIS_ORDERS( ARRAY,ORDERS,RANK,ASTAT )
```

Arguments

ARRAY	An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.
ORDERS	An integer front-end vector that is at least **RANK(ARRAY)** in length. The values returned in **ORDERS** are one or more of: **CMF_SERIAL_ORDER** **CMF_NEWS_ORDER** **CMF_SEND_ORDER**
RANK	An integer scalar. The array rank is returned in **RANK**.
ASTAT	An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

Result

None.

Restrictions

- This procedure returns **CMF_SERIAL_ORDER** only for axes that are explicitly declared **:SERIAL** or **CMF_SERIAL_ORDER**, not for local axes that are declared with **:PROCS** or **:PDESC** syntax.
- This procedure is incompatible with the **cmf** option **-noaxisreorder**. When compiled with this option, the procedure returns **CMF_NEWS_ORDER** for all axes.

Copyright © 1993 Thinking Machines Corporation

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.
- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_ALIAS_DETAILED

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_GET_GLOBAL_LOCATION

Returns as an integer the location (a parallel memory address) of the element of the argument **ARRAY** selected by the argument **OFFSETS**.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
INTEGER = CMF_GET_GLOBAL_LOCATION( ARRAY,OFFSETS,ASTAT )
```

Arguments

ARRAY — An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.

OFFSETS — An integer front-end vector that is at least **RANK(ARRAY)** in length and that contains an index offset for each dimension of **ARRAY**. The offset values index **ARRAY** as though all dimensions were declared as zero-based.

ASTAT — An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

If alias safety is nonzero and **OFFSETS** address memory out of the bounds of the argument **ARRAY**, a nonzero value is returned in **ASTAT**.

Result

Integer scalar representing a parallel memory address.

Restrictions

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.

Copyright © 1993 Thinking Machines Corporation

- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_SET_ALIAS_LOCATION

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_GET_LOCAL_EXTENTS

Fills the **EXTENTS** argument array with the axis extents of **ARRAY**'s per-processor subgrid, and the **RANK** argument with the rank of **ARRAY**'s per-processor subgrid.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
CALL CMF_GET_LOCAL_EXTENTS( ARRAY,EXTENTS,RANK,ASTAT )
```

Arguments

ARRAY	An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.
EXTENTS	An integer front-end vector that is at least **RANK(ARRAY)** in length. The subgrid extents (excluding padding, if any) are returned in **EXTENTS**.
RANK	An integer scalar. The subgrid rank is returned in **RANK**.
ASTAT	An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

Result

None.

Restrictions

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.
- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

Copyright © 1993 Thinking Machines Corporation

See Also

CMF_ALIAS_DETAILED

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_GET_LOCAL_STRIDES

Fills the **STRIDES** argument array with the elemental strides of the axes of **ARRAY**'s per-processor subgrid, and the **RANK** argument with the rank of **ARRAY**'s per-processor subgrid.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
CALL CMF_GET_LOCAL_STRIDES( ARRAY,STRIDES,RANK,ASTAT )
```

Arguments

ARRAY	An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.
STRIDES	An integer front-end vector that is at least **RANK(ARRAY)** in length. The subgrid elemental axis strides are returned in **STRIDES**.
RANK	An integer scalar. The subgrid rank is returned in **RANK**.
ASTAT	An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

Result

None.

Description

The returned values should be interpreted such that: the innermost subgrid-local axis has a unit stride, while the outermost subgrid-local axis has a stride equal to the product of the strides of all inferior axes, multiplied by the local (per-processor) extent of the axis with the next smaller stride.

Copyright © 1993 Thinking Machines Corporation

Restrictions

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.
- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_ALIAS_DETAILED

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_GET_PHYSICAL_LOCATION

Returns as an integer the location (a parallel memory address) of the element of the argument **ARRAY**'s local subgrid selected by the argument **OFFSETS**.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
INTEGER = CMF_GET_PHYSICAL_LOCATION( ARRAY,OFFSETS,ASTAT )
```

Arguments

ARRAY — An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.

OFFSETS — An integer front-end vector that is at least **RANK(ARRAY)** in length and contains a subgrid-relative index offset for each dimension of **ARRAY**. The offsets are zero-based indices addressing the in-processor subgrid of the global array. The values that correspond to purely inter-processor dimensions (those whose local extent is 1) must be zero.

ASTAT — An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

If alias safety is nonzero and **OFFSETS** address memory out of the bounds of the argument **ARRAY**, a nonzero value is returned in **ASTAT**.

Result

Integer scalar representing a parallel memory address.

Restrictions

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.

Copyright © 1993 Thinking Machines Corporation

- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_SET_ALIAS_LOCATION

Copyright © 1993 Thinking Machines Corporation

CMF_GET_PROCESSOR_MASKS

Fills the **MASKS** argument array with integer bitmasks that describe the layout of **ARRAY**'s axes across processors, and fills the **RANK** argument with **ARRAY**'s rank.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
CALL CMF_GET_PROCESSOR_MASKS( ARRAY, MASKS, RANK, ASTAT )
```

Arguments

ARRAY	An allocated CM array of any type, including an array alias, *or* an integer front-end array that is a valid array descriptor.
MASKS	An integer front-end vector that is at least **RANK(ARRAY)** in length. Each element is filled with an integer bitmask such that the intersection of any two elements is zero and the union of all masks is less than or equal to the value of **CMF_NUMBER_OF_PROCESSORS()-1**. Each bit of a given element is set if and only if the corresponding bit of the physical processor address is allocated to the axis of the argument **ARRAY** which corresponds to that element of **MASKS**.
RANK	An integer scalar. The rank of **ARRAY** is returned in **RANK**.
ASTAT	An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

Result

None.

Restrictions

- The aliasing procedures behave incorrectly if the **ARRAY** argument is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.

Copyright © 1993 Thinking Machines Corporation

- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

See Also

CMF_ALIAS_DETAILED

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_REMOVE_ALIAS

Deallocates the front-end auxiliary and substructure memory associated with an alias descriptor.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
CALL CMF_REMOVE_ALIAS( ALIAS, ASTAT )
```

Arguments

ALIAS An integer front-end array that is the descriptor of a generated array alias. (Any attempt to use the descriptor as an alias after removal is an error.)

ASTAT An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

If alias safety is nonzero and **ALIAS** is not a generated array alias, a nonzero value is returned in **ASTAT** and no deallocation takes place.

Result

None.

See Also

CMF_ALIAS_TYPE
CMF_ALIAS_DETAILED

Copyright © 1993 Thinking Machines Corporation

CMF_SET_ALIAS_LOCATION

Modifies an alias so that the base location of the CM array it represents is the parallel memory address in **LOCATION**.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'
CALL CMF_SET_ALIAS_LOCATION( ALIAS, LOCATION, ASTAT )
```

Arguments

ALIAS An integer front-end array that is the descriptor of a generated array alias.

LOCATION An integer scalar that contains a parallel memory address. The value is that returned by **CMF_GET_GLOBAL_LOCATION** or **CMF_GET_PHYSICAL_LOCATION**.

ASTAT An integer scalar into which the error status of the operation is placed. Zero indicates success; a positive value indicates failure.

If alias safety is nonzero and **ALIAS** is not a generated array alias, a nonzero value is returned in **ASTAT**.

Result

None.

Restrictions

- The aliasing procedures are incompatible with the run-time safety checking enabled with the **cmf** command option **-safety**.

Copyright © 1993 Thinking Machines Corporation

See Also

CMF_GET_GLOBAL_LOCATION
CMF_GET_PHYSICAL_LOCATION

Copyright © 1993 Thinking Machines Corporation

CMF_SET_ALIAS_SAFETY_LEVEL

Sets level of run-time alias safety checking.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_SET_ALIAS_SAFETY_LEVEL( SAFETY_LEVEL )
```

Arguments

SAFETY_LEVEL Integer scalar. The value zero indicates the default safety level; any positive value indicates extra safety checking. Negative values are not supported.

The former safety level is returned in the argument.

Result

None.

Description

Run-time alias safety checking may be set to perform either the default checks only or extra checks. Error status of an alias procedure is returned in the argument **ASTAT**.

At the higher safety level, the system checks for:

- attempts to construct an alias that addresses memory outside the range of memory addressed by the alias target
- attempts to dereference pointer elements of the descriptors passed to an alias routine (which is likely to result in a segmentation violation, if the descriptor argument is invalid or erroneous in any way)
- axis descriptor masks of a form inappropriate to the current execution environment

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

See Also

CMF_GET_ALIAS_SAFETY_LEVEL

Copyright © 1993 Thinking Machines Corporation

4.4 Alias Safety Checking: Error Codes

This section describes the error messages that are returned in the **ASTAT** argument when an aliasing operation fails. Most reflect default-level safety checking in the alias package.

If the environment is set to a higher level of safety checking (with a positive argument to **CMF_SET_ALIAS_SAFETY_LEVEL**), the system checks for the following conditions. If necessary, it returns in **ASTAT** one of the error codes identified as "with bounds checking enabled."

- attempts to construct an alias that addresses memory outside the range of memory addressed by the alias target
- attempts to dereference pointer elements of the descriptors passed to an alias routine (which is likely to result in a segmentation violation, if the descriptor argument is invalid or erroneous in any way)
- axis descriptor masks of a form inappropriate to the current execution environment

The error code identified as "with mask checking enabled" refers to the argument **MASK_CHECK** to **CMF_ALIAS_DETAILED**.

CMF_error_null_array
: This code is returned if an attempt is made to alias an array with a null extent.

CMF_error_sparse_mask
: This code is returned if an attempt is made to create an alias with a discontiguous processor allocation mask along some axis, when the current execution environment does not support such a mask.

CMF_error_axis_order
: This code is returned if an attempt is made to alias a descriptor with an invalid or otherwise unrecognized axis ordering along some axis, or to generate an alias with an invalid or otherwise unrecognized axis ordering along some axis.

CMF_error_cannot_intern
: This code is returned if an attempt to intern a run-time system geometry when generating an alias fails for any reason. No alias is generated in this case.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_error_bad_argument

This code is returned if one of the arguments to an alias support routine is out of range or otherwise invalid and no more specific error code applies.

CMF_error_bad_geometry

This code is returned if the alias target CM array descriptor (by convention, the argument prototyped with the name **ARRAY**) is determined to be corrupt in its geometry components.

CMF_error_mask_mismatch

With mask checking enabled, this code is returned if an attempt is made to generate an alias such that its geometry involves a garbage mask which does not match the garbage mask of the alias target (by convention, the argument prototyped with the name **ARRAY**).

CMF_error_fe_home

This code is returned by the **CMF_ALIAS_DETAILED** subroutine if its **ARRAY** argument is a front-end array that is not a valid array descriptor.

CMF_error_fe_memory

This code is returned if an attempt to allocate heap memory for alias descriptor substructures on the control processor fails.

CMF_error_bad_type

This code is returned if an attempt is made to generate a type alias (as with **CMF_ALIAS_TYPE**) of an unsupported type. Currently supported type constants are:

CMF_LOGICAL
CMF_S_INTEGER
CMF_FLOAT
CMF_DOUBLE
CMF_COMPLEX
CMF_DOUBLE_COMPLEX

CMF_error_bad_opcode

This code is returned if an attempt is made to generate an alias using an undefined or unsupported **RESIZE_MODE** argument to **CMF_ALIAS_TYPE**.

Copyright © 1993 Thinking Machines Corporation

CMF_error_bounds_violation

With bounds-checking enabled, this code is returned if an attempt is made to construct an alias referencing CM memory that was not referenced by the original alias target (by convention, the argument prototyped with the name **ARRAY**).

CMF_error_no_axis

This code is returned if an attempt is made to alias a target array in such a fashion that a run-time geometry axis of the target array that is not used in the user layout of that array is indexed by an **OFFSETS** argument of the alias routine.

CMF_error_no_alias

This code is returned if an attempt is made to remove an alias, and the argument alias provided is not recognized as a generated alias.

CMF_error_aligned_geometry

This code is returned if an attempt is made to generate an alias for a target array that is detectably aligned to another CM array.

CMF_error_runtime_layer

This code is returned if the current execution environment does not support the alias operation being attempted, or if the user code was linked with a run-time system not appropriate to the architecture supported by the linked utility library.

Copyright © 1993 Thinking Machines Corporation

4.5 Examples of Detailed-Alias Construction

Example 1

This program aliases a 1D NEWS-ordered array to a 2D **:SERIAL**, **:NEWS** array. The alias treats the local (on-processor) portion of the target **:NEWS** array as a separate **:SERIAL** axis. The program output shown afterward gives the layout and contents of the target array and the alias.

```
      PROGRAM SIMPLE_ALIASING
C     Get size of 1D array to use in aliasing program
      INTEGER N

      PRINT *,'Simple array aliasing program'
      PRINT *,'What 1D array size do you want? '
      READ  *,N
      CALL MAIN(N)
      END PROGRAM SIMPLE_ALIASING

      SUBROUTINE MAIN(N)
C     Define array and alias, then call printing subroutine

      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER N,A(N)
CMF$  LAYOUT A(:NEWS)
      INTEGER SUBGRID_SIZES(1), OLD_PMASKS(1)
      INTEGER A_ALIAS(CMF_SIZEOF_DESCRIPTOR)
      INTEGER EXTENTS(2), SGLENGTHS(2), ORDERS(2)
      INTEGER PMASKS(2), OFFSETS(2)
      LOGICAL MASK_CHECK
      INTEGER PROCS,RANK
      INTEGER I,J,STATUS

C Zero the real (target)array and then self-index its elements
      A=0
      FORALL (I=1:N) A(I) = I

C Get information about subgrid size and pmasks of array
      PROCS = CMF_NUMBER_OF_PROCESSORS()
      CALL CMF_GET_LOCAL_EXTENTS(A,SUBGRID_SIZES,RANK,STATUS)
      CALL CMF_GET_PROCESSOR_MASKS(A,OLD_PMASKS,RANK,STATUS)
```


Copyright © 1993 Thinking Machines Corporation

```
C Define properties of the alias array:
      RANK = 2

C Axis 1 of alias is SERIAL order axis, with same subgrid
C size as in the original array
      ORDERS(1)=CMF_SERIAL_ORDER
      EXTENTS(1)=SUBGRID_SIZES(1)
      SGLENGTHS(1)=SUBGRID_SIZES(1)
      PMASKS(1)=0
      OFFSETS(1)=0

C Axis 2 of alias is NEWS order axis, with extent equal to
C the number of CM processors executing the program
      ORDERS(2)=CMF_NEWS_ORDER
      EXTENTS(2)=PROCS
      SGLENGTHS(2)=1
      PMASKS(2)=PROCS-1
      OFFSETS(2)=0

C Don't bother checking the alias's garbage mask
      MASK_CHECK=.FALSE.

C Define the alias array
      CALL CMF_ALIAS_DETAILED
     $    (A_ALIAS,RANK,EXTENTS,ORDERS,SGLENGTHS,
     $    PMASKS,OFFSETS,MASK_CHECK,A,STATUS)

C Print out the contents of both arrays
      CALL DISPLAY_ARRAYS(A,N,A_ALIAS,EXTENTS(1),EXTENTS(2))
      END SUBROUTINE MAIN

      SUBROUTINE DISPLAY_ARRAYS(ARRAY,SIZE,ALIAS,SUBGRID,PROCS)
C Print contents and descriptions of target array and alias

      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER SIZE,SUBGRID,PROCS
      INTEGER ARRAY(SIZE), ALIAS(SUBGRID,PROCS)
CMF$  LAYOUT ARRAY(:NEWS), ALIAS(:SERIAL,:NEWS)
      INTEGER I,J
      PRINT *,' '
      PRINT *,'Here''s the original 1D array:'
      PRINT *,' '
      CALL CMF_DESCRIBE_ARRAY(ARRAY)
      PRINT *,' '
      PRINT '(1000I4)', (ARRAY(I), I=1,SIZE)
```


Copyright © 1993 Thinking Machines Corporation

```
      PRINT *,' '
      PRINT *,'And here''s the 2D alias array:'
      PRINT *,' '
      CALL CMF_DESCRIBE_ARRAY(ALIAS)
      PRINT *,' '
      DO I=1,SUBGRID
         PRINT '(1000I4)', (ALIAS(I,J), J=1,PROCS)
      END DO
      END SUBROUTINE DISPLAY_ARRAYS
```

Example 1 produces the following output when executed on a 16-processor CM-5: a 4-node, 16-vector-unit partition under the CM Fortran switches **-vu** and **-padding**.

```
% a.out
Simple array aliasing program
What 1D array size do you want?
116

Here's the original 1-D array:
    ...
    user_rank                :  1
    axes_extents             :  116
    …
Machine geometry id: 0x147500, rank: 1, column major
    Machine geometry elements:    128
    Overall subgrid size:         8
    Axis 0:
         Extent:        128 (16 physical x 8 subgrid)
         Off-chip:      4 bits, mask = 0xf
         Subgrid:       length = 8, axis-increment = 1

 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48
49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64
65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
97  98  99 100 101 102 103 104 105 106 107 108 109 110 111 112
113 114 115 116
```


Copyright © 1993 Thinking Machines Corporation

```
And here's the 2-D alias array:
    ...
    user_rank               :  2
    axes_extents            :  8 16
    ...
Machine geometry id: 0x147780, rank: 2, column major
    Machine geometry elements:    128
    Overall subgrid size:         8
    Axis 0 (serial axis):
         Extent:        8 (1 physical x 8 subgrid)
         Off-chip:      0 bits, mask = 0x0
         Subgrid:       length = 8, axis-increment = 1
    Axis 1:
         Extent:        16 (16 physical x 1 subgrid)
         Off-chip:      4 bits, mask = 0xf
         Subgrid:       length = 1, axis-increment = 1

  1   9  17  25  33  41  49  57  65  73  81  89  97 105 113   0
  2  10  18  26  34  42  50  58  66  74  82  90  98 106 114   0
  3  11  19  27  35  43  51  59  67  75  83  91  99 107 115   0
  4  12  20  28  36  44  52  60  68  76  84  92 100 108 116   0
  5  13  21  29  37  45  53  61  69  77  85  93 101 109   0   0
  6  14  22  30  38  46  54  62  70  78  86  94 102 110   0   0
  7  15  23  31  39  47  55  63  71  79  87  95 103 111   0   0
  8  16  24  32  40  48  56  64  72  80  88  96 104 112   0   0
```


Copyright © 1993 Thinking Machines Corporation

Example 2

This program uses array aliasing in performing a 5-point stencil.

```
     PROGRAM ALIAS_BASIC

C Loop over problem size
      DO
         WRITE(6,*)'NX,NY (enter "0 0" to exit) ?'
         READ(5,*)NX,NY
         IF (NX.LE.0) EXIT
         CALL TIMING(NX,NY)
      END DO
      STOP
      END

      SUBROUTINE TIMING(NX,NY)
      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER ALIAS(CMF_SIZEOF_DESCRIPTOR), EXTENTS(6)
      DOUBLE PRECISION, ARRAY(NX,NY) :: X, Y
CMF$  LAYOUT ALIAS(:SERIAL), EXTENTS(:SERIAL), X(,), Y(,)

C Initialize data
      X=0.D0
      FORALL (I=1:NX,J=1:NY) X(I,J)=COS((I+J)/(0.2D0*NX))
      Y=X

C Get number of loops for program check
      WRITE(6,*)'# of loops for program check?'
      READ(5,*)NLOOPS

C Conventional way
      DO (NLOOPS) TIMES
         Y=0.5*Y+0.125*
     .       (CSHIFT(Y,DIM=1,SHIFT=-1)+CSHIFT(Y,DIM=1,SHIFT=+1)
     .       +CSHIFT(Y,DIM=2,SHIFT=-1)+CSHIFT(Y,DIM=2,SHIFT=+1))
      END DO

C Set up alias
      CALL MAKE_ALIAS(X,ALIAS,EXTENTS)
```


Copyright © 1993 Thinking Machines Corporation

```
C Stencil way
      DO (NLOOPS) TIMES
      CALL STENCIL(ALIAS,EXTENTS(1),
     &      EXTENTS(2),EXTENTS(3),EXTENTS(4))
      END DO

C Compare results
      WRITE(6,*)'maximum error: ',MAXVAL(ABS(X-Y))

C Get number of loops for program timing
      WRITE(6,*)'# of loops for program timing?'
      READ(5,*)NLOOPSC TIME STENCIL
      CALL CM_timer_clear(0)
      CALL CM_timer_clear(1)
      CALL CM_timer_clear(2)
      CALL CM_timer_clear(3)
      CALL CM_timer_start(0)
      DO (NLOOPS) TIMES
         CALL STENCIL(ALIAS,EXTENTS(1),EXTENTS(2),
     &      EXTENTS(3),EXTENTS(4))
      END DO
      CALL CM_timer_stop(0)
      WRITE(6,*)'total .....'
      CALL CM_timer_print(0)
      WRITE(6,*)'on-node data movement .....'
      CALL CM_timer_print(1)
      WRITE(6,*)'off-node data movement .....'
      CALL CM_timer_print(2)
      WRITE(6,*)'computation .....'
      CALL CM_timer_print(3)

C Remove alias
      CALL CMF_REMOVE_ALIAS(ALIAS,IERROR)
      RETURN
      END

     SUBROUTINE MAKE_ALIAS(XDESC,ALIAS,EXTENTS)

C This routine assumes XDESC describes an array with only
C :NEWS axes. Here, the array should be at most 3-dimensional.
C The constructed alias addresses the same memory with twice
C as many axes -- one :SERIAL and one :NEWS axis for each of
C the original axes. The :SERIAL axes are all to the left.
C The :NEWS axes are all off-processor.
c The axis extents of the alias are returned in EXTENTS.
```


Copyright © 1993 Thinking Machines Corporation

```
      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER POPCOUNT
      INTEGER, ARRAY(CMF_SIZEOF_DESCRIPTOR) :: XDESC, ALIAS
      INTEGER,ARRAY(6)::EXTENTS,ORDERS,SUBGRIDS,PMASKS, OFFSETS
CMF$  LAYOUT XDESC(:SERIAL), ALIAS(:SERIAL)
CMF$  LAYOUT EXTENTS(:SERIAL), ORDERS(:SERIAL)
CMF$  LAYOUT SUBGRIDS(:SERIAL),PMASKS(:SERIAL),OFFSETS(:SERIAL)

C Form subgrid extents (and get the number of dimensions)
      CALL CMF_GET_LOCAL_EXTENTS(XDESC,SUBGRIDS,NDIMS,IERROR)
      IF (NDIMS.GT.3) WRITE(6,*)'dimension too high'
      DO IDIM=NDIMS+1,2*NDIMS
         SUBGRIDS(IDIM)=1
      END DO

C Get processor masks
      DO IDIM=1,NDIMS
         PMASKS(IDIM)=0
      END DO
      CALL CMF_GET_PROCESSOR_MASKS(XDESC,PMASKS(NDIMS+1),
     &   NDIMS,IERROR)

c Lay out the axes (first half :SERIAL, second half :NEWS)
      DO IDIM=1,NDIMS
         ORDERS(IDIM)=CMF_SERIAL_ORDER
         ORDERS(NDIMS+IDIM)=CMF_NEWS_ORDER
      END DO

C Determine axis extents
      DO IDIM=1,NDIMS
         EXTENTS(IDIM)=SUBGRIDS(IDIM)
         EXTENTS(IDIM+NDIMS)=2**POPCOUNT(PMASKS(IDIM+NDIMS))
      END DO

C Determine offsets
      DO IDIM=1,2*NDIMS
         OFFSETS(IDIM)=0
      END DO

C Form alias
      CALL CMF_ALIAS_DETAILED(ALIAS,2*NDIMS,EXTENTS,ORDERS,
     &   SUBGRIDS,PMASKS,OFFSETS,.FALSE.,XDESC,IERROR)
      RETURN
      END
```


Copyright © 1993 Thinking Machines Corporation

```
      INTEGER FUNCTION POPCOUNT(III)
C Count the number of nonzero bits in an integer argument
      JJJ=III
      POPCOUNT=0
      DO (32) TIMES
         IF (IAND(JJJ,1).NE.0) POPCOUNT=POPCOUNT+1
         JJJ=JJJ/2
      END DO
      RETURN
      END

      SUBROUTINE STENCIL(X,M1,M2,N1,N2)
      DOUBLE PRECISION X(M1,M2,N1,N2), Y(0:M1+1,0:M2+1,N1,N2)
      DOUBLE PRECISION, ARRAY(0:M2+1,N1,N2) :: GHOST1, GHOST2
      DOUBLE PRECISION, ARRAY(0:M1+1,N1,N2) :: GHOST3, GHOST4
CMF$  LAYOUT X(:SERIAL,:SERIAL,:BLOCK=1:PROCS=N1, &
CMF      :BLOCK=1:PROCS=N2)
CMF$  LAYOUT Y(:SERIAL,:SERIAL,:BLOCK=1:PROCS=N1,  &
CMF      :BLOCK=1:PROCS=N2)
CMF$  LAYOUT GHOST1(:SERIAL,:BLOCK=1:PROCS=N1,:BLOCK=1:PROCS=N2)
CMF$  LAYOUT GHOST2(:SERIAL,:BLOCK=1:PROCS=N1,:BLOCK=1:PROCS=N2)
CMF$  LAYOUT GHOST3(:SERIAL,:BLOCK=1:PROCS=N1,:BLOCK=1:PROCS=N2)
CMF$  LAYOUT GHOST4(:SERIAL,:BLOCK=1:PROCS=N1,:BLOCK=1:PROCS=N2)

C Copy in bulk values
      Y=0.D0
      CALL CM_timer_start(1)
      Y(1:M1,1:M2,:,:)=X(1:M1,1:M2,:,:)

C Copy over ghost cells
      GHOST1(0:M2+1,:,:)=Y(1,      0:M2+1,:,:)
      GHOST2(0:M2+1,:,:)=Y(M1,     0:M2+1,:,:)
      GHOST3(0:M1+1,:,:)=Y(0:M1+1,1,      :,:)
      GHOST4(0:M1+1,:,:)=Y(0:M1+1,M2,     :,:)
      CALL CM_timer_stop(1)
      CALL CM_timer_start(2)
      GHOST1=CSHIFT(GHOST1,DIM=2,SHIFT=+1)
      GHOST2=CSHIFT(GHOST2,DIM=2,SHIFT=-1)
      GHOST3=CSHIFT(GHOST3,DIM=3,SHIFT=+1)
      GHOST4=CSHIFT(GHOST4,DIM=3,SHIFT=-1)
      CALL CM_timer_stop(2)
      CALL CM_timer_start(1)
      Y(M1+1,   0:M2+1,:,:)=GHOST1(0:M2+1,:,:)
      Y(0,      0:M2+1,:,:)=GHOST2(0:M2+1,:,:)
      Y(0:M1+1,M2+1,   :,:)=GHOST3(0:M1+1,:,:)
```


Copyright © 1993 Thinking Machines Corporation

```
      Y(0:M1+1,0,      :,:)=GHOST4(0:M1+1,:,:)
      CALL CM_timer_stop(1)

C Perform stencil operation
      CALL CM_timer_start(3)
      Y(1:M1,1:M2,:,:)=0.5*Y(1:M1,1:M2,:,:)
     .   +0.125*(Y(0:M1-1,1:M2,:,:)+Y(2:M1+1,1:M2,:,:)
     .   +Y(1:M1,0:M2-1,:,:)+Y(1:M1,2:M2+1,:,:))
      CALL CM_timer_stop(3)

C Copy back values
      CALL CM_timer_start(1)
      X(1:M1,1:M2,:,:)=Y(1:M1,1:M2,:,:)
      CALL CM_timer_stop(1)
      RETURN
      END
```


Copyright © 1993 Thinking Machines Corporation

Example 3

NOTE: This program is incompatible with "vector-length padding" of subgrids. Thus, it executes only on the CM-5 and, under the vector-units model, only when compiled with the **cmf** switch **-nopadding**.

```
      PROGRAM DETAILED_ALIASES
      IMPLICIT NONE
      INTEGER N

      WRITE (6,*) 'enter N: '
      READ (5,*) N

      CALL BOTH(N)

      END


      SUBROUTINE BOTH(N)
      IMPLICIT NONE
      INCLUDE '/usr/include/cm/CMF_defs.h'

      INTEGER N
      REAL *8 A(N,N),B0(N,N),B1(N,N)
      INTEGER AALIAS(CMF_SIZEOF_DESCRIPTOR)
      INTEGER B1ALIAS(CMF_SIZEOF_DESCRIPTOR)
      INTEGER LEXTENTS(4),GEXTENTS(4),RANK,ASTAT
      INTEGER ORDERS(4),P1,P2,PMASKS(4),OFFSETS(4)
      INTEGER I,J
CMF$  LAYOUT A(:NEWS,:NEWS)
CMF$  LAYOUT B0(:NEWS,:NEWS)
CMF$  LAYOUT B1(:NEWS,:NEWS)
CMF$  LAYOUT AALIAS(:SERIAL)
CMF$  LAYOUT B1ALIAS(:SERIAL)
CMF$  LAYOUT LEXTENTS(:SERIAL)
CMF$  LAYOUT GEXTENTS(:SERIAL)
CMF$  LAYOUT ORDERS(:SERIAL)
CMF$  LAYOUT PMASKS(:SERIAL)
CMF$  LAYOUT OFFSETS(:SERIAL)
```


Copyright © 1993 Thinking Machines Corporation

```
      B0 = 0
      B1 = 0
      FORALL (I = 1:N, J = 1:N)
     $   A(I,J) = J + 1000*I

      CALL CMF_GET_LOCAL_EXTENTS(A,LEXTENTS,RANK,ASTAT)

      IF (RANK .NE. 2) THEN
         WRITE (6,*) 'Confusion: rank = ',RANK
         GOTO 999
      ENDIF

      GEXTENTS(1) = LEXTENTS(1)
      GEXTENTS(2) = LEXTENTS(2)

      ORDERS(1) = CMF_SERIAL_ORDER
      ORDERS(2) = CMF_SERIAL_ORDER
      ORDERS(3) = CMF_NEWS_ORDER
      ORDERS(4) = CMF_NEWS_ORDER

      CALL CMF_GET_PROCESSOR_MASKS(A,PMASKS,RANK,ASTAT)

      P1 = PMASKS(1)
      P2 = PMASKS(2)

      PMASKS(1) = 0
      PMASKS(2) = 0
      PMASKS(3) = P1
      PMASKS(4) = P2

      OFFSETS(1) = 0
      OFFSETS(2) = 0
      OFFSETS(3) = 0
      OFFSETS(4) = 0

      LEXTENTS(3) = 1
      LEXTENTS(4) = 1

      GEXTENTS(3) = 1
      GEXTENTS(4) = 1
      DO I=1,32
         PRINT *,P1,P2,GEXTENTS(1),GEXTENTS(2),GEXTENTS(3),
     $      GEXTENTS(4)
         IF (IAND(P1,1) .EQ. 1) THEN
            GEXTENTS(3) = GEXTENTS(3)*2
```


Copyright © 1993 Thinking Machines Corporation

```
         ENDIF
         IF (IAND(P2,1) .EQ. 1) THEN
            GEXTENTS(4) = GEXTENTS(4)*2
         ENDIF
         P1 = ISHFT(P1,-1)
         P2 = ISHFT(P2,-1)
         IF (P1+P2 .EQ. 0) EXIT
      END DO

      IF (N .NE. (GEXTENTS(1) * GEXTENTS(3))) THEN
         WRITE(6,*) 'length problem on axis 1'
         GOTO 999
      ENDIF
      IF (N .NE. (GEXTENTS(2) * GEXTENTS(4))) THEN
         WRITE(6,*) 'length problem on axis 2'
         GOTO 999
      ENDIF

      CALL CMF_ALIAS_DETAILED(AALIAS,4,GEXTENTS,ORDERS,LEXTENTS,
     $   PMASKS,OFFSETS,.TRUE.,A,ASTAT)

      CALL CMF_ALIAS_DETAILED(B1ALIAS,4,GEXTENTS,ORDERS,
     $   LEXTENTS,PMASKS,OFFSETS,.TRUE.,B1,ASTAT)

      B0 = A + CSHIFT(A,DIM=1,SHIFT=1)

      CALL FOO(B1ALIAS,AALIAS,LEXTENTS(1),LEXTENTS(2),
     $   GEXTENTS(3),GEXTENTS(4))

      IF (ANY(B0 .NE. B1)) THEN
         WRITE(6,*) 'FAIL'
      ELSE
         WRITE(6,*) 'PASS'
      ENDIF

999   CONTINUE
      RETURN
      END


      SUBROUTINE FOO(B,A,SG1,SG2,N1,N2)
      IMPLICIT NONE
      INTEGER SG1,SG2,N1,N2
      REAL *8 A(SG1,SG2,N1,N2),B(SG1,SG2,N1,N2)
      REAL *8 OLD_EDGE(1,SG2,N1,N2),NEW_EDGE(1,SG2,N1,N2)
```


Copyright © 1993 Thinking Machines Corporation

```
CMF$   LAYOUT A(:SERIAL,:SERIAL,:NEWS,:NEWS)
CMF$   LAYOUT B(:SERIAL,:SERIAL,:NEWS,:NEWS)
CMF$   LAYOUT OLD_EDGE(:SERIAL,:SERIAL,:NEWS,:NEWS)
CMF$   LAYOUT NEW_EDGE(:SERIAL,:SERIAL,:NEWS,:NEWS)

      OLD_EDGE(1,:,:,:) = A(1,:,:,:)
      NEW_EDGE = CSHIFT(OLD_EDGE,DIM=3,SHIFT=1)

      B(1:SG1-1,:,:,:) = A(1:SG1-1,:,:,:) + A(2:SG1,:,:,:)
      B(SG1,:,:,:) = A(SG1,:,:,:) + NEW_EDGE(1,:,:,:)

      RETURN
      END
```


Copyright © 1993 Thinking Machines Corporation

5 FORALL Optimizations

This section describes enhancements to the **FORALL** statement that permit it to generate faster parallel code; it also notes cases where **FORALL** executes serially.

See the *CM Fortran Reference Manual* for the definition of the **FORALL** statement.

5.1 Operations Now Optimized

5.1.1 Gathers/Scatters on Serial Dimensions

FORALL can be used to express gather/scatter operations. An example of a scatter is:

```
FORALL (I=1:N) A(X(I),I) = B(I)
```

A gather is:

```
FORALL (I=1:N) B(I) = A(X(I),I)
```

When **FORALL** does a gather or scatter on a serial axis, the compiler can determine that the operation is entirely local and uses the fast indirect addressing hardware. See Section 2, above, for information on which axes the compiler recognizes as serial.

Examples of optimized indirect addressing:

```
      INTEGER SOURCE(NS,NP), DEST(NP), INDEX(NP)
CMF$  LAYOUT SOURCE(:SERIAL,), DEST(), INDEX()
      ...
      FORALL (I=1:NP) DEST(I)=SOURCE((I),I)
```

and,

```
      INTEGER, ARRAY(NS,NP) :: SOURCE, DEST, INDEX
CMF$  LAYOUT SOURCE(:SERIAL,),DEST(:SERIAL,)
CMF$  LAYOUTINDEX(:SERIAL,)
      ...
      FORALL (I=1:NS,J=1:NP)
     &   DEST(I,J)=SOURCE(INDEX(I,J),J)
```


Copyright © 1993 Thinking Machines Corporation

This form of **FORALL** on serial dimensions gives performance comparable to the utility library procedures **CMF_AREF/ASET_1D**.

Implementation Note: Here is a detailed definition of the class of indirect references that can be handled without **GET** or **SEND**. Let **A** be an array of rank *n+s,* laid out with *n* **:NEWS** axes and *s* **:SERIAL** axes. A **SERIAL-INDIRECT** memory reference is given by a **FORALL** expression with **FORALL** indices **J1,...,JM, I1,...,IN** with **M <= S**. The index on **:NEWS** axis **K** must be **IK**. The index on **:SERIAL** axis **Jk** can be any elemental expression involving constants, the **FORALL** indices, and other conforming **SERIAL-INDIRECT** memory references. Two **SERIAL-INDIRECT** references are conforming if the arrays have the same "reduced geometry" (i.e., the same geometry with serial axes removed).

For example:

```
 A(:NEWS,:NEWS,:SERIAL,:SERIAL)
 B(:SERIAL,:NEWS,:SERIAL,:NEWS)
 C(:SERIAL,:NEWS,:NEWS)
 D(:SERIAL,:SERIAL,:NEWS,:NEWS)

 FORALL(J1,J2,I1,I2)
&          D(J1,J2,I1,I2) =
&          A(I1,I2,I1+J1,B(J2,I1,I2 + C(J1,I1,I2),I2))
```

5.1.2 Spread Operations on Serial Dimensions

FORALL can be used to express spread operations. When done on serial axes the compiler can determine that the operation is entirely local and generates local memory accesses instead of communication routines.

```
      REAL A(10,10), B(10)
CMF$  LAYOUT A(:SERIAL,:NEWS), B(:NEWS)
      ...
      FORALL(I=1:10) A(I,:) = B(:)
```

See Section 2, above, for details on which axes the compiler recognizes as serial.

Copyright © 1993 Thinking Machines Corporation

5.1.3 Scan Operations

FORALL now generates run-time scan routines, including conditional and segmented scans and scans of scans, rather than generating the less efficient reduction-of-spread expressions. **FORALL** now gives performance comparable to the utility library procedures **CMF_SCAN_***combiner*.

Simple Scans

- Upward inclusive 1D scan, axis 1:

```
FORALL(I=1:N) A(I) = SUM(B(1:I))
```

- Upward exclusive 1D scan, axis 1

```
FORALL(I=1:N) A(I) = SUM(B(1:I-1))
```

- Upward inclusive 2D scan, axis 1:

```
FORALL(I=1:N,J=1:M) A(I,J) = SUM(B(1:I,J))
```

- Upward inclusive 2D scan, axis 2:

```
FORALL(I=1:N,J=1:M) A(I,J) = SUM(B(I,1:J))
```

Segmented Scan

- Upward inclusive segmented unconditional:

```
 FORALL (I=1:N) A(I) =
&    SUM(B(MAXVAL([1:I],SEGMENT(1:I)):I))
```

Scan of Scan

```
FORALL(I=1:N,J=1:M) A(I,J) = SUM(B(1:I,1:J))
```


Copyright © 1993 Thinking Machines Corporation

5.1.4 Comparison with Utility Library CMF_SCAN_

The scan capabilities of **FORALL** are largely, but not entirely, the same as those of the utility library procedures **CMF_SCAN_***combiner*. The comparative combiners are:

CMF_SCAN_*combiner*	**FORALL** expression
ADD	**SUM**
MAX	**MAXVAL**
MIN	**MINVAL**
mult scan (not available)	**PRODUCT**
LOGICAL IOR	**ANY**
LOGICAL IAND	**ALL**
LOGICAL IEOR	**MOD(COUNT(),2)==1**
INTEGER IOR	(not available)
INTEGER IEOR	(not available)
INTEGER IAND	(not available)

For example, a segmented add scan with **FORALL** can be written:

```
 FORALL(I=1:N)
&      A(I) = SUM(B(MAXVAL([1:I],SEGMENT(1:I)):I))
```

To do a copy scan, you can use a vector-valued subscript instead of the sum of a triplet expression. The triplet is `MAXVAL([1:I],SEGMENT(1:I))` as the lower bound and I is the upper bound.

The following examples show variations in direction, inclusion, and segment mode, as defined for **CMF_SCAN_***combiner*. Note that this form of **FORALL** can make use of two different masks:

- The reduction mask, the array-valued **MASK** argument to the reduction function, indicates which elements participate in the reduction.
- The **FORALL** mask, a scalar, indicates which elements are assigned.

Copyright © 1993 Thinking Machines Corporation

Each example is shown first with a reduction mask implicitly **.TRUE.** and then with a reduction **MASK** argument.

- 1D upward, inclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(1:I))
FORALL(I=1:N) A(I) = SUM(B(1:I),MASK=MASK(1:I))
```

- 1D downward, inclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(I:N))
FORALL(I=1:N) A(I) = SUM(B(I:N),MASK=MASK(I:N))
```

- 1D upward, exclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(1:I-1))
FORALL(I=1:N) A(I) = SUM(B(1:I-1),MASK=MASK(1:I-1))
```

- 1D downward, exclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(I+1:N))
FORALL(I=1:N) A(I) = SUM(B(I+1:N),MASK=MASK(I+1:N))
```

- 1D upward, inclusive, segment

  Note that the segment in this form must start with a **.TRUE.** element.

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],MASK=SEGMENT(1:I)):I))
```

  In this form, the segment starts with an implicit **.TRUE.**.

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAX(1,MAXVAL([1:I],
&   MASK=SEGMENT(1:I))):I)))
```

  The intrinsic **MAXVAL** returns most-negative-integer when its mask is null. Therefore, where segment is **.FALSE.** and the **MAXVAL** expression returns **MOST_NEG_INT**, the **SUM** expression would be **B(MOST_NEG_INT:I)**, which is an error. The expression **MAX(1, MAXVAL(...))** causes the **SUM** expression to be **1:I** even if segment is **.FALSE.**.

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAX(1,
&   MAXVAL([1:I],MASK=SEGMENT(1:I))):I))
```


Copyright © 1993 Thinking Machines Corporation

- 1D downward, inclusive, segment:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I:MINVAL([I+1:N],MASK=SEGMENT(I+1:N))-1))
```

- 1D upward, exclusive, segment:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],MASK=SEGMENT(1:I)):I-1))
```

- 1D downward, exclusive, segment:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I+1:MINVAL([I+1:N],
&   MASK=SEGMENT(I+1:N))-1))
```

- 1D upward, inclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],MASK=SEGMENT(1:I)):I))
```

- 1D upward, exclusive, start

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I-1],MASK=SEGMENT(1:I-1)):I-1))
```

- 1D downward, inclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I:MINVAL([I:N],MASK=SEGMENT(I:N))))
```

- 1D downward, exclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I:MINVAL([I:N],MASK=SEGMENT(I:N))-1)
```

5.1.5 Send-with-Combiner Operations

The **FORALL** statement generates parallel send-with-combiner instructions for *n*-to-*m*-dimensional scatters when the possibility of data collisions exists. For each dimension of the destination array, supply an index array that is conform-

Copyright © 1993 Thinking Machines Corporation

able with the source array, and then use a reduction function to combine multiple values being sent to the same destination element.

For example, a 1D-to-1D send-with-add operation is written as:

```
FORALL(I=1:8) A(I)=SUM(B(1:1000),MASK=V(1:1000).EQ.I)
```

where
`A` is `A(8)` of numeric type.
`B` is `B(1000)` of numeric type.
`V` is `V(1000)` of type integer.

A 1D-to-1D send-with-add operation that adds in the original destination value is written as:

```
FORALL(I=1:N) A(I) = A(I) + SUM(B(:), MASK=V(:).EQ.I)
```

For a 2D-to-2D send-with-add, use an index array (conformable with the source array) for each dimension of the destination array:

```
 FORALL(I=1:N,J=1:M)
&       OUT(I,J) =
&       SUM(IN(:,:),
&           MASK=(X(:,:).EQ.I).AND.(Y(:,:).EQ.J))
```

A 1D-to-2D send-with-add operation is written as:

```
 FORALL(I=1:N,J=1:M)
&       OUT(I,J)=
&       SUM(IN(:), MASK=X(:).EQ.I .AND. Y(:).EQ.J)
```

A restriction on this use of **FORALL** is that it generates parallel instructions *only if* the rank of **OUT** plus the rank of **IN** is 7 or less. For arrays of higher rank, use the utility library procedure **CMF_SEND_***combiner* for best performance.

5.2 Expressions Now Parallelized

The **FORALL** statement now accepts the following expressions and generates parallel expressions.

- References to all intrinsic functions and a few specific function calls that are listed below under “restrictions.”

Copyright © 1993 Thinking Machines Corporation

Previous versions of **FORALL** always serialized the transformational functions **CSHIFT**, **EOSHIFT**, **DIAGONAL**, **DOTPRODUCT**, **FIRSTLOC**, **LASTLOC**, **MATMUL**, **MAXLOC**, **MINLOC**, **PACK**, **PROJECT**, **REPLICATE**, **RESHAPE**, **SPREAD**, **TRANSPOSE**, **UNPACK**. and the inquiry functions **DSIZE**, **DSHAPE**, **DLBOUND**, **DUBOUND**, and **RANK**. As of Version 2.0, **FORALL** accepts these functions, except in the cases noted below.

- Use of a **FORALL** index variable in the following ways:
  - in a triplet subscript, such as `A(1:I)` or `B(I:I+5:2)`
  - in an array constructor, such as `[I]` or `[1:I]`
  - as an argument to a statement function, such as `FOO(I)`

5.3 Temporary Restrictions

FORALL has several temporary inefficiencies, where it either executes serially or fails to generate the most efficient parallel instructions.

- Temporarily, **FORALL** executes serially if it references:
  - the intrinsic function **RESHAPE** with a **PAD** argument, with triplet **SOURCE**, or with **MOLD** or **ORDER** arguments not specified with literal constants

- Temporarily, **FORALL** executes serially if an index variable is used in any of the following expressions:
  - front-end array expression
  - unaligned binary triplet expression, such as `B(0:I) + B(I:2I)`
  - multiple-term array constructor, such as `[B(I),C(I),D(I)]`
  - multiple-dimension expression to **MAXLOC** or **MINLOC**, such as `MAXLOC(B(I,:,:))`
  - **REPLICATE** along triplet axis, such as `REPLICATE(B(1:I),DIM=1,NCOPIES=8)`

Copyright © 1993 Thinking Machines Corporation

- Temporarily, **FORALL** does not generate the most efficient parallel instructions for array transfers between the parallel processors and the control processor. A simple CM-to-FE transfer operation is expressed as:

  ```
  FORALL (I=1:N) FE(I) = CM(I)
  ```

  In the current release, this statement generates a serial **DO** loop with a **READ_TO_PROCESSOR** or **WRITE_FROM_PROCESSOR**. It is better to use the utility procedures **CMF_FE_ARRAY_TO/FROM_CM**.

5.4 Permanent Restrictions

The following cause **FORALL** to execute serially:

- Reference to a character variable, such as `STRING(I)`.
- Reference to an external function.
- "Too many" triplet axes, such as `A(1:I,1:J,1:K,1:L)`. The operation executes serially if the number of vector axes (those with colons) plus the number of distinct **FORALL** indices referenced is above a certain threshold. The threshold in the current release is 7. The expression above references 4 vector axes and 4 indices, for a total of 8, and thus executes serially.
- **CSHIFT** or **EOSHIFT** of a triplet axis using a **FORALL** index, such as `CSHIFT( B(1:I),1,1 )`.
- A non-constant **DIM** argument, that is, a dimension not known at compile time, to any reduction function, or to the transformational functions **CSHIFT** or **EOSHIFT**, **FIRSTLOC** or **LASTLOC**, **PROJECT**, **REPLICATE**, **SPREAD**, or to the inquiry functions **DUBOUND**, **DLBOUND**, or **DSIZE**.

Another restriction is that **FORALL** does not support assumed-size arrays (those declared with * as the last axis extent) or the type **CHARACTER**.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

5.5 Using Alternative Forms of FORALL

Because of **FORALL**'s expressive power, there are often several ways to express a given parallel operation. However, the implementations, and thus the performance characteristics, of the alternative forms are usually different. One form may be better or worse than another depending on other factors, such as the size or layout of the array.

For example, both the following statements extract the diagonal elements from one matrix and store them in the diagonal of another matrix:

```
FORALL (I=1:N) A(I,I) = B(I,I)

FORALL (I=1:N,J=1:N,I.EQ.J) A(I,J) = B(I,J)
```

The first statement is a 1-dimensional algorithm. The compiler generates a **GET** instruction to a 1-dimensional array from a 2-dimensional array, and then a **SEND** instruction to transfer the result to the 2-dimensional destination.

The second statement is a 2-dimensional algorithm. The compiler generates a 2-dimensional array assignment under a mask.

Copyright © 1993 Thinking Machines Corporation

6 Other Optimizations

Additional optimizations in this version of CM Fortran include:

- global optimizer (**-O**) improvements
- run-time alignment checking of dynamic arrays
- optimized single-precision memory accesses (in vector-unit code)
- array geometry checked in caller, not in subprogram
- CMSSL speeds some CM Fortran code when linked in

6.1 Global Optimizer Improvements

The global optimizer (**-O** switch) now performs some dead-code elimination and copy propagation. Copy propagation is one pass, useless assignment elimination is a second pass.

6.1.1 Dead-Code Elimination

As an example of dead-code elimination, consider this subroutine:

```
SUBROUTINE FOO(C,A,B,N)
IMPLICIT NONE
INTEGER N
REAL *8 C(N),A(N),B(N)
REAL *8 TMP(N)

TMP = A + B
C = TMP

RETURN
END
```

Previous versions of the compiler generated 3 loads and 2 stores for this subroutine. The current version generates 2 loads and 1 store.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

6.1.2 Copy Propagation

As an example of copy propagation, consider these assignments:

```
A = A + 1
A = A + 1
A = A + 1
```

If we copy propagate the above, we get

```
A = A + 1
A = (A + 1) + 1        ! one copy prop
A = ((A+1) + 1) + 1    ! two copy props
```

If we now evaluate this, assuming **A** is 0 on entry, we get 3 from the first set of statements and 6 from the second. If, however, we have

```
B = A + 1
B = B + 1
A = B + 1
```

we can copy propagate to

```
B = A + 1
B = (A + 1) + 1
A = ((A + 1) + 1) + 1
```

and now the two stores to **B** can be killed, and we get **A = 3** when we execute it. In essence, you can't copy past the point at which any quantity on the right-hand side is modified. In the second example, the statement **B = B + 1** satisfies this condition because we can replace **B** with **A + 1** and still have a correct program with no dead code removed.

6.1.3 Second, or "Post," Run of Optimizer

Another enhancement is that the optimizer, which previously ran once after the compiler's "build" phase, now runs a second time after the "middle end" phase. A great deal of generated code can be moved out of loops by this second run.

Copyright © 1993 Thinking Machines Corporation

6.1.4 Selectively Disabling New Optimizer Features

If you suspect that these new optimizer features are causing problems, you can selectively disable them with the following internal switches:

`-nocprop`	disables copy propagation
`-nodce`	disables dead-code elimination
`-nopostoptimize`	disables the second optimizer run

Copyright © 1993 Thinking Machines Corporation

6.2 Alignment of Dynamic Arrays

Previous releases often generated excessive communication for operations involving deferred-shape arrays (**ALLOCATABLE** and **POINTER**), since the compiler cannot determine their associated geometry at compile time.

Judicious use of the **ALIGN** directive can sometimes help avoid needless communication for arrays of different shapes. Dynamic arrays of the same shape are sometimes recognized as such by the compiler (as described below). If not, the compiler generates a new routine, reported by **-list** as **BLOCKMOVE**, which does geometry comparison at run time and chooses the most efficient code for an assignment where at least one operand is a dynamic array.

6.2.1 The Compiler's View of Dynamic Arrays

For static arrays and local assumed-shape arrays, the compiler generates code for the allocation of memory upon entry to the program unit. This code assigns the geometry that will be defined and used at run-time. For dynamic arrays, however, space is not actually allocated until the **ALLOCATE** statement executes. The reason for this is that dynamic arrays may be allocated and deallocated more than once in each program unit, and the control flow of the code may dictate that one **ALLOCATE** may be executed at a different point in the execution than another **ALLOCATE**. For example:

```
      SUBROUTINE FOO(N)
      REAL, ALLOCATABLE, ARRAY(:) :: A

      IF (N .GT. 100) GOTO 100
      ALLOCATE(A(100))
      A = [1:100]
      GOTO 200
100   ALLOCATE(A(N))
      A = [1:N]

200   PRINT *, A
      DEALLOCATE(A)
      END
```

Because of this, the compiler must generate the call to assign (intern) the geometry at the point that the array is allocated. Unfortunately, the compiler must also still be able to refer to the geometry at any place that the array is referenced, even though the array has not been assigned a geometry. In order to do this, the com-

Copyright © 1993 Thinking Machines Corporation

piler allocates a temporary that will hold the result of the call to the run-time routine **intern_geometry**; when the geometry is needed at compile time, code that represents a fetch of the value stored in that temporary is generated.

Since the dynamic array's size may be different at different points in the execution of the program, no assumptions about the geometry of the array are made with respect to other arrays that may be the same size at a given point in the execution. In previous releases, the compiler would therefore generate a **SEND** when it encountered references to dynamic arrays:

```
      SUBROUTINE FOO(N)
      REAL,POINTER,ARRAY(:) :: A
      REAL B(N)

100   ALLOCATE(A(N))

      B = [1:N]
200   A = B
300   PRINT *, A
      DEALLOCATE(A)
      END
```

In this example, the assignment of array **B** to array **A** at the line labeled 200 required a **SEND** of **B**, even though the two arrays are conformable.

6.2.2 ALIGN Directives and Dynamic Arrays

The **ALIGN** directive allows the compiler to make assumptions about how a dynamic array will be used. The compiler can then attempt to generate more efficient code.

The **ALIGN** directive may be used with dynamic arrays in one of two ways:

- Align a dynamic array with a non-dynamic array:

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH X(I)
```

- Align a dynamic array with another dynamic array:

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH B(I)
```

Aligning Dynamic Arrays with Non-Dynamic Arrays

You can **ALIGN** an allocatable array or array pointer with an explicit-shape or assumed-shape array. In either situation, the compiler assigns to the dynamic array the same geometry that it assigns to the target array. For example, in this case,

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH X(I)

      ALLOCATE(A(N))
 100  X = A
```

the array **A** will be assigned the same geometry as the array **X**. Therefore, the compiler should not generate a **SEND** for the assignment at the statement labeled 100.

Similarly, in the case,

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(1000)
CMF$  ALIGN A(I) WITH X(I)

      ALLOCATE(A(1000))
 100  X = A
```

the compiler should not generate a **SEND** for the assignment labeled 100.

Copyright © 1993 Thinking Machines Corporation

Aligning Dynamic Arrays with Other Dynamic Arrays

You can **ALIGN** one allocatable array or array pointer with another dynamic array:

```
        SUBROUTINE FOO(N)
        INTEGER N
        REAL*8, ALLOCATABLE :: A(:), B(:)
        REAL*8 X(N)
CMF$    ALIGN A(I) WITH B(I)

        ALLOCATE(A(N))
        ALLOCATE(B(N))
 100    B = A
```

In this case, there should not be a **SEND** generated for the assignment labeled 100.

6.2.3 Using ALIGN Effectively with Dynamic Arrays

Align Multiple Arrays

If the desired effect is to have dynamic arrays aligned with both static and dynamic arrays, use one static array as the base destination of the **ALIGN** directive:

```
        SUBROUTINE FOO(X,N)
        INTEGER N
        REAL *8, ALLOCATABLE :: A(:), B(:)
        REAL*8 X(N+5)
        REAL*8 C(N+5)
        REAL *8 ANS
CMF$    ALIGN A(I) WITH X(I)
CMF$    ALIGN B(I) WITH X(I)
CMF$    ALIGN C(I) WITH X(I)
```

In this example, the arrays **A**, **B**, and **C** are all aligned with the base destination array **X**, which is a dummy argument array. The compiler then views all four arrays as having the same geometry.

Copyright © 1993 Thinking Machines Corporation

Avoid Using Allocatables Unnecessarily

In many cases, an assumed-shape dummy or local array may be just as useful as a dynamic array. For example, if the array shape is determined by an argument passed into a subprogram, then an assumed-shape array will work as well.

Instead of this:

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:)
      ALLOCATE(A(N))
```

Do this:

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL*8 A(N)
```

Be Careful!

As always when you use directives, make sure that an **ALIGN** directive with dynamic arrays does not conflict with the code. Ill-defined code sequences will result in undefined results. For example:

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:)
      REAL*8 X(N+5)
      REAL*8 C(N+5)
      REAL*8 ANS
CMF$  ALIGN A(I) WITH X(I)

 100  ALLOCATE(A(-200:0))

      A(-100:0) = X(1:101)
```

The compiler assumes that the array **A**, when allocated, will begin at the same lower bound as the array **X**. Yet, while the dummy array **X**'s lower bound is 1, the allocatable array **A**'s lower bound will be –100 at the time of allocation. In this situation, confusion reigns.

Copyright © 1993 Thinking Machines Corporation

6.3 Compile-Time Shape Checking

The bounds expressions used in an **ALLOCATE** statement may enable the compiler to determine whether a deferred-shape array is conformable with other arrays. In the absence of directives, conformable arrays are known to have the same geometry, and the compiler can avoid generating communication for operations between them.

The compiler checks the bounds expressions in **ALLOCATE** statements for:

- **ALLOCATABLE** arrays
- Array pointers that are not in **COMMON**

If the bounds are specified with:

- Constant expressions
- Invariant dummy arguments
- **DUBOUND** or **DLBOUND** expressions referencing other arrays (the compiler also accepts the Fortran 90 **UBOUND/LBOUND**).

the compiler can determine whether the deferred-shape array is conformable with other arrays used with it, including other deferred-shape arrays, as well as static, automatic, and dummy arrays.

The compiler cannot make assumptions about shape in the following cases (and so does not check bounds):

- Array names that appear in more than one **ALLOCATE** statement in the program unit (even if the statements are within a conditional construct and only one will be executed)
- Array pointers in **COMMON** (since they may have been reallocated in another program unit)

For example, the compiler can determine that these two arrays are conformable because the **ALLOCATABLE**'s upper bound is a constant:

```
PARAMETER (N=1000)
INTEGER ARRAY(N) :: A
INTEGER, ARRAY(:), ALLOCATABLE :: B
...
ALLOCATE (B(N))
```

The compiler can determine that these two pointers are conformable because **B**'s upper bound is derived from **A**'s with **DUBOUND**:

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

```
INTEGER, ARRAY(:), POINTER :: A,B
READ *,N
...
ALLOCATE (A(N))
ALLOCATE (B(DUBOUND(A,1)))
```

If the second example had been written

```
READ *,N
...
ALLOCATE (A(N))
...
ALLOCATE (B(N))
...
B=A
```

the compiler would not try to prove that **N** had not changed between the **ALLOCATE** statements. Instead, it would generate **BLOCKMOVE** code before the assignment to check for alignment at run time. (See the release notes for 2.1 Beta 2.0 for information on run-time alignment checking of dynamic arrays.)

In addition, the compiler does not check for multiple uses of a bounds expression within a single **ALLOCATE** statement unless the expression meets the criteria given above. For example, it would generate **BLOCKMOVE** code for operations between these two arrays:

```
READ *,N
...
ALLOCATE (A(N), B(N))
```

6.3.1 Run-Time Geometry Checking

When the compiler cannot compare the shapes of dynamic arrays (as described in the previous section), it generates code reported by **-list** as **BLOCKMOVE**. This item indicates code that compares the geometries of source and destination arrays at run time when either operand is a deferred-shape array. The code then generates memory accesses if possible; if not, **NEWS** communication if possible; or **SEND** routines as a last resort.

Note, however, that there may still be excess overhead even when **BLOCKMOVE** translates to local memory accesses. This is because **BLOCKMOVE** causes the compiler to break the current PN code block and move data to a temporary (in case communication is generated at run time).

Copyright © 1993 Thinking Machines Corporation

6.4 Single-Precision Memory Accesses in Vector Unit Code

Conditional and unconditional stores of single-precision **REAL** data have been speeded up significantly for the CM-5 vector units. Based on run-time checking of alignments and counts, the system can sometimes replace single-precision stores at vector length 8 with faster double-precision stores at vector length 4.

This release also optimizes loads and stores of single-precision **COMPLEX** data (for vector units only). The system can load or store a single **COMPLEX** result at vector length 8 by performing a double-precision **REAL** memory access at vector length 8 (instead of two single-precision memory accesses at vector length 8). This requires interleaving the data into a double register.

For example,

```
COMPLEX, ARRAY(N1,N2,N3):: A, B, C
...
A = B+C
```

previously generated two single-precision stores. Now, it can generate one double-precision store.

Copyright © 1993 Thinking Machines Corporation

6.5 A Change: Geometries of Array Arguments

In previous releases, the geometry associated with a CM array argument was determined upon entry to a subroutine based on the declaration in the subroutine. In the current release, the geometry of a CM array argument is determined from the descriptor of the actual argument (at the caller level).

A side effect (unsupported) of the previous behavior was that it allowed limited forms of equivalencing to occur at program boundaries. The new behavior prevents this.

For example, the following program fails because the geometry used for **B** is the geometry stored in **A**'s descriptor, thus violating the CM Fortran restriction against changing array shape across program boundaries:

```
      REAL A(10,100)
CMF$  LAYOUT A(:SERIAL,:NEWS)
      CALL FOO(A)
      ...
      SUBROUTINE FOO(B)
      REAL B(2,5,100)     ! Error: shape mismatch
CMF$  LAYOUT B(:SERIAL,:SERIAL,:NEWS)
      ...
```

Similarly, the following program fails because of the attempt to change array **C**'s type and shape in the subroutine:

```
      COMPLEX C(8,128)
CMF$  LAYOUT C(:BLOCK=8 :PROCS=1,:BLOCK=1 :PROCS=128)
      CALL BAR(C)
      ...
      SUBROUTINE BAR(D)
      REAL D(16,128)      ! Error: type/shape mismatch
CMF$  LAYOUT D(:BLOCK=16:PROCS=1,:BLOCK=1 :PROCS=128)
      ...
```


Copyright © 1993 Thinking Machines Corporation

6.6 Linking with CMSSL to Enhance Performance

The following CM Fortran procedures are enhanced by linking with CMSSL Version 3.1: **CMF_RANDOM**, **CMF_SORT**, **CMF_ORDER**, **CMF_RANK**, and the intrinsic **MATMUL**.

The procedure **CMF_RANDOM_LONG_S_INTEGER** will be enhanced by linking with CMSSL Version 3.2.

Note that when linking with CMSSL Version 3.1, the use of the **cmf** switches **-nopadding** and **-noaxisreorder** is not supported.

6.6.1 Linking with CMSSL on the CM-5

To compile a program called **p1** on a CM-5 and link it with the CMSSL, issue one of the following command lines at the UNIX prompt:

In the CM Fortran vector-units model:

```
% cmf -cm5 -vu -o p1 p1.fcm -lcmsslcm5vu
```

In the CM Fortran (SPARC) nodes model:

```
% cmf -cm5 -sparc -o p1 p1.fcm -lcmsslcm5
```

In the CM Fortran single-node model:

```
% cmf -cm5 -vu -node -o p1 p1.fcm -lcmsslcm5vu-node
```

Add **-cmmd_root /usr/cmmd/3.0** before **-lcmsslcm5vu-node** if CMMD 3.0 (or higher) is not the default on your system.

6.6.2 Linking with CMSSL on the CM-2/200

To compile a program called **p1** using the slicewise execution model on a CM-200, issue the following command at the UNIX prompt:

```
% cmf -slicewise -cm200 -o p1 p1.fcm -lcmssl-cm200-s
```

To use the slicewise execution model on a CM-2, issue the following command:

```
% cmf -slicewise -cm2 -o p1 p1.fcm -lcmssl-cm2-s
```


Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

7 Global/Local Programming

Global/local programming extends the data parallel model provided by CM Fortran, by allowing 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.

At this 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).

Global/local programming extends the data parallel model provided by CM Fortran, by allowing 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.

NOTE

This is the first release of global/local programming in CM Fortran. Your feedback is welcome as we plan the further development of this functionality.

A global/local application begins with a global main program, 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.

Copyright © 1993 Thinking Machines Corporation

At any time thereafter, the application can take explicit control of the nodes by calling a *local routine*. Invoking the local routine temporarily transforms the application into a nodal program, executing in message-passing style:

- Each node operates independently while executing the local routine.
- Global arrays (defined and allocated by the global program) can be passed as arguments to local routines. CM arrays and front-end arrays are treated differently:
  - Local routines operate on parallel arrays (known in CM Fortran as CM arrays) in place, with each node operating on its own portion, or *subarray*, of the array.

    Because the operations occur in place, local code can alter the value of a parallel array.
  - Each node operates on its own copy of serial arrays (known in CM Fortran as front-end arrays). The global program must make the copies before the local routine can use them.

    Because local code cannot pass values back to the host, it cannot alter values in global serial arrays.
- While executing local code, the nodes may use CMMD message-passing functions to communicate, synchronize, and share data with other nodes.
- Local code cannot perform I/O (including **`PRINT`** statements). Neither can it communicate with the partition manager. (From CMMD's point of view, the local routines function as hostless programs.)
- When the end of the original local routine is reached, all nodes synchronize, then return control to the partition manager to continue the global program.
- Local routines must be subroutines; they cannot return values to the global CM Fortran program.

The global/local model can be thought of in terms of threads of control, and in terms of the visibility of data. A global program has a single thread of control. Under that control, all data in the system is visible to all processors: system software performs interprocessor communication when needed. When the global program invokes a local routine, it yields its single control in favor of many threads of control (one per node). Each node follows its own control path until the end of the local routine. When all nodes have finished the local routine, con-

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

trol returns to the global program's single thread of control. Within the scope of the local routine, each node acts upon (and sees) only its own data. If communication of data or of control among nodes is required, the application must perform that communication by including explicit message-passing code (that is, calls to the CMMD message-passing library) in the local routines.

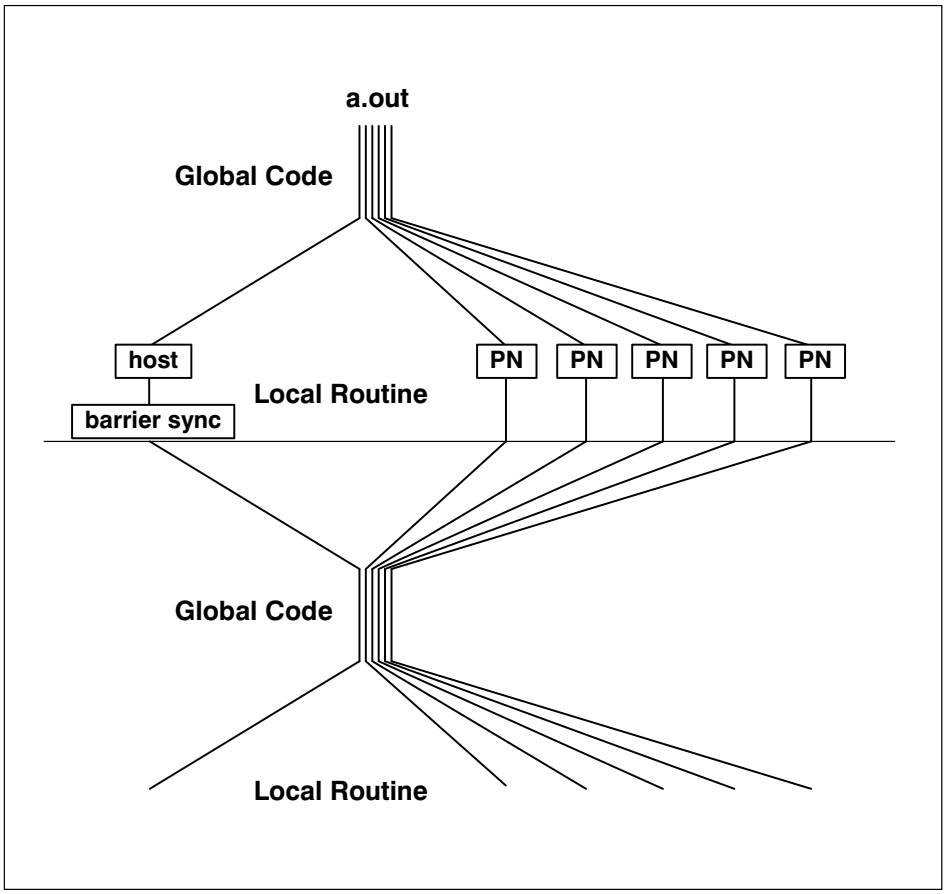

Figure 7. Programmer's view: Global/local programming.

The following sections describe global/local programming in more detail, then explain how to construct a global/local program.

Copyright © 1993 Thinking Machines Corporation

7.1 Flow of Control

When a CM Fortran program calls a local routine, one copy of the local routine is invoked on each CM-5 node. The copies of the local routine run independently; they may use any appropriate control structure.

CM system software synchronizes the nodes before the local routine begins execution; it synchronizes them again when control returns to the global CM Fortran program. Between these two times, any synchronization is up to the user program, and must be accomplished, either explicitly or implicitly, with CMMD calls. (**`CMMD_sync_with_nodes`** is an example of a CMMD function that synchronizes the nodes explicitly; **`CMMD_reduce_to_nodes_v`** is an example of a CMMD function that synchronizes the nodes implicitly.)

NOTE

If any CMMD calls are made during the local routine, the routine must ensure that all such calls have completed, and that the network is empty, before returning control to the global program.

More specifically, the program must guarantee that when a given node returns from that instance of the local routine, no more messages of any sort will be directed at that node by any other node. In particular, this means that if you're sending active messages, you need to keep track and make sure that all active messages aimed at a node have been sent and received before you allow that node to exit from the local routine.

Failure to follow this rule will probably crash your program.

Local routines may call other local subroutines or functions, but may not call any global routines. At this first release, they may not perform any I/O.

Local routines are always subroutines; they are not allowed to return values to the global program.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

7.2 Global/Local Data

The only global data visible to local routines is that which is passed to them as arguments. Local routines cannot see data in global common blocks.

Local routines may create their own arrays, common blocks, and so on. Data in these structures is visible only within the scope of the routine that creates them, and only on the calling node. (To pass such data from one node to another, CMMD calls must be used.)

Arguments passed to local routines may include global parallel arrays, global serial arrays, and scalar values. **`CHARACTER*`** arguments are not supported.

7.3 Arrays

Global/local programs use four kinds of arrays:

- global parallel arrays

  Arrays allocated by the global program, which are spread across the memory of the VUs, with a subgrid of similar size and location in each VU's memory bank. (CM Fortran calls these CM arrays.)

- global serial arrays

  Arrays allocated by the global program, which live in the partition manager's serial memory. (CM Fortran calls these front-end arrays.)

- local parallel arrays

  Arrays allocated by a local routine, which are spread across the 4 VU memory banks of that node.

- local serial arrays

  Arrays allocated by a local routine which live in the node's serial memory.

Copyright © 1993 Thinking Machines Corporation

7.3.1 Parallel Arrays

Global parallel arrays can be passed directly into local routines as arguments. Each node can then act upon the "subarray" of the global array resident in its VUs. The rank of the subarray is the same as that of the global CM array, and serial axes are preserved. Subarrays are indexed beginning with 1 (for CM Fortran code) or 0 (for C code) for all nodes, regardless of each subarray's position within the global array.

Assume, for example, a 1024-element vector **X**, on a 32-node partition. Each node would hold 32 elements of the vector, 8 elements per VU. Global code sees **X** as a single vector, 1024 elements long. When it passes **X** to a local routine — **CALL LOCAL_ROUTINE(X)** — the local code on each of the 32 nodes sees **X** as signifying its own 32-element array.

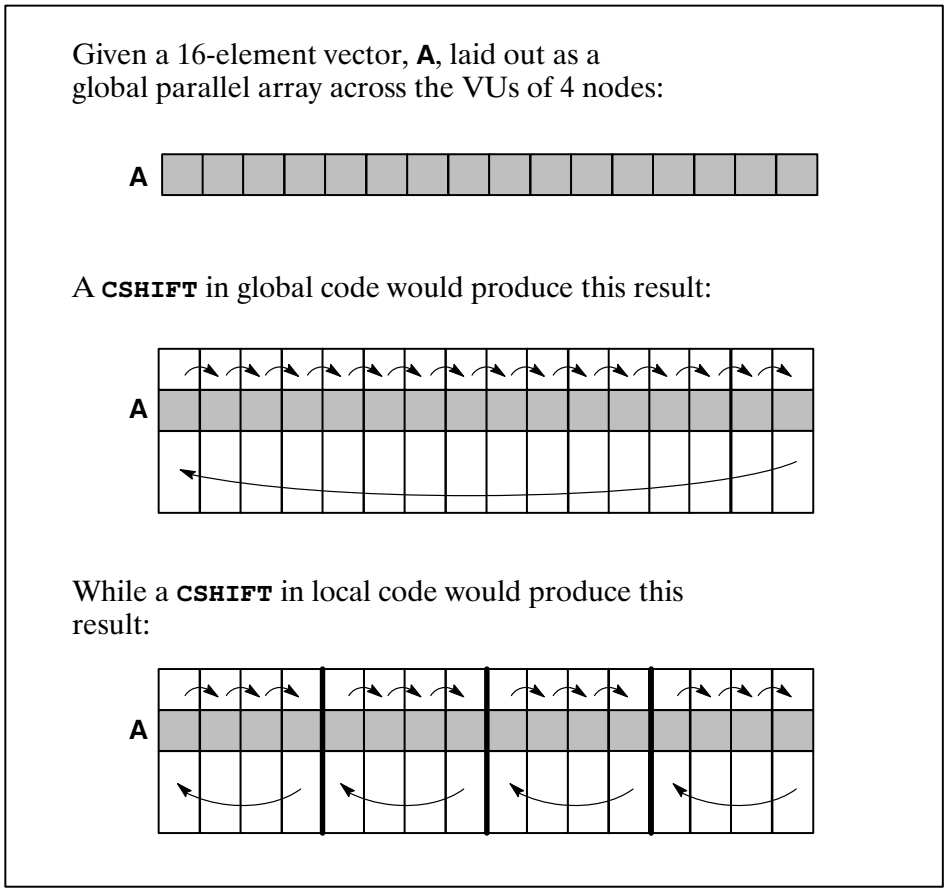

Figure 8. Computing on arrays and subarrays.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

Local code thus treats these arrays as if they were local. For example, if each node were to do a **CSHIFT** on X during the local routine, the shift would take place on each group of 32 elements, independently of the rest; a call to the **DSIZE** intrinsic function would return a size of 32; and so on. (See Figure 2.) If a node wants to find out where its elements reside within the global array, it uses CMGL library procedures.

Note that the global parallel array and the subarrays seen by the local routines on the nodes refer to exactly the same data in parallel CM memory. Any changes made to subarray data during a local routine will therefore be visible to global code after the return from the local routine. Should the global code then further change the array data and re-call the local routine, the new changes would be visible to the local routine.

> **Implementation Note:** When a parallel array is passed to a local routine, each node receives a descriptor defining its elements. This is entirely transparent to local routines written in CM Fortran. Local routines written in C, however, use information within the descriptor to access the VU memory that holds the array.

Local parallel arrays are created within the local routines, and are visible only within the scope of those routines. Like subarrays, local parallel arrays are divided into 4 similarly sized and shaped "subgrids", one per VU memory bank. Local routines can thus treat both types of parallel arrays in the same manner.

Any CMMD calls that can handle parallel arrays can handle either global or local parallel arrays within local routines. The one distinction the programmer must remember is that garbage elements for local parallel arrays can be expected to be in identical positions on all nodes, whereas garbage elements for global arrays may vary widely from node to node: a given node's subarray may contain some, all, or no garbage elements. Remember, too, that CMMD functions know nothing about garbage elements. They ignore the garbage masks and send the entire array.

Two subroutines, **CMGL_local_to_global** and **CMGL_global_to_local**, allow a local routine to translate between the global and local view of a parallel array. Section 7.5 describes these function calls.

Copyright © 1993 Thinking Machines Corporation

7.3.2 Serial Arrays

Global serial arrays, created by global CM Fortran programs, reside on the partition manager (PM). Local serial arrays, created by local routines, reside in microprocessor memory on each node.

Local routines have limited access to global serial arrays:

- First, the global program must call **CMGL_BROADCAST_SERIAL_ARRAY**, providing a new name (a "handle") for the array. This function allocates serial memory on each node and copies the array into that memory. (Each node receives a copy of the entire array.)
- Second, the global program passes the handle as an argument to a local routine.
- Local code on any given node can then access that node's copy of the serial routine. It cannot access or alter the original array, which remains on the PM.

Serial arrays are relatively static. Once created, they cannot be updated by the global program to reflect any changes to the front-end array, and they cannot be deallocated. Thus, they are best used for such static purposes as table lookups.

The following example copies a 4-integer (16-byte) vector, **Y**, into microprocessor memory, and then passes the starting address of that memory as an argument to a local routine:

```
HANDLE = CMGL_BROADCAST_SERIAL_ARRAY(Y,16)
CALL LOCAL_ROUTINE(HANDLE)
```

Unlike global parallel arrays, global serial arrays cannot be modified within the scope of local routines. The reasons for this are that each node is working on a local copy of the serial array, not on the array itself (which remains in the partition manager's memory), and that neither local routines nor any procedures they may call can pass data to the partition manager or back to the global program. If data from a serial array must be made visible to some global procedure from a local routine, the local routine must transfer the data from a serial array into a global parallel array. Note that this requires the global procedure to have allocated the parallel array before calling the local routine. Note also that the minimum size for the parallel array is four elements per node (one element per VU).

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

(Local code may, of course, modify any local serial arrays or scalar data that are visible within the scope of the local routine; it is only the modification of global serial data that is prohibited.)

Copyright © 1993 Thinking Machines Corporation

7.4 Restrictions

Global/local applications must observe certain restrictions:

- Local routines may call other local subroutines or functions. They must not call global CM Fortran code.
- Local routines may not return values to global routines.
- Local routines cannot use **`CHARACTER*`** arguments.
- Local routines may not modify scalar arguments.
- The only global data visible within the scope of local routines is data that has been passed as arguments to the local routines. In particular, local routines cannot access data in global common blocks.
- Any global array to be passed as an argument to a local routine must be 1-based. Arrays with lower bounds other than 1 are not supported.
- Parallel I/O cannot be performed within the scope of a local routine. (At this release, no I/O can be performed within the scope of a local routine.)
- If any CMMD communications are performed within the scope of a local routine, the program must ensure that the network is empty before the local routine returns control to the global program.

7.5 Global/Local Library Functions

Global/local programming makes use of three specialized library routines: one that transfers global serial arrays from front-end memory to node memory, and two that provide information on the mapping between global parallel arrays and subarrays.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMGL_BROADCAST_SERIAL_ARRAY

Allocates space for, and copies, a serial array from the partition manager to every node.

CM Fortran Syntax

```
INCLUDE '/usr/include/cm/cmgl.h'
INTEGER = CMGL_BROADCAST_SERIAL_ARRAY( serial_array, num_bytes )
```

Arguments

serial_array	The global serial array to be copied onto the nodes.
num_bytes	Scalar integer. Number of bytes to be copied; also, the amount of microprocessor memory to be allocated for the array.

Result

Integer scalar representing the starting address of the new array, which can then be passed as an argument to one or more local routines.

Description

This function is to be called by the global portion of a global/local program.

Given a CM Fortran front-end array (a "serial array") as argument, **CMGL_BROADCAST_SERIAL_ARRAY** first allocates microprocessor memory for an identical array on every node, then copies the front-end array into that memory. The nodal array remains allocated throughout the remainder of the program; it cannot be deallocated.

For example:

```
HANDLE = CMGL_BROADCAST_SERIAL_ARRAY(THIS_ARRAY, 1024)
CALL LOCAL_ROUTINE(HANDLE)
```

If this function is called twice on the same array, it will allocate and broadcast two separate nodal arrays.

Copyright © 1993 Thinking Machines Corporation

The *num_bytes* argument is determined by the number of elements in the array (the product of all its axis lengths) and by the storage size of each element in bytes. For the allowed array element types, the storage sizes are:

```
INTEGER*4, REAL, LOGICAL                    4 bytes
INTEGER*8, DOUBLE PRECISION, COMPLEX        8 bytes
DOUBLE COMPLEX                              16 bytes
```

For example, to transfer a 2-dimensional array of double-precision values, one might write:

```
SUBROUTINE GLOBAL_ROUTINE( ... )
DOUBLE PRECISION   DOUBLES_ARRAY(2,3)
INTEGER            ARRAYLOC
ARRAYLOC = CMGL_BROADCAST_SERIAL_ARRAY(DOUBLE_ARRAY, ( 2 * 3 * 8 ))
CALL LOCAL_ROUTINE(ARRAYLOC)
...

SUBROUTINE LOCAL_ROUTINE(SERIAL_ARRAY)
DOUBLE PRECISION   SERIAL_ARRAY(2,3)
...
```

Note that the amount of data specified by *num_bytes* will be sent, whether it matches the array size or not. No checking will be done. Similarly, if the *serial_array* argument does not identify a serial array, the wrong data will be transferred to the nodes.

Error Conditions

If you try to call this subroutine from a local routine, your program will fail to link.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMGL_local_to_global

Given local array indices, provides global array indices.

CM Fortran Syntax

```
CALL CMGL_LOCAL_TO_GLOBAL( array, l_index, g_index )
```

C Syntax

```
void CMGL_local_to_global( CMRT_desc_t array,
                           int *l_index, int *g_index )
```

Arguments

array	A subarray (that is, the calling node's portion of a global parallel array). For C routines, the argument type is a CMRT array descriptor (**CMRT_desc_t**); for CM Fortran routines, it is the data type of *array*.
l_index	A one-dimensional integer array whose size is equal to the rank of *array*. It contains the coordinates of an element within the subarray, *array*. In CM Fortran, it is an **INTENT(IN)** argument.
g_index	A one-dimensional integer array whose size is equal to the rank of *array*. It receives the coordinates within the global array of the element identified by *l_index*. In CM Fortran, it is an **INTENT(OUT)** argument.

Description

CMGL_local_to_global converts a set of local coordinates within a subarray to an equivalent set of global coordinates within the associated global array. Both sets of coordinates assume a 1-based array. If the local indices are out of bounds for the subarray, *g_index* will be filled with –1s.

CMGL_local_to_global must be called only from local routines, and only on subarrays. This procedure is a subroutine in CM Fortran, a function returning void in C.

Copyright © 1993 Thinking Machines Corporation

Example

Imagine a 2-dimensional global array named **A**, dimensioned 16 x 16, divided across four nodes. Each node wants to find out where in the global array its subarray begins.

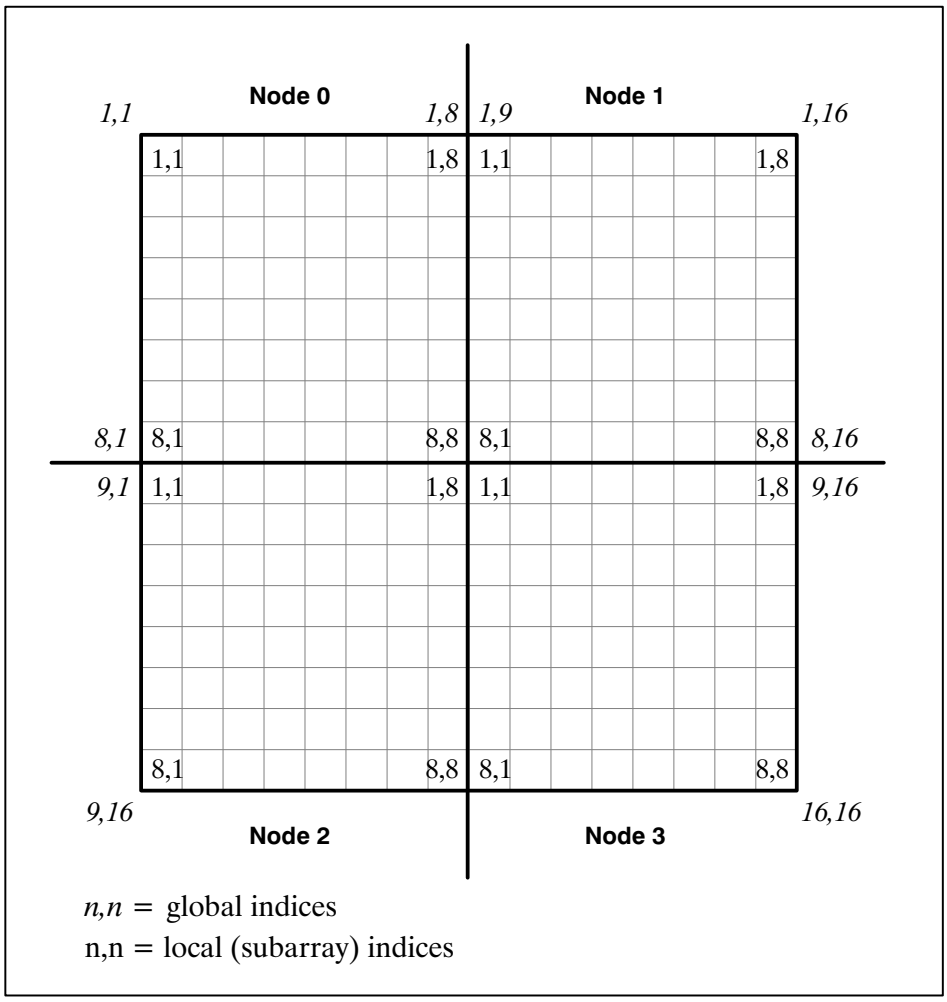

Figure 9. A 2D array and its four subarrays.

Each node will call **CMGL_local_to_global**; for each call, array will be **A** and *l_index* will be a two-element array containing two 1s. After the call executes, *g_index* will contain 2 integers representing the position in the global array of each subarray's starting element. For Node 0, this will be 1,1; for Node 1, it might be 1,9; for Node 2, 9,1; for Node 3, 9,9.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

Error Conditions

Calling **`CMGL_local_to_global`** or **`CMGL_global_to_local`** on an array which is not a subarray (that is, it is not a global parallel array that was passed to the local routine as an argument), causes an error that the compiler cannot catch at compile time. As a result, such errors will probably cause your program to crash.

Calling either of these routines from a global program will cause a failure at link time.

Copyright © 1993 Thinking Machines Corporation

CMGL_global_to_local

Given local array indices, provides global array indices.

CM Fortran Syntax

```
CALL CMGL_GLOBAL_TO_LOCAL( array, l_index, g_index, local )
```

C Syntax

```
void CMGL_local_to_global( CMRT_desc_t array,
                           int *l_index,int *g_index )
```

Arguments

array — A subarray (that is, the calling node's portion of a global parallel array). For C routines, the argument type is a CMRT array descriptor (**CMRT_desc_t**); for CM Fortran routines, it is the data type of *array*.

l_index — A one-dimensional integer array whose size is equal to the rank of *array*. It receives the coordinates within the subarray, *array*, equivalent to the element identified by *g_index*. If the requested element is not in the subarray, the values in *l_index* are undefined. In CM Fortran, *l_index* is an **INTENT(OUT)** argument.

g_index — A one-dimensional integer array whose size is equal to the rank of *array*. It contains the coordinates of an element within the global array of which *array* is a subarray. In CM Fortran, it is an **INTENT(IN)** argument.

local — In CM Fortran, a logical scalar that is set to **.TRUE.** if the subarray contains the element specified by *g_index*, and to **.FALSE.** otherwise. In C, a pointer to an integer that will be set to 1 if the subarray contains the specified element, and to 0 otherwise.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

Description

CMGL_global_to_local converts a set of global coordinates within a global array to an equivalent set of local coordinates within the subarray on the calling node. A separate argument states whether the requested global coordinates are in fact within this node's subarray. All coordinates assume a 1-based array.

CMGL_global_to_local must be called only from local routines, and only on subarrays. It returns no value. It is a subroutine in CM Fortran, and a function returning void in C.

Example

Consider again the 2-dimensional global array, **A**, 16 x 16, divided among 4 nodes. Suppose that all nodes in a CM Fortran local routine call **CMGL_global_to_local**, specifying A for *array* and providing a *g_index* containing the global coordinates 16,16. After the call, Node 3 sees *local* set to **.TRUE.**, and sees the coordinates 8,8 in *l_index*. Nodes 0, 1, and 2 see *local* set to **.FALSE.**; they should not check the contents of l_index, which will be undefined, and hence meaningless.

Error Conditions

Calling **CMGL_local_to_global** or **CMGL_global_to_local** on an array which is not a subarray (that is, it is not a global parallel array that was passed to the local routine as an argument), causes an error that the compiler cannot catch at compile time. As a result, such errors will probably cause your program to crash.

Calling either of these routines from a global program will cause a failure at link time.

Copyright © 1993 Thinking Machines Corporation

7.6 Program Construction

A global/local program contains:

- A global main program, written in CM Fortran. Execution of the global/local program begins with the main program, as usual.
- Zero or more global procedures called by the main program.
- One or more local routines called by the main program or by one or more of the global procedures within its scope.
- Zero or more local functions or subroutines called within the scope of the local routine(s).

It requires the following files:

- One or more files for the global CM Fortran program (that is, the main program and any global procedures called within its scope).
- One or more files for the local routines, and for any subroutines or functions called within their scope. These must be separate from files containing the global portion of the program. CM Fortran local routine files have the `.fcm` suffix; C local routine files have the `.c` suffix.
- A prototype file, which defines the interface between a global CM Fortran program and the local routines it calls.

7.6.1 How to Write the CM Fortran Global Program

There must be a main program unit written in CM Fortran. Execution begins with the main program, as usual. This program unit, and any global procedures it calls, declare and allocate global parallel arrays (CM arrays) and global serial arrays (front-end arrays) that will be used throughout the program; perform parallel and serial I/O for the program; and perform whatever computations are best done in a data parallel style, without needing node-level intervention by the application.

Any CM arrays that are to be passed as arguments to local routines must be 1-based. Lower bounds other than 1 are not handled correctly in this implementation of the global/local interface. This is the only current restriction on the global portions of the program.

Copyright © 1993 Thinking Machines Corporation

7.6.2 How to Call a Local Routine

A CM Fortran program invokes a local routine in the same way that it invokes a global CM Fortran routine, regardless of the language in which the local routine is written. The following example calls the local routine **LR1** with three arguments: a 2-dimensional global parallel array **X**; an integer **I**; and a scalar, 25.0:

```
CALL LR1(X, I, 25.0)
```

The next example adds a global serial array **Y** of 4 integers (16 bytes) to the argument list:

```
HANDLE=CMGL_BROADCAST_SERIAL_ARRAY(Y,16)
CALL LR2(X, I, 25.0, HANDLE)
```

7.6.3 How to Write a Local Routine in CM Fortran

Declaration

The declaration of a local routine written in CM Fortran looks like the definition of a global CM Fortran subroutine.

Arguments

Parallel array arguments must be declared to be assumed-shape arrays of the rank and type of the actual argument, as shown in the example below. Lower bounds other than 1 are not allowed for global arrays to be passed to subarrays; subarrays will automatically be given lower bounds of 1.

Scalar arguments must be declared as being of the same type as the actual arguments are.

Serial array arguments (for arrays previously transferred to the nodes via **CMGL_BROADCAST_SERIAL_ARRAY**) are declared as arrays of the same rank and extent as the global serial arrays to which they correspond.

Copyright © 1993 Thinking Machines Corporation

Here is an example of a local routine with one two-dimensional parallel array argument, one integer argument, one real argument, and one serial array argument:

```
SUBROUTINE LOCAL_ROUTINE(X,I,R,Y)
INTEGER X(:,:)
INTEGER I
REAL R
INTEGER Y(2,2)
X=X+Y(1,1)
...
...
RETURN
END
```

Features

All of CM Fortran, with certain exceptions, is available to local routines written in CM Fortran:

- I/O cannot be done from local routines.
- Care should be taken when using global arrays that have non-canonical layouts; subarrays of such arrays may not contain the elements you would expect them to have.

Subarrays are indexed beginning with 1 within the local routine, regardless of the subarray's position within the global array. The routines **CMGL_global_to_local** and **CMGL_local_to_global** (described in Section 7.5) can be used to determine the position of a subarray element within the global array, and vice versa.

All CM Fortran intrinsics (**DSIZE**, **DLBOUND**, etc.) are available to the local routine; they operate strictly within the local view of the subarray as a complete and independent array.

When operating on subarrays with array notation, the programmer need not be concerned with garbage data. However, when accessing the elements of a subarray serially, or when using message-passing functions, care must be taken to operate only within the bounds given by **DLBOUND(1)** and **DUBOUND**, in order not to compute on garbage. Different subarrays of a given global parallel array may have different amounts of garbage data (all, none, or something in between).

Copyright © 1993 Thinking Machines Corporation

Consider, for example, a 30-element vector, laid out on 4 nodes. Each node will contain an 8-element vector. On the first 3 nodes, all 8 elements will hold meaningful values; on the fourth node, only 6 elements will be useful; the last 2 elements will be "garbage elements." A call to **DUBOUND** will warn you that Node 4 should not compute on elements 7 and 8.

Local routines may have common data. Data in a local common block are visible only to the local routine on the node. If data from one local common block is to be seen by another node, it must be sent to that node as a CMMD message. There is no provision for sharing global common blocks with local code, or vice versa.

IMPORTANT

If an application performs CMMD communications within a local routine, the application must ensure that the network is empty before the local routine returns to the global program.

7.6.4 How to Write a Local Routine in C

Declarations

The declaration of a local routine written in C looks like the declaration of a regular C function that returns void. The name given to the local C routine must be that used to reference it in the global CM Fortran program.

Arguments

Since C has no concept of parallel arrays, a subarray argument is passed to a local C routine by passing an array descriptor, with the data type **CMRT_desc_t**, as shown below. The descriptor contains information for the subarray on the particular node.

Scalar arguments must be declared to be the same type as the corresponding actual arguments in the call.

Copyright © 1993 Thinking Machines Corporation

Here is an example of a local routine with one parallel array argument, one integer argument, and one float argument:

```
#include <cm/rts.h>
local_routine(x,i,r)
CMRT_desc_t x;
int i;
float r;
{
...
...
}
```

Features

All the functionality of C, with the exception of I/O, is available to the programmer of a local routine written in C. However, because C does not understand that array descriptors represent parallel arrays, C routines must deal with array descriptors explicitly. (They will probably use DPEAC or CDPEAC code in order to do this.)

In particular, the programmer must use the information in the descriptor to avoid computing on any garbage data that may exist in the subarray. (The forthcoming *CM Fortran Performance Guide* and *VU Programmer's Handbook* will carry information on array descriptors. The *VU Programmer's Handbook* will also explain how to call DPEAC and CDPEAC code from C, in order to access VU memory.)

The procedures **`CMGL_global_to_local`** and **`CMGL_local_to_global`** (described in Section 7.5) are also available to C routines to assist in understanding which element of a global array is represented by which element in a given node's subarray, and vice versa. C programmers should remember that these functions require and provide 1-based coordinates for all their arguments, even though the CMRT descriptors are 0-based.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

IMPORTANT

If an application performs CMMD communications within a local routine, the application must ensure that the network is empty before the local routine returns to the global program.

7.6.5 The Prototype File

A special *prototype file* must be provided for CM Fortran programs that call local routines. This file contains a prototype for each local routine that is called from global code, and thus defines the interface between the global and local portions of the program.

The name of the prototype file must have the suffix **`.proto`**.

(NOTE: This file may become unnecessary in future versions of the compiler.)

Prototypes

A prototype describes a local function's arguments and calling environment. For example, a prototype for a local routine LR, written in CM Fortran, having one array argument, one integer argument, and one real argument, would be

```
LR(array X, integer I, real R):host_cmf:node_cmf;
```

For the same routine written in C, the prototype would be

```
LR(array X, integer I, real R):host_cmf:node_c;
```

Note that all prototypes end with a semicolon (;).

The actual names given to the arguments in the prototype do not matter. They do not need to match the dummy argument names in the definition of the local routine. They are included in the prototype specification to allow prototypes to conform as closely as possible to ANSI prototypes.

Copyright © 1993 Thinking Machines Corporation

Note that prototypes are available only for local routines called from the global CM Fortran program. You should not write prototypes for local subroutines or functions that are called by other local routines.

A program can have multiple **.proto** files.

Data Types

The following data type names are available:

```
logical (or logical*4)
integer (or integer*4)
integer*8
real (or real*4)
double precision (or real*8)
complex (or complex*8)
double complex (or complex*16)
serial
array (or CMRT_desc_t)
```

The type `array` is used for global parallel arrays of all data types: `integer`, `real`, etc. The type `serial` is used for global serial arrays.

Use

The prototype file is input to a tool called **cmmd_wrapper_gen**. This tool generates two output files, one containing wrapper functions to be linked into the scalar executable and one containing wrapper functions to be linked into the **.pe** executable. These wrapper functions perform all the necessary manipulation required to get local routines to work, such as building the local descriptors on the nodes. Programmers do not need to understand the output created by the tool; they only need to know how to write the prototype file.

7.6.6 Compiling and Linking

CM Fortran Version 2.1 provides a **-local** switch to handle the compiling, linking, and wrapper-generating details involved in a global/local program. The **-cm5** and **-vu** switches must also be used. (Remember, global/local programs can run only on CM-5 systems equipped with VUs.)

Copyright © 1993 Thinking Machines Corporation

Unlike the other compiler switches, the **-local** switch must appear before each **.fcm** or **.c** file that has local code, thereby letting the driver know that they are special.

So, in a program that has just one file with global CM Fortran code and one file with local CM Fortran code, the compile line looks like this:

```
% cmf -cm5 -vu foo_global.fcm
           -local foo_local.fcm foo.proto
```

If there are several local files, then:

```
% cmf -cm5 -vu foo_global.fcm
      -local foo_local1.fcm -local foo_local2.fcm
      -local foo_local3.fcm foo.proto
```

Header Files

Global programs that call **CMGL_BROADCAST_SERIAL_ARRAY** must include the header file **cmgl.h** in the global code.

Local routines that call CMMD functions must include a CMMD header file:

- For CM Fortran routines,

  ```
  INCLUDE '/usr/include/cm/cmmd_fort.h'
  ```

  or

  ```
  #include <cm/cmmd_fort.h>
  ```

  if the **.FCM** source file extension is used.

- For C routines,

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


Copyright © 1993 Thinking Machines Corporation

7.6.7 Debugging

Prism 2.0, running on CMOST 7.3 or later, can be used to debug a global/local program. You will need two copies of Prism, one to debug the global program units and one to debug the local program units.

Follow this procedure to get two graphical Prism's started on program **a.out**:

[In one window]

```
% prism a.out
```

(prism starts up)

Set a breakpoint and execute **run** to get your program going.

[In another window]

```
% cmps
```

(find out the pid of your process)

```
% prism -node a.out <pid>
```

(2nd copy of prism starts up)

All nodes come up in "interrupted" state. You can set breakpoints in your local code if you wish, then you should **continue** to get all nodes running again.

At this point, you have two copies of Prism running,. You can use the global Prism to perform any debugging operations on the global part of the program, and the nodal Prism to perform any debugging operations on the local part of your program. Refer to the *Prism User's Guide* for information on using Prism.

For example, consider this global program

```
PROGRAM MAIN
REAL, ARRAY(1024, 1024) :: A
...
CALL LOCAL_ROUTINE(A)
...
END
```

that calls this local routine

Copyright © 1993 Thinking Machines Corporation

```
SUBROUTINE LOCAL_ROUTINE(A)
REAL, ARRAY(:,:) :: A
...
END
```

In this example, if you **print a** from the global Prism, you will get a 1024 x 1024 array. If you use nodal Prism to break in **LOCAL_ROUTINE**, and you **print a** there, you will see a smaller array in each PN, representing that PN's subarray. For instance, if your partition has 32 PN's, then you will see a 32 x 1024 array in each PN. Nodal Prism will glue these together along a new axis, so if you **print a** when your current pnset is **all**, you see a 32 x 32 x 1024 array.

Important caveat: In order to do debugging operations on CM arrays from the global Prism (the global **print a** in the above example), all PN's must be "running" from the point of view of nodal Prism. That is, they cannot be interrupted or stopped at nodal Prism breakpoints. If any PN's are stopped, the global **print** will hang until they are continued.

If you do not have Prism 2.0 or are not on a CMOST 7.3 system, you can use the **pndbx** debugger in place of nodal Prism. Follow the same startup instructions as above, but type **pndbx** instead of **prism -node**. Remember, **pndbx** is a command-line debugger only. Also, **pndbx** will not do the data aggregation that is described above. In the above example, **pndbx** will show you a 32 x 1024 array for one PN at a time. The caveat about nodes having to be running applies if you are using **pndbx** for your node-level debugger.

7.6.8 Profiling

Prism Versions 1.2 and 2.0 do not support node-level profiling. Do not use the CM Fortran switch **-cmprofile** together with **-local**.

Copyright © 1993 Thinking Machines Corporation

8 I/O from CM Fortran

This chapter describes support for parallel I/O to the Scalable Disk Array and DataVault, as well as several enhancements to the language I/O statements and to the utility library I/O procedures.

8.1 Parallel I/O with READ and WRITE

The CM Fortran **READ** and **WRITE** statements now support I/O to the Scalable Disk Array (SDA) and the DataVault from both the control processor and the parallel processors. Prior to this release, the I/O statements only supported serial I/O between the control processor and the UNIX file system. Any parallel I/O to a CM storage device had to be accomplished via the file operations procedures in the CM Fortran utility library.

CM Fortran I/O statements now support the following file systems:

- The UNIX file system, which resides on a serial computer.
- The SFS, or Scalable File System, which is a UNIX file system that resides on the Scalable Disk Array (CM-5 only).
- The CMFS, or CM File System, which resides on the DataVault (CM-5 or CM-2/200).

See the CM I/O documentation for more information on the two CM file systems.

All CM Fortran I/O statements can be used on files of these file systems.

When the target file system is on the SDA or DataVault, the **READ** and **WRITE** statements can perform either serial or parallel I/O.

8.1.1 Specifying a Target File

For systems with more than one kind of storage device installed, the device affected by a CM Fortran I/O operation is governed by the file's pathname. This does not depend on the setting of the environment variable **CMFS_PATHTYPE**, as described in the CM I/O documentation.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

NOTE

When you perform I/O by using the utility library routines rather than the CM Fortran I/O statements, CMFS *does* use the value of **CMFS_PATHTYPE** to determine which device is affected by the I/O operation.

Note that front-end data can be written to the SDA or DataVault, including character data, although the transfer done serially (in a single stream). (Writing character data to the DataVault will not be supported on CM-2/200 systems.)

Specifying a Scalable Disk Array File

Files on a Scalable Disk Array (SDA) are managed by the Scalable File System, which shares name space with the UNIX file system. Your system administrator sets up the system such that some directories are mounted to the SDA and other directories are on the UNIX file system. To specify a target file on an SDA, you have two choices:

- Specify the full pathname of a file that is currently mounted on the SDA. For example, assume the system administrator has mounted the SFS to the directory **/sda**.

```
OPEN (9, FILE='/sda/dir1/data2')
```

- Or, **cd** to the directory that is mounted to the SDA and specify just the file name.

```
OPEN (9, FILE='data2')
```

When you specify a file name alone, the location of the file is assumed to be the directory that is current when you execute the code.

Copyright © 1993 Thinking Machines Corporation

Specifying a DataVault File

To specify a target file on a DataVault, include the DataVault name in the file's path. The pathname is of the form *dvname*:*pathname*. For example, this code opens the file **/dir1/data2** on a DataVault called **dv1**:

```
OPEN (9, FILE='dv1:/dir1/data2')
```

8.1.2 Examples

File Copy from DataVault to SDA

This example shows how to copy a file from a DataVault to an SDA.

```
      PROGRAM COPY_DATA
      IMPLICIT NONE
      INTEGER A(100)
CMF$  LAYOUT A(:NEWS)

      OPEN (9, FILE='dv6:/dir1/file1',
     $    ACCESS='SEQUENTIAL',
     $    FORM='UNFORMATTED',
     $    STATUS='OLD')

      OPEN (10, FILE='/sda/dir1/file1',
     $    ACCESS='SEQUENTIAL',
     $    FORM='UNFORMATTED',
     $    STATUS='NEW')

      READ (9, ERR=99) A
      WRITE (10, ERR=99) A
      CLOSE (9)
      CLOSE (10)
      STOP

99    PRINT *, 'error'
      END
```


Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

I/O To and From the SDA

This program prompts the user for a number of bytes. That many bytes are written to the SDA and then read back in. Results are checked for accuracy.

```
      PROGRAM IO_TEST
      DO
        WRITE(6,'(" NBYTES? (ENTER 0 TO EXIT) ",$)')
        READ(5,*)NBYTES
        IF (NBYTES.LE.0) EXIT
        CALL SUB(NBYTES)
      END DO
      STOP
      END

      SUBROUTINE SUB(NBYTES)
      CHARACTER*(*),PARAMETER :: FILENAME='/TEST1/TEST'
      DOUBLE PRECISION TIMEOFDAY, START_TIME
      COMPLEX*16, ARRAY(NBYTES/16) :: X, Y
CMF$  LAYOUT X(), Y()

C Make up write array
      X=SQRT(DCMPLX(DBLE([1:DUBOUND(X,1)]),
     &  0.5d0*DUBOUND(X,1)))

C Initialize read array
      Y=DCMPLX(0.d0,0.d0)

C Time write
      OPEN(20,FILE=FILENAME,STATUS='UNKNOWN',
     &  FORM='PHYSICAL')
      START_TIME=TIMEOFDAY()
      WRITE(20)X
      WRITE_TIME=TIMEOFDAY()-START_TIME
      CLOSE(20)

C Time read
      OPEN(20,FILE=FILENAME,STATUS='OLD',
     &  FORM='PHYSICAL')
      START_TIME=TIMEOFDAY()
      READ(20)Y
      READ_TIME=TIMEOFDAY()-START_TIME
      CLOSE(20)
```


Copyright © 1993 Thinking Machines Corporation

```
C Check results
      III=COUNT(X.NE.Y)
      IF (III.NE.0) WRITE(6,*)III,' ERRORS!!!!!!!!'

C Report times
      AMOUNT=NBYTES/2.**20
      WRITE_RATE=NBYTES/WRITE_TIME/2.**20
      READ_RATE=NBYTES/READ_TIME/2.**20
      WRITE(6,'("                          WRITE
     &        READ")')
      WRITE(6,'("    MB         SEC      MB/S
     &  SEC     MB/S")')

      WRITE(6,'(5F10.3)')
     &  AMOUNT,WRITE_TIME,WRITE_RATE,READ_TIME,READ_RATE
      WRITE(6,*)

      RETURN
      END
```

The following code is in a file with the suffix **.c**:

```
long float timeofday_()
{
    long int t1[2],t2[2];
    int gtod;

    gtod=gettimeofday(t1,t2);
    return ((double)t1[0]+0.000001 * (double)t1[1]);
}
```

8.1.3 When is I/O Parallel?

Parallel I/O refers to transferring data in multiple streams between the CM processing elements and an external device. Thus, it is faster than serial I/O. The **READ** and **WRITE** statements perform parallel I/O if all of the following conditions apply.

- The data involved is a CM array.

Copyright © 1993 Thinking Machines Corporation

- The file is on the SDA or DataVault.
- The file data is unformatted.

Thus, *any* of the following conditions results in serial I/O:

- Front-end data is involved.
- The file is in the UNIX file system on a serial computer.
- The file data is formatted.

Any data subject to an implied **DO** loop within a **READ** or **WRITE** statement is transferred serially. Use whole arrays or array sections to get parallel I/O.

For direct-access I/O involving a DataVault, performance improves when you read or write records that are multiples of 512 bytes or arrays of multiples of 512 bytes. This applies only to I/O of CM arrays.

Copyright © 1993 Thinking Machines Corporation

8.2 Enhancement of Other I/O Statements

8.2.1 PHYSICAL Keyword for OPEN

A third keyword, **PHYSICAL**, can now be supplied to the **FORM=** specifier in an **OPEN** statement, instead of the values **FORMATTED** and **UNFORMATTED**. Usually, CM Fortran arrays are written in array element order. This requires a transpose which takes time and temporary space. Using **FORM=PHYSICAL** eliminates the transpose, speeding up the parallel I/O operation. This is useful, for example, if you are writing arrays temporarily to a file in order to gain working storage.

Use **FORM=PHYSICAL** only if the following conditions apply:

- The order of array elements in the file is not important.
- Data is read into arrays of the same shape and layout as those from which it is written.

The **PHYSICAL** keyword is supported for sequential access files only. The order of array elements within a record is undefined.

8.2.2 New Specifiers in OPEN Statement

This version supports four new specifiers for the **OPEN** statement: **POSITION=**, **ACTION=**, **DELIM=**, and **PAD=**.

POSITION= Specifier

The **POSITION=** specifier indicates how the file being opened is to be positioned. The file must be connected for sequential access. A new file is positioned at its initial point. An existing file is positioned according to the keyword specified:

REWIND Positions the file at its initial point.

APPEND Positions the file so that the endfile record is the next record, if it has one. If the file does not have an endfile record, the file is positioned at its terminal point.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

ASIS Leaves unchanged the position of an already connected file. If the file is not yet connected, **ASIS** leaves the position unspecified.

If the specifier is omitted, the default is **ASIS**.

ACTION= Specifier

The **ACTION=** specifier indicates which actions on the file are permissible.

READ Specifies that no **WRITE**, **PRINT**, or **ENDFILE** statements may refer to this connection.

WRITE Specifies that no **READ** statements may refer to this connection.

READWRITE Permits any I/O statements to refer to this connection.

If **READWRITE** is included in the set of allowed actions for a file, both **READ** and **WRITE** must also be included in that set. For an existing file, the specified action must be included in the set of allowed actions for the file. If the file is new, the processor creates the file with a set of allowed actions that includes the specified action.

If the specifier is omitted, the default is **READWRITE**.

DELIM= Specifier

The **DELIM=** specifier indicates which character will be used to delimit character constants written with list-directed or namelist formatting. All instances of that character within the character constant are doubled. The file must be being connected for formatted I/O.

APOSTROPHE The apostrophe will be used to delimit character constants.

QUOTE The quote will be used to delimit character constants.

NONE The character constant will not be delimited.

If the specifier is omitted, the default is **NONE**. The **DELIM=** specifier is ignored during input of a formatted record.

Copyright © 1993 Thinking Machines Corporation

PAD= Specifier

The **PAD=** specifier indicates whether an input record should be padded.

YES	A formatted input record is logically padded with blanks when an input list is specified and the format specification requires more data from a record than the record contains.
NO	The input list and the format specification must not require more characters from a record than the record contains.

If the specifier is omitted, the default is **YES**.

8.2.3 New Specifiers in INQUIRE Statement

Version 2.1 provides four new specifiers for use with the **INQUIRE** statement: **POSITION=**, **ACTION=**, **DELIM=**, and **PAD=**.

POSITION= Specifier

The variable in the **POSITION=** specifier is assigned one of the following values:

REWIND	The file is connected by an **OPEN** statement for positioning at its initial point.
APPEND	The file is connected for positioning before its endfile record or at its terminal point.
ASIS	The file is connected without changing its position.

If there is no connection or if the file is connected for direct access, the value is undefined.

ACTION= Specifier

The variable in the **ACTION=** specifier is assigned one of the following values:

READ	The file is connected for input only.
WRITE	The file is connected for output only.
READWRITE	The file is connected for both input and output.

Copyright © 1993 Thinking Machines Corporation

If there is no connection, the value is undefined.

DELIM= Specifier

The variable in the **DELIM=** specifier is assigned one of the following values:

APOSTROPHE	The apostrophe is to be used to delimit character data written by list-directed or namelist formatting.
QUOTE	The quotation mark is to be used to delimit character data.
NONE	Character data will not be delimited.

If there is no connection, or the connection is not for formatted I/O, the value is undefined.

PAD= Specifier

The variable in the **PAD=** specifier is assigned one of the following values:

NO	The connection of the file to the unit included the **PAD=** specifier, and its value was **NO**.
YES	The value of the **PAD=** specifier was **YES**, or the specifier was omitted.

8.2.4 New Format for NAMELIST

CM Fortran now supports additional **NAMELIST** syntax, as an alternative to the Fortran 90 syntax. The Fortran 90 format for a **NAMELIST** I/O file is:

```
&NAMELIST_NAME
        data data data data data
 /
```

The **NAMELIST_NAME** begins with **&**, and the file ends with **/**.

In this release, CM Fortran also supports:

```
$NAMELIST_NAME
         data data data data data
$END
```


Copyright © 1993 Thinking Machines Corporation

The **NAMELIST_NAME** begins with **$**, and the file ends with **$END**.

8.2.5 BACKSPACE and REWIND on Redirected Streams

CM Fortran now supports applying the **BACKSPACE** and **REWIND** statements to standard I/O streams (**stdin** or **stdout**, default units 5 and 6) when redirected to files.

8.2.6 Printing First Character in Output Streams

CM Fortran no longer ignores the first character in formatted output to **stderr** and **stdout** (default units 0 and 6). Previous releases treated the output device as a line printer and the first character as a carriage control character, which meant that it was ignored. The current release prints or writes the first character.

Consider this example:

```
      PROGRAM OUTPUT
      WRITE(6,100) 'No leading blank'
      WRITE(6,100) ' One leading blank'
100   FORMAT(A)
      END
```

Previous releases gave:

```
% output
o leading blank
One leading blank
```

This release gives:

```
% output
No leading blank
 One leading blank
```


Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

8.3 Utility Library Procedures

The CM Fortran utility library I/O procedures are now, for most purposes, redundant with language statements. We recommend that you use the **READ** and **WRITE** statements instead of utility library I/O, with a couple of exceptions.

- Continue using the utility library procedures to perform parallel I/O via sockets or devices, such as CM-HIPPI. **READ** and **WRITE** support parallel I/O only to the SDA or DataVault.
- Use the utility read procedures to read files previously written with utility write procedures.

See the *CM Fortran Utility Library Reference Manual* for information about the individual utility library procedures.

8.3.1 New Utility Library Procedures

CMF_FILE_LLSEEK (UNIT, OFFSET, IOSTAT)

This procedure is like **CMF_FILE_LSEEK** except that it take a 64-bit **OFFSET** argument. This is useful for moving the file pointer in serial order ("SO") files over a range greater than 2 Gbytes.

Seeking on parallel order files with **CMF_FILE_LSEEK** is not limited to 2 Gigabytes, since it seeks by whole-array units rather than by bytes. If you do use **CMF_FILE_LLSEEK** on a parallel order file, be sure to coerce the 32-bit result of **CMF_SIZEOF_ARRAY_ELEMENT** (which supplies the **OFFSET** argument) to **INTEGER*8**.

CMF_FILE_LTRUNCATE (UNIT, LENGTH, IOSTAT)

This procedure is like **CMF_FILE_TRUNCATE** except that it take a 64-bit **LENGTH** argument. This is useful for truncating or extending serial order ("SO") files to a size greater than 2 Gbytes.

Truncating/extending parallel order files with **CMF_FILE_TRUNCATE** is not limited to 2 Gigabytes, since it operates by whole-array units rather than by bytes. If you do use **CMF_FILE_LTRUNCATE** on a parallel order file, be sure to

Copyright © 1993 Thinking Machines Corporation

coerce the 32-bit result of **CMF_SIZEOF_ARRAY_ELEMENT** (which supplies the **LENGTH** argument) to **INTEGER*8**.

CMF_FILE_OPEN_READONLY (UNIT, PATH, IOSTAT)

This procedure is comparable to **CMF_FILE_OPEN**, except that it prevents the file from being changed by a write procedure. Using **CMF_FILE_OPEN_READONLY** is like using the CM Fortran **OPEN** statement with the **READONLY** specifier.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

CMF_FILE_LLSEEK

Offset the file pointer a specified distance within a CM file.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_LLSEEK( UNIT, OFFSET, IOSTAT )
```

Arguments

UNIT — An **INTEGER*4** variable, parameter, or literal constant containing a valid unit number in the range 0 through 2999. The unit number is established in the call to **CMF_FILE_OPEN** that creates the file in the CM file system. This **UNIT** number is not related to the standard Fortran unit numbers used in the **OPEN** statement.

OFFSET — **INTEGER*8**. An offset from the current position in the specified file. This argument is specified differently for serial order and parallel order files. See the description below for more information.

IOSTAT — An **INTEGER*4** variable into which the status of the I/O operation will be placed. A positive or zero value indicates success; a negative value indicates failure.

Result

None.

Description

This subroutine moves the file pointer within a CM file. The new position is computed by adding the specified offset to the current position of the file pointer.

Seeking in Serial Order Files

For serial order files (those written with **CMF_CM_ARRAY_TO_FILE_SO**), the offset is given in bytes. To calculate the offset, multiply the number of bytes in the array's ele-

Copyright © 1993 Thinking Machines Corporation

ment type by the number of elements to traverse. This allows you to seek to an arbitrary element in the file.

The **INTEGER*8** type of the **OFFSET** argument permits an offset larger than that accepted by the comparable procedure **CMF_FILE_LSEEK**. The latter procedure is supported on all CM systems, but the **OFFSET** argument is limited by its 32-bit type to 2 Gigabytes. **CMF_FILE_LLSEEK** permits moving the file pointer over a larger range, but note the restriction below on CM system support for the **INTEGER*8** type.

Seeking in Parallel Order Files

Before performing a seek operation on a parallel order file (one written with **CMF_CM_ARRAY_TO_FILE**), you must perform a read or write operation on the file to establish its geometry.

For parallel order files, you can only seek to the beginning of whole arrays. To compute the offset, you need not specify the size of the array, since this information is contained in the file geometry. You need specify only the size of an array's elements using the utility function **CMF_SIZEOF_ARRAY_ELEMENT**. To seek over multiple arrays, call the function **CMF_SIZEOF_ARRAY_ELEMENT** on each array and add the results. (See the example given under **CMF_FILE_LSEEK**.)

Since this function returns an **INTEGER*4** result, you must coerce the result to **INTEGER*8** before passing it as an argument to **CMF_FILE_LLSEEK**.

Example

The use of **CMF_SIZEOF_ARRAY_ELEMENT** to seek over parallel order files is shown under the description of **CMF_FILE_LSEEK**. To use **CMF_FILE_LLSEEK** for this purpose, you must coerce the result of **CMF_SIZEOF_ARRAY_ELEMENT** to **INTEGER*8**. This can be done with the intrinsic function **INT**, supplying the bit-length **64** as the **KIND** argument:

```
... INT(CMF_SIZEOF_ARRAY_ELEMENT(ARRAY),64)
```

or by assignment of the function result to an **INTEGER*8** temp.

```
INTEGER*8 TEMP
...
TEMP = CMF_SIZEOF_ARRAY_ELEMENT( ARRAY )
CALL CMF_FILE_LLSEEK( UNIT, TEMP, IOSTAT )
```


Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

Restrictions

- The **INTEGER*8** type is supported only on CM-5 systems with vector units.
- For parallel order files on CM-5 systems, the element size of a seek operation must be the same as the element size of the read or write operation that established the geometry of the file when it was first opened.
- If the file was written with **CMF_CM_ARRAY_TO_FILE_FMS**, you must use **CMF_FILE_LSEEK_FMS** to perform a seek operation on it.

Copyright © 1993 Thinking Machines Corporation

CMF_FILE_LTRUNCATE

Change the size of a CM file.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_LTRUNCATE( UNIT, LENGTH, IOSTAT )
```

Arguments

UNIT An **INTEGER*4** variable, parameter, or literal constant containing a valid unit number in the range 0 through 2999. The unit number is established in the call to **CMF_FILE_OPEN** that creates the file in the CM file system. This **UNIT** number is not related to the standard Fortran unit numbers used in the **OPEN** statement.

LENGTH An **INTEGER*8** specifying the new length of the file. This argument is specified differently for serial order and parallel order files. See the **Description** section below for more information.

IOSTAT An **INTEGER*4** variable into which the status of the I/O operation will be placed. A positive or zero value indicates success; a negative value indicates failure.

Result

None.

Description

This subroutine changes the size of the specified CM file to **LENGTH**. The file is extended or shortened by the difference between the **LENGTH** argument and the file's current length. If the file was previously larger than **LENGTH**, the extra data is lost. If the file was previously smaller than **LENGTH**, the file is extended to **LENGTH**.

The **INTEGER*8** type of the **LENGTH** argument permits a length larger than that accepted by the comparable procedure **CMF_FILE_TRUNCATE**. The latter procedure is

Copyright © 1993 Thinking Machines Corporation

supported on all CM systems, but the **LENGTH** argument is limited by its 32-bit type to 2 Gigabytes. **CMF_FILE_LTRUNCATE** permits a longer file length, but note the restriction below on CM system support for the **INTEGER*8** type.

Deriving LENGTH for Serial Order Files

For serial order files (those written with **CMF_CM_ARRAY_TO_FILE_SO**), the length is given in bytes. To calculate the length for the file, multiply the number of bytes in the array's element type by the number of array elements to be contained in the file.

Deriving LENGTH for Parallel Order Files

Before performing a truncate operation on a parallel order file (one written with **CMF_CM_ARRAY_TO_FILE** or **CMF_CM_ARRAY_TO_FILE_FMS**), you must perform a read or write operation on the file to establish its geometry.

Parallel order files can only be reduced or enlarged by whole arrays. To compute the length of the array, you need not specify the size of the array, since this information is contained in the file geometry. You need specify only the size of an array's elements using the function **CMF_SIZEOF_ARRAY_ELEMENT**. You can increase or decrease the size of the file by more than one array by calling **CMF_SIZEOF_ARRAY_ELEMENT** on several arrays in succession, adding the returned values together, and supplying the cumulative result as the **LENGTH** argument. (See the example given under the description of **CMF_FILE_LSEEK**.)

Restrictions

- The **INTEGER*8** type is supported only on CM-5 systems with vector units.
- **CMF_FILE_LTRUNCATE** requires the file to be open for writing.
- For parallel order files on CM-5 systems, the element size of a truncate operation must be the same as the element size of the read or write operation that established the geometry of the file when it was first opened.

Copyright © 1993 Thinking Machines Corporation

CMF_FILE_OPEN_READONLY

Opens a CM file for read-only access and attaches the file to the **UNIT**.

Syntax

```
INCLUDE '/usr/include/cm/CMF_defs.h'

CALL CMF_FILE_OPEN_READONLY( UNIT, PATH, IOSTAT )
```

Arguments

UNIT	An **INTEGER*4** variable, parameter, or literal constant that is a valid unit number in the range 0 through 2999.
PATH	A **CHARACTER** string containing the pathname for the file to be opened.
IOSTAT	An **INTEGER*4** variable into which the status of the I/O operation will be placed. A positive or zero value indicates success; a negative value indicates failure.

Result

None.

Description

This subroutine opens a CM file in read-only mode and attaches the file to the **UNIT**. You must supply this unit number to other CM file system procedures when you wish to operate on this file. A run-time error is reported if an attempt is made to write to this file. Note that the CM file system unit numbers are not related to standard Fortran unit numbers used with the **OPEN** statement.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

8.4 Note on I/O Unit Numbers

CM Fortran allows unit numbers in the range 0 through 2999. The unit number becomes associated with a file when it is used as the **UNIT** argument to the **OPEN** statement or the utility procedure **CMF_FILE_OPEN**.

NOTE

It is possible to use each unit number twice, once with **OPEN** and again with **CMF_FILE_OPEN**, since the two sets of unit numbers are distinct. However, this practice invites confusion in programming.

8.5 Note on Back-Compatibility of Utility I/O

The CMFS I/O routines underlying the utility library procedures **CMF_CM_ARRAY_TO/FROM_FILE** no longer pad array data to 16-byte "CM word" boundaries. However, to preserve compatibility with files written under previous CM Fortran releases, the CM Fortran procedures add the padding on writes and remove it on reads. No code changes are needed in CM Fortran programs.

REMINDER

Always read a CM file with the same mechanism that was used to write it. That is, read with **READ** if written with **WRITE**, read with **CMF_...SO** if written with the **_SO** routine, and so on.

Copyright © 1993 Thinking Machines Corporation

9 Array Pointers

CM Fortran provides *array pointers* as an alternative to allocatable arrays for managing storage at run time. Array pointers are arrays declared with the attribute **POINTER**, along with type and rank, but without extents. The extents are supplied to the **ALLOCATE** statement, which allocates the storage at run time and associates it with the pointer. The pointer can then be used like any other CM array name.

For arrays that are local to a program unit, pointers provide the same capabilities as allocatable arrays. The difference between them is that array pointers can appear in common, whereas allocatable arrays cannot. Pointers thus give you the ability to manage global memory dynamically.

NOTE

This release provides a subset of the Fortran 90 pointer functionality. The subset does not include the **TARGET** attribute or the pointer assignment statement. You cannot declare pointers and targets separately and then associate them at run time.

9.1 Declaring Array Pointers

An array can acquire the **POINTER** attribute either in an attributed type declaration or by means of the **POINTER** statement:

```
REAL, POINTER, ARRAY(:) :: A,B
```

or,

```
...
POINTER A,B
```

The array's shape must be specified as a *deferred-shape*, meaning a comma-separated list of colons, one for each dimension. The deferred-shape specification can be attached to the array name in a type declaration or **POINTER** statement,

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

or it can be supplied with a separate **DIMENSION** or **ARRAY** specification. (It cannot be attached to the **POINTER** attribute in a type declaration.) For example, here are several equivalent ways to declare three array pointers, **P**, **Q**, and **R**.

```
INTEGER, POINTER :: P(:), Q(:), R(:,:)
```

or,

```
INTEGER P,Q,R
POINTER P(:), Q(:), R(:,:)
```

or,

```
INTEGER, POINTER, ARRAY(:) :: P,Q
INTEGER, POINTER, ARRAY(:,:) :: R
```

or,

```
INTEGER P,Q,R
DIMENSION P(:), Q(:), R(:,:)
POINTER P,Q,R
```

9.2 Allocating Pointer "Target" Arrays

Like allocatable arrays, the targets of array pointers are allocated and deallocated by execution of the statements **ALLOCATE** and **DEALLOCATE**. The **ALLOCATE** statement specifies the shape of the array, allocates the storage, and associates it with the pointer.

```
REAL, POINTER, ARRAY(:) :: A
...
ALLOCATE (A(N))
...
DEALLOCATE (A)
```

The **DEALLOCATE** statement disassociates the pointer and releases the target storage. It is an error to deallocate a pointer array that is not currently allocated.

The pointer can become associated with other storage by another **ALLOCATE** statement. It is not an error to allocate a pointer array that is already allocated; in this case, the pointer becomes associated with the new storage, and the previous target storage becomes inaccessible.

Copyright © 1993 Thinking Machines Corporation

Use the intrinsic inquiry function **ASSOCIATED** to check the allocation status of an array pointer.

```
ASSOCIATED( POINTER )
```

For example:

```
IF (ASSOCIATED(P)) THEN
   PRINT *, 'P is already allocated.'
ELSE
   ALLOCATE (P(0:M))
ENDIF
```

Notice that the inquiry function that reports the allocation status of pointers is different from the one provided for allocatable arrays. The allocation status of allocatable arrays is reported by the intrinsic function **ALLOCATED**.

Also, those familiar with Fortran 90 will notice that **ASSOCIATED** in CM Fortran takes only one argument. The second Fortran 90 argument, **TARGET**, is not provided, since CM Fortran provides no mechanism to associate a target with more than one pointer at a time.

9.3 Using Array Pointers

Once allocated, an array pointer can be used like any other CM array. It can be a primary in an expression, the left-hand side of an array assignment, an argument to a procedure, or an item in an I/O list. An array pointer cannot, however, be a function result.

Pointers are particularly useful for declaring dynamic arrays in common to avoid having to pass them as arguments. For example, a program that operates on numerous dynamic arrays can declare their pointers in common, perhaps in an include file, so that all program units have access to them.

```
REAL, POINTER, ARRAY(:) :: P1,P2,P3, ... P100
COMMON /C/ P1,P2,P3, ... P100
```


Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

If you do use array pointers as arguments, notice several constraints on declaring dummy array arguments as pointers:

- If a dummy array argument is a pointer, the actual argument passed to it must also be a pointer (not necessarily allocated). In addition, the calling procedure must contain an interface block that declares the dummy argument as a pointer.

```
INTEGER, POINTER, ARRAY(:,:,:) :: ACTUAL_D3
...
ALLOCATE (ACTUAL_D3 (N,M,N+M))
...
INTERFACE
   SUBROUTINE SUB(D3)
   INTEGER, POINTER, ARRAY(:,:,:) :: D3
END INTERFACE

CALL SUB( ACTUAL_D3 )
...
```

- An actual argument that is a pointer may be passed to a non-pointer dummy argument. In this case, no interface block is required.

- A dummy array argument that is a pointer may not have the **INTENT** attribute.

Copyright © 1993 Thinking Machines Corporation

9.4 Summary of Syntax

This section provides the formal syntax of the features related to array pointers. The **POINTER** attribute, **POINTER** statement, and **ASSOCIATED** function are new with Version 2.1 The **ALLOCATE** and **DEALLOCATE** statements appear in the *CM Fortran Reference Manual*, Version 2.0 Beta. They are extended in the current release to operate on pointers as well as on allocatable arrays.

9.4.1 POINTER Attribute

An array can be given the **POINTER** attribute in a type declaration statement, optionally using an array-shape specification with the array name. The array-shape specification must be a deferred-shape specification. If it is not supplied within the attributed type declaration, it must be supplied with a **DIMENSION** or **ARRAY** attribute earlier in the program unit.

An (attributed) *type-decl-stmt* is:
type-spec [**,** *attr-spec*]... **::** *entity-decl(s)*

An *attr-spec* may be:
POINTER

An *entity-decl* may be:
array-name [(*deferred-shape-spec-list*)]

A *deferred-shape-spec-list* is:
: [**,** **:**]**...**

The rank of the array is equal to the number of colons in the *deferred-shape-spec-list*.

- The **INTENT** attribute must not be specified for *array-name*.
- If the **DIMENSION** or **ARRAY** attribute for an array name has been specified previously in the scoping unit, the array-shape specification must have been a *deferred-shape-spec-list*.

Copyright © 1993 Thinking Machines Corporation

9.4.2 POINTER Statement

A pointer statement specifies a list of objects (array names) that have the **POINTER** attribute, optionally adding an array-shape specification. If the array-shape specification is supplied, it must be a deferred-shape specification.

A *pointer-stmt* is:

POINTER [**::**] *object-name* [(*deferred-shape-spec-list*)]
[, *object-name* [(*deferred-shape-spec-list*)])...

A *deferred-shape-spec-list* is:

: [**,** **:**]...

The rank of the array is equal to the number of colons in the *deferred-shape-spec-list.*

- The **INTENT** attribute must not be specified for *object-name*.
- If the **DIMENSION** or **ARRAY** attribute for an *object-name* has been specified previously in the scoping unit, the array-shape specification must have been a *deferred-shape-spec-list*.

9.4.3 ALLOCATE Statement

The **ALLOCATE** statement dynamically creates allocatable arrays and pointer targets.

An *allocate-stmt* is:

ALLOCATE (*allocation(s)*)

An *allocation* is:

allocate-object (*explicit-shape-spec(s)*)

An *allocate-object* is:

variable-name

An *explicit-shape-spec* is:

[lower-bound **:** *] upper-bound*

- Each *allocate-object* must be an **ALLOCATABLE** array that is not currently allocated or an array **POINTER**.
- A bound in an *allocation explicit-shape-spec* must not be an expression containing an array inquiry function whose argument is any other object in the same **ALLOCATE** statement.

Copyright © 1993 Thinking Machines Corporation

- The number of *explicit-shape-spec(s)* in an *allocation* must be the same as the rank of the deferred-shape array.

CM Fortran does not implement Fortran 90's optional **STAT=** specifier in the **ALLOCATE** statement for capturing an error condition. If an error occurs during execution of the **ALLOCATE** statement, program execution is terminated.

9.4.4 DEALLOCATE Statement

The **DEALLOCATE** statement causes an allocatable array to be deallocated or a pointer target to be deallocated and the pointer to be disassociated.

A *deallocate-stmt* is:
DEALLOCATE (*allocate-object(s)*)

An *allocate-object* is:
variable-name

- Each *allocate-object* must be an **ALLOCATABLE** array that is currently allocated or an array with the attribute **POINTER** that is currently allocated.

Deallocating an array that is not currently allocated causes an error condition in the **DEALLOCATE** statement, which terminates execution of the program. CM Fortran does not implement Fortran 90's optional **STAT=** specifier in the **DEALLOCATE** statement.

9.4.5 ASSOCIATED Function

ASSOCIATED(POINTER)

Returns the association status of the argument **POINTER**. (The Fortran 90 **TARGET** argument is not supported.)

The **POINTER** argument is an array pointer of any type.

The result is a logical scalar, **.TRUE.** if **POINTER** is currently associated with a target, and **.FALSE.** otherwise.

Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

10 64-Bit Integers and the Kind Mechanism

CM Fortran provides the 64-bit integer data type, for use only under the vector-unit execution model on CM-5. This type may be used both for scalar data on the partition manager and for parallel data on the vector units. It will not be supported on CM systems without vector units.

10.1 Type Specifications

CM Fortran now supports seven types for scalar and parallel data on CM-5 systems with vector units (along with character type for scalars only). The seven types, along with their lengths in bits and their predefined symbolic names, are shown below. For complex types, the length is that of the base type (real or double).

Data Type	Length	Symbolic Name
LOGICAL	32	(none)
INTEGER*4	32	_SINGLE_INT_
INTEGER*8	64	_DOUBLE_INT_
REAL*4	32	_SINGLE_
REAL*8	64	_DOUBLE_
COMPLEX*8	32	_SINGLE_COMPLEX_
COMPLEX*16	64	_DOUBLE_COMPLEX_

DOUBLE PRECISION (see REAL*8)
DOUBLE COMPLEX (see COMPLEX*16)

Integer and floating-point values default to the 32-bit length. The **cmf** compiler provides an option, **-double_precision**, that causes real values only to default to the 64-bit length.

You can declare the desired length of a variable by using the CM Fortran language extension *type*length*. For example:

```
INTEGER*8 A
COMPLEX*16 Z
```

Alternatively, you can use the Fortran 90 *kind* mechanism, described below.

Copyright © 1993 Thinking Machines Corporation

10.1.1 KIND Keyword

Fortran 90 describes the keyword **KIND**, which associates a length with the type in a specification statement. The value of **KIND**, called the *kind type parameter*, must be a scalar integer *initialization expression*, meaning a compile-time constant. In CM Fortran, it can be any constant expression that evaluates to 32 or 64. For example:

```
INTEGER(KIND=64) A
INTEGER(KIND=_DOUBLE_INT_) B
REAL(KIND=_DOUBLE_) C

INTEGER LL
PARAMETER (LL=64)
INTEGER(KIND=LL) D
REAL(KIND=LL) E
```

10.1.2 Typed Constants

Fortran 90 introduces the notion of typed constants. To specify the type (including kind) of a literal constant, append an underscore and then an integer constant expression that evaluates to a valid **KIND** number (32 or 64). For example, a constant of type **REAL(KIND=_DOUBLE_)** could be written either as **1.0D0** or as **1.0__DOUBLE_**.

Note the double underscore when the appended underscore of the typed constant is combined with the prepended underscore of the CM Fortran symbolic name for the kind type parameter:

```
INTEGER LL
PARAMETER (LL=64)
INTEGER(KIND=LL) D
D = 1234_64 + 1234_LL + 1234__DOUBLE_INT_
```

Given the rules of implicit coercion in expressions (see the *CM Fortran Reference Manual),* a typed constant provides a way to coerce variables to the desired length. (An alternative is the intrinsic functions described below in Section 6.2). For example, suppose you want to coerce a 32-bit integer N to length 64:

```
N = N + 0_64
```


Version 2.1, November 1993
Copyright © 1993 Thinking Machines Corporation

10.2 Enhanced Intrinsic Functions

Another component of the Fortran 90 kind mechanism is the intrinsic inquiry function **KIND**, which takes a scalar or array argument and returns the kind type parameter of the argument. The returned value is itself a scalar of the default integer type (**INTEGER*4** in CM Fortran). For example,

```
INTEGER I
INTEGER*8 J
...
PRINT *, KIND(I)        ! prints 32
PRINT *, KIND(J)        ! prints 64
```

The **KIND** function can be referenced in specification statements as well as in executable statements.

In addition, the intrinsic functions **INT** and **NINT**, which convert a value of any numeric type to an integer, have been enhanced to take an optional **KIND** argument. The **KIND** argument is an initialization expression (constant) that specifies the desired length of the result. Examples of **INT** referenced with the **KIND** argument are:

```
INTEGER*8 BIGINT
INTEGER*4 SMALLINT
...
BIGINT = INT(SMALLINT,KIND=64)
...
BIGINT = INT(SMALLINT,KIND=KIND(BIGINT)) !KIND intrinsic
```

Another example uses a conversion function to move array **SMALLINT** into the high 32 bits of **BIGINT**. The following construction does *not* express this intention; the shift is performed in 32 bits before the coercion to 64 bits.

```
BIGINT = ISHFT( SMALLINT, +32 )  ! Shift before coercion
```

The desired behavior (coerce and then shift by 32 positions) is expressed by:

```
BIGINT = ISHFT( INT( SMALLINT,KIND=64 ),+32 )
```

The comparable expression using a typed constant is:

```
BIGINT = ISHFT( SMALLINT+0_64, +32 )
```


Copyright © 1993 Thinking Machines Corporation

The three elemental intrinsics that inquire about the bit-level representation of integer and logical values behave as follows:

- **POPCNT** and **POPPAR**, which report the population count and population parity of the argument, always return an **INTEGER*4** result, for any argument of type **INTEGER*4**, **INTEGER*8**, or **LOGICAL**.
- **LEADZ**, which reports the number of leading 0 bits before a 1 bit is encountered, returns an **INTEGER*4** result for **INTEGER*4** and **LOGICAL** arguments, but returns **INTEGER*8** for arguments of type **INTEGER*8**.

10.3 Utility Library Support

The 64-bit integer type can be used for any CM array argument to procedures in the CM Fortran utility library, except the array aliasing routines. It may not be used, however, for integer scalar arguments. For front-end array arguments, such as the array argument to the dynamic allocation utilities, an **INTEGER*8** argument is accepted, but the extra length is not used. When passing a type as an argument to a utility procedure, use the predefined constant **CMF_LONG_S_INTEGER**.

For generating pseudo random numbers, the utility library provides a new procedure specifically for 64-bit integers:

```
CMF_RANDOM_LONG_S_INTEGER (DEST, LIMIT)
```

This utility takes only an **INTEGER*8** argument, and it runs the cellular automaton that generates the values for 64 generations. The **LIMIT** argument is also an **INTEGER*8**. The other random number utility, **CMF_RANDOM**, accepts CM arrays of any numeric type, including **INTEGER*8**, but it runs the automaton for only 32 generations, and it accepts only a 32-bit **LIMIT** argument.

10.3.1 64-Bit Send Addresses

A send address can now be declared as an **INTEGER*4**, **INTEGER*8**, **DOUBLE PRECISION**, or **REAL*8**. The **INTEGER*8** type is limited to the CM-5 with vector units, with CM Fortran running under the **-vu** execution model.

Copyright © 1993 Thinking Machines Corporation

The CM-2/200 computes send addresses as 4-byte values; the CM-5 uses 8-byte send addresses. Both platforms accept either 4-byte or 8-byte send-address arrays, but only CM-5 systems with vector units accept the **INTEGER*8** type.

There may be a performance penalty for using 4-byte addresses on the CM-5, as the system coerces the values to 8-byte length. There is no performance penalty for using 8-byte send-address arrays on the CM-2/200, although there is some waste of memory. When you are using CM-5 systems with vector units only, it is preferable to use **INTEGER*8** send addresses. If not, for maximum portability, CM Fortran programs should declare send-address arrays as **DOUBLE PRECISION** or **REAL*8**. **INTEGER*4** send-address arrays should only be used in programs to be run on the CM-2/200, and only when conserving memory is an issue.

10.4 Restrictions on 64-Bit Integers

- **INTEGER*8** values cannot be used in **READ**, **WRITE**, and **PRINT** statements as anything other than the I/O list items (the data to be transferred). This type cannot be used for unit numbers, record numbers, status variables, and so on.
- **INTEGER*8** values cannot be used as scalar integer arguments to the CM Fortran utility library procedures. This type can be used for front-end array arguments, but the second 32 bits of the argument are ignored.
- As with **INTEGER*4**, the CM system does not detect integer overflow for **INTEGER*8** calculations. For example, multiplying 2**62 by 2**62 gives incorrect results without warning. Further, the partition manager and the vector units produce different incorrect results if overflow occurs while converting an **INTEGER*8** to an **INTEGER*4**.
- If an **INTEGER*8** is used as the index variable in a **DO** loop, the total number of iterations cannot exceed (2**31)–1.

Copyright © 1993 Thinking Machines Corporation