PCS+ Tool Chain User's Guide
Version 3.0+

including changes made at CMU through November 1993

Susan Hinrichs

November 15, 1993

Contents

1 Introduction

PCS+ is an experimental version of PCS developed at Carnegie Mellon. It is based on version 3.0 of the Programmed Communication Services Tool Chain (PCS) supported by Intel. Much of the user interface is identical, and the user is referred to the standard Intel documentation for more detailed information[3].

The major functional change between PCS 3.0 and PCS+ is the added support for network phases. Instead of supporting only one set of statically defined networks, PCS+ lets the programmer define multiple sets (or phases) of networks. At run time one phase is active at a time. This supports the assumption that a parallel program goes through different phases of communication. The basic phase switching software was developed by Thomas Warfel[6]. See Sections 2.3 and 2.4.2 for more detailed information about defining and using network phases.

The PCS+ tool chain is experimental and not officially supported by Intel. Other iWarp users are free to try it out and report any problems they find. However, understand that support will be at best sporadic.

1.1 Tool chain overview

PCS+ tools provide the programmer with direct access to iWarp's communication hardware. See [1] for details on the iWarp hardware. PCS+ can be used in C language user programs to provide systolic and memory based communication, or PCS+ and C can be used as the target language for parallel program generators as in Fx[2] and Assign[4]. The PCS+ tool chain is divided into two sets: array global and cell local. The array global tools work on *array modules*, groups of cell programs working together. These tools use the array wide view to define networks and allocate resources. The cell local tools work on cell programs. These tools only require the local view of a single cell such as a single cell compiler. **iwpp** and the PCS+ communication functions operate at the cell level, and **iwcb**, **iwmd**, and **iwamld** work at the array level. See Figure 1 for a picture of how the PCS+ tool chain fits together.

In using the iWarp tool chain, the user or program generator is responsible for creating two sets of files: the *array program* and the *cell programs*. The array program is uses the array global view to map programs on cells and define communication patterns. The array program is executed by **iwcb** at compile time. The cell programs are loaded on individual cells at run time and use only a cell local view. The same cell program can be loaded onto multiple cells.

1.2 Cell naming

User programs operate on logical cell arrays. At load time these logical cells are mapped one to one onto physical cells. For both physical and logical arrays, a cell is named by the row and column it is in. For example, a 4 by 4 physical array would be numbered as shown in Figure 2

Cells ID's are represented by the following structure:

```
typedef struct cellid_record {
  int type;
  int row;
  int col;
} cellid_type;
```

For the rest of this paper, all references to cell ID's mean logical cell ID's unless otherwise specified. For logical array cells, the `type` field is set to `ARRAY`.

In addition to regular array cells, an iWarp system may also have auxiliary cells like system interface cells (SIBs) or disk interface cells. If these cells exist, they are part of an optional, extra column. For a $n \times m$ physical array, column -1 is between columns 0 and $m-1$. To address an auxiliary cell the `type` field is set to `AUX`, and the

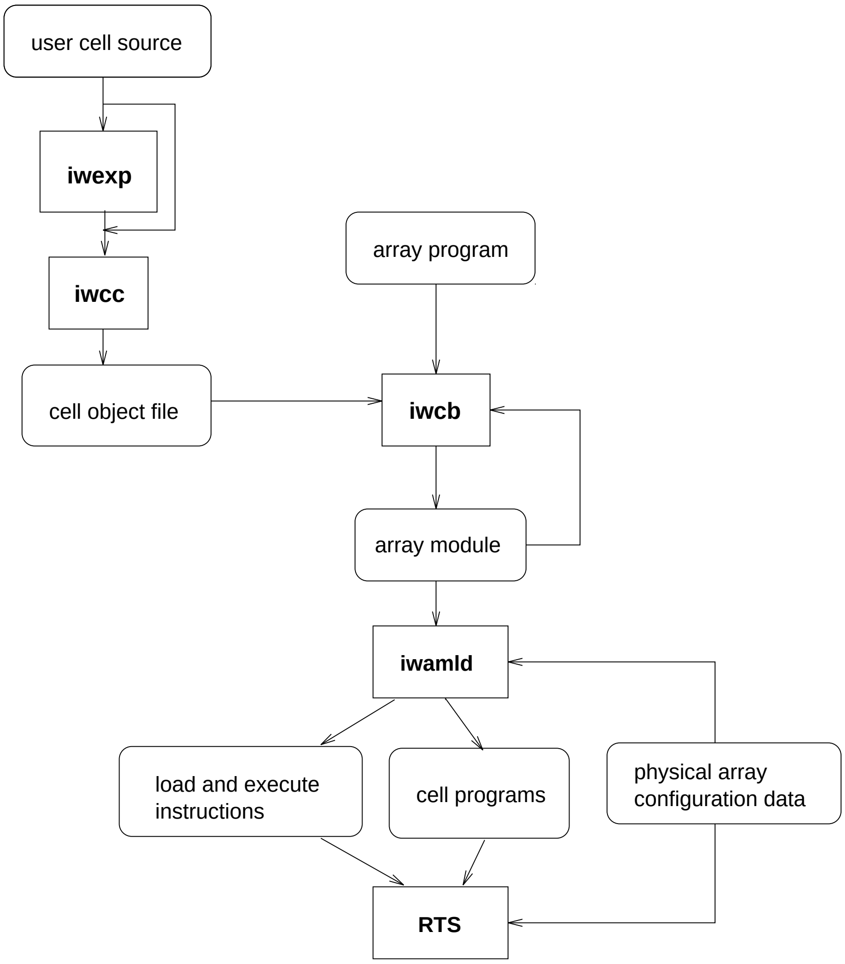

Figure 1: Flow chart of the PCS tool chain.

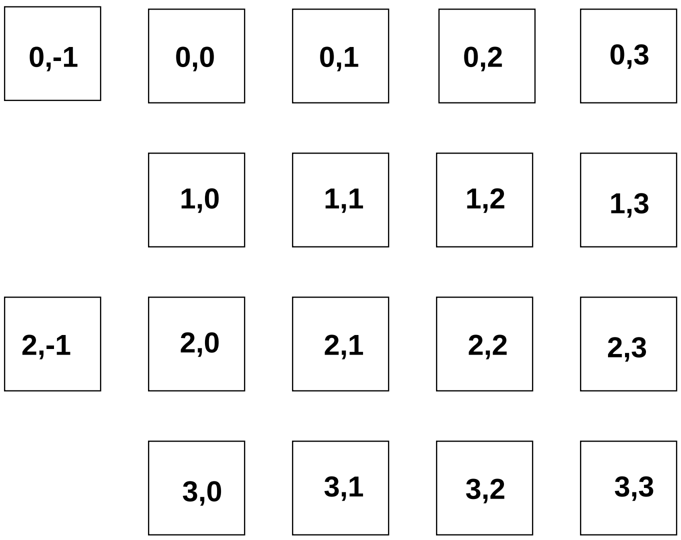

Figure 2: Example of naming in a 4 by 4 physical array with 2 auxiliary cells.

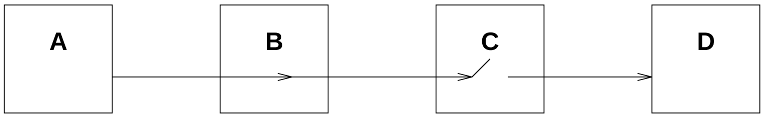

Figure 3: The end cells A and D are in local mode. Cell B is in express mode. Cell C is in local mode.

row and column fields are set to the appropriate rows and columns. The addresses and locations of two auxiliary cells are shown in Figure 2.

The location of auxiliary cells is system specific. At load time, PCS+ initialization checks to ensure that the target physical array has all auxiliary cells used by the user program. Since auxiliary cells are system dependent, array modules that explicitly reference auxiliary cells cannot be included in other array modules.

2 Defining array modules

The PCS+ array combiner **iwcb** follows an array program to combine cell programs and communication descriptions into a single array module. The user defines the mapping of cell programs and communication descriptions in the array program. The resulting array module contains all data needed to run the collection of cell programs on an iWarp array, so the array module can be executed using **iwamld** or it can be used as a building block in other array programs to create larger array modules.

2.1 Network classification

iWarp supports the creation of long-lived communication connections. Unlike the message passing communication of traditional distributed memory machines, long-lived connections can survive for multiple messages and multiple program sections. PCS+ supports the *network* as an abstraction for these long-lived connections. In the array program, the programmer can define static networks between cells. At run time the cell program accesses these communication networks through *ports*. Ports provide access to networks much the same way file descriptors provide access to files in Unix environments.

A legal PCS+ network follows a simple path through the cells. If the starting and ending nodes are the only nodes that can read from or write to the network, this is a *point-to-point* network. Additional cells along the network path can tap into the network. Cells with taps into the network can also read from or write to the network. Each link in the network is unidirectional.

A cell on a network can be in *local* or *express* mode with respect to that network. A network can be viewed as a switch at each cell. If the cell is in local mode, the switch is open, so all data on the network will stop at the cell and the cell is free to put more data on the network. If the cell is in express mode, the switch is closed, so all data will pass that cell until a express to local transition is made. Figure 3 shows one example network in various states.

Endpoints of the network are always in local mode. A tap cell is in express mode by default. It changes to local mode when a message addressed to it arrives or when a stop condition is met. Sending or receiving a close message or calling `skip_msg` will return the network to express mode.

2.2 Hierarchical array modules

There are two intuitive ways to think about combining array modules: *overlaying* and *nesting*. The difference between these models deals with the cell addressing used by cell programs within a particular module. In the overlay model, the cell programs in the included array module use the cell naming of the external module. Consider placing array module B in Figure 4. Array module B is placed at cell (2,2) in a 4 by 4 array. In the overlay model, the cell program on cell (0,0) in array module B now considers itself on cell (2,2). The environment of the included array module is changed. This is useful for loading utilities like a synchronization package or for loading predefined networks.

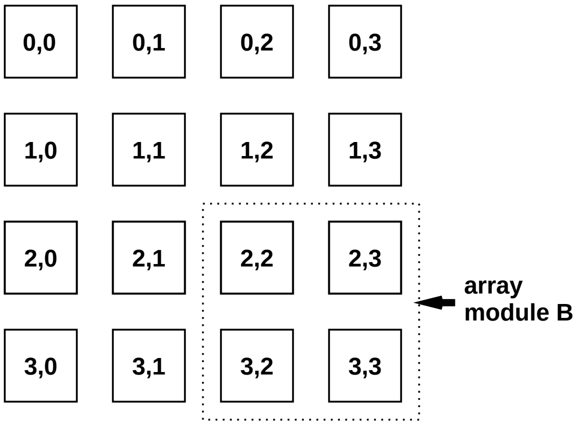

Figure 4: Array module B placed on a 4 by 4 array.

The nesting interpretation is necessary to use array modules as hierarchical building blocks. In the nesting interpretation, the cells in the included array module maintain their original cell identity; the external environment of the included array module is not disturbed. In the example in Figure 4 the cell program on (0,0) in array module B still considers itself on cell (0,0). If that cell program sends to the cell to its right, it still sends to cell (0,1). In this model the cell program on a cell uses the logical cell ID of the most deeply nested array module that encloses the cell.

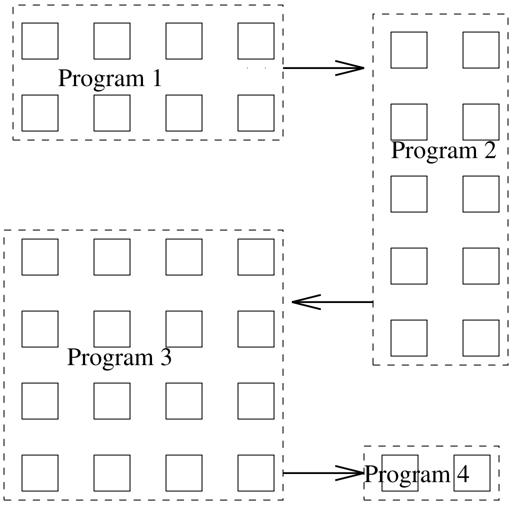

Figure 5: Four array modules pipelined together by including instances of the array modules in a 6 by 6 array module and connecting external ports.

Some sense of an "external" port is also needed to use array modules as building blocks. By writing to or reading from an external port, the cell program can communicate with a program outside its locally defined array module. By connecting external ports, one can create a pipeline of different programs. For instance, one could pipeline a matrix operation program, a signal processing program, and an image processing program. See Figure 5 for a picture of four separately defined array modules pipelined together.

This nesting interpretation is much like the hierarchical model used in computer aided design. For instance while defining an ALU, you may first define an adder. You would create a module that defined the adder and define a set of inputs and outputs for the module. Later in your design, you could treat the adder as a black box by properly using the inputs and outputs. Similarly in the nesting model, you can define an array module that performs matrix multiplication and define some input and output external ports. Later you can use the matrix multiply module in another program as a black box by properly using the input and output external ports.

In order to cleanly intergrate external ports into our network definition functions, consider the following model for defining array modules. All array modules are defined from already existing modules. Each cell is an instance of the lowest level module. The instance ID of a cell is predefined to be the row and column cell ID you would expect, so a cell ID is a special case of an instance ID. When defining a new array module, the programmer can create new instances of already defined array modules in the new array module. Each array module instance is named by an instance ID. Each instance can have ports associated with it. Regular cell ID's have local ports associated with them. Other instances can access the external ports defined in that array module.

2.3 Network phases

PCS supports the definition of *static* networks, i.e. networks the exist for the life of the program. On a machine like iWarp with a fairly small number of resources for networks, this is a severe restriction. In general, a network is not needed for the entire lifetime of the program. Instead many programs have different phases of communication requirements. For example, a program could first have a phase that needed nearest neighbor communication and then a phase that required hypercube communication connections.

PCS+ supports the abstraction of *phases*. In the array program each network can be attached to a particular phase. Then at run time only one phase is active at a time, so a network is only valid when its phase is active. If no phases are defined, the array module defaults to one phase, and all networks are defined in the context of the default phase.

2.4 Array program functions

This section gives a brief overview of the functions available to the user to define array programs. With the exception of phases, the meaning is not changed from PCS 3.0. See the man pages and standard Intel documentation for more information.

In all array programs include **pcs/cb_def.h**. This file defines the PCS+ compile time types and declares array program functions. Generally, all references to instance pointers may be instances pointers or cell ID pointers because a cell ID is a special case of an instance ID.

2.4.1 Network Functions

```
port_type *create_input_port(port_name, instance)
char *port_name;
instance_type *instance;

port_type *create_output_port(port_name, instance)
char *port_name;
instance_type *instance;
```

These functions create a port of the specified type. A pointer to the new port is returned. The returned pointer is not connected to any network or mapped to any external port. At run time, the cell corresponding to the instance pointer can retrieve this port by its name *port_name*. The cell programs use ports to access networks.

```
network_type *create_network(src_port, dst_port, route)
```

```
port_type *src_port;
port_type *dst_port;
route_type *route;
```

create_network() creates a network from port *src_port* to port *dst_port*. A pointer to the new network is returned. The new network has *src_port* and *dst_port* connected to either end. A connected port can access the network at run time. *route* is a linked list of type `route_type`. It indicates the route the network should follow. If *route* is NULL, a default route is used, that first goes along the row and then along the column.

route must include the source cell, destination cell, and all intermediate cells where the route turns. For example, to specify a route from cell (0,0) to (1,2) turning at cell (1,0), the linked list would contain the following cells: (0,0), (1,0), and (1,2). There are several functions to build up routes.

```
route_type *make_route(cell, route)
cellid_type *cell;
route_type *route;

route_type *make_directed_route(cell, direction, route)
cellid_type *cell;
int direction;
route_type *route;

route_type *add_to_route(cell, route)
cellid_type *cell;
route_type *route;

route_type *add_to_directed_route(cell, direction, route)
cellid_type *cell;
route_type *route;
```

The route construction functions come in two flavors. The *make_route* functions build up routes by adding new cell IDs to the beginning of the list. The *add_to_route* functions build up routes by adding new cell IDs to the end of the list. *make_route* and *add_to_route* calculate the direction the route should go. For grids there is no choice about the direction along a row or column to the next entry. For a torus, there is a decision about whether to use the back edge. In the torus case *make_route* and *add_to_route* pick the direction that results in the least distance between the cells. *make_directed_route* and *add_to_directed_route* let the user specify the next direction of the route.

Here are two examples of specifying a route from (0,0) to (7,7) on a 8 by 8 grid.

```
cell00.type = cell70.type = cell77.type = ARRAY;
cell00.row = 0; cell00.col = 0;
cell70.row = 7; cell70.col = 0;
cell77.row = 7; cell77.col = 0;
route = make_route(&cell00,
                        make_route(&cell70,
                                    make_route(&cell77, NULL)));

route = add_to_route(&cell00, NULL);
route = add_to_route(&cell70, route);
route = add_to_route(&cell77, route);
```

Suppose the route is specified on a 8 by 8 torus. To route across the face of the array, the direction must be specified.

```
route = make_directed_route(&cell00, DOWN,
```

```
                make_directed_route(&cell70, RIGHT,
                      make_directed_route(&cell77, SELF, NULL)));

   route = add_to_directed_route(&cell00, DOWN, NULL);
   route = add_to_directed_route(&cell70, RIGHT, route);
   route = add_to_directed_route(&cell77, SELF, route);
```

```
network_type *join_network(nw1, nw2)
network_type *nw1;
network_type *nw2;
```

join_network() creates a new network out of two already existing networks. The resources used by *nw1* and *nw2* are absorbed by the new network, so *nw1* and *nw2* are not valid networks after this call.

```
int connect_port(port, nw)
port_type *port;
network_type *nw;
```

connect_port connects the previously defined *port* to the previously created network *nw*. In order to connect *port* the network must pass through the cell *port* is created on. Once a port is connected to a network, that port can access the network at run time.

```
int alias_port(new_port, orig_port)
port_type *new_port, *orig_port;
```

This function aliases output ports. On iWarp, an output logical channel can be redirected. This function specifies that `new_port` and `orig_port` refer to the same output logical channel. At run time the `set_port(thisport)` directs that output channel as specified by port `thisport`.

2.4.2 Phase functions

```
phase_type *create_phase(phase_name)
char *phase_name;
```

This function creates a communication phase named *phase_name*. It returns a pointer to the new phase structure. At run time the cell program can retrieve the phase information by using *phase_name*.

```
add_to_phase(nw, phase)
network_type *nw;
phase_type *phase;
```

This function adds networks to the phase pointed at by *phase*. The network argument *nw* can be a network pointer returned by *create_network* or *join_network*, it can be a network list returned by *overlay*, or it can be an instance type returned by *create_instance*. In any case, *add_to_phase* adds all networks associated with *nw* to the phase *phase*.

2.4.3 Inclusion functions

```
int place(instance, prog_name)
instance_type *cell;
char *prog_name;
```

This function maps the cell program *prog_name* to *instance*. If instance is not a cell ID, it maps the program to the logical (0,0) cell in the array module instance. The code corresponding to *prog_name* is stored in the resulting array module.

```
int place_lib(prog_name)
char *prog_name;
```

This function maps the cell program *prog_name* to all possible cells. At load time, the array loader figures out which cells do not exist. This function was developed to place libraries without needing to change makefiles, although regular object files can also be placed with this function tool.

The array loader interprets the suffix of the program to determine whether it is a library. If the suffix is 'a' and it is the only program placed on the cell, the array loader assumes this is an unloaded cell and loads the default dummy program. See Section 5.1 for a discussion of dummy programs.

instance. If instance is not a cell ID, it maps the program to the logical (0,0) cell in the array module instance. The code corresponding to *prog_name* is stored in the resulting array module.

```
net_list_type *overlay(offset, module_name)
cellid_type *offset;
char *module_name;
```

This function includes the previously defined array module *module_name* at *offset* using the overlay model described in Section 2.2. This function is useful to include predefined networks and utilities. *overlay* returns a list of the networks defined in the array module. This list can be used the attach all the networks to a particular phase.

```
instance_type *create_instance(offset, instance_name, module_name)
cellid_type *offset;
char *instance_name;
char *module_name;
```

This function creates an instance of the array module named *module_name* at *offset*. The array module is included following the nesting interpretation described in Section 2.2. An instance pointer is returned that refers to the new array module instance. In the cell program this instance pointer can be retrieved by using the name *instance_name*.[1]

```
int map_external_port(port, external_port_name)
port_type *port;
char *external_port_name;
```

This function maps *port* to an external port in the current array module called *external_port_name*. *port* should not be connected to a network or mapped to another external port in the current array module. If the current array module is included in another array module *port* can be referenced by the name *external_port_name*. Only external ports defined in the outermost array module can be referenced when that array module is included in another array module. To make an included external port visible when the current array module is included elsewhere, the external port must be mapped to another external port name in the current array module.

```
port_type *get_external_port(external_port_name, instance)
char *external_port_name;
instance_type *instance;
```

[1] There is a naming conflict problem in the current version. Suppose you are including array modules `array1` and `array2`. Both `array1` and `array2` include different cell programs both named `cellprog.o`. Since both array modules are unpacked in the current working directory, one version of `cellprog.o` will overwrite the other. The solution is to unpack array modules in directories named by the array module name.

This function returns a pointer to the external port named *external_port_name* defined in the array module referenced by *instance*. The external port pointer can be used as an argument to *create_network*, *connect_port*, and *map_external_port*.

```
int set_logical_channel_count(count, phase)
int count;
phase_type *phase;
```

This function specifies how many logical channels are available for user defined networks. Suppose the operating system leaves 16 logical channels, and the cell program wants to use a message passing library that requires 5 logical channels. Then the array program should set the logical channel count to $16 - 5$.

2.5 Automatic placement and routing

If cell ID's are not specified in the array program functions, **iwcb** will automatically place and route the user defined cells and networks. Instead of using regular cell ID's, you can create instances of the special type *cell*. Then use this instance pointer as a cell ID in other functions like *port_create* that require a cell specification. Here is an example of creating and using a cell instance.

```
instance_type *cell_ptr;
port_type *port_ptr;

cell_ptr = create_instance(NULL, "cell1", "cell");
port_ptr = create_output_port("out_port", cell_ptr);
```

If no row and column cell ID's are used in the array program, **iwcb** will automatically search for a feasible cell placement that minimizes the number of networks mapped over a single physical bus. Since only one cell instance is mapped to a logical cell, the number of cell instances cannot be greater than the number of logical cells.

Due to limitations in the mapping and routing function, only point to point networks can be automatically routed, and array modules cannot be automatically routed on a logical torus (using the **-w** option in **iwcb**). Since the user does not know where the cells will end up, it is not safe to use *reserve_logical_channel* and set up run time connections with automatic placement and routing.

2.6 iwcb description

iwcb is the iWarp array combiner. It creates the array module from the user provided array program. The array program defines communication networks, includes other array modules, and maps cell programs to cells. **iwcb** compiles the array program and links it with the combiner library and any other user specified libraries. The resulting program is executed and creates an array module that can be loaded and executed by **iwamld**. The main function in the user's array program must be called *user_main()*. *user_main()* is passed two arguments, the number of rows and columns in the logical array.

```
user_main(row_num, col_num)
int row_num, col_num;
```

The combiner library **libcb.a** contains definitions of the array program functions and other supporting functions. The programmer can create extra libraries to define frequently used networks or special purpose utility functions.

The array module contains the cell program object files and a file that describes the user defined networks. Since the array module contains copies of the cell program object files, the programmer must be careful when setting up the makefile. The array module target should depend on all the cell program object files to ensure the new cell programs are ready when **iwcb** is executed. If the array module uses cell programs "cellprog.o" and "sibprog.o", the makefile rule for the array module should be:

```
array_module:     array_module.c cellprog.o sibprog.o
         iwcb -o array_module array_module.c
```

If the array program has not changed since **iwcb** was last executed, **iwcb** will not recreate the network description file but only include the newly compiled cell programs.

2.7 iwcb command line options

`iwcb [options]` *array program support file**

The options to **iwcb** are as follows:

-V Print the version number and continue.

-VV Print the version number, print usage information, and exit.

-d Enable the printing of additional debugging statements.

-row *#* Set the number of rows in the grid. Defaults to 8.

-col *#* Set the number of columns in the grid. Defaults to 8.

-w Allow routes to include the back edges of the torus. By default only the grid edges are used.

-M*dir* Add *dir* to the list of directories searched while looking for included array modules. By default **iwcb** searches the current directory and **/iwarp/misc/pcs+/lib/modules**.

-D*def* Compile the array program with *def* defined.

-I*dir* Add *dir* to the list of directories searched while looking for include files. By default **iwcb** searches **/iwarp/misc/pcs+/include**.

-l*lib* Link *lib* with array program.

-L*dir* Add *dir* to the list of directories searched while looking for libraries. By default **iwcb** searches **/iwarp/misc/pcs+/lib**.

-o *output* Set the name of the array module to *output*. Defaults to *array.out*.

array program User C program that describe the array's communication networks and cell program to cell assignments.

support file File that contains C functions that should be compiled and linked with the array program.

3 Displaying array modules

An array module consists of the cell programs and a network description file. The array module display program **iwmd** displays the mappings and networks of the array module in an X window. Additional information about the object pointed at by the mouse is displayed in an information window at the top of the screen. **iwmd** takes one argument which is the name of the array module. The phase menu option enables the user to view different communication phases defined for the array module. [2]

[2]This tool may not be available if motif libraries are not available.

4 Writing and compiling user cell programs

The networks defined in the user's array program are mapped onto the low level iWarp communication hardware. Once the cell program retrieves a port for the network, it can use any communication library that can use ports or the lower level pct's, including pathlib, the PCS+ data movement library, the ESPL spooling library, or the raw built in communication functions.

By using **iwexp** the PCS+ data movement routines can be expanded to their definitions to avoid function call overhead. The data movement library **-lpcs** is linked in by **iwamld** to define all data movement functions that are not expanded by **iwexp**. Many data movement functions are standard C preprocessor macros, so they are expanded in either case. These macros are defined in **pcs/pcs_includes.h**.

4.1 iWarp messages

Cell programs exchange data over networks in the form of *messages*. A message consists of data preceded by an open message marker and followed by a close message marker. These markers are necessary to tell the hardware that a particular network on a cell is "in" a message. Nested messages are not allowed, so a close message must be sent before another open message marker can be sent.

If a cell taps into the middle of a network, the network changes from express to local mode with the sending or receiving of an open message marker. The network returns to express mode with the sending or receiving of a close message marker. When the cell is not in a message, the network is always in express mode so messages addressed to other cells can pass through it.

Sending message markers is not essential for point to point networks, because the endpoint cells are alway in local mode, so the hardware does not need to make decisions about whether the cell is in a message.

4.2 PCS+ communication functions

All programs that use the data movement routines should include **pcs/pcs_def.h**. Most PCS+ data movement functions have not changed from PCS 3.0. See the standard Intel documentation for additional information.

4.2.1 Initialization functions

```
pcs_init(ports, num_ports, instances, num_instances)
port_list_type *ports;
int num_ports;
instance_list_type *instances;
int num_instances;

pcs_de_init();

typedef struct {
    char port_name[PORT_NAME_LEN];
    port_type **port_ptr;
} port_list_type;

typedef struct {
    char instance_name[INSTANCE_NAME_LEN];
    instance_type **instance_ptr;
} instance_list_type;
```

pcs_init() must be run before any PCS calls are made. *pcs_init()* reads initialization tables created by **iwamld** for information to configure the logical channels, initialize ports, and open input and output connections. It also initializes port pointers and instances pointers passed in the port list and the instance list.

The port list is an array of port name and port pointer address pairs. *pcs_init()* looks up the port name in the cell's port table and sets the port pointer to the corresponding port address if it is found or sets the port pointer to NULL. Similarly, the instance list is an array of instance name and instance pointer address pairs. *pcs_init()* looks up the instance name in the cell's instance table and sets the instance pointer to the corresponding instances address if it is found or sets the instance pointer to NULL. Macros NUM_PORTS and NUM_INSTANCES calculate the number of entries in the port list or instance list arrays. Port lists and instance lists can be pre-initialized as shown below.

```
port_type *in_port[2], *out_port[2];
port_list ports = {
{ "left" , &in_port[0] },
{ "up", &in_port[1] },
{ "right", &out_port[0] },
{ "down", &out_port[1] }};
```

pcs_de_init() closes all connections opened by *pcs_init()*. No PCS functions should called after *pcs_de_init()*.

For additional debugging information, the variables `Debug_init` and `Debug_de_init` may be set to TRUE before *pcs_init()* and *pcs_de_init()* are called.

4.2.2 Utilility functions

```
cellid_type *get_cellid()

void get_array_size(row, col)
int *row, *col;
```

get_cellid() returns the logical cell ID of the current cell. *get_array_size()* returns in the two arguments the number of rows and columns in the logical array. The logical cell ID and the logical array size are assigned by **iwcb** in the array module. The logical cell ID is determined by the most deeply nested array module that encloses the cell. Similarly, the size of the logical array is determined by the size of the most deeply nested array module that encloses the cell.

```
instance_type *get_instance()

instance_type *get_parent_instance(inst)
instance_type *inst;

char *get_instance_name(inst)
instance_type *inst

bool InstanceEqual(inst1, inst2)
instance_type *inst1;
instance_type *inst2;
```

get_instance() returns an instance pointer to the most deeply nested array module instance that encloses the current cell. *get_parent_instance()* returns a pointer to the instance that encloses the parameter *inst*. If *inst* has no enclosing instance, NULL is returned. *get_instance_name()* returns the name of the parameter *inst*. *InstanceEqual()* returns true if the two pointers point to the same instances.

```
port_type *get_port(port_name)
char *port_name;

int get_index(port_name)
char *port_name;

port_type *get_index_port(index)
int index;
```

get_port() returns a pointer to the port named *port_name*. *get_index* translates the port name to a unique index that is common to all cells in the array module. *get_index_port* returns the port corresponding to the port name index calculated by *get_index*.

```
bool PtEqual(cell1, cell2)
cellid_type cell1;
cellid_type cell2;

bool auxiliary(cell)
cellid_type cell;
```

PtEqual() returns TRUE if the two cell IDs are the same. *auxiliary()* returns TRUE if *cell* is of type AUX.

4.2.3 Phase switch functions

```
phase_type get_phase(phase_name)
char *phase_name;

set_phase(phase)
phase_type *phase;

phase_type wait_for_phase()
```

get_phase returns a pointer to the phase named *phase_name*. This phase should have been defined in the array program.

set_phase and *wait_for_phase* are both methods to switch phases. To switch phases within the array module, all cells must participate. *set_phase* enables a cell to actively change the phase to the phase pointed at by *phase*. All cells must change to the same phase. *wait_for_phase* passively changes phase. It waits to be notified of the new phase and returns a pointer to that phase. Obviously, at least one cell in the array must perform an active switch.[3] The passive phase switch comes at a cost. Each passive cell in a P cell system, slows down the synchronization by $1/P$.

The passive phase switch is useful for dummy cells that do not observe phases otherwise. For example such a dummy cell may run the following loop.

```
while (last_phase != wait_for_phase());
```

4.2.4 Synchronization functions

The synchronization package that supports the phase switching functions is available for general use.

[3]Depending on the synchronization package used more active cells may be required. With the synch 3, one cell in each row must be active.

```
arsync(val)
int val;

int slavesync()
```

`arsync` performs a barrier synchronization over the whole system. The argument `val` is passed between cells during the synchronization. `slavesync` waits until a synchronization token reaches the cell. Then it passes on the synchronization token and returns the value argument. During a synchronization step, each cell must either perform an active synchronization with *arsync* or a passive synchronization with *slavesync*. For the slow synchronization package, at least one cell must be "active". For the fast synchronization package, at least one cell on each row must be "active".

```
int pcs_service_pending_net_action()
```

This function enables passive cells to combine waiting for phase switches and synchronizations. This functions waits until a token arrives over the synchronization network. It determines whether this is a phase switch or a synchronization, performs the appropriate action, and returns the value. If it was a phase switch, the return value with be the phase switch id. If it was a synchronization, the return value will be the synchronization value with the bit 31 set. This function makes dummy passive cell programs very simple.

```
while (pcs_service_pending_net_action() != last_phase);
```

4.3 iwexp description

iwexp expands all occurrences of PCS calls to their definitions of pathlib calls. Before any PCS program is compiled it can be run through **iwexp**. Before it is run through **iwexp** it must be run through **iwcpp**. Generally, you can use the script **iwpp** to call **iwexp** and **iwcpp**. **iwpp** writes the expanded source file to standard output.

As mentioned before PCS calls that are not expanded by **iwpp** have their defining code linked in by **iwamld**, so the preprocessing step is not necessary. It should only be used to avoid function call overhead.

4.4 iwpp command line options

`iwpp [options]` *source file*

The options are as follows:

-V Print the version number and continue.

-VV Prints the version number, prints usage information, and exits.

-d Prevent intermediate files from being erased.

-I*include directory* Passed on to **iwcpp**, so include files in that directory will be used.

-D*define* Passed on to **iwcpp**, so the symbol *define* is defined during C preprocessing.

source file The user program to be expanded.

5 Loading and running array modules

5.1 iwamld description

iwamld takes an array module as input and does the following:

- Verifies there are sufficient resources on the target physical array to run the logical array.
- Creates cell initialization tables.
- Links cell programs with initialization tables and run time libraries.
- Either directly loads and executes the array module or creates a script to load and execute the array module on the physical array.

Additional libraries may be linked by passing -l and -L arguments to **iwamld**. These arguments are passed onto **iwld**.

If the same files are placed on two cells and the same arguments are passed to those cells, those cells will share the same cell executable. This reduces the amount of time spent linking and compiling.

If no program has been assigned to a particular cell and a network is routed through that cell, a "dummy" program is loaded on that cell. The dummy program must call *pcs_init()* to set up the logical channels so networks can be routed through the cell. By default the dummy program is **/iwarp/misc/pcs+/lib/pcs-dummy.o**. If you want to use your own dummy program, set the **PCSDUM** environment variable to that path name. The default dummy program performs no phase switches.

5.2 iwamld command line options

iwamld *array-module* `[options]`

The options are as follows:

-V Print the version number and continue.

-VV Prints the version number, prints usage information, and exit.

-d Prints additional debugging information.

-e The execute flag. In addition to checking and linking, it executes the array program on the physical array. By default **iwamld** will execute the array module. It uses **rsh** to access remote hosts if necessary.

-n The no execute flag. This flag prevents **iwamld** from directly executing the array program. Instead it creates a script call **go** to load and execute the array program on the designated machine.

-H *host_name* Specifies the target iWarp array. By default it loads the array module on the host specified by the environment variable IWARP_HOSTNAME.

-S *sib_num* Specifies the target iWarp SIB. By default it uses the SIB specified in the environment variable IWARP_SIBNUM.

-sync *count* Specifies how many logical channels the synchronization library requires. The default library requires 3. A slower synchronization library requires 2.

-pct *mask* Indicates which communication resources can be used by the networks. The default value depends on which synchronization library is used, 0xFFFF with the default synchronization library or 0x7FFF with the faster library.

-o *run script* Specifies the same of the run script. By default it is named "go".

-m *msg* Specifies a message to be displayed by **iwstat** when the program is executed.

iwcc *arguments* Any other arguments like *-l* or -L will be passed on to **iwcc** when the run time libraries are linked in.

-*row[..row],col[..col]* [,*arg* *] Defines arguments to be used with the cell program on virtual cell (*row,col*).

array_ module Produced by **iwcb**. This describes the array program to be loaded and contains the cell programs.

6 Known errors and work arounds as of November 10, 1993

The current version does not handle attaching a network to multiple phases. To be safe only attach a network to one phase. In general, the interactions of phase switching and submodules have not been worked out. Current work by Tom Warfel is expanding the model to deal with switching submodules.

There is the possibility for port name conflicts between phases. If port "foo" is defined in phase 1 and phase2 , it is not clear which will be returned from `get_port(``foo'')`. Another function like `get_phase_port()` may be necessary.

iwamld does not always detect changes in arguments that require creation of new intermediate files. For instances changing the `-sync` argument will not cause a new network description file to be create. To ensure that everything is updated for array module `foo`, execute `/bin/rm -f foo-*` before executing **iwamld**.

Auxiliary cells can be connected in the back loop of a row and a column of the array, but the current configuration file is too limited to describe connections in both dimensions. At CMU one system (stargazer) has such an auxiliary cell that is connected in the backloops for row 4 and column 3. In order to route over the column backloop, the programmer must use the `-hib_hack` argument for **iwcb** and **iwamld**.

The default dummy program performs no phase switches. If your program performs phase switchs, you must load your program dummy program to take this into account. Eventually the `pcs_pending_net_action` function will be used to create a general purpose dummy program.

7 Documentation locations

Postscript versions of CMU documentation about the iWarp can be found online in **/iwarp/doc** and **/iwarp/misc/pcs+/doc**. This document is in **/iwarp/misc/pcs+/pcs/pcs+.ps**. [5] describes how to use the iWarp at CMU. A postscript version of this paper can be found in **/iwarp/doc/local.ps**.

Intel documentation can be found online in the directory **/iwarp/docs**. Man pages can be accessed by the **iwman** command.

Examples of programs using PCS+ are in **/iwarp/misc/pcs+/examples**. There is a simple two cell example in **simple**. A simple example that addresses the SIB cell is in **sib**. Several examples that exercise multiple phases can be found in **phase**. An example that overlays and nests submodules is is **module**.

The **/iwarp** directory is available via afs as **/afs/cs/project/iwarp/ccom**.

References

[1] S. Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[2] T. Gross, D. O'Hallaron, S. Hinrichs, G. Lueh, J. Stichnoth, and J. Subhlok. Compiling Task and Data Parallel Programs for iWarp. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, 1992.

[3] Intel. *iWarp Programmer's Guide*, August 1992.

[4] D. O'Hallaron. The Assign Parallel Program Generator. In *Proceedings of the 5th Distributed Memory Computer Conference*, 1991.

[5] David O'Hallaron. *iWarp Local Guide*, 1990.

[6] T. Warfel. Supporting Array Level Subroutines. Thesis proposal.