## Chapter 3

# Writing CM/AVS Modules

You can develop new code or adapt existing code to be compatible with the CM/AVS environment. You may combine your modules in a network with modules from other sources.

This chapter talks about the aspects of code that are unique to handling parallel arrays, including the topics listed below:

- The Parallel Field Type
- Using AVS Field Routines on Parallel Fields
- Allocating Parallel Fields
- Accessing Field Data and Coordinates

The chapter includes a sample module, and concludes with instructions for compiling and debugging your modules.

## 3.1 The Field Type

The data that is passed between AVS modules is categorized by type. Only the field data type is pertinent to CM/AVS.

An AVS field is an *n*-dimensional array of byte, short integer, integer, floating-point, or double-precision floating-point numbers. AVS fields contain some descriptive information, such as the number of dimensions and the type of coordinate mapping, but the bulk of a field is in its data and coordinate arrays.

A field is defined in a *computation space* where the axes are orthogonal and each data point is unit distance away from its neighbors along any axis.

This computation space is mapped into a *coordinate space* in one of three ways;

*uniform* — The coordinate space is determined by minimum and maximum values along each axis and is mapped directly onto the Cartesian grid defined by these extents.

*rectilinear* — The neighbors along any axis may be different distances apart; for each axis there is a separate array that gives the mapping from computation to coordinate space.

*irregular* — Each data point also has an explicit coordinate stored with it; this can be used to represent curvilinear volumes. The connectivity is still topologically rectilinear.

Fields are described in further detail in the *AVS Developer's Guide*.

## 3.2 The Parallel Field Type

The defining feature of a *parallel field* is the distribution of its data across the CM-5 processing nodes. The field's coordinate array may also be stored on the processing nodes. If the mapping from computation space to coordinate space is rectilinear or irregular, then the coordinate array is automatically placed on the processing nodes. If the mapping is uniform, then the few floating point-numbers that describe the mapping are left on the partition manager; the coordinate array for a uniform field is never put on the processing nodes.

In this discussion, we use the term *parallel field* for a field whose data is distributed over the CM-5 processing nodes. The term *serial field* refers to a field whose data lives in the memory of some scalar machine: either your local workstation or a CM-5 partition manager. When this distinction does not matter, we simply use the term *field*. CM/AVS provides routines for allocating and accessing *parallel* fields.

### 3.2.1 Declaring a Parallel Field

Parallel fields are declared in the same way as standard AVS fields. In C*, a parallel field is declared as a structure or as a pointer to a structure, as appropriate:

```
AVSfield *field;
```

In CM Fortran, a parallel field is declared as an opaque integer, which should be operated on only with AVS or CM/AVS routines:

```
integer field
```

### 3.2.2 Passing a Parallel Field

Parallel fields must be passed as single arguments to CM Fortran functions. This means that any CM Fortran module must include in its module description function a call to **AVSset_module_flags** with the **SINGLE_ARG_DATA** flag set:

```
call AVSset_module_flags(SINGLE_ARG_DATA)
```

For coroutine modules that output parallel fields, the **COROUT_UNPACK_ARGS** flag must also be set. In CM Fortran, module flags may be combined by using the **IOR** intrinsic:

```
      call AVSset_module_flags(IOR(SINGLE_ARG_DATA,
     $      COROUT_UNPACK_ARGS ))
```

## 3.3 Using AVS Field Routines on Parallel Fields

Most of the standard AVS field access routines work correctly on parallel fields; the exceptions are the ones that that touch the field data or coordinates. CM/AVS provides equivalents for these, as listed in Table 1.

**Table 1. Standard AVS field routine equivalents for parallel fields.**

| Standard AVS Field Routine | CMAVS Replacement |
|---|---|
| **AVSfield_alloc** | Use **CMAVSfield_alloc** or **CMAVSdata_alloc**. |
| **AVSfield_free** | Use **AVSdata_free**. |
| **AVSfield_data_offset** **AVSfield_data_ptr** | Convert the field into a CMF array or C* pvar with **CMAVSfield_data_get**. |
| **AVSfield_points_offset** **AVSfield_points_ptr** | Use these routines only for UNIFORM CM/AVS fields. For others, use **CMAVSfield_points_get**. |
| **AVSfield_reset_minmax** | Use **CMAVSfield_reset_minmax**. |
| **AVSfield_copy_points** | Use only for UNIFORM CM/AVS fields. For others, use **CMAVSfield_copy_points**. |
| **AVSbuild_field** **AVSbuild_2d_field** **AVSbuild_3d_field** | *Obsolete after AVS 2.0.* |

## 3.4 Allocating Parallel Fields

There are two ways that a parallel field can come into being: it may be explicitly allocated, or an input port may be declared as parallel, causing the received field to reside on the processing nodes.

### 3.4.1 Parallel Input Ports

To direct a module to distribute data on the CM-5 processing nodes, use the **PARALLEL** flag with **AVScreate_input_port**. (This flag may be **OR**'ed with others, such as **REQUIRED**.) If the **PARALLEL** flag is not set, the data is placed on the partition manager.

For example, a CM Fortran module that reads an image and processes it on the processing nodes might contain this input port definition:

```
     inport = AVScreate_input_port('input field',
$              'field 2D 4-vector byte',
$              IOR(REQUIRED, PARALLEL))
```

The AVS Network Editor displays ports for parallel fields and serial fields the same way, and allows connections between the two.

**NOTE**

A connection between an AVS module and a CM/AVS module can work only if the modules *are not linked in the same binary.* If this condition is not met, the results may appear to be correct at first; however, errors may appear later.

### 3.4.2 Explicit Allocation

To allocate a CM/AVS field, one may call **CMAVSdata_alloc** or **CMAVSfield_alloc**.

Given a dimension array and a string describing the desired field, **CMAVSdata_alloc** returns a parallel field.

```
output = CMAVSdata_alloc
              ("field 2D scalar byte",dims)
```

The string describing the field is the same as that used by **AVSdata_alloc**.

**NOTE**

**CMAVSdata_alloc** may be used only to allocate fields. If the string describes any other type, an error is raised.

**CMAVSfield_alloc** takes an AVS field as a template and allocates a corresponding parallel field. The new field may take its dimensions from the template or from an explicit dimensions array. This CM Fortran code fragment allocates an output field with the same properties as the input field:

```
iresult = AVSfield_make_template(input, template)
output = CMAVSfield_alloc(template,0)
```

The resulting output field is a duplicate of the input field, and it is guaranteed to be on the processing nodes, even if the input field was not.

## 3.5 Accessing Field Data and Coordinates

CM/AVS provides some special routines to gain access to a parallel field's data and coordinates.

### 3.5.1 Access Routines

There are two CM/AVS routines that give access to the data and coordinates in a CM/AVS field: **CMAVSfield_data_get** and **CMAVSfield_points_get**:

```
void:void *
    CMAVSfield_data_get(AVSfield *field, shape S);

float:void *
    CMAVSfield_points_get(AVSfield *field shape S);
```

The C* interface to these routines takes both a field and a shape, and returns a pointer to a parallel variable. The parallel pointer refers to the coordinates in the first argument (**AVSfield**). You must pre-allocate the shape using **CMAVSfield_alloc_points_shape**. You may use the same shape to construct pointers to any fields that have the same rank and dimensions.

In CM Fortran, the situation is slightly more complicated because both these routines construct and return arrays of arbitrary rank. There is no way to express such a generic array constructor in CM Fortran itself, so the routines instead return an opaque object that can be passed to a routine expecting a CM Fortran array.

This approach is similar to that used by the CM Fortran utility function **CMF_ALLOCATE_ARRAY**. CM/AVS uses a small array on the partition manager to hold an array descriptor. This one-dimensional array of integers must have length **CMF_SIZEOF_DESCRIPTOR**, which is defined in **/usr/include/cm/CMF_defs.h**.

```
      include '/usr/include/cm/CMF_defs.h'
      include '/usr/include/cm/cmavs.inc'
      integer field

      integer desc(CMF_SIZEOF_DESCRIPTOR)
CMF$LAYOUT desc(:serial)


      call CMAVSfield_data_get(field, desc)

      call CMAVSfield_points_get(field, desc)
```

The **CMF$LAYOUT** directive is not actually needed, but its use in the documentation and example code emphasizes that **desc** must not be a parallel array. The descriptor array returned by **CMAVSfield_data_get** or **CMAVSfield_points_get** may be passed to any routine expecting a CM Fortran array. Section 3.5.5 shows how to declare the layout of these arrays.

Note that both the C* and CM Fortran access routines are really returning pointers to a memory location on the processing nodes. Be careful not to refer to one of these pointers after freeing a field; it will no longer refer to valid data.

### 3.5.2 Primitive Data Types

When you declare a C* parallel variable or a CM Fortran array, the primitive data type must correspond with the AVS type, as shown in Table 2.

**Table 2. Primitive data types.**

| AVS Type | C* Type | CM Fortran Type |
|---|---|---|
| **AVS_TYPE_BYTE** | unsigned char | integer |
| **AVS_TYPE_SHORT** | short | integer |
| **AVS_TYPE_INTEGER** | integer | integer |
| **AVS_TYPE_REAL** | float | real |
| **AVS_TYPE_DOUBLE** | double | double-precision |

CM/AVS byte and short fields are promoted to integers for CM Fortran, since CM Fortran does not support parallel arrays of bytes or shorts, and it is simpler to manipulate integer fields. During this promotion, shorts are sign-extended to form integers, and bytes are not sign extended: shorts are in the range –32768 <= x <= 32767 and bytes are in the range 0 <= x <= 255.

**NOTE**

Even though the field data is promoted to integers, the *min_data* and *max_data* values are still kept as bytes and shorts. With Version 5.0, AVS provides **AVSfield_get_minmax_as_int** and **AVSfield_set_minmax_as_int**, which automatically coerce shorts and bytes to ints.

When you convert the coordinates in a CM/AVS field to a C* parallel variable or CM Fortran array, the result is always stored as single-precision floating-point numbers.

### 3.5.3 Data Array Layout

An AVS field is essentially an *n*-dimensional Cartesian grid, where each point in the grid may contain a single value or a vector of values. The length of this vector is given by the *veclen* member of the **`field`** structure.

In C*, the field data is stored in an *n*-dimensional shape. In this shape, we allocate a 1-dimensional per-processor array of length *veclen* using the appropriate primitive data type.

In CM Fortran, the field data is stored in an array with *n*+1 dimensions. The first axis has **`:SERIAL`** ordering (elements along this axis reside in the same physical processor) and length *veclen*. We call this serial axis the “vector axis.” To make it easier to write modules that are independent of vector-length, this vector axis is present even for scalar fields; in this case it is of length one. The remaining axes have **`:NEWS`** ordering.

### 3.5.4 Coordinate Array Layout

There is only one valid type for coordinate arrays: single-precision floating point. This applies to coordinates on the processing nodes or on the partition manager.

Uniform fields *always* have their coordinates stored on the partition manager.

Irregular fields are placed in a floating-point array with *ndim*+1 axes, where *ndim* is the dimensionality of the data array. The first *ndim* are **`:NEWS`** axes whose length is given by the corresponding entry in the dimensions array. The remaining axis is **`:SERIAL`** and of length *nspace*, where *nspace* is the dimensionality of the space in which the data points exist.

Rectilinear fields are placed in a 1-dimensional floating-point array with a single **`:NEWS`** axis. The length of the array is the sum of the lengths of all axes in the field.

### 3.5.5 Declaring the Arrays

Assume that you have a 2-dimensional uniform field, with a 4-vector of bytes at every point. In CM Fortran, the field data would be loaded into an array declared

```
     integer array(4, x, y)
CMF$LAYOUT(:serial, :news, :news)
```

where *x* and *y* are the lengths of the field's axes.

If you have a 3-dimensional scalar field, the field data would be loaded into an array declared

```
     integer array(1, x, y, z)
CMF$LAYOUT(:serial, :news, :news, :news)
```

## 3.6 Luminance Module Example

As a simple example, consider a module that takes the luminance of an image. In AVS, an image is represented by a 2-dimensional field with a 4-vector of bytes at every point. The coordinate mapping is usually uniform.

The luminance of an image is a weighted sum of the color components at each pixel. The first byte in each 4-vector is the alpha component; this component is typically used to store opacity, and it is not used to compute the luminance. The remaining bytes are the red, green, and blue components; we combine these, using weights appropriate for the NTSC luminance. This choice of weights makes our simple routine compatible with the AVS **luminance** module.

### 3.6.1 Luminance Module Code

A copy of the module **luminance.fcm** is included with other examples in the directory

**/usr/examples/cmavs**

Below is the CM Fortran code for the module. Note that the bytes of the image are automatically promoted to integers by CM/AVS; this makes it easier to deal with byte fields in CM Fortran. Note also that we have a separate routine, **luminance_compute**, which extracts parallel arrays from the CM/AVS fields and passes them to the function that actually computes the luminance.

```
C*********************************************************************
C A luminance module
C*********************************************************************

C---------------------------------------------------------------------
C Describe the module to AVS
C---------------------------------------------------------------------
      subroutine AVSinit_modules
      implicit none
      include 'avs/avs.inc'
      include 'cm/cmavs.inc'
      integer iport, oport
      external luminance_compute

C Set the module name and type
      call AVSset_module_name('luminance CM', 'filter')

C Make sure we pass in the args as integers
      call AVSset_module_flags(IOR(SINGLE_ARG_DATA,
     $      IOR(COOPERATIVE,REENTRANT)))

C Create an input port for the required field input
      iport = AVScreate_input_port('input field',
     $     'field 2D 4-vector byte',
     $     IOR(REQUIRED, PARALLEL))

C Create an output port for the result
      oport = AVScreate_output_port('output field',
     $     'field 2D scalar byte')

      call AVSset_compute_proc(luminance_compute)

      return
      end

C---------------------------------------------------------------------
C Unpack the structure members we need, create CMF arrays that point
C to the field data, and call the routine that does the real work
C---------------------------------------------------------------------
      integer function luminance_compute(in, out)
      implicit none
      include 'avs/avs.inc'
      include '/usr/include/cm/CMF_defs.h'
      include 'cm/cmavs.inc'

      integer in, out

      integer indesc(CMF_SIZEOF_DESCRIPTOR),
     $     outdesc(CMF_SIZEOF_DESCRIPTOR)
      integer iresult, dims(2)
```

```
C Now get pointers to the arrays containing the AVS field data
      call CMAVSfield_data_get(in,indesc)
      iresult= AVSfield_get_dimensions(in, dims)

C If there is already output data, deallocate it.
      if (out .ne. 0) then
         call AVSdata_free("field",out)
      endif
      out = CMAVSdata_alloc("field 2D scalar byte",dims)

C Get a pointer to the output data
      call CMAVSfield_data_get(out,outdesc)

C Copy the points from input to output
      iresult= CMAVSfield_copy_points(in,out)

C Call the real function that does the work
      call luminance_internal(indesc, outdesc, dims(1), dims(2))

C Return 1 to indicate success
      luminance_compute = 1
      return
      end

C---------------------------------------------------------------------
C The real workhorse
C---------------------------------------------------------------------
      subroutine luminance_internal(in, out, x, y)

      integer x, y
      integer in(4,x,y), out(1,x,y)
CMF$LAYOUT in(:serial,:news, :news), out(:serial,:news, :news)

C
C Set up the weights for NTSC luminance
C
      double precision red_weight, green_weight, blue_weight

      parameter (red_weight   = .299,
     $     green_weight = .587,
     $     blue_weight  = .114)


      out(1,:,:) = in(2,:,:) * red_weight +
     $     in(3,:,:) * green_weight +
     $     in(4,:,:) * blue_weight

      return
      end
```

## 3.7 The CM/AVS Header Files

The routines that you write must include the standard AVS header files. In addition, they must include the CM/AVS files that define all the appropriate symbols and return types for the CM/AVS routines.

The header file for CM Fortran routines is

**/usr/include/cm/cmavs.inc**

The include file for C* routines is

**<cm/cmavs.h>**

## 3.8 The CM/AVS Libraries

The CM/AVS subroutine and coroutine libraries are listed below.

For a sparc processor:

| | |
|---|---|
| CMF subroutine | **libcmavsflow_f_cm5_sparc_sp.a** |
| CMF coroutine | **libcmavssim_f_cm5_sparc_sp.a** |
| C* subroutine | **libcmavsflow_c_cm5_sparc_sp.a** |
| C* coroutine | **libcmavssim_c_cm5_sparc_sp.a** |

For a vector unit processor:

| | |
|---|---|
| CMF subroutine | **libcmavsflow_f_cm5_vu_sp.a** |
| CMF coroutine | **libcmavssim_f_cm5_vu_sp.a** |
| C* subroutine | **libcmavsflow_c_cm5_vu_sp.a** |
| C* coroutine | **libcmavssim_c_cm5_vu_sp.a** |

These libraries act in conjunction with the standard AVS libraries:

| | |
|---|---|
| FORTRAN subroutine | **libflow_f.a** |
| FORTRAN coroutine | **libsim_f.a** |
| C subroutine | **libflow_c.a** |
| C coroutine | **libsim_c_.a** |

When you link a CM/AVS module, specify the CM/AVS library first, then the corresponding AVS library. To build a CM Fortran subroutine module, for example, link against **libcmavsflow_f_cm5_sparc_sp.a** first, then against **libflow_f.a**.

## 3.9 Compiling a Module

To compile a CM Fortran subroutine module, start a shell on a CM-5 compile server and invoke the CM Fortran compiler:

For a sparc processor:

```
cmf -cm5 -sparc -o module_name module_name.fcm \
-lcmavsflow_f_cm5_sparc_sp -L/usr/avs/lib -lflow_f \
-lgeom -lutil -lm
```

For a vector unit processor:

```
cmf -cm5 -vu -o module_name module_name.fcm \
-lcmavsflow_f_cm5_vu_sp -L/usr/avs/lib -lflow_f \
-lgeom -lutil -lm
```

To compile a C* subroutine module, start a shell on a CM-5 compile server and invoke the C* compiler:

For a sparc processor:

```
cs -cm5 -sparc -o module_name module_name.cs \
-lcmavsflow_c_cm5_sparc_sp -L/usr/avs/lib -lflow_c \
-lgeom -lutil -lm
```

For a vector unit processor:

```
cs -cm5 -vu -o module_name module_name.cs \
-lcmavsflow_c_cm5_vu_sp -L/usr/avs/lib -lflow_c \
-lgeom -lutil -lm
```

To compile *coroutine* modules, replace **flow** with **sim** in the **lcmavsflow...** and **lflow_c** library names above.

## 3.10 Debugging a Module

AVS reports run-time errors in a dialog box. However, the run-time error messages are not as detailed as those issued by a debugger, and some problems may appear to be downstream from the actual error.

To obtain detailed debugging messages, follow these steps:

1. Compile all the files for your module with the **-g** switch.

2. Select **Read Remote Modules** (or **Read Modules**, if you are on the partition manager) to add your module to the palette.

3. Start a shell on the machine that will run this module (for CM/AVS modules, this is the partition manager).

4. In this shell, change to the directory containing your module and invoke **avs_dbx**:

   **avs_dbx -debug prism** *your_module*

   The **-debug** switch lets you specify your preferred debugger. You may substitute **"prism -C"** (including the quotes) for **prism** .

5. Drag your module into a network. It will not fire immediately. Instead, you will see the following message in the window where you invoked **avs_dbx**:

   *your_module* `instance waiting, fire when ready...`

6. Set the desired breakpoints.

7. Launch the module by telling the debugger to run it.

## 3.11 Getting Help

Man pages for all the CM/AVS modules are viewable through AVS after you have "read" them following the instructions in Sections 2.3 and 2.4. To view them:

1. In the shell where you will invoke the AVS kernel, set the environment variable AVS_HELP_PATH as follows:

C shell:

```
setenv AVS_HELP_PATH /usr/doc/cmavs/modules
```

Bourne or Korn shell:

```
AVS_HELP_PATH=/usr/doc/cmavs/modules
export AVS_HELP_PATH
```

2. Open the AVS network editor.

3. In the module library list, find the CM/AVS module whose man page you want to view. Using the right mouse button, select the button on the right of the module icon.

4. On the resulting pop-up menu, select **Show Module Documentation**. The man page will appear in the AVS viewer.

## 3.12 Multiple-Module Binaries

CM/AVS modules may be linked together into a single binary in exactly the same way as AVS modules. With the exception noted below, this is desirable, because it enables multiple modules in a network to run in a single process.

**NOTE**

A connection between an AVS module and a CM/AVS module can work only if the modules *are not* linked in the same binary.

Field transfer between modules in a single process can be considerably faster than field transfer between modules in separate processes. The former involves a simple pointer copy, while the latter uses sockets to transfer all the data.