Appendix D

Global/Local Programming

CM-5 C* provides support for a programming model in which a global C* program can call local C* functions running on the individual nodes of a partition. This is known as the *global/local* programming model. The local functions can communicate via calls to the CMMD message-passing library.

The global/local programming model lets you obtain the advantages of both data parallel and message-passing programming.

You might want to do local programming

- If you want explicit control (via CMMD) over the communication in your program, because use of the global data parallel model is too restrictive.
- If use of the standard data parallel model is creating too many code blocks (and thereby degrading performance) because your program has an intricate but local control flow.

You might want to use C* global/local programming rather than simply using C and CMMD in order to take advantage of the vector units.

Note these points with regard to global/local programming in C*:

- Global/local programs can run only on CM-5s with vector units.
- Global/local programming is also available in CM Fortran. Issues involved in C* global/local programming are similar to those involved in CM Fortran, but the interface is not identical. Users familiar with global/local programming in CM Fortran should still read this section.

Copyright © 1990–1995 Thinking Machines Corporation

D.1 Overview

A C* global/local application begins with a global C* main program, executing in data parallel fashion; that is, it lays out the elements of parallel variables across all nodes in the partition in which it is running. and it operates on these parallel variables in a global, synchronous manner.

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

- Each node operates independently while executing the local function.
- The global program can pass parallel and scalar variables to the local functions. These variables are treated differently:
  - Parallel variables are passed by reference to the local functions, with each node operating on its own portion of the parallel variable. The local code can alter the values of its portion of a parallel variable.
  - Scalar variables are passed by value to the local functions; each node operates on its own copy. The local code cannot reference or alter values in the global program's scalar variables.
- While executing local code, the nodes can use CMMD message-passing functions to communicate, synchronize, and share data with other nodes.
- 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.

Figure 102 gives a conceptual view of global/local programming. Note that different instances of the local function can take different amounts of time to complete; they wait for the last node to complete before returning control to the global program.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

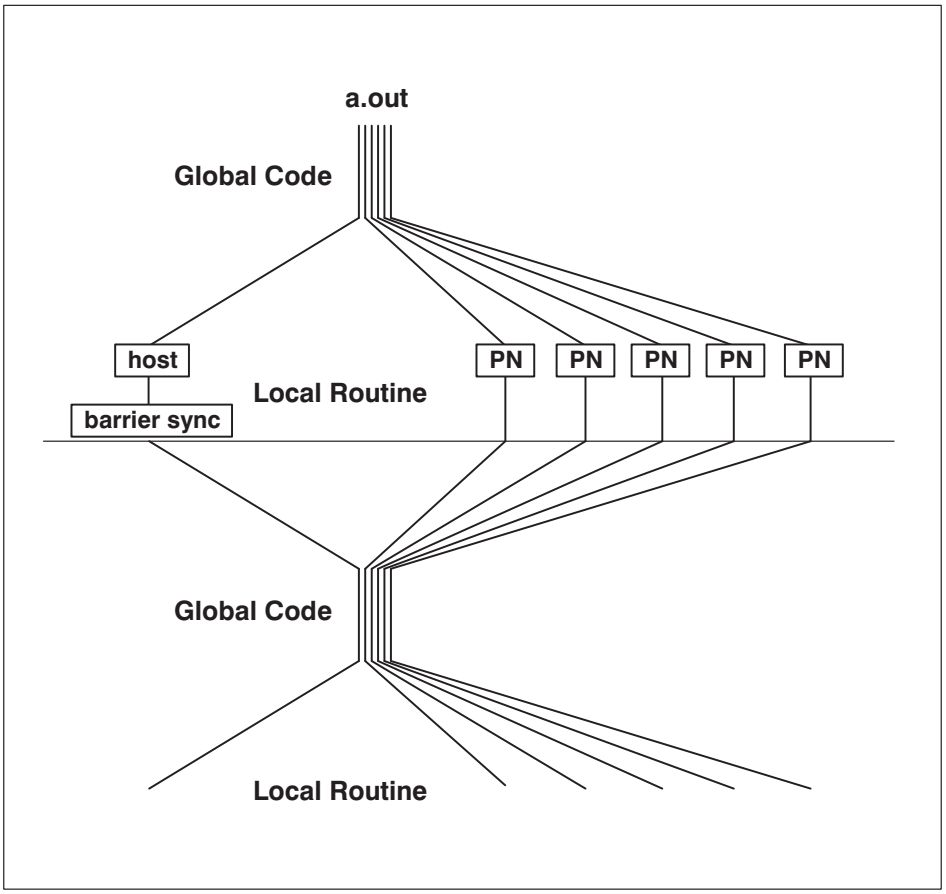

Figure 102. Programmer's view: Global/local programming.

Section D.2 goes into more detail about control flow in global/local programs.

Section D.3 explains shape and context in global/local programs.

Section D.4 describes the use of parallel and scalar data in global/local programs.

Section D.6 describes how to write a global/local program.

Section D.7 describes some inquiry functions available for use in global/local programs.

Section D.8 describes how to compile and link the program.

Copyright © 1990–1995 Thinking Machines Corporation

Section D.9 describes how to use Prism to debug and analyze the performance of global/local programs.

Section D.10 contains a sample global/local program.

D.1.1 Limitations

Note these limitations in global/local programming. (Some are discussed in more detail elsewhere in this chapter.)

- Local functions can call other local functions, but they cannot call a global function.
- Local functions cannot return values to global functions.
- Local functions cannot perform parallel I/O.
- The only way global scalar values can be accessed by local functions is by passing them, by value, to the local function.
- If any CMMD communication is performed within the scope of a local function, the program must ensure that the network is empty before the local function returns control to the global program.
- It takes a global function hundreds of microseconds to call a local function and then return.
- A global C* program can only call local C* functions. These local C* functions can, however, subsequently call local CM Fortran or C functions.

D.2 Flow Control in a Global/Local Program

When a C* 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 C*

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

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. If you're sending active messages, you need to 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 can call other local functions, but cannot call any global functions.

D.3 Shape and Context

When a global C* program calls a local function, there is no current shape, and the current context for all shapes is **`everywhere`**. In other words, the local function does *not* inherit the shape and context from the global function from which it was called; they must be explicitly set within the local function.

Copyright © 1990–1995 Thinking Machines Corporation

You can use shapes in a local routine that were declared in the global portion of the program. Each node sees only the positions of the shape that are local to its four vector units. The **positionsof** function, for example, applied to the shape in a local function, returns the number of positions in the shape allocated on the vector units of the node.

Note that this makes it possible for a shape in a local function to have a size of 0, if the shape is laid out such that it has no positions on a particular node; a shape size of 0 is not allowed in global code.

To learn about the way C* (and the CM-5 run-time system) lays out shapes on the vector units, see Appendix A.

You can also declare a local shape within a local function. Copies of such a shape exist independently on the vector units of each node; each copy is visible only on its own node.

D.4 Global and Local Data

The only variables in a global function that are visible to a local function are those that the global function passes to it as arguments; the global function can pass both scalar variables and pointers to parallel variables. Local functions cannot operate on any other global variables.

Local functions can create their own parallel and scalar variables. These variables are visible only on the calling node. To pass such data from one node to another, you must use CMMD calls.

D.4.1 Parallel Variables

As mentioned above, a global function can pass a global parallel variable to a local function; the parallel variable must be passed by reference. Each node sees only the elements of this parallel variable that are local to its vector units.

Assume, for example, a 1024-element parallel variable **x**, on a 32-node partition. Each node would hold 32 elements of the parallel variable, 8 elements per VU. Global code sees **x** as a single parallel variable, 1024 elements long.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

When it passes **x** to a local function, the local code on each of the 32 nodes sees **x** as signifying its own 32-element parallel variable. Each node can operate on its own version of **x** independently; for example, a call to **to_torus** within the local function would shift data within each group of 32 elements.

Changes that the local code makes to the global parallel variable are visible to the global code after the return from the local function.

You can use global/local inquiry functions to find out the relationship between the global and local coordinates of a parallel variable. See Section D.7.

If a parallel variable has a shape that has a size of 0, as discussed in Section D.3, operations on the parallel variable take place as if there were no active positions; for example, a reduction operation on the parallel variable returns the identity for the operation. See Section 6.5.1.

To learn about the way C* (and the CM-5 run-time system) lays out elements of a parallel variable on the vector units, see Appendixes A and B.

Local parallel variables are created within the local functions, and are visible only within the scope of those functions. Like the local version of a global parallel variable, local parallel variables are divided into four subgrids, one per vector unit. Local functions can thus treat both types of parallel variable in the same manner.

Any CMMD calls that can handle parallel variables can handle either global or local parallel variables within local functions. The one distinction is that garbage elements for local parallel variables are in identical positions on all nodes, whereas garbage elements for global parallel variables may vary widely from node to node: a given node's subset of a global parallel variable 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. (See Section A.1.2 for a discussion of garbage elements.)

D.4.2 Scalar Variables

Global scalar variables reside in the memory of the partition manager. Local scalar variables, created by a local routine, reside in the microprocessor memory on each node.

Copyright © 1990–1995 Thinking Machines Corporation

Unlike global parallel variables, global scalar variables cannot be modified or referenced by a local function. The local function can only modify its copy of the global scalar variable when the variable is passed by value to the local function. If data from a scalar variable must be made visible to some global procedure from a local routine, the local routine must transfer the data from a scalar variable into a global parallel variable. Note that this requires the global function to have allocated the parallel variable before calling the local function. Note also that the minimum size for the parallel variable is 32 elements per node (eight elements per VU).

D.5 I/O

Global programs can perform both serial and parallel I/O.

Local functions cannot currently perform parallel I/O. They *can* perform serial I/O, but note these points:

- When using the "streams" I/O interface defined by **<stdio.h>**, you need to flush I/O buffers at appropriate places in the program. Appropriate places to flush might be before leaving global mode (on the global side) and before leaving local mode (on the local side). To flush in global mode, use code like this:

```
printf(...);
fprintf(foofile, ...);

fflush(stdout);
fflush(foofile); /* Empty buffers before start of
                    local. */

call local();
```

  To flush in local mode, use code like this:

```
local() {
    printf(...);
    fflush(stdout); /* Empty buffer before
                         return to global. */
  }
```

  Flushing behavior changes, depending on both default stdio file I/O settings and on choice of file. For instance, standard output to a

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

terminal flushes when a newline is written to the buffer, or upon a read from standard input (if that is also a terminal). On the other hand, if standard output is piped to a file, then the flushing will occur under different conditions.

- You should be careful not to refer to a file that is on a parallel file system. This won't work, since I/O to such a file would be parallel.

D.6 Writing a Global/Local Program

A global/local C* program contains

- A global main program. Execution of the global/local program begins with the main program, as usual.
- One or more global functions written in C* and called by the main program.
- One or more local functions called by the main program or by one or more of the global functions within its scope.
- Zero or more local functions that may be called by other local functions.

It requires these files:

- One or more files for the global C* program (that is, the main program and any global functions called within its scope).
- One or more files for the local C* functions, and for any functions called within their scope. These must be separate from files containing the global portion of the program.
- A prototype file, which defines the interface between a global C* program and the local functions it calls. This file is described in Section D.6.3.

Copyright © 1990–1995 Thinking Machines Corporation

D.6.1 Calling the Local Function

A global function calls a local function in the same way it would call any function. Parallel variables must be passed by reference. For example, to call the local function **lf** and pass it a parallel variable **p1**, a scalar integer **i**, and the scalar value 25, do this:

```
shape [128]s;
int:s p1;
int i;

lf(&p1, i, 25);
```

D.6.2 Writing the Local Function

You write a local C* function in the same way you would write a global C* function. Thus, the local code for the function **lf**, called above, might look like this:

```
lf(int:void *p1, i, 25) {
    with(shapeof(*p1)) {
         /* ... */
    }
}
```

Keep in mind these points, discussed in earlier sections:

- You must explicitly set the current shape and the context.
- The local function for each node can operate only on its elements of a parallel variable passed to it.
- You can use global/local inquiry functions to find out the relationship between global and local coordinates of a parallel variable. See Section D.7. If you call any of these functions, include the header file **<csgl.h>**.
- You cannot perform parallel I/O from within the local function.
- You can include calls to the CMMD library to communicate with other nodes. If you include CMMD calls, you must also include the header file **<cm/cmmd.h>**.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

D.6.3 Writing the Prototype File

You must provide a special *prototype file* for C* programs that call local functions. This file contains a prototype for each local function that is called from global code, and thus defines the interface between the global and local portions of the program.

The prototype file can have any name; it must have the suffix **.proto**. A global/local program can contain multiple prototype files.

Prototypes

A prototype describes a local function's arguments and calling environment. For example, a prototype for a local routine **lr**, written in C*, having one parallel variable argument and one scalar integer argument, would be

```
lr(array p1, int i):host_cs:node_cs;
```

Note these points:

- Give all parallel variables the type **array**; this type corresponds to a pointer to a parallel variable in the function definition. Give all scalar variables the type they receive in the function definition (**int**, **double**, etc.); all types are allowed.
- The actual names given to the arguments in the prototype do not matter. They do not need to match the argument names in the definition of the local function. They are included in the prototype specification to allow prototypes to conform as closely as possible to ANSI prototypes.
- Each prototype must end with the syntax

  ```
  :host_cs:node_cs;
  ```

  The semicolon is required. The prototype can include newlines.
- Use prototypes only for local routines called from the global C* program. Do not write prototypes for local functions that are called by other local functions.

Copyright © 1990–1995 Thinking Machines Corporation

Use

The prototype file is input to a tool called **cmmd_cswrapper_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 functions to work. You don't need to understand the output created by the tool; you only need to know how to write the prototype file.

See Section D.8 for information on compiling and linking prototype files.

D.7 Global/Local Inquiry Functions

C* provides three inquiry functions that let you determine the mapping between global and local parallel variables. Note these points in using the functions:

- If you call any of these functions, you must include the header file **<csgl.h>**.
- You can call these functions only from a local function.
- These functions have corresponding versions in the CM Fortran. Unlike the CM Fortran versions, however, all parallel variables, arrays, and processor numbers in the C* versions are 0-based.

D.7.1 Determining the Global Coordinates from the Local Coordinates

Use the function **CMC_local_to_global** to determine the mapping between coordinates of an element of a parallel variable as seen locally and its coordinates as seen globally. The function's definition is

```
#include <csgl.h>

extern int CMC_local_to_global(shape x_shape,
                               int *l_index,
                               int *g_index);
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

where

`x_shape` is the local shape of a parallel variable **`x`**. The parallel variable must have been passed as an argument to the local function from which this function is called.

`l_index` is a 1-dimensional array of integers whose size is equivalent to the rank of **`x`**. It contains the local coordinates of an element in **`x`**.

`g_index` is a 1-dimensional array of integers whose size is equivalent to the rank of **`x`**. It receives the global coordinates of the same element in the global view of **`x`**.

The return value is 1 if **`l_index`** contains valid coordinates for the node from which it is called; the return value is 0 otherwise. If **`l_index`** is invalid, **`g_index`** is filled in with −1s.

You can call **`CMC_local_to_global`** only from local functions.

Example

Imagine a 16 x 16 global parallel variable named **`A`** divided across four nodes. Each node wants to find out where in the global parallel variable its local portion begins.

Copyright © 1990−1995 Thinking Machines Corporation

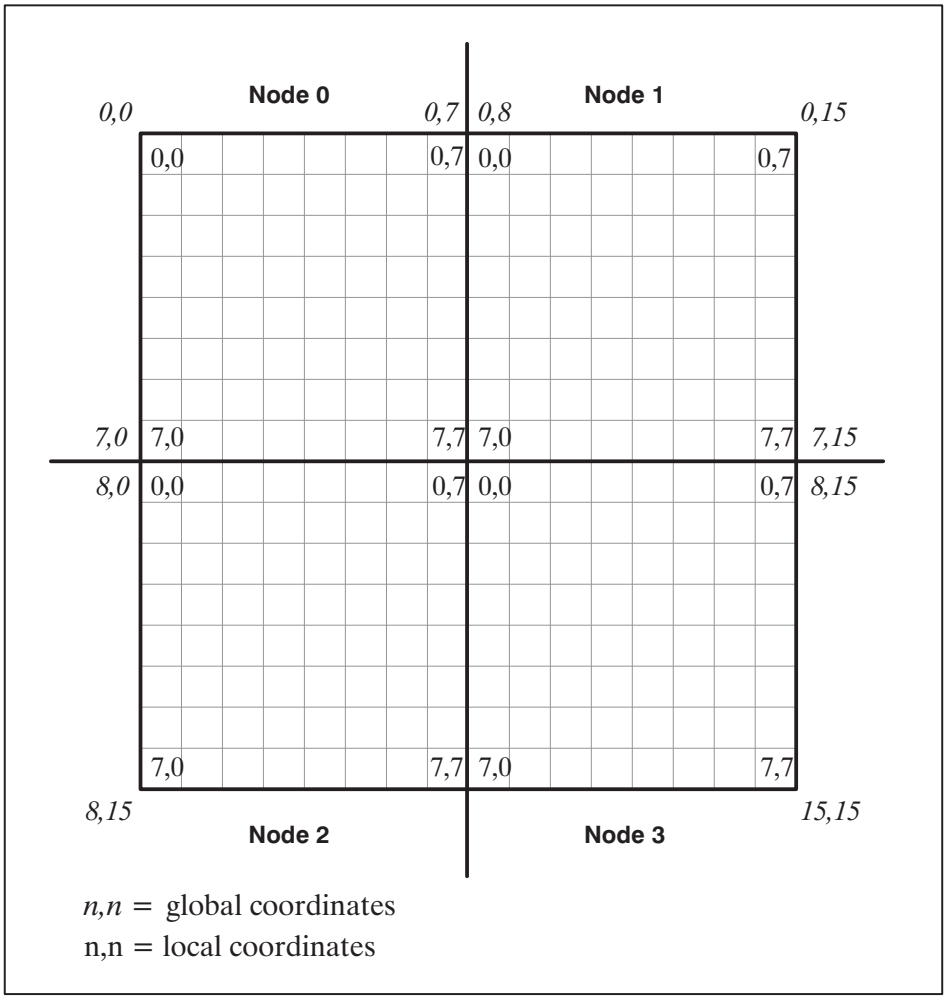

Figure 103. A 2-dimensional global parallel variable and its four local portions.

Each node calls **`CMC_local_to_global`**; for each call, the **`x`** argument is **`A`** and **`L_INDEX`** is a two-element array containing two 1s. After the call executes, **`G_INDEX`** will contain two integers representing the position in the global parallel variable of each local parallel variable's starting element. For Node 0, this will be 0,0; for Node 1, it might be 0,8; for Node 2, 8,0; for Node 3, 8,8.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

D.7.2 Determining the Local Coordinates from the Global Coordinates

Use the function **CMC_global_to_local** to determine the mapping between coordinates of an element of a parallel variable as seen globally and its coordinates as seen locally. Its definition is

```
#include <csgl.h>

extern int CMC_global_to_local(shape x_shape,
                               int *g_index,
                               int *l_index);
```

where

x_shape is the local shape of a parallel variable **x**. The parallel variable must have been passed as an argument to the local function from which this function is called.

g_index is a 1-dimensional array of integers whose size is equivalent to the rank of **x**. It contains the global coordinates of an element in the global view of **x**.

l_index is a 1-dimensional array of integers whose size is equivalent to the rank of **x**. It receives the local coordinates of the same element in the local view of **x**.

The return value is 1 if the coordinates in **g_index** represent an element on the node from which the function is called; the return value is 0 otherwise. If the return value is 0, **l_index** is filled in with −1s.

You can call **CMC_global_to_local** only from local functions.

Example

Consider again the 2-dimensional parallel variable **A**, 16 x 16, divided among 4 nodes, as shown in Figure 103. Suppose that all nodes in a C* local routine call **CMC_global_to_local**, specifying **A** for the argument **x** and providing a **g_index** containing the global coordinates 15,15. After the call, Node 3 sees the coordinates 7,7 in **l_index**. Nodes 0, 1, and 2 will have a return value of 0 and the contents of **l_index** set to −1,−1, since the global coordinates 15, 15 do not represent an element on these nodes.

Copyright © 1990−1995 Thinking Machines Corporation

D.7.3 Determining the Node Number from the Global Coordinates

Use the function **CMC_global_to_physical_index** to determine the physical node on which the parallel-variable element with the specified global coordinates is located, along with the element's local coordinates. Its definition is

```
extern int CMC_global_to_physical_index(shape x_shape,
                                        int *g_index,
                                        int *l_index);
```

where

x_shape is the local shape of a parallel variable **x**. The parallel variable must have been passed as an argument to the local function from which this function is called.

g_index is a 1-dimensional array of integers whose size is equivalent to the rank of **x**. It contains the global coordinates of an element in the global view of **x**.

l_index is a 1-dimensional array of integers whose size is equivalent to the rank of **x**. It receives the local coordinates of the same element in the local view of **x**.

The function returns the number of the node on which **g_index** is located, or −1 if **g_index** is not a valid set of coordinates. If **g_index** is invalid, **l_index** is filled with −1s.

You can call **CMC_global_to_physical_index** only from local functions.

Example

Once again consider the 2-dimensional parallel variable **A**, 16 x 16, divided among 4 nodes as shown in Figure 103. In this case, we want to find out the node on which the element of **A** with global coordinates 9,9 is located. See Figure 104.

We call **CMC_global_to_physical_index** from a local function, passing 9,9 as its **g_index** argument. The function returns 3, since the global coordinates 9,9 are located on node 3, and sets **l_index** to 1,1, since these are the corresponding local coordinates on node 3.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

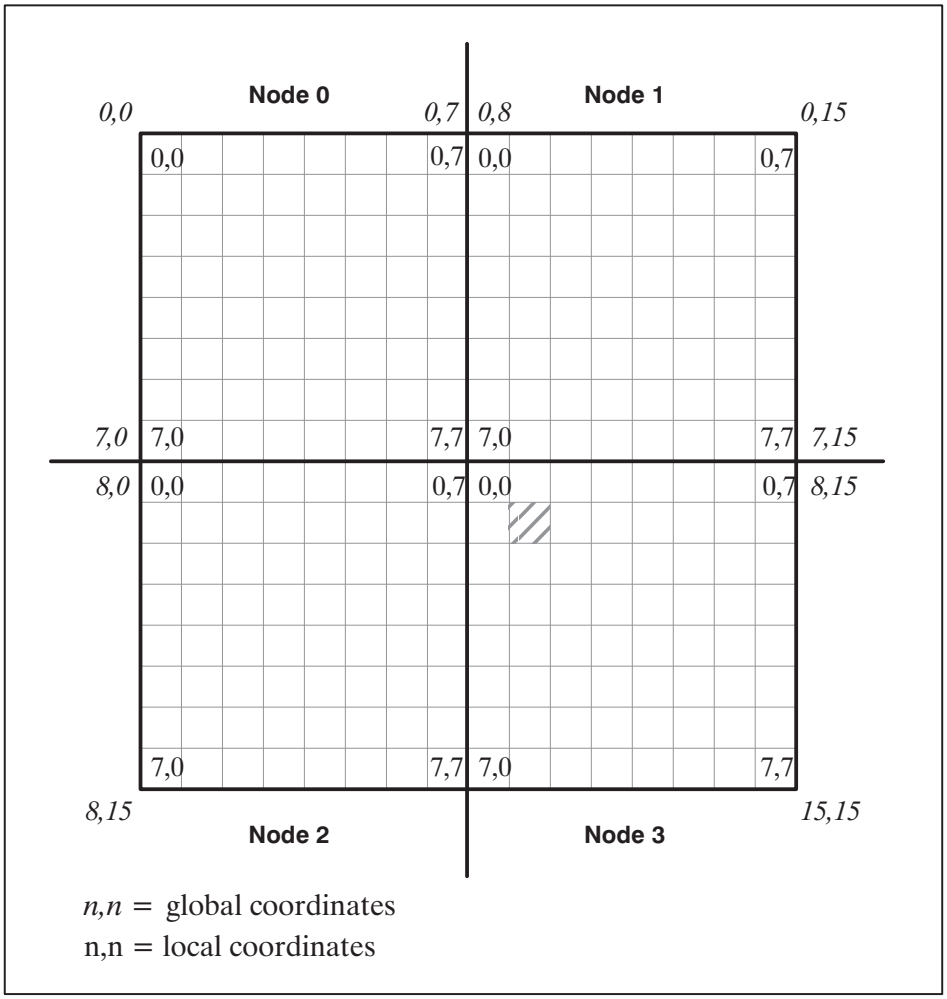

Figure 104. Global coordinates 9,9 for a 2-dimensional parallel variable laid out on four nodes.

D.8 Compiling and Linking

Use the **cs** command's **-local** option to handle the compiling, linking, and wrapper-generating details involved in a global/local program. You must also use the **-vu** option (if it's not the default at your site), since global/local programs can run only on CM-5s with vector units.

Copyright © 1990–1995 Thinking Machines Corporation

Unlike other compiler options, the **-local** option must appear before each **.cs** file that has local code, thereby letting the driver know that they are special.

You must also include the global/local prototype file discussed in Section D.6.3.

So, in a program that has one file with global C* code and one file with local C* code, the compile line would look like this:

```
% cs -vu foo_global.cs -local foo_local.cs foo.proto
```

If there are several local files, the compile line would look like this:

```
% cs -vu foo_global.cs \
     -local foo_local1.cs -local foo_local2.cs \
     -local foo_local3.cs foo.proto
```

You can also use object files. For example,

```
% cs -c -vu -o foo_global.o foo_global.cs
% cs -c -vu -o foo_local.o foo_local.cs
% cs -vu foo_global.o -local foo_local.o foo.proto
```

D.9 Debugging

You can use Prism 2.0 or later, running on CMOST 7.3 or later, to debug a global/local program. (Note that, to be debugged, the program must be compiled and linked with the **-g** option.) You need two copies of Prism, Global Prism to debug the global program units and Node Prism to debug the local program units.

Follow this procedure to get two graphical Prisms started on program **a.out**:

1. Start up Prism on the global program:

   ```
   % prism a.out
   ```

   (Prism starts up, with the source code of **a.out**'s main file displayed.)

2. Select **PN Debug** from Prism's **Utilities** menu to start up Node Prism.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

3. Set a breakpoint in the program and execute the Prism command **`run`** to start your program.

At this point, Node Prism starts up on the local program. All nodes come up in the "interrupted" state. You can set breakpoints in your local code if you wish; you should then issue the **`continue`** command to get all nodes running again. **NOTE:** Prism may not display the local file on which you want to work; in that case, you would have to issue the **`file`** command to display the correct file.

With the two copies of Prism running, you can use Global Prism to perform any debugging operations on the global part of the program, and Node Prism to perform any debugging operations on the local part of your program. Refer to the *Prism User's Guide* for more information on using Prism.

NOTE

To do debugging operations on parallel variables from Global Prism (for example, printing their values), all PNs must be "running" from the point of view of Node Prism. That is, they cannot be interrupted or stopped at Node Prism breakpoints. If any PNs are stopped, the global **`print`** will hang until they are continued.

If you do not have Prism 2.0 (or later) or are not on a CMOST 7.3 (or later) system, you can use the **`pndbx`** debugger in place of Node Prism. Follow the same startup instructions as above; clicking on **PN Debug** will start up **`pndbx`** instead of Node Prism. See the *CMMD User's Guide*, Version 3.0 or earler, for information on using **`pndbx`**.

D.9.1 Performance Analysis

Prism does not currently support node-level performance analysis. Do not use the C* option **`-cmprofile`** together with **`-local`**.

Copyright © 1990–1995 Thinking Machines Corporation

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

D.10 Sample Program

Here is a sample global/local program.

D.10.1 Global Program

Here is the global program:

```
#include <csshape.h>

/* A "fizzical" shape is one that has been whacked on
   to obtain actual distribution across the machine,
   while still respecting the padding rules which currently
   apply.  As such, it is two dimensional, with the first index
   giving the actual PN coordinate, and the second index varying
   over the VUs and VU elements on that machine (which is to say,
   it has size 32, with 0-7 on the first VU, 8-15 on the second,
   16-23 on the third, and 24-31 on the fourth.) */

shape fizzical;

void allocate_fizzical() {
  unsigned long int extents[2];
  int pmasks[2];
  CMC_axis_order_t axis_orderings[2];
  unsigned long int weights[2];
  int subgrid_lengths[2];
  int subgrid_sequence[2];

  unsigned long int npe = positionsof(physical);

  /* NPE is the number of VUs, not the number of uniprocessor CPUs. */

  extents[0] = npe/4; /* One per PN. */
  extents[1] = 32;    /* VUs-per-PN(4) times minumum padded size(8). */
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
  subgrid_lengths[0] = 1; /* Request 1-by-8 rectangles, per VU. */
  subgrid_lengths[1] = 8;

  pmasks[0] = npe - 4; /* Address PNs. */
  pmasks[1] = 3;       /* Address VUs. */

  fizzical = allocate_detailed_shape(
                        & fizzical,
                        2,
                        extents,
                        0, /* weights */
                        0, /* axis_orderings */
                        pmasks,
                        subgrid_lengths,
                        0); /* subgrid sequence */

}

report_fizzical(shape S, int:fizzical * x) {
  int i, j;

  printf("For %d by %d by %d global shape, local information is:\n",
     dimof(S,0), dimof(S,1), dimof(S,2));

  printf("node #  ldim 0  ldim 1  ldim 2  lposns  self #  orig 0  orig 1
orig 2  nhbr 0  nbhr 1  nbhr 2\n");

  for (i = 0; i < dimof(fizzical,0); i++)
    printf("%6d  %6d  %6d  %6d  %6d  %6d  %6d  %6d  %6d  %6d  %6d
%6d\n",

i,[i][0]*x,[i][1]*x,[i][2]*x,[i][3]*x,[i][4]*x,[i][10]*x,[i][11]*x,[i][12
]*x,[i][20]*x,[i][21]*x,[i][22]*x);

  printf("\n");
}

shape [17][17][17]S;
shape [16][16][16]T;
shape [15][15][15]U;

extern void gll(int:void * a, int:void * b, unsigned long int npe);

test() {
  int:current a;
  int:fizzical b;

  a = 0;
  with (fizzical) everywhere b = 0;
  gll(&a, &b, positionsof(physical)/4);
```


Copyright © 1990–1995 Thinking Machines Corporation

```
  report_fizzical(current, &b);
}

main() {

  allocate_fizzical();
  with (S) test();
  with (T) test();
  with (U) test();
}
```

D.10.2 Local Program

Here is the local program **gll**, called by the global program above. Note that it uses some of the global/local inquiry functions.

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

extern long int CMRT_pe_limit();

void gll(int:void * a, int:void * b, unsigned long int npe) {
  int l[3], g[3], other_g[3], t, u;
  shape A = shapeof(*a);
  shape B = shapeof(*b);

  /* Record the shape of the local version of A */
  [0][0]*b = dimof(A, 0);
  [0][1]*b = dimof(A, 1);
  [0][2]*b = dimof(A, 2);
  [0][3]*b = positionsof(A);

  /* Set up a local coordinate for some local_to_global work. */
  l[0] = 0;
  l[1] = 0;
  l[2] = 0;

  /* This is an involved way to figure out our PN index.
     Note that we use shape B, which is "fizzical" on the
     global side, which guarantees that there is something
     allocated on the local side. */
  (void) CMC_local_to_global(B, l, g);
  /* Zero passed for output parameter -- we only want the index. */
  u = CMC_global_to_physical_index(B, g, 0);

  /* Do the same within the local view of shape A.
```


Copyright © 1990–1995 Thinking Machines Corporation

```
    NOTE THAT -1 can be returned (e.g., on a 32-pn machine) */
  (void) CMC_local_to_global(A, l, g);
  t = CMC_global_to_physical_index(A, g, l);

  /* Mark the allocated locations. */
  with(A) everywhere *a = t;

  /* Record result of local_to_global. */
  [0][4]*b = t;
  [0][10]*b = g[0];
  [0][11]*b = g[1];
  [0][12]*b = g[2];

  /* Exchange information in a ring */
  CMMD_send_and_receive(
              (u+1)%%npe, CMMD_DEFAULT_TAG,
              other_g, sizeof(g),
              (u-1)%%npe, CMMD_DEFAULT_TAG,
              g, sizeof(g));

  /* Record the exchanged information. */
  [0][20]*b = other_g[0];
  [0][21]*b = other_g[1];
  [0][22]*b = other_g[2];
}
```

D.10.3 Prototype File

And here is the prototype file for the local program **`gll`**:

```
gll(array a, array b, unsigned long int npe):host_cs:node_cs;
```


Copyright © 1990–1995 Thinking Machines Corporation