Chapter 13

Global/Local Programming

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

As of this CM Fortran 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).

NOTE

Your feedback on global/local programming in CM Fortran 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 © 1989–1994 Thinking Machines Corporation

At any point, 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 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 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 called from global 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

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

finished the local routine, control 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.

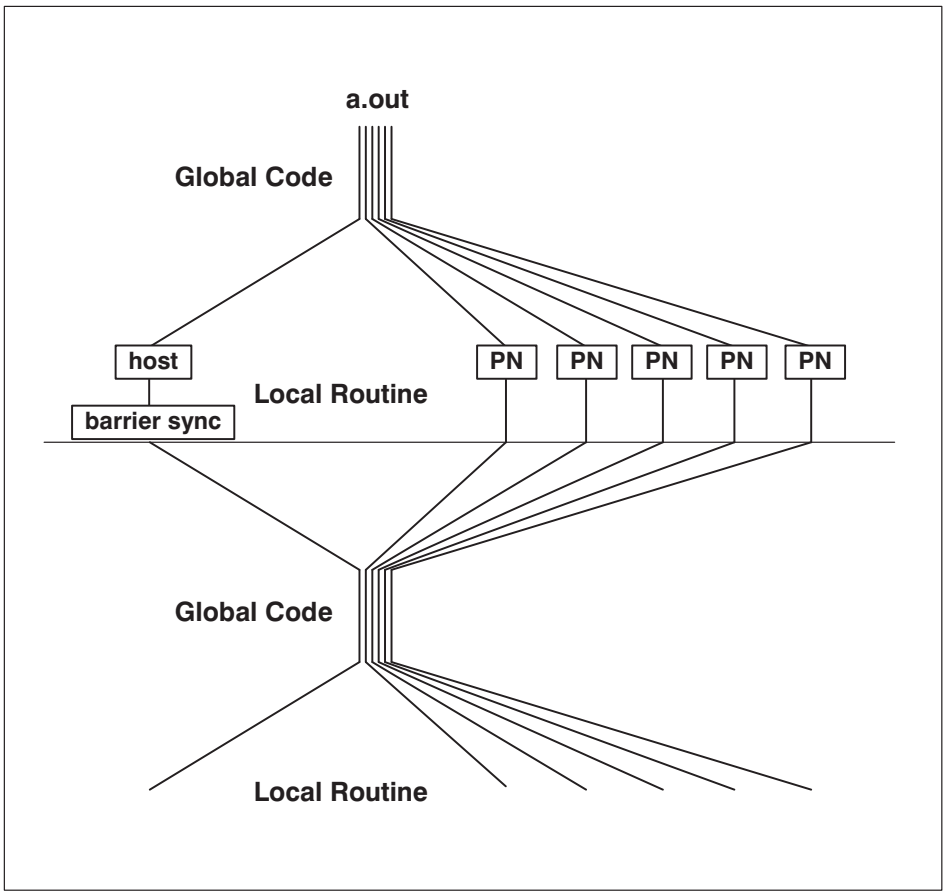

Figure 28. Programmer's view: Global/local programming.

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

Copyright © 1989–1994 Thinking Machines Corporation

13.1 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, including **`PRINT`** statements.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

13.2 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 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.

13.3 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.
- *global serial arrays:* Arrays allocated by the global program, which live in partition manager serial memory.
- *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.

13.3.1 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.

Copyright © 1989–1994 Thinking Machines Corporation

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.

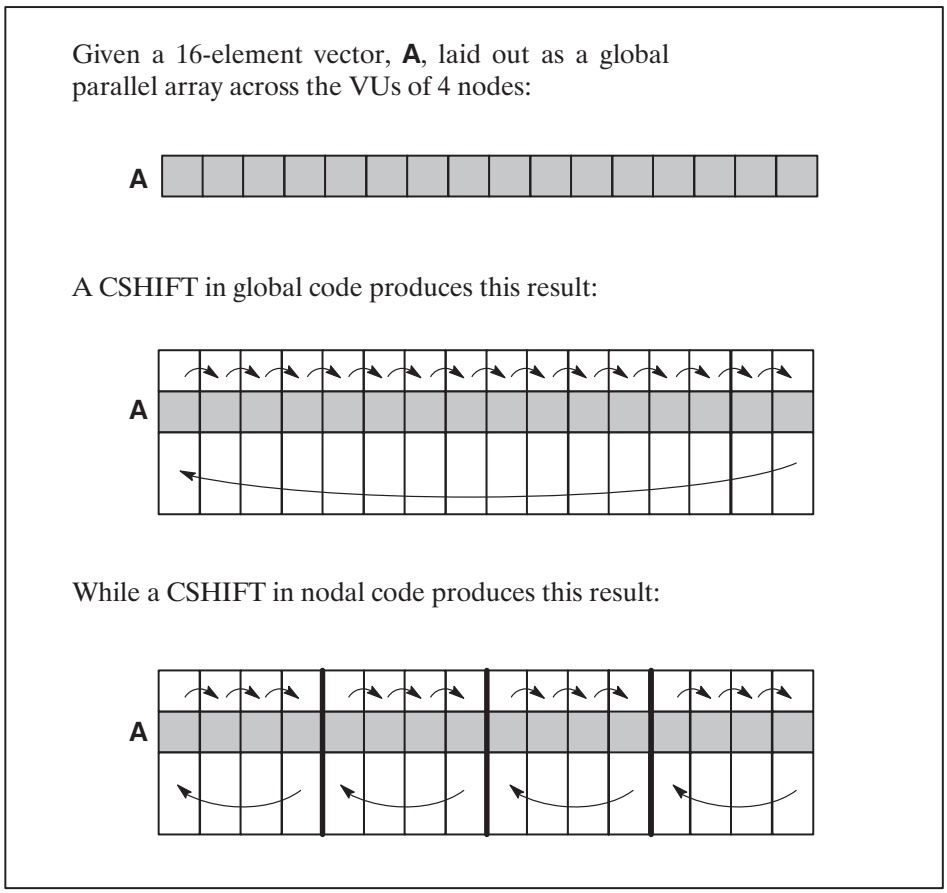

Figure 29. Computing on arrays and subarrays.

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 **SIZE** intrinsic function would return a size of 32; and so on. (See Figure 5.) If a node wants to find out where its elements reside within the global array, it uses CMGL library procedures.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

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. 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. A third, **`CMGL_global_to_physical`**, provides information on the mapping between global array indices and the corresponding physical node. Section 13.6 describes these procedures.

13.3.2 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.

Copyright © 1989–1994 Thinking Machines Corporation

Local routines have no access to global serial arrays, but the arrays can be broadcast:

- 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 PM memory), and that neither local routines nor any procedures they may call can pass data to the PM 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).

(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.)

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

13.4 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 for CM Fortran code or 0-based for C code. Arrays with lower bounds other than that 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.

13.5 Performance

The time it takes a global program to call a local subroutine and then return is hundreds of microseconds.

Copyright © 1989–1994 Thinking Machines Corporation

13.6 Global/Local Library Routines

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

Version 2.2, October 1994
Copyright © 1989–1994 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 '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 © 1989–1994 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 2.2, October 1994
Copyright © 1989–1994 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

```
CMGL_local_to_global( ARRAY, L_INDEX, 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 serial 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 serial 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.

Result

None.

Copyright © 1989–1994 Thinking Machines Corporation

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 for CM Fortran code or a 0-based array for C code. If the local indices are out of bounds for the subarray, **G_INDEX** is 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.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Example

Imagine a 16 x 16 CM Fortran global array named **A** divided across four nodes. Each node wants to find out where in the global array its subarray begins.

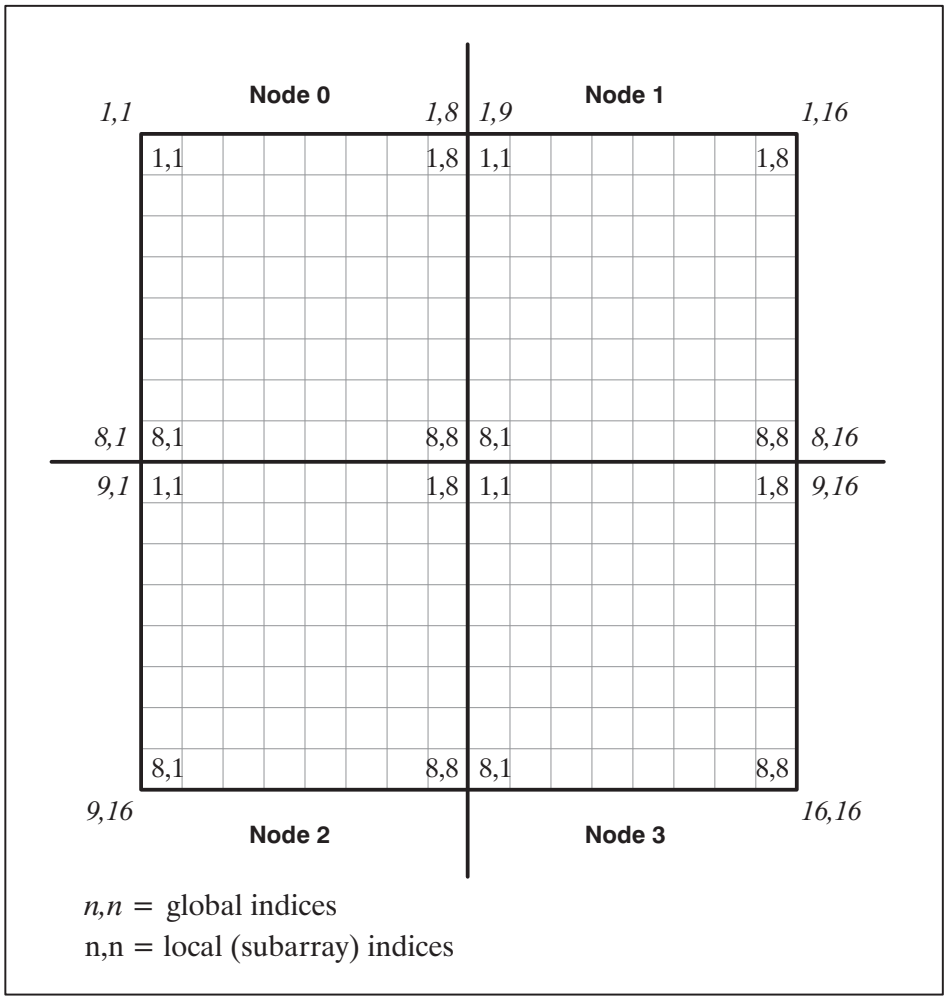

Figure 30. A 2D array and its four subarrays.

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 will be 1,9; for Node 2, 9,1; for Node 3, 9,9.

Copyright © 1989–1994 Thinking Machines Corporation

Error Conditions

Calling **CMGL_local_to_global** or **CMGL_global_to_local** on an array that 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.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

CMGL_global_to_local

Given global array indices, provides local array indices.

CM Fortran Syntax

```
CALL CMGL_GLOBAL_TO_LOCAL( ARRAY, G_INDEX, L_INDEX, LOCAL )
```

C Syntax

```
CMGL_global_to_local( ARRAY, G_INDEX, L_INDEX, LOCAL );
```

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**.

G_INDEX — A one-dimensional serial 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.

L_INDEX — A one-dimensional serial 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.

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 is set to 1 if the subarray contains the specified element, and to 0 otherwise. In CM Fortran, **LOCAL** is an **INTENT(OUT)** argument.

Result

None.

Copyright © 1989–1994 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 for CM Fortran code or a 0-based array for C code.

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 that 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.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

CMGL_global_to_physical

Given global array indices, provides the number of the corresponding physical node.

CM Fortran Syntax

```
CALL CMGL_GLOBAL_TO_PHYSICAL( LOCAL_ARRAY, G_INDEX, PN, L_INDEX )
```

C Syntax

```
PN = CMGL_global_to_physical( LOCAL_ARRAY, G_INDEX, L_INDEX );
```

Arguments

LOCAL_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 **LOCAL_ARRAY**.

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

PN In CM Fortran, the number of the physical node on which **G_INDEX** resides. **PN** is a scalar integer with a value in the range 0 through *number of nodes − 1*. **PN** is an **INTENT(OUT)** argument.

L_INDEX A one-dimensional serial integer array whose size is equal to the rank of **LOCAL_ARRAY**. It receives the coordinates within the subarray, **LOCAL_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.

Result

For CM Fortran, none. For C, an integer scalar, representing the number of the physical node.

Copyright © 1989–1994 Thinking Machines Corporation

Description

CMGL_global_to_physical, given a set of global coordinates within a local array, provides the number of the physical node containing those coordinates. All coordinates assume a 1-based array for CM Fortran code or a 0-based array for C code.

In CM Fortran, **CMGL_global_to_physical** assigns to **PN** the number of the physical node containing the coordinates specified in **G_INDEX**. In C, the function returns the number of the physical node. Both versions assign to **L_INDEX** the coordinates, within the subarray on that node, of the location specified in **G_INDEX**.

If **G_INDEX** is out of range, **CMGL_global_to_physical** returns −1 and assigns −1 to **L_INDEX**.

CMGL_global_to_physical must be called only from local routines, and only on subarrays. It is a subroutine in CM Fortran, and a function returning an integer in C.

Version 2.2, October 1994
Copyright © 1989−1994 Thinking Machines Corporation

Examples

Imagine a 16 x 16 CM Fortran global array named **A** divided across four nodes.

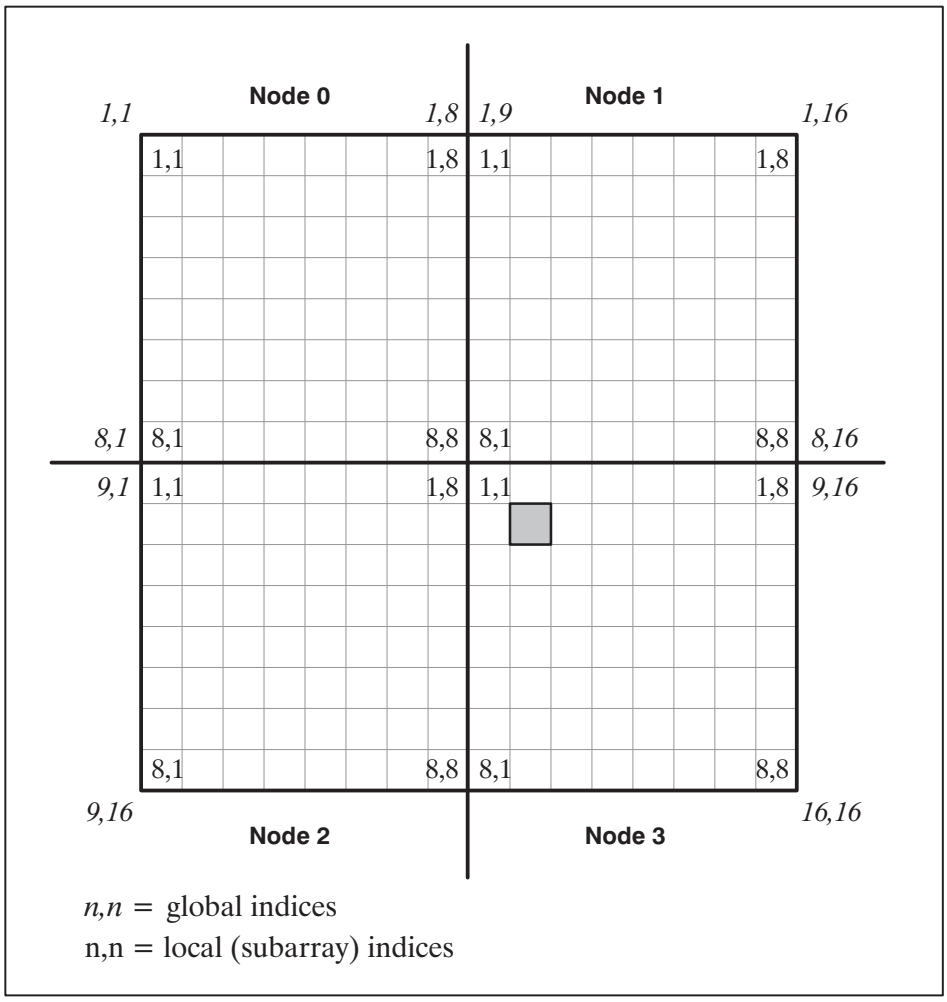

Figure 31. A 2D array and its four subarrays.

If **CMGL_global_to_physical** is called from code running on any of the four nodes and given the shaded square, 10,10, as **G_INDEX**, **PN** becomes 3 and **L_INDEX** becomes 2,2.

Copyright © 1989–1994 Thinking Machines Corporation

The following example shows a way to use **CMGL_global_to_physical**:

```
C ********************************************************
C  Function name:         FIND_PREV_PN
C
C  Purpose:               Find out what PN is next to this
C                         one in the -X direction
C ********************************************************

       INTEGER FUNCTION FIND_PREV_PN(LOCAL_ARRAY)
       INTEGER LOCAL_ARRAY(:,:,:)
CMF$LAYOUT LOCAL_ARRAY(:NEWS,:NEWS,:NEWS)

       INTEGER L_INDEX(3), G_INDEX(3), PN
CMF$LAYOUT L_INDEX(:SERIAL), G_INDEX(:SERIAL)

C First get the global coordinate of our local origin:

       L_INDEX(0) = 1
       L_INDEX(1) = 1
       L_INDEX(2) = 1
       CALL CMGL_LOCAL_TO_GLOBAL(LOCAL_ARRAY, L_INDEX, G_INDEX)

C Adjust global coordinate:

       G_INDEX(0) = G_INDEX(0) - 1

C Now convert to 'local', though it'll be someone else's local,
C since G_INDEX(0) is not in this array.
C This returns PN==-1 if it's out of bounds.

       CALL CMGL_GLOBAL_TO_PHYSICAL(LOCAL_ARRAY, G_INDEX,
      &                             PN, L_INDEX)

       FIND_PREV_PN = PN
       RETURN
       END
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Error Conditions

Calling **`CMGL_global_to_physical`** on an array that 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 the routine from a global program will cause a failure at link time.

Copyright © 1989–1994 Thinking Machines Corporation

13.7 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. Prototype files have the `.proto` suffix.

13.7.1 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 CM arrays and 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.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

13.7.2 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 a local routine 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)
```

13.7.3 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.

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

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 © 1989–1994 Thinking Machines Corporation

Here is an example of a local routine with one 2-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

Most 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 1.6) can be used to determine the position of a subarray element within the global array, and vice versa.

All CM Fortran intrinsics (**SIZE**, **LBOUND**, 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 **LBOUND(1)** and **UBOUND**, 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).

Version 2.2, October 1994
Copyright © 1989–1994 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 **UBOUND** 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.

13.7.4 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 © 1989–1994 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 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. (A C routine 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 *VU Programmer's Handbook* includes information on array descriptors, along with an explanation of 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 1.6) 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. The C versions of these functions require and provide 0-based coordinates for all their arguments.

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.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

13.7.5 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 semicolons (;).

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.

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.

Copyright © 1989–1994 Thinking Machines Corporation

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 serves as 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.

13.7.6 Compiling and Linking

CM Fortran provides a **-local** switch to handle the compiling, linking, and wrapper-generating details involved in a global/local program. Ordinarily, the CM Fortran driver is explicitly built for a VU-equipped system; if not, the **-cm5** and **-vu** switches must also be used. (Remember, global/local programs can run only on CM-5 systems equipped with VUs.)

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.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

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 foo_global.fcm -local foo_local.fcm foo.proto
```

If there are several local files, then:

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

Object files may be used as well, as in:

```
% cmf foo_global.o -local foo_local1.o
        -local foo_local2.o -local foo_local3.o
        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 '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 © 1989–1994 Thinking Machines Corporation

13.7.7 Debugging

Prism 2.0 can be used to debug both the global and the local part of a global/local CM Fortran program. However, you will need two copies of Prism (one to debug the global part and one to debug the local part).

To get two graphical Prisms started on your program, you can follow this procedure. (Assume your program is called **a.out**.)

1. In one window, issue the command:

   ```
   % prism a.out
   ```

   Prism starts up.

2. Set a breakpoint in **a.out** and issue a **run** command to get your program going.

3. Note the program's process ID, which is listed in a Prism message.

4. Then issue the **prism -node** command, specifying the name of the program and its process ID:

   % prism -node a.out *pid*

   This starts the second copy of Prism. You may have to use the **file** command to manually set Prism to the top-level local file. All nodes will come up in interrupted state. You can set breakpoints in your local code if you wish; then you should issue a **cont** command to get all nodes running again.

At this point you have two copies of Prism running. You can use Global Prism to perform any debugging operations on the global part of you program, and Node 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.

Note, however, that if an array does not neatly match the size of the machine, so that some PNs have less data than others (or possibly no data at all) then you must debug on a node-by-node basis; printing data with the PN set **all** in Node Prism will give incomplete, incorrect, or meaningless results.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

NOTE

In order to do debugging operations on CM arrays from Global Prism, all PNs must be running from the point of view of Node Prism. That is, they cannot be interrupted or stopped at nodal Prism breakpoints. If any PNs are stopped, the global **`print`** will hang until they are continued.

13.7.8 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`**.

However, profiling the global part of a global/local program is supported. Prism shows how much total time is spent in a local routine, but you cannot determine how that time is being spent. For this purpose, compile the global part of the code with **`-cmprofile`**, compile the local part without **`-cmprofile`**, and then link it all together.

Copyright © 1989–1994 Thinking Machines Corporation