7 Developing iWarp Programs

This chapter provides an introduction to developing and executing programs for the iWarp system. Because parallel programs are highly application specific, it is impossible to provide a cookbook recipe for parallel program development. This chapter simply provides some basic examples to illustrate the fundamentals of program development and execution. These examples include the following:

- a single-cell "Hello World" program
- a four-cell "Hello World" program
- a single-cell program using PathLib communication
- a matrix multiplication program using PathLib communication
- a three-cell send and receive program using PCS communication
- a three-cell send and receive program including the cell on the Sun Interface Board (SIB) and using PCS communication
- a two-cell send and receive program using spooling and PCS communication
- an array program for a multiple-cell program that combines existing array modules using PCS communication with auto-placement and auto-routing

A Single-Cell Example: "Hello World"

The following example is a simple single-cell example that illustrates the steps involved in developing and executing a program for the iWarp system. The example program is the following "Hello World" program:

```
#include <stdio.h>

main()
{
  printf("hello world\n");
}
```

Assuming the program is stored in the file hello.c, use the iwcc command to compile the program and store the output in the file hello.out. Chapter 2 provides a complete description of the iwcc command.

```
iwcc -o hello  hello.c
```

You now have an image in the file *hello* that is ready to load and execute. Now you must locate an array for program execution. Each array is attached to a SIB, and array names have the following form:

```
/dev/isibn
```

where *n* is the array number. Configuration information about each array is shown in the configuration files located in directory /*rootdir*/etc*x.x*/*cfg*, where *rootdir* is the root directory pathname where the iWarp software is located on your system, and *x.x* corresponds to the software release number, such as 3.0. This directory contains text files that describe the configuration of each iWarp array on the system. The following is an example of a typical directory layout:

```
% ls /iwarp/etc3.0/cfg

     bluejay.0  stargazer.0.motd
     bluejay.0.motd stargazer.1
     stargazer.0  stargazer.1.motd
```

In this example, there are two iWarp arrays on the host named ‘‘stargazer”, and they are attached to /dev/isib0 and /dev/isib1. There is also one array on the host named ‘‘bluejay” attached to /dev/isib0. The .motd files in the directory provide the ‘‘message” of the day for the given system. For example, if we look at stargazer.0.motd, we might see the following:

```
% cat /iwarp/etc/cfg/stargazer.0.motd

     System is up and reliable
```

You can also use the cat command to view the configuration file.

```
% cat /iwarp/etc/cfg/stargazer.0
```

The configuration file contains a great deal of information about the array. At the end of the file, there is a diagram that shows how the array is set up. The following is an example of an array configuration diagram:

```
array z[2,2] = (  | |),
           (s a-b),
           (  | |),
           (- c-c);
```

This diagram shows a four-cell array set up as a 2 x 2 rectangular array. In the diagram, the s indicates the SIB cell. The other letters used for cell designations correspond to attribute sets that are described in the configuration file. The letter c corresponds to the default set of cell attributes.

We can now use the lgo command to execute our program. The following command executes the program on cell 0:

```
lgo -r hello 0
```

Chapter 3 provides a complete description of the lgo command.

“Hello World” on Four Cells

The ‘‘hello world” program for four cells would be the following:

```
#include <stdio.h>
#include <iwsys/getcfg.h>

main()
{
  struct iwcfg  cfg;

  getcfg(&cfg);
  printf("Hello world from cell %d\n", cfg.cellid);
  exit(0);
}
```

In this program we've added a call to exit, a structure called "cfg," and a call to function getcfg. The program also includes a header file, found in /iwarp/include/iwsys/getcfg.h, to describe the structure. The call to getcfg fills in the data structure, providing array information to the program. For this program, we only use the number of the cell on which the program is running, which is stored in cfg.cellid. The following commands compile and execute the program on four cells:

```
% iwcc -o hello hello.c -lrts
% lgo -r hello 0..3
```

The following commands show the program output:

```
% ls stdout*
    stdout0000  stdout0001  stdout0002  stdout0003
% cat stdout0000
    Hello world from cell 0
% cat stdout0001
    Hello world from cell 1
% cat stdout0002
    Hello world from cell 2
% cat stdout0003
    Hello world from cell 3
```

A Single-Cell Program Using PathLib Communication

The examples shown so far have illustrated the mechanics of executing programs on an iWarp array, but they have not discussed the most challenging aspect of parallel programming on the iWarp system: program communication between cells. Chapter 4 provided an introduction to the PathLib library of communication functions. This library provides functions to implement program-based communication in programs developed for the iWarp system. Appendix B also provides descriptions of the PathLib functions.

This section provides an example of a single-cell program that uses PathLib functions to communicate with itself. The example uses the include file, /iwarp/include/pathlib/pl.h to allow access to PathLib functions. In this example we are assuming a 2 x 2 array, configured as shown in Figure 7-1.

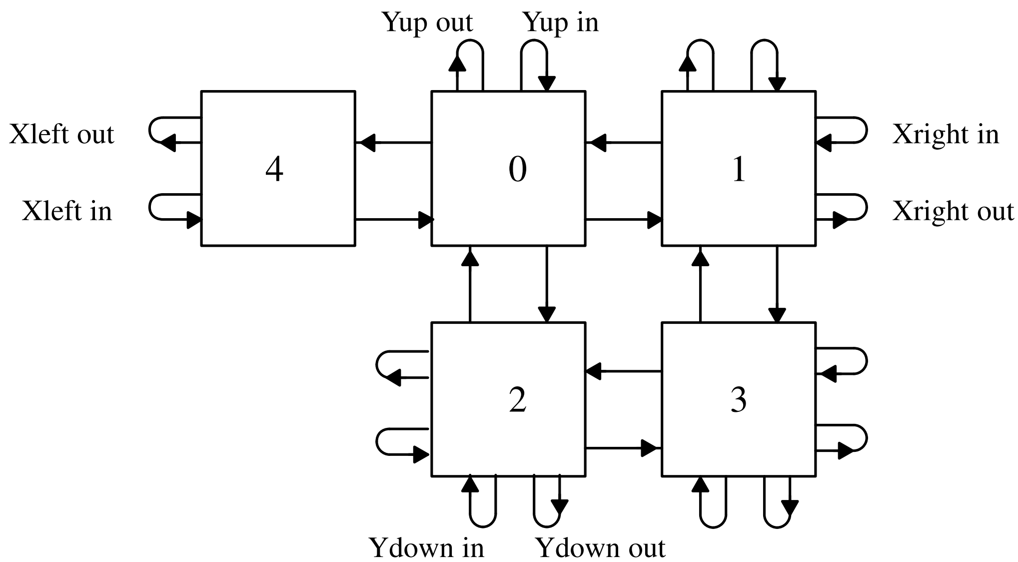

Figure 7-1: 2 x 2 Array Configuration

Note that cell 4 in the configuration is the interface cell, and cells 0 through 3 are the array cells. In general, with an N-cell system, the array cells are numbered 0 through N–1, and the interface cell is numbered N.

For this example, the program is loaded and run on cell 0. Cell 0 sends out a channel to the right, and the channel returns to cell 0 on the left after passing through cell 1 and cell 4. The following is the example program.

```
/* The channel is sent out to the right and is returned to this cell
 * on the left after traveling around the entire path of cells that
 * make up this cell's row in the iWarp array. */

#include <stdio.h>
#include <pathlib/pl.h>
#include <iwsys/getcfg.h>

main()
{
  int      route_len;
/* number of words in routing info */
  pl_pcti    i_chan;
/* logical channel descriptor
*/
  pl_pcti    o_chan;
/* logical channel descriptor
*/
  unsigned   dest_id[1];
/* name of the destination cell
*/
  unsigned   header;
/* storage for input header
*/
  int i;
```

```
int tmp;
struct iwcfg  cfg;

/* Find out which cell this is. */
getcfg(&cfg);

/* Initialize the PL module. Use two PCTEs (one in, one out). */
pl_init(0x3);

/* Reserve one pcte in the cell to the
      right and one in your own cell */
pl_rpe_configure(0, 0x1, 0, 0, 0x2);

/* Only cell 0 is going to do anything. */
if (cfg.cellid != 0)  {
  pl_exit();
  exit(0);
}

/* Create a logical channel from this cell to itself
    in the X-right direction. */
route_len = 1;
dest_id[0] = cfg.cellid|PL_DESTINATION;
o_chan = pl_send_oc(PL_GATE0, PL_DIR_XR,
                  dest_id, route_len);
pl_send_om(PL_GATE0, PL_NOMATCH);

/* Accept the logical channel just created. */
i_chan = pl_recv_oc(PL_GATE1, &header);
if (pl_recv_om(PL_GATE1, &header) != PL_SUCCESS)  {
  fprintf(stderr, "Open message not received.\n");
  exit(1);
}

/* Send and receive 1000 words over the channel. */
printf("Cellid %d: start\n", cfg.cellid);
for (i=0; i<1000; i++) {
  pl_sendi(PL_GATE0, i);
  tmp = pl_recvi(PL_GATE1);
  if (tmp != i)  {
    fprintf(stderr, "Error: sent %d, recd %d\n",
        i, tmp);
    exit(1);
  }
}
printf("cellid %d: finish\n", cfg.cellid);

/* Terminate the output channel */
pl_send_cm(PL_GATE0);
pl_send_cc(o_chan, PL_GATE0);

/* Terminate the input channel */
if (pl_recv_cm(PL_GATE1) != PL_SUCCESS)  {
  fprintf(stderr, "Close message not received.\n");
```

```
    exit(1);
  }
  if (pl_recv_cc(i_chan, PL_GATE1) != PL_SUCCESS)  {
    fprintf(stderr, "Close connection not received.\n");
    exit(1);
  }

  /* Give the pathways time to empty out. */
  pl_exit();
  exit(0);
}
```

A section by section discussion of the example is helpful to understanding how PathLib communication works.

```
/* Find out which cell this is. */
getcfg(&cfg);
```

Getcfg is a system call that, upon success, fills in the array configuration information in structure cfg. The structure is defined in include file $(ROOTDIR)/iwsys/getcfg.h. This function call is made to obtain the cell id which is now available as cfg.cellid.

```
/* Use two PCT entries (one in, one out). */
pl_init(0x3);
```

pl_init initializes the Pathlib module, indicating it needed to access 2 of the 20 PCTE's. Bit mask 0x3 represents bits 0 and 1 indicating that PCTE's 0 and 1 are needed.

```
/* Reserve one PCT entry in the cell to the right and one
      in your own cell. */
pl_rpe_configure(0, 0x1, 0, 0, 0x2);
```

pl_rpe_configure establishes the allocations of logical channel resources in neighbor cells. The first four arguments represent RPE's for the four directions Xleft, Xright, Yup, and Ydown. The last argument supplies which RPE are used for output connections. The arguments to pl_rpe_configure are bit masks representing which RPE's to use. There are a total of 20 RPEs numbered 0 through 19. Examining the arguments we see that for this example:

- Xleft uses no RPE
- Xright uses RPE 0
- Yup uses no RPE
- Ydown uses no RPE
- RPE 1 is used for output connections

Notice that RPEs 2 through 15 are not used.

```
/* Create a logical channel from this cell to
   itself in the X-right direction */
dest_id[0] = cfg.cellid|PL_DESTINATION;
o_chan = pl_send_oc(PL_GATE0, PL_DIR_XR, dest_id, route_len);
```

Pl_send_oc creates a connection bound to a stream gate routed in an initial direction with routing information. Looking at each of the arguments:

PL_GATE0 — A stream gate. There are two gates available, GATE0 and GATE1. Each gate has an input and an output side. When you write (send) to the output, the data is written out to the logical channel to which the output side of the gate is bound. When you read from the input side, data is read (receive) from the logical channel to which the input side of the gate is bound.

PL_DIR_XR — Specifies the initial direction of the connection, in this case xright. This is used to determine in which direction the first routing control word will be sent.

dest_id — An array of routing information words. Each routing control word indicates a destination cell by naming the cell and encoding the routing action to be performed when the header reaches (is matched at) the destination. In this example, we are sending to ourselves, so the first and only routing control word has cell 0 as the destination and PL_DESTINATION as the action for the connection to perform at the destination.

route_len — The number of routing information words in the array.

o_chan is an ID to a logical connection on the pathway unit; pl.h provides a data type for this, pl_pcti. It is the index of the PCT allocated for the connection.

```
pl_send_om(PL_GATE0, PL_NOMATCH);
```

pl_send_om sends a message header via the outbound connection. PL_GATE0 is the streamgate bound to an output PCT entry which is the origin of the connection. “PL_NOMATCH” tells the open message to continue until the end of the connection. Open messages can stop before reaching the end of a connection if you want them to. Normally only PL_NOMATCH will be used in the destination ID. At this point, we have created a connection to ourself and have sent a message to ourself as well.

```
/* Accept the logical channel just created */
i_chan = pl_recv_oc(PL_GATE1, &header);
if (pl_recv_om(PL_GATE1, &header) != PL_SUCCESS)  {
  fprintf(stderr, ”Open message not received.\n”);
  exit(1);
  }
```

pl_recv_oc consumes the header for the connection. Whenever an open connection is consumed, an input connection is created on the cell that does the consuming. This input connection is returned from pl_recv_oc. pl_recv_om reads in the message header that was sent. Any pathlib call that tries to read a particular type of data off the pathways first checks to see whether the desired type of data is there. If PL_SUCCESS is not returned, we know something has gone wrong, since the open message we sent was not the first thing received off the pathway. Now we are ready to send and receive data.

```
/* Send and receive 1000 words over the channel. */
printf(”Cellid %d: start\n”, cfg.cellid);
for (i=0; i<1000; i++) {
  pl_sendi(PL_GATE0, i);
  tmp = pl_recvi(PL_GATE1);
  if (tmp != i)  {
    fprintf(stderr, ”Error: sent %d, recd %d\n”, i, tmp);
    exit(1);
  }
}
```

In the for loop we send out of PL_GATE0 and receive from PL_GATE1. When the for loop completes, we must clean up the connections.

```
/* Terminate the output channel */
pl_send_cm(PL_GATE0);
pl_send_cc(o_chan, PL_GATE0);
```

pl_send_cm closes the message and pl_send_cc takes down the output connection.

```
/* Terminate the input channel */
if (pl_recv_cm(PL_GATE1) != PL_SUCCESS)  {
  fprintf(stderr, "Close message not received.\n");
  exit(1);
}
if (pl_recv_cc(i_chan, PL_GATE1) != PL_SUCCESS)  {
  fprintf(stderr, "Close connection not received.\n");
  exit(1);
}
```

pl_recv_cm consumes the message trailer and pl_recv_cc closes the input connection.

Since the output connection on cell 0 goes out the xright direction and comes in the xleft, this connection goes through cell 4 and cell 1. To ensure that the pathways are programmed correctly, cell 1 and 4 also need to cooperate. For this to happen, this same program will be loaded and run on cells 0, 1 and 4 of the array. You will notice in the code that all the cells initialize the pathway library (PL_INIT) and configure RPI's (pl_rpe_configure). If their cell id is not 0, they call pl_exit() and exit(). pl_exit() should always be called by a pathlib program before exit(). Many pathlib calls will not actually finish until after the routine returns. pl_exit() waits until all pathlib calls have finished.

In our example, all cells must call pl_init() and pl_rpe_configure() before cell 0 can start executing. This is because a connection header will stop dead at a cell if it has not yet had its RPE's configured.

Now we must compile the program. Notice that we are linking with the pathway library (–lpl).

```
% iwcc –o prog1 prog1.c –lpl –lrts
```

Next, we load and execute prog1 using lgo. As explained above, this program needs to execute on cells 0, 1, and 4 on a four cell array. However, you will notice that we load and execute it on the entire array, including the interface cell (cell 4), because it avoids selectively choosing cells where the program will execute.

```
% lgo –r prog1 0..4
% cat stdout0000
    cellid 0: start
    cellid 0: finish
```

This example runs the program on a four-cell array. To run the program on a different size array, change the range of the array size. For example, to run the code on a ten-cell you enter the following:

```
% lgo –r prog1 0..10
```

Matrix Multiplication Using PathLib Communication

The following example solves a matrix multiplication on a 2 x 2 iWarp array. The matrices are 4 x 4. Figure 7-2 shows the contents of matrix A and B, and the desired contents of the resulting matrix C.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \\ 26 & 26 & 26 & 26 \\ 42 & 42 & 42 & 42 \\ 58 & 58 & 58 & 58 \end{bmatrix}$$

Matrix A Matrix B Matrix C

Figure 7-2: Contents of Matrices for Matrix Multiplication Example

Our strategy is to distribute one row of matrix A to each cell in the array. Each cell can then compute a row of the resulting matrix. When each cell finishes computing, the resulting rows are collected. Figure 7-3 shows our four-cell configuration.

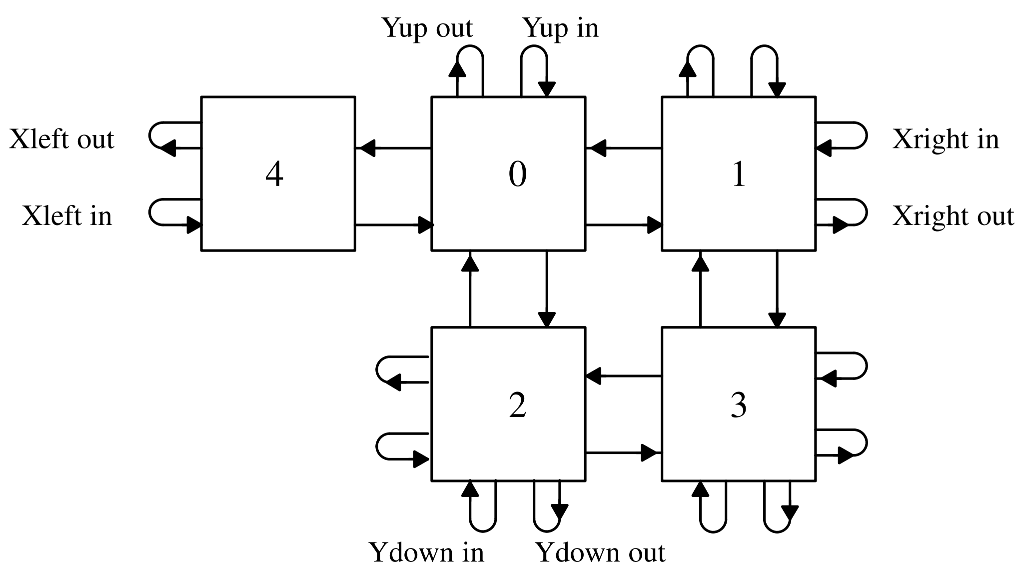

Figure 7-3: 2 x 2 Array Configuration

We will collect the result matrix on cell 0, using the following paths to return the results from the other cells:

- cell 1: Xleft to cell 0
- cell 2: Yup to cell 1, Xleft to cell 0
- cell 2: Xright to cell 3, Yup to cell 1, Xleft to cell 0

The following is the program.

```
/*  mat.c  */

#include <stdio.h>
#include <iwsys/getcfg.h>
#include <pathlib/pl.h>
int     mata[4][4] = {
        1,  2,  3,  4,
        5,  6,  7,  8,
        9, 10, 11, 12,
       13, 14, 15, 16,
};
int     matb[4][4] = {
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1,
};
int     matc[4][4] = {
        0, 0, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0,
};
void   do_row();
void   collect_results();
void   send_results();
main()
{
  int i;
  int j;
  struct iwcfg  cfg;
  int  x, y;  /* Position within the array. */
  /* Find out which cell this is. */
  getcfg(&cfg);

  if ((cfg.height < 2) || (cfg.width < 2))  {
    fprintf(stderr, "Array must be at least 2x2.\n");
    exit(1);
  }
  x = cfg.cellid % cfg.width;
  y = cfg.cellid / cfg.width;
  if ((x > 1) || (y > 1))  /* Only a 2x2 array is needed. */
    exit(0);
   /* Initialize pathlib. */
  /* Reserve four PCTEs. */
  pl_init(0xF);
  /* Set up PCTs for neighbors */
```

```
  /* Reserve one PCT in the cell to the left, one in the
         cell to the right, one in the cell above, and one
         in this cell. */
  pl_rpe_configure(0x1, 0x2, 0x4, 0x0, 0x8);

  /* Carry out mata[cellid][*] * matb */
  do_row(x + y*2);

  if ((x == 0) && (y == 0))  {
    collect_results();
    printf("result of mat a * mat b\n\n");
    for (i=0; i<4; i++) {
      for (j=0; j<4; j++) {
        printf("%2d ", matc[i][j]);
      }
      printf("\n");
    }
    printf("\n");
  } else {
      send_results(x, y);
  }

  /* Wait for the pathways to clear out. */
  pl_exit();
  exit(0);
}

void  do_row(row)
  int row;
{
  int col;
  int i;
  int sum;

  for (col = 0; col < 4; col++) {
    sum = 0;
    for (i = 0; i < 4; i++) {
      sum += mata[row][i] * matb[i][col];
    }
    matc[row][col] = sum;
  }
}

void  collect_results()
{
  static char  errmsg[] = "%s expected but not received.\n";
  pl_pcti  i_chan;
  unsigned  header;
  int  fromid;
  int  i;
  int  j;

  for (i = 1; i < 4; i++) {
```

```
    /* Receive the new connection. */
    i_chan = pl_recv_oc(PL_GATE1, &header);
    if (pl_recv_om(PL_GATE1, &header) != PL_SUCCESS)  {
      fprintf(stderr, errmsg, "Open message");
      exit(1);
    }

    /* First data word is the cellid of the sender. */
    fromid = pl_recvi(PL_GATE1);

    /* Receive a row of the data. */
    for (j = 0; j < 4; j++)  {
      matc[fromid][j] = pl_recvi(PL_GATE1);
    }

    /* Close the input connection. */
    if (pl_recv_cm(PL_GATE1) != PL_SUCCESS)  {
      fprintf(stderr, errmsg, "Close message");
      exit(1);
    }

    if (pl_recv_cc(i_chan, PL_GATE1) != PL_SUCCESS)  {
      fprintf(stderr, errmsg, "Close connection");
      exit(1);
    }
  }
}

/* cell 1: xl to cell 0 */
/* cell 3: yu to cell 1, xl to cell 0 */
/* cell 2: xr to cell 3, yu to cell 1, xl to cell 0 */

void  send_results(x, y)
  int  x, y;
{
  pl_pcti  o_chan;
  unsigned  headers[2];
  unsigned  header_count;
  unsigned  start_dir;
  int  i;
  int  rownum = x + y*2;

  if ((x == 1) && (y == 0))  {
        start_dir = PL_DIR_XL;
        headers[0] = 0 | PL_DESTINATION;
        header_count = 1;

  } else if ((x == 1) && (y == 1))  {
        start_dir = PL_DIR_YU;
        headers[0] = 1 | PL_TURN_XL;
        headers[1] = 0 | PL_DESTINATION;
        header_count = 2;

  } else {  /* x == 0 && y == 1 */
        start_dir = PL_DIR_YU;
        headers[0] = 0 | PL_DESTINATION;
```

```
        header_count = 1;
    }

    o_chan = pl_send_oc(PL_GATE0, start_dir, headers, header_count);

    /* Send an open message to the end of the connection. */
    pl_send_om(PL_GATE0, PL_NOMATCH);

    /* Let receiver know what cell data is coming from. */
    pl_sendi(PL_GATE0, rownum);

    /* Send result row. */
    for (i = 0; i < 4; i++)  {
      pl_sendi(PL_GATE0,matc[rownum][i]);
    }

    /* Close the output connection. */
    pl_send_cm(PL_GATE0);
    pl_send_cc(o_chan, PL_GATE0);
}
```

As with our previous example, we compile and execute the program as follows:

```
% iwcc -o mat mat.c -lpl -lrts
% lgo -r mat 0..4
% cat stdout0000
    result of mat a * mat b
      10 10 10 10
      26 26 26 26
      42 42 42 42
      58 58 58 58
```

Three-Cell PCS Example

This example shows a simple send and receive application that runs on a three-cell network. The application consists of a cell program that runs on each cell of the network and an array program that sets up the cell network. A makefile is provided to show the build process for the application.

Figure 7-4 shows the network configuration of the three-cell application.

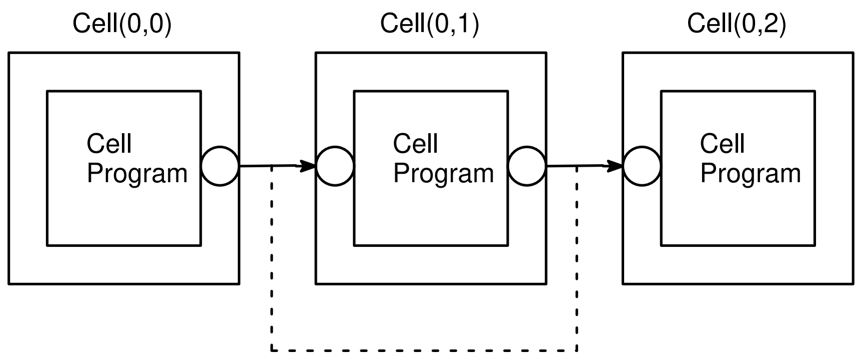

Figure 7-4: Three-Cell Application

Figure 7-4 shows a single network consisting of three cells in an array: cell(0,0), cell(0,1), and cell(0,2). This kind of network is called a multi-drop network. A multi-drop network is a single network that is accessed by multiple cells. In this example, the initial network is created between cell(0,0) and cell(0,2). The network passes through cell(0,1), which the dashed line indicates. After creating the initial network, cell(0,1) taps into the network allowing it to communicate with the other cells in the network. Cell(0,0) can send data directly to cell(0,2) without interrupting operations on cell(0,1). But because cell(0,1) shares the network, it cannot send data while a message is passing through it. The PCS calls manage this waiting operation for you.

Array Program

An array program defines the communication networks for an array module and assigns cell programs to cells. The example array program in this section does the following:

- creates the ports on cell(0,0) and cell(0,2)
- creates a cell network between these ports
- creates a port on cell(0,1) and connects the port to the existing network between cell(0,0) and cell(0,2).
- places cell programs on cell(0,0), cell(0,1), and cell(0,2)

At the beginning of your array program, specify the include files you need to access the PCS combiner library. To access the data movement library, include the *pcs/cb_def.h* file. To access the C standard I/O library, include the *stdio.h* file.

```
#include <pcs/cb_def.h>
#include <stdio.h>
```

You declare an array program as a function and specify the variables for the program. The name of the function is user_main.

```
user_main() {
```

You declare the structure variables for the network, the cells, and the ports.

```
    network_type *nw;
    cellid_type src, dst;
    port_type *src_port, *dst_port;
```

To configure the network between cell(0,0) and cell(0,2), specify the cell IDs for the cells using a cellid_type structure.

```
    src.type = dst.type = ARRAY;
    src.row = src.col = dst.row = 0;
    dst.col = 2;
```

The constant ARRAY sets the cell type to an array cell.

To create a network, configure the ports for the cells and create the network by connecting the ports. Cell(0,0), the source cell, has one output port. Create this port with the create_output_port call. Cell(0,2), the destination cell, has one input port. Create this port with the create_input_port call.

```
    src_port = create_output_port("Xout", &src);
    dst_port = create_input_port("Xin", &dst);
```

Create the network between the two ports using the create_network call.

```
    nw = create_network(src_port, dst_port, NULL);
```

This call creates a network between the src_port on cell(0,0) and the destination port on cell(0,0). The NULL value as the third parameter chooses default routing for the network when the array is combined.

After creating the cell network, you place the cell program on cell(0,0) and cell(0,2).

```
    place(&src, "cellprog.o");
    place(&dst, "cellprog.o");
```

To add cell(0,1) to the cell network, you create the cell's output port and input port. You can then tap into the existing network by connecting the input and output ports for this cell. After connecting cell(0,1) to the network, you place the cell program onto the cell.

```
    src.col = 1;
    src_port = create_output_port("Xout", &src);
    dst_port = create_input_port("Xin", &src);
    connect_port(src_port, nw);
    connect_port(dst_port, nw);
    place(&src, "cellprog.o");
}
```

Cell Program

In this example, the same cell program is placed on each cell of the cell network. The cell program does the following:

- initializes the communication connections
- gets the cell ID for the cell that the program is running on
- on cell(0,0) the program sends data to cell(0,1) and cell(0,2)
- on cell(0,1) the program receives data from cell(0,0) and sends data to cell(0,2)
- on cell(0,2) the program receives data from cell(0,0) and cell(0,1)
- closes the communication connections

At the beginning of your cell program, specify the include files you need to access the PCS data movement library. To access the data movement library, include the *pcs/pcs_def.h* file. To access the standard C I/O library, include the stdio.h file.

```
#include <pcs/pcs_def.h>
#include <stdio.h>
```

The following line defines the number of data items you are sending.

```
#define DATA_SIZE 16
```

To set up ports, you declare the input and output ports. The names Xout and Xin are the names for the cell program's ports. To use the ports, you set them up as a list of available ports in an array of port_list_type structures, as follows:

```
port_type *Xout, *Xin;
port_list_type ports[] = {
{ "Xout", &Xout },
{ "Xin", &Xin }};
```

The port names must be unique within a cell and the names must match port names specified in the array program.

You declare a cell program as a main program and specify the variables for the program.

```
main() {
  int i;
  cellid_type *self, dst;
  int data[DATA_SIZE];
  int data1[DATA_SIZE];
```

Before you make any PCS calls in your cell program, you use the pcs_init call to initialize your cell program. This call configures the logical channels and ports for the cell program, and opens the program's communication connections.

```
pcs_init(ports, NUM_PORTS(ports), NULL, 0);
```

The ports variable specifies the list of ports. You can use the NUM_PORTS macro to return the number of ports set for the cell program. There are no array module instances specified. The call initializes the ports named in the port list structure.

Get the cell ID for the cell on which the cell program is running by using the get_cellid call.

```
self = get_cellid();
```

This call assigns a pointer of type cellid_type to the self variable.

If the cell program is running on cell(0,0), the program initializes the data and sends the data to the two destination cells.

```
if (self->col == 0) { /* This code runs on cell(0,0) */
  for (i = 0; i < DATA_SIZE; i++)
  data[i] = i;
```

The program sends data using systolic data transfer. To do this, bind the output port Xout to the gate GATE0 as a systolic gate, using the bind_systolic_gate call.

```
  bind_systolic_gate(Xout, GATE0);
```

To send the data, you specify the following:

- the port and gate you are sending the data through
- the address and size in bytes of the data you are sending
- the destination cell in the form of a cellid_type structure

To send the data, you use the send_msg call. Before sending the data, set the cell's type and array coordinates. Setting the cell's type to ARRAY indicates that the cell is an array cell. The destination cell is set to cell(0,1).

```
  dst.type = ARRAY;
  dst.row = 0;  dst.col = 1;
  send_msg(Xout, GATE0, (char *)data,
          DATA_SIZE*sizeof(int), &dst);
```

To send the data to cell(0,2), use the same gate and port, but specify the new destination cell.

```
  dst.col = 2;
  send_msg(Xout, GATE0, (char *)data,
          DATA_SIZE*sizeof(int), &dst);
  printf("Send messages\n");
}
```

If the cell program is running on cell(0,1), the program receives data from cell(0,0) and sends the data to cell(0,2).

```
else if (self->col == 1) {  /* This code runs on cell(0,1) */
```

The program receives data using systolic data transfer. Bind the input port Xin to the gate GATE0 as a systolic gate, using the bind_systolic_gate call. Receive the data using the receive_msg call.

```
  bind_systolic_gate(Xin, GATE0);
  receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
  printf("Received message\n");
```

The program sends data using systolic data transfer. Bind the output port Xout to the gate GATE0 as a systolic gate, using the bind_systolic_gate call. Send the data using the send_msg call.

```
  bind_systolic_gate(Xout, GATE0);
  dst.type = ARRAY;
  dst.row = 0;    dst.col = 2;
  send_msg(Xout, GATE0, (char *)data,
          DATA_SIZE*sizeof(int), &dst);
  printf("Send message\n");
  }
```

If the cell program is running on cell(0,2), the program receives data from cell(0,0) and cell(0,1). The program receives data using the systolic data transfer. The order in which the data arrives is not guaranteed The data from cell(0,1) may arrive before the data from cell(0,0).

To receive data through a systolic port, bind the input port Xin to the gate GATE0 as a systolic port, using the bind_systolic_gate call. Receive the data using the receive_msg call.

```
    else { /* This code runs on cell(0,2). */
      bind_systolic_gate(Xin, GATE0);
      receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
      receive_msg(Xin, GATE0, (char *)data1, DATA_SIZE*sizeof(int));
      printf("Received messages\n");
      }
```

To close the program, you use the pcs_de_init call. This call closes all opened connections for the program. You should not use any PCS calls after this call.

```
    pcs_de_init();
    exit(0);
  }
```

Building the Application

You use the PCS tools as follows to build PCS applications:

- use the PCS preprocessor to preprocess cell programs (this is an optional step)
- use the iWarp C compiler to compile cell programs
- use the PCS array combiner to create the array module for your application
- use the PCS array loader to load and run the array module

Preprocessing your cell program optimizes the source code before you compile it. To preprocess the example cell program, enter the following:

```
% iwpp cellprog.c > cellprog.pp.i
```

The preprocessor directs output to standard output, so redirect the output to a new source file.

You compile the preprocessed cell program with the iWarp C compiler. Enter the following command to compile these files:

```
% iwcc -O -c cellprog.pp.i
```

The -O option requests the compiler to generate optimized code. The `-c` option requests that the compiler produce an object file, but suppress linking. Linking and loading is done later by the PCS array loader.

The name of the array program in this example is arrayprog.c. You create the array module by submitting the array program to the PCS array combiner. The array combiner assumes the cell programs exist and are compiled to object code.

Enter the following command line to create the array module:

```
% iwcb -row 1 -col 3 -o three_cell arrayprog.c
```

The -row option requests that the array module run on an array grid with one row. The -col option requests that the array module run on an array grid with 3 columns. The -o option requests that the array module be named three_cell. The array program name is arrayprog.c.

Use the PCS array loader to load the array module. Enter the following command line to load the array module without running it:

```
% iwamld three_cell -H stargazer -S 0 -n
```

The array module name is three_cell. The `-H` option specifies loading the array module onto the iWarp host system named stargazer. The `-S options specifies executing the array module onto the iWarp system attached to SIB 0.` The `-n` option requests not to run the array module.

This generates a run script named go. To execute the array module, enter the following:

```
% go
```

Makefile

You can use a makefile to build your application. This makefile is set up to make sure the cell program is preprocessed and compiled before the PCS array combiner is called. You can use the makefile to skip the preprocessing step. The PCS array loader loads the array module on the iWarp array that is attached to SIB 0 on the stargazer host.

```
.SUFFIXES: .i
CC= iwcc

all: three_cell
     iwamld three_cell -H stargazer -S 0

three_cell:   arrayprog.c cellprog.o
     iwcb -row 1 -col 3 -o three_cell arrayprog.c

# The next two targets run preprocessing
cellprog.i:   cellprog.c
      iwpp cellprog.c > cellprog.i

cellprog.o:  cellprog.i
      $(CC) -O -c cellprog.i
```

To have the makefile perform all the above functions, enter the following command:

```
% make all
```

The make command automatically builds your PCS application.

SIB Example

This PCS example shows a send and receive application that uses the SIB cell. The SIB cell is the first auxiliary cell of the iWarp array. The cell program on the SIB cell sends data to cell(0,0), and cell(0,0) passes this data to cell (0,1). Figure 7-5 shows the network configuration of the three-cell application using the SIB cell.

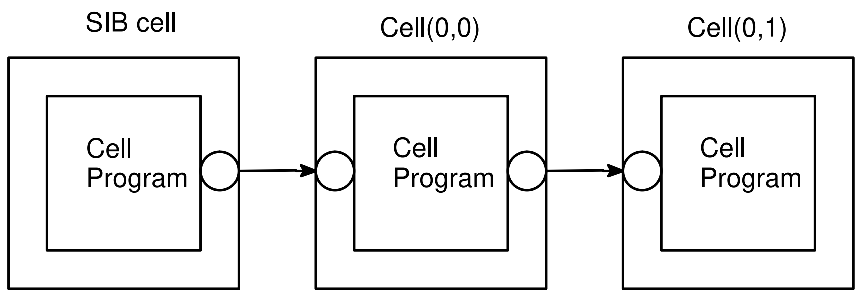

Figure 7-5: Three-Cell Application with an SIB

Figure 7-5 shows three cells with two point-to-point networks. One network is between the SIB cell and cell(0,0). The second network is between cell(0,0) and cell(0,1). The SIB cell sends data to cell(0,0). The cell program on cell(0,0) receives the data and sends it to cell(0,1). The cell program on cell(0,1) receives that data.

Array Program

The array program does the following to create a cell network using the SIB and two cells:

- creates the output port for cell(0,0) and the input port for cell(0,1)
- creates the network between cell(0,0) and cell(0,1)
- places the cell programs on these cells
- creates the output port for the SIB cell and the input port for cell(0,0)
- creates the network between the SIB cell to cell(0,0)
- places the cell program on the SIB cell

```
#include <pcs/cb_def.h>
#include <stdio.h>

user_main()
{
  network_type *nw;
  cellid_type src, dst;
  port_type *src_port, *dst_port;

  /* Set up the ports for cell(0,0) and cell(0,1). */
  src.type = dst.type = ARRAY;
  src.row = src.col = dst.row = 0;
  dst.col = 1;
```

```
  src_port = create_output_port("Xout", &src);
  dst_port = create_input_port("Xin", &dst);

  /* Create the cell network. */
  nw = create_network(src_port, dst_port, NULL);

  place(&src, "cellprog.o");
  place(&dst, "cellprog.o");

  /* Set up the SIB cell. */
  dst = src;
  /* The SIB cell is the first auxiliary cell, cell(0,0). */
  src.type = AUX;
  src.col = -1;
  src_port = create_output_port("Xout", &src);
  dst_port = create_input_port("Xin", &dst);

  /* Create the network between the SIB cell and cell(0,0). */
  nw = create_network(src_port, dst_port, NULL);

  place(&src, "cellprog.o");
}
```

Cell Program

For this example, the same cell program runs on each cell of the cell network. The cell program does the following:

- initializes the communication connections
- gets the cell ID for the cell the program is running on
- on the SIB cell, sends data to cell(0,0)
- on cell(0,0), receives data from the SIB and sends data to cell(0,1)
- on cell(0,1), receives data from cell(0,0)
- closes the communication connections

```
#include <pcs/pcs_def.h>
#include <stdio.h>

#define DATA_SIZE 16

port_type *Xout, *Xin;

/* List of requested ports */
port_list_type ports[] = {
{ "Xout", &Xout },
{ "Xin", &Xin }};

main()
{
```

```
  int i;
  cellid_type *self;
  int data[DATA_SIZE];

  /* Initialize the cell program and get the cell ID. */
  pcs_init(ports, NUM_PORTS(ports), NULL, 0);

  self = get_cellid();

  /* If the SIB cell, bind output gate and send data. */
  if (auxiliary(*self))
  {
    for (i = 0; i < DATA_SIZE; i++)
      data[i] = i;
    bind_systolic_gate(Xout, GATE0);

    /* Send data to cell (0,0). */
    send_msg(Xout, GATE0, (char *)data,
                DATA_SIZE*sizeof(int), NULL);
    printf("Send message\n");
  }
  /* If cell(0,0), receive data and send it to cell(0,1). */
  else if (self->col == 0)
  {
    bind_systolic_gate(Xin, GATE0);
    /* Receive data from SIB */
    receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));

    bind_systolic_gate(Xout, GATE0);
    send_msg(Xout, GATE0, (char *)data,
                DATA_SIZE*sizeof(int), NULL);
    printf("Send message\n");
  }
  /* If cell(0,1), receive data. */
  else
  {
    bind_systolic_gate(Xin, GATE0);
    receive_msg(Xin, GATE0, (char *)data, DATA_SIZE*sizeof(int));
    printf("Received message\n");
  }
  pcs_de_init();
  exit(0);
}
```

PCS Spooling Example

This example is a two-cell program that uses spooling to send and receive messages in the background.

NOTE

Use of spooling can cause programs that contain multiclock operations as the third word of a compute-and-access instruction to be unreliable. For a complete description of this problem and associated programming suggestions, please refer to the `spooling` manual page.

Array Program

The array program defines the communication networks for the array module and assigns the cell programs to the cells. The array program does the following:

- creates input and output ports for the cell programs
- creates the cell networks
- places cell program code on the cells

```
#include <pcs/cb_def.h>
#include <stdio.h>

user_main()
{
  network_type *nw;
  cellid_type src, dst;
  port_type *src_port, *dst_port;

  src.type = dst.type = ARRAY;
  src.row = src.col = dst.row = 0;
  dst.col = 1;

  /* Create input and output ports for cell programs. */
  src_port = create_output_port("Xout", &src);
  dst_port = create_input_port("Xin", &dst);

  /* Create the cell network. */
  nw = create_network(src_port, dst_port, NULL);

  /* Place the cell programs on the cells. */
  place(&src, "cellprog.o");
  place(&dst, "cellprog.o");
```

Cell Program

The following cell program runs on each cell of the cell network. The cell program does the following:

- initializes the cell program
- gets the cell ID for the cell that the program is running on
- for cell(0,0), the cell program sends data to cell(0,1) as follows:
  - — sets up systolic data transfer by binding the output port to a systolic gate
  - — sends an open message
  - — binds the output spool gate
  - — starts the message spool by sending the message block with a background process
  - — waits for the completion of the send operation
  - — sends a close message
  - — unbinds the output spool gate
- for cell(0,1), the cell program does the following to receive data from cell(0,0):
  - — sets up systolic data transfer by binding the input port to a systolic gate
  - — receives an open message
  - — bind an input spool gate
  - — receives the message block in a background process
  - — waits for the completion of the receive operation
  - — receives the close message
  - — unbinds the input spool gate

```
#include <stdio.h>
#include <pcs/pcs_def.h>
#define DATA_SIZE 16

port_type *Xout, *Xin;
port_list_type ports[] = {
{ ”Xout”, &Xout },
{ ”Xin”, &Xin }};
main() {

  int i;
  cellid_type *self;
  int data[DATA_SIZE];
  pl_spool spool;

  /* Initialize networks and ports. */
  pcs_init(ports, NUM_PORTS(ports), NULL, 0);

  self = get_cellid();
  /* Sending cell */
  if (self->col == 0) {
```

```
    for (i = 0; i < DATA_SIZE; i++)
      data[i] = i;

    bind_systolic_gate(Xout, GATE0);
    /* Systolically send open message. */
    send_open_msg(Xout, GATE0, NULL);

    /* Get spool gate */
    bind_spool_gate(Xout, sizeof(int), SPOOL_WORD);

    /* Start message spool. */
    send_block_bg(Xout, (char *)data, DATA_SIZE*sizeof(int));

    /* Wait for spool to finish. */
    await_bg_complete(Xout);

    /* Send close message */
    send_close_msg(Xout, GATE0);
    /* Unbind spool gate */
    unbind_spool_gate(Xout);

    printf("Send message\n");
  }
    /* Receiving cell */
  else
  {
    /* Systolically receive an open message. */
    bind_systolic_gate(Xin, GATE0);
    receive_open_msg(Xin, GATE0);

    /* Get spool gate */
    bind_spool_gate(Xin, sizeof(int), SPOOL_WORD);

    /* Start spool */
    receive_block_bg(Xin, (char *)data, DATA_SIZE*sizeof(int));

    /* Wait for spool to finish. */
    await_bg_complete(Xin);

    /* Systolically receive close message. */
    receive_close_msg(Xin, GATE0);

    /* Unbind spool gate */
    unbind_spool_gate(Xin);

    printf("Received message\n");
  }
  pcs_de_init();
  exit(0);
}
```

Automatic Placement and Routing

The example in this section shows how to do auto-placement and auto-routing of cell networks. It also shows how to combine multiple array modules to create an iWarp application.

Because you configure your cell network using the array program, this example demonstrates auto-placement and auto-routing by showing only the array program source code. You can find the complete code for this example online in the /iwarp/pcs/examples directory.

Auto-placement and auto-routing of cell networks lets the PCS array combiner map the cell network for your application to the physical cells of a target iWarp array. To configure your application for auto-placement and auto-routing, you specify a general network configuration in your array program. When you combine your application, the array combiner adds the physical configuration information. The array combiner finds an optimal configuration from the available cells on the physical array.

Array Program

The following array program uses auto-placement and auto-routing to create networks. It configures two existing array modules and one cell program to create a new array module. The first array module is a multiple-cell application that sets up a race between addition and subtraction operations. The second array module is a multiple-cell application that sets up a race between division and multiplication operations. Each module prints its own local winner and sends the name of its winner to an external single cell running its own cell program. The single cell is treated as another module. The cell program chooses a winner based on which message it receives first and prints the name of the winner.

The array program does the following:

- creates an instance of the two array modules and the single cell
- creates the input ports for the cell program
- maps the external ports for the existing array modules to port pointers in the cell program
- creates a network but does not specify the cell IDs for the network. This lets the array combiner determine the mapping for the network from each existing array module's external port to the cell program's input ports.

The array program does not specify where the cell program is placed and does not specify the network routing.

You create the array program as a user_main program and create three instance pointers: two for the array modules and one for the cell. The cell is treated as an instance.

```
#include <stdio.h>
#include <pcs/cb_def.h>
user_main() {

  instance_type*race1, /* Module for add vs. subtract */
               *race2, /* Module for multiplication vs. division */
               *judge; /* The cell program is the judge. */

  port_type  *in1,    /* Judge's input port from race1 */
             *in2,    /* Judge's input port from race2 */
             *out1,   /* Race1's external output port  */
             *out2;   /* Race2's external output port  */
```

You create the three instances using the create_instance call. The first parameter is NULL, because the program uses auto-placement; you usually specify a cell ID for a physical cell with this parameter. The second parameter is the local name of the array module. The third parameter is the name of the instance. In the following code, the two calls create array module instances. The names of the instances are “arr1” and “arr2.”

```
race1=create_instance(NULL,"race1","arr1");
race2=create_instance(NULL,"race2","arr2");
```

When creating an instance of a cell program, you must use “cell” as the third parameter of create_instance. In the code that follows, the third parameter, “cell”, indicates that the instance is a cell program.

```
judge=create_instance(NULL,"judge3","cell");
```

The array module must already exist. When combining array modules, use the –M option to tell the array combiner where to look for the module. By default, the array combiner looks for array modules in the /iwarp/lib directory.

The following calls create the external ports for the existing array modules. The calls to the get_external_port call make out1 and out2 point to the external port on the first and second array modules, respectively. These external ports were defined when you created the array modules. The first parameter in the get_external_port call is the name of the external port used when the array modules were created. The second parameter is the instance pointer for the array module.

```
out1=get_external_port("ext_out",race1);
out2=get_external_port("ext_out",race2);
```

You use the create_input_port call to set up the input ports on the judge cell. The first parameter defines the name of the port, and the second one tells which cell to define them on.

```
in1=create_input_port("in1",judge);
in2=create_input_port("in2",judge);
```

Use the create_network call to create the networks. Create two networks by connecting the two modules to the judge cell program. The first parameter is a pointer to the external output port for the array module. The second is a pointer to the input port. The third parameter specifies the network route. For automatic routing, the route parameter is NULL. The array combiner does the network routing. Use this method to create a network with auto-placement, when you do not know on which cells the cell programs will be placed.

```
create_network(out1,in1,NULL);
create_network(out2,in2,NULL);
```

The place call places the cell program cell.o onto the judge cell.

```
    place(judge,"cell.o");
}
```

You do not place the cell programs for the array modules, because the cell programs for the modules were placed when the array modules were created.