A Compile Time Model for Composing Parallel Programs

Susan Hinrichs

April, 1994

CMU-CS-94-108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many distributed memory machines support connection-based communication instead of or in addition to connection-less message passing. Connection-based communication can be more efficient than message passing because the resources are reserved once for the connection and multiple messages can be sent over the connection. While long-lived connections enable more efficient use of the communication system in some situations, managing connection resources adds another level of complexity to programming such machines. iWarp is an example of a distributed memory machine that supports long-lived connections. To aid the iWarp programmer and program generator tools, we developed a tool chain that enables the programmer to define connections and compose parallel programs. The communication tool chain has been in use for four years in various forms. In that time, we have found many benefits and a few pitfalls in our model. This paper describes the design of the programming model and tools and discusses our experiences with this implementation.

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University.

Keywords: Parallel systems, communication architecture, parallel programming, connection-based communication, programming tools

1 Introduction

For many distributed memory systems, connection-based communication is another option to connectionless message passing communication. With traditional message passing systems, communication resources (buffers, memory, network bandwidth, etc.) are reserved for each message and released after the message has completed. This reservation time is reflected in the start up overhead for each message, so long messages are needed to amortize this cost.

With connection-based communication, the communication resources are reserved once when the connection is opened. Since the connection is long-lived, multiple messages can be sent over the connection, so the cost of resource reservation is amortized over more data. For regular, frequently used communication patterns, long-lived connections improve communication performance. However, working with long-lived connections adds complexity to programming the parallel system, because the programmer must be aware of the communication patterns. By contrast, message passing systems do not require as much communication configuration information from the programmer.

iWarp is one distributed memory system that supports long-lived connections with hardware. The Intel Paragon, Inmos transputer systems, and ATM networks are other examples of systems that support connection-based communication with varying degrees of hardware and software. These long-lived connections are called logical channels, virtual channels, or virtual connections on various architectures.

To aid programmers and program generators with the task of managing long-lived connections for iWarp, we developed the Programmed Communication Service (PCS) tool chain. The PCS tool chain provides several abstractions to aid programming in a connection-based model. PCS provides the user with the *network* abstraction that describes connections at a level higher than the hardware implementation. The tool chain takes care of the hardware level resource allocation given the user network definitions. PCS supports *array modules* to encapsulate the node programs and network definitions that make up a parallel program. This encapsulation allows array modules to be reused and composed into a larger array modules. PCS also uses the concept of *phases* of a computation. The programmer can associate network definitions with different program phases. At run time only one phase is active at a time, so the resources for inactive networks can be shared with the active networks. This allows more networks to be defined with a limited set of communication resources.

We have been using versions of the PCS tool chain since 1990. In this paper, we describe the tool chain features to date and discuss the benefits and short comings of our current implementation. Section 2 describes the connection model used by PCS. The major PCS abstractions are described in Section 3. In Section 4, we discuss the benefits and limitations of the programming model and the current implementation. Section 5 describes other communication systems and how they relate to PCS. In Section 6, we present our conclusions.

2 Connection model

In many cases, a parallel program uses well understood, static communication patterns (eg. grids, rings, or hypercubes). In such cases, it is beneficial to set up the connections once and use them many times. Using long-lived connections yields two benefits over dynamic connections. First, reusing connections reduces the start up overhead of communication, because the route does not need to be renegotiated with each message between the same pair of nodes. Second, long-lived connections ensure the order of messages sent over a connection, so these messages do not need to be buffered in memory but instead can be read directly from the communication agent[HG92]. This direct data consumption is necessary to effectively implement systolic algorithms.

PCS is designed to work with distributed memory machines that support long-lived connections. The tool chain assumes the target machine supports *direct* connections between nodes that are not necessarily adjacent. Data sent over a direct connection require no software intervention on the intermediate nodes once the connection has been initialized.

The PCS model assigns connection resources at compile time, so the programmer must write an *array program* that defines the required communication connections. By the time the run time programs are loaded, the connection resources have been pre-computed, so opening connections and negotiating resources requires minimal execution time. Safely opening long-lived connections at run time without pre-computed information can be a tricky operation.

Connection implementation differs between different systems. The current PCS implementation was developed for iWarp[B+88, B+90], so for an example of connection configuration issues, we describe iWarp's connection hardware support. iWarp supports connections with a limited number of communication queues. Data from different queues are multiplexed over the physical bus on a word by word basis. For two nodes to communicate, they build a *connection* by configuring queues on adjacent nodes to forward data to each other. The connection requires one queue on each communicating node and each intermediate node. When initializing the connection, communicating nodes must know which queues are used on their neighboring nodes to correctly initialize the flow control hardware.

Figure 1 shows connections in a system where each node has two queues. No more connections can involve node (1,1), since all of its queues are already in use. If a new long-lived connection is opened at run time, the source node must ensure that there are sufficient queues along the projected route. For example, if nodes (0,1) and (1,2) both attempt to open a connection to node (0,2) as shown by the dashed lines in Figure 1, only one connection will succeed since only one queue remains free on node (0,2). To ensure resource availability at run time, the node must communicate via a deadlock free channel, which is potentially error prone and/or expensive. The short-lived connections used in message passing avoid this problem by using deadlock avoidance techniques described in [DS87].

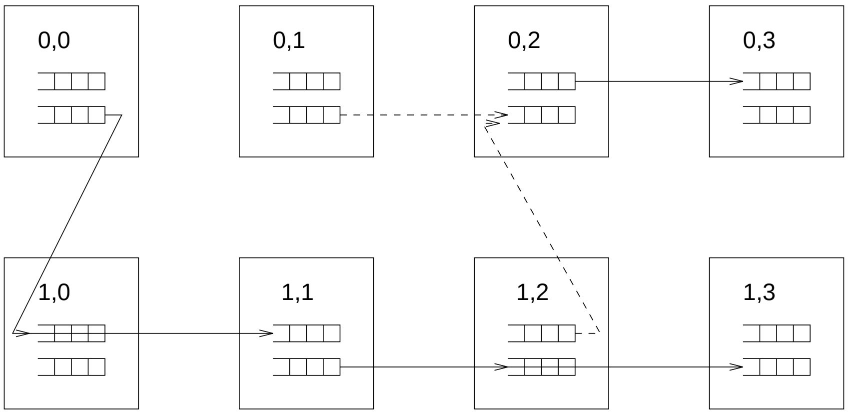


Figure 1: Example of connections and communication queue usage. The solid lines show connections. The dashed lines show attempted connections.

PCS assumes a fairly low level MIMD programming model. Each node can potentially run a unique program, though for the sake of compilation simplicity most parallel programs consist of a few node programs run on many nodes. By not strongly enforcing a programming model, PCS can be used to define communication structure for a number of different programming models.

Mapping run time programs to nodes is also defined in the array program. For run time communication

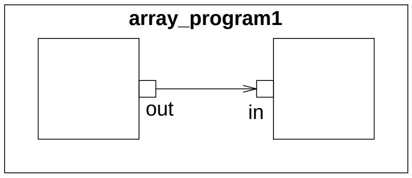
From array program

```
inport = create_port(node(0,0), "in");
outport = create_port(node(0,1), "out");
nw = create_network(outport, inport);
```

From node program

```
pcs_init();
if (self == node(0,0)) {
   port = get_port("out");
   send_msg(port, data);
}
if (self == node(0,1)) {
   port = get_port("in");
   recv_msg(port, data);
}
```

(a)



(b)

Figure 2: Part (a) shows code that defines a network at compile time and run time code that uses the network. Part (b) shows a picture of the connections defined for the resulting two node parallel program.

efficiency, the PCS model assumes that at most one process is loaded on each node. Multi-threaded node programs can be supported in this model, but without gang scheduling, fully multiprogrammed nodes require additional safety mechanisms.

The topology of the target machine is not intrinsic to the communication model. iWarp is connected in a two dimensional torus, which is a two dimensional grid with wrap around edges called *back edges*. For concreteness, the remainder of this paper assumes that the target machine is connected in a torus topology.

3 PCS abstractions

The PCS tool set supports connections and program constructions defined at compile time. This compile time construction avoids the problems of run time communication resource allocation. Here we describe the major abstractions supported in the PCS tool chain.

3.1 Network compilation

With the PCS tool chain, the programmer or program generator creates an array program in addition to the node programs that are executed on the nodes at run time. This array program defines connections between nodes and defines node program to node mappings. Figure 2 shows segments from an array program and node program and the corresponding run time communication structure.

At run time nodes communicate over connections called *networks*. To define a connection between two nodes, the array program defines *ports* on the input and output nodes and defines a network between the two ports. At run time, the node program retrieves the ports on its node and uses these ports to access the network. Therefore, port names are local to the node. If the node programs used the network name directly to access connections, the nodes would have to agree on a global network naming scheme. The example in Figure 2 defines one network. At run time node (0,0) accesses this network with the “out” port, and node (0,1) accesses this network with the “in” port.

The system supports several alternatives for placing node programs on nodes and routing networks. The programmer can explicitly place all node programs and explicitly route all networks. The programmer can also place all node programs but not specify routes for some of the networks. The current implementation does simple row then column routing for unspecified routes. The programmer can also leave both node placement and network routing unspecificed. In this case, the system uses an automatic placement and routing algorithm defined in [MKS89]. This search algorithm tries to optimize the use of the limited number of communication hardware resources. The framework could easily be altered to use other placement and routing algorithms, e.g. simulated annealing or other circuit routing techniques.

Figure 3 shows a flow chart of the tool chain. The PCS tools add two steps to the traditional compile and execute cycle of a uniprocessor: array module combining and array module loading. The *array combiner* compiles and executes the array program to create an *array module*. The combiner calculates which communication resources should be used for each network and writes this information in the *network definition file*, in effect compiling the networks. The resulting array module contains the network definition file and the node programs, encapsulating all information needed to execute the parallel program.

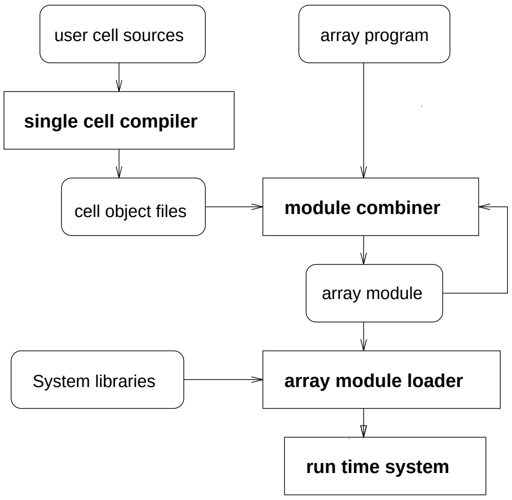


Figure 3: Flow chart of the PCS tool chain.

The *array module loader* loads an array module onto a specified physical system. It calculates a mapping from the logical grid and logical connections to the physical system. Physical nodes are addressed by their row and column location in the physical grid. Similarly nodes in array module are addressed by their row and column location in the logical array module. The array module loader calculates the mapping from the logical node ID's to the physical node ID's. Figure 4 shows a case where a 2×4 array module is loaded onto a 4×4 node array. The user can specify which processor should correspond upper left node in the array module. The logical to physical mapping in Figure 4 is $map(lr, lc) \rightarrow (lr + 1, lc)$. This array module has a connection that uses the back-edge from logical node (0,0) to (1,0). In this mapping, this connection must be routed through two extra physical nodes, since physical nodes (1,0) and (2,0) are not adjacent across the back edge.

After the logical to physical resource mapping is calculated, the array module loader loads the node programs and the necessary network initialization information on each node. After a global synchronization, the nodes read in the compiled network data to initialize the user defined connections.

Once the networks have been initialized, the programmer can access the networks with any library, assembler function, or system call, because the run time data movement functions are orthogonal to the

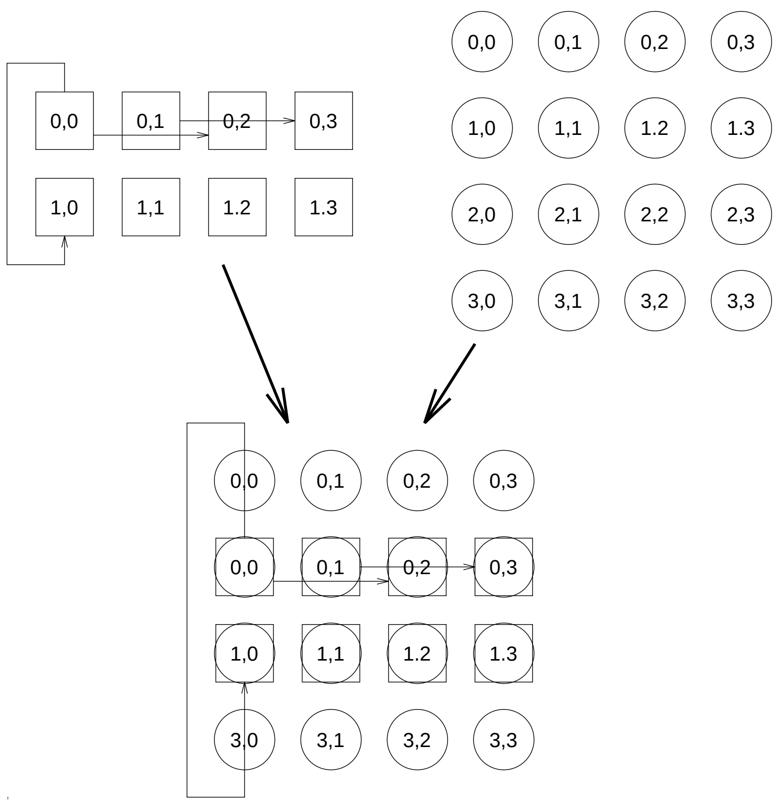


Figure 4: The 2x4 array module mapped onto a 4x4 physical array. The back edges of the array module are mapped through several physical nodes.

definition of the networks. The port structure contains detailed information about the resources used for the corresponding network. This information can be used to access the network at run time. For example on iWarp, the port structure can be used to directly access the register-mapped communication interface (e.g. `send_word(gate, port->queue, data)`). The port can also be used in a higher level data movement routine much as sockets or file pointers are used (e.g. `send_msg(port, data_block)`).

3.2 Module composition

The array module encapsulates all the details of a parallel program. The next obvious step is to compose smaller array modules into bigger modules, i.e. use array modules as building blocks. By composing array modules, the programmer can create a pipeline of several different programs or reuse the same subprogram in several different contexts.

PCS supports two styles of combining array modules: *merging* and *nesting*. These models differ in how they handle a node's ID. At run time, the node program uses a node ID to determine which program branches to take, which data to access, etc.. In the merge model, the included node program assumes the node ID of the logical node it is mapped to. This model is useful for loading utilities or networks, where the node ID is unimportant or should adapt. The array module in Figure 5(a) shows array module B included using the merge model. Node (0,0) in array module B assumes the node ID of the new array module, (2,2). By contrast, the nesting model preserves the node's original ID. This model is necessary to hierarchically

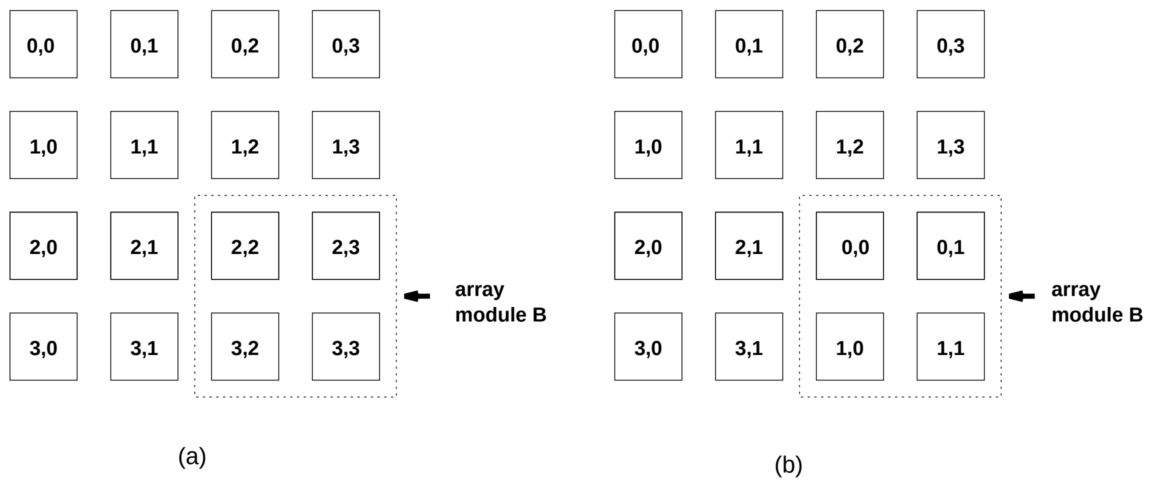


Figure 5: Array module B is included following the merge model in part (a). In part (b), array module B is included following the nesting model.

compose array modules. The array module in Figure 5(b) shows array module B included using the nesting model. In this case node (0,0) from array module B maintains its original grid ID. If the program on that node sends a message to the right, it would still send to node (0,1) not node (2,3).

Computer Aided Design (CAD) tools provide motivation for the nesting model. When designing a processor in a CAD tool, the user may first design a register unit with well-defined inputs and outputs. Then later in the design, the user can include an instance of the register unit. The main design does not know the details of the register unit design, but by properly using input and outputs, the main design can treat the register unit as a black box. Similarly, a person creating a parallel program may first create a Fast Fourier Transform (FFT) module with well-defined inputs and outputs. Then later the programmer can use an instance of the FFT module as a black box by properly feeding the inputs and using the outputs.

In PCS all array modules are created out of building blocks, *instances* of lower level array modules. The lowest level module is a node. Each node instance has a default instance ID, which is the same as its grid ID. By combining a set of nodes, a higher level array module is created. In turn instances of this array module can be composed to create a still higher level array module. The example in Figure 6 shows two instances of the simple array module described in Figure 2 composed by the nesting model into a larger array module. This example shows three levels of modules: the nodes, two instances of the array module `array_program1`, and one instance of the `nested_array` array module.

Since the composition follows the nesting model, the node program on the lower left node still assumes that its ID is (0,0), so it executes the `send_msg` statement in the node program. Similarly, the lower right node assumes it is still node (0,1), so it executes the `recv_msg` statement. If this array module was composed with the merging model, only the nodes on the top row of the array module would execute the guarded node program statements. The nesting model automatically keeps track of translating the logical node IDs to reflect their original ID. Without nesting composition, the node programmer would have to rewrite node programs to not rely on their node IDs or to reflect their new locations. With the nesting model, the programmer can create array modules without concern for their final location.

To complete the hierarchical composition, the nesting model must support communication between array modules. Figure 7 shows an example of a nested array module for a two dimensional FFT computation that also uses inter-module connections. Code segments from the 1D-FFT and 2D-FFT array programs are shown in Figure 8.

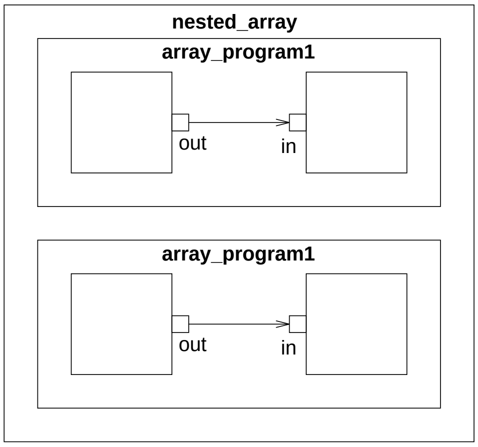


Figure 6: Nested composition of two instances the simple array module from Figure 2.

Module instances communicate through inputs and outputs defined as ports. As described in Section 3.1, nodes can access local ports, which are the only type of ports available at run time. Higher level array modules can access external ports, which are mapped to ports in the next lowest level. In Figure 7 external port "2D.in" is mapped to external port "FFT.in", which is mapped to local port "in". The code in Figure 8 shows this mapping with the call to `map_external_port` that makes these ports externally visible. Only the ports associated with the highest level of the array module instance are accessible in the array program. The external ports can be connected by networks as the "FFT.out" port in row-fft and the "FFT.in" port in col-fft are connected in Figure 7. The external port abstraction exports ports to higher levels, enabling nodes to be connected to other nodes outside their local array module without requiring the lowest level array program to consider all possible external connections.

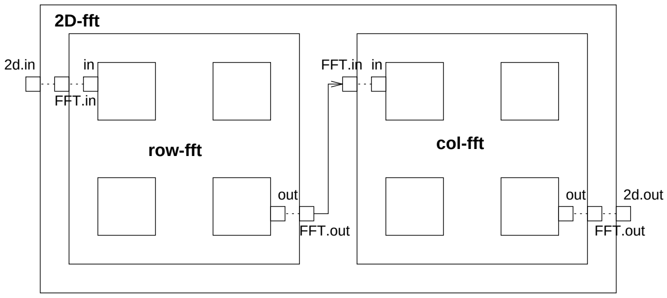


Figure 7: Array module composition of 2D-FFT array module from two 1D-FFT array modules. The dotted lines show mappings between local and external ports.

The node program for the 1D-FFT module is shown in the bottom of Figure 8. All nodes in the 2D-FFT module execute this code. The 1D-FFT module reads input from its input port, performs a series of 1D-FFTs, and writes the transposed data to the output port. In this example, the same 1D-FFT module is used for the row FFT and the column FFT. If a local port is not bound in the array program, it will be a NULL port at run time. In this example, by checking the value of the port, the node program decides between reading from the connection and reading from file I/O.

From array program for 1D-FFT

```
node.row = node.col = 0;
port = create_port(&node, "in");
map_external_port(port, "FFT.in");
node.row = row_num-1; node.col = col_num-1;
create_port(&node, "out");
map_external_port(port, "FFT.out");
```

From array program for 2D-FFT

```
node.row = node.col = 0;
rowinst = create_instance(&node, "row-fft", "fft");
in_port = get_external_port("FFT.in", rowinst);
map_external_port(in_port, "2d.in");

node.col = 2;
colinst = create_instance(&node, "col-fft", "fft");
out_port = get_external_port("FFT.out", colinst);
map_external_port(out_port, "2d.out");

out_port = get_external_port("FFT.out", rowinst);
in_port = get_external_port("FFT.in", colinst);
create_network(out_port, in_port, NULL);
```

From node program

```
if (self == node(0,0)) {
   in = get_port("in");
   if (in != NULL)
        recv_msg(in, data, sizeof(data));
   else read(0, data, sizeof(data));
}
distribute_data(data, self);

for (i = 0; i < n/P; i++)
   fft(data[i], n);

transpose(data, dataout);
merge_data(dataout, self)
if (self == node(1,1)) {
   out = get_port("out");
   if (out != NULL)
         send_msg(out, dataout, sizeof(dataout));
   else  write(1, dataout, sizeof(dataout));
}
```

Figure 8: Excerpts from array program that defines the 1D-FFT and 2D-FFT array modules, and excerpts of the run time node program executes on all nodes.

3.3 Communication resource sharing

On iWarp at most 20 communication queues are available on each node. This limits the number of connections that can be defined at a given point in time. One important observation is that programs go through different *phases* of communication patterns. When a particular communication pattern is not in

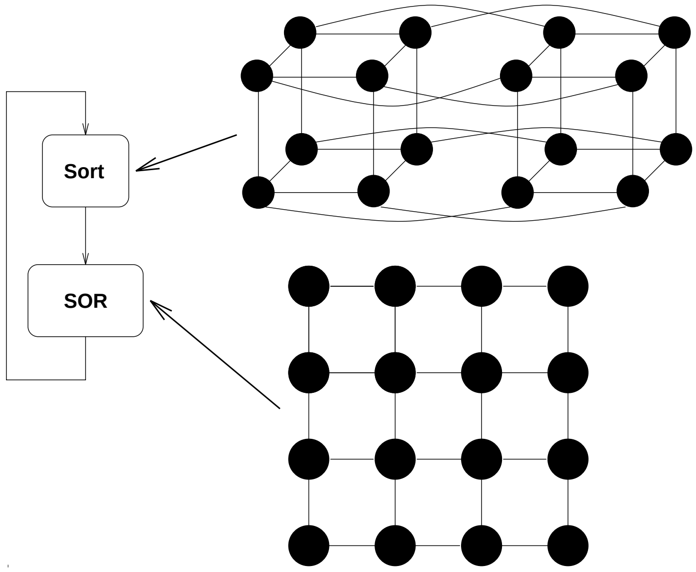


Figure 9: Program that alternates between two communication phases.

use, its resources can be used for other patterns. Thus, the communication resources can be time shared. This network phase switching is analogous to reusing registers via a context switch on a multiprogrammed uniprocessor. For example, Figure 9 shows a program that alternates between a phase that communicates using a hypercube pattern and a phase that communicates over a grid pattern.

Switching between communication patterns is conceptually simpler than general dynamic connection creation, because all nodes make the phase switch at a common point in the program. At compile time, the networks are defined as usual. In addition, the networks are assigned a phase ID with the `add_to_phase()` function. To switch between phases at run time, the node retrieves a pointer to the phase and calls the `set_phase()` function. The `set_phase()` function must first synchronize the system before the new communication pattern information can be loaded to ensure the network configuration is consistent across the system. In fact the synchronization is the bulk of the phase switch time in our current implementation. Figure 10 shows code segments that define and use the phases shown in Figure 9.

From array program

```
phase1 = create_phase("hyper");
nw = create_hyper_net();
add_to_phase(nw, phase1);

phase2 = create_phase("grid");
nw = create_grid_net();
add_to_phase(nw, phase2);
```

From node program

```
set_phase(get_phase("hyper"));
/* Hypercube-based calculation */

set_phase(get_phase("grid"));
/* Grid-based calculation */
```

Figure 10: Array program and node program segments that switch between hypercube and grid topologies.

Even for systems where hardware resource pressure is not so severe, phases can serve as a useful abstraction. For example on some systems, it may be a good idea to allocate more bandwidth or buffer space to the networks in the currently active phase.

3.4 Viewing array modules

In addition to the other compilation tools, there is an array module display program. This program uses information from the array module to create a picture of the defined networks, submodules, phases, and program mappings. Figure 11 shows the display of a module that defines networks for a transpose that utilizes the machine's full bisection bandwidth. Additional information is displayed at the top of the screen about the object indicated by the mouse. By choosing entries from the phase menu, networks from different phases are displayed.

Originally, we had planned to extend the module display program to enable graphical editing of the array module. At the time, we decided such a tool would only be useful for occasional or novice PCS programmers, who where not our target audience. In hindsight, editing would be quite handy for making small changes to array modules created by other users or other program generators.

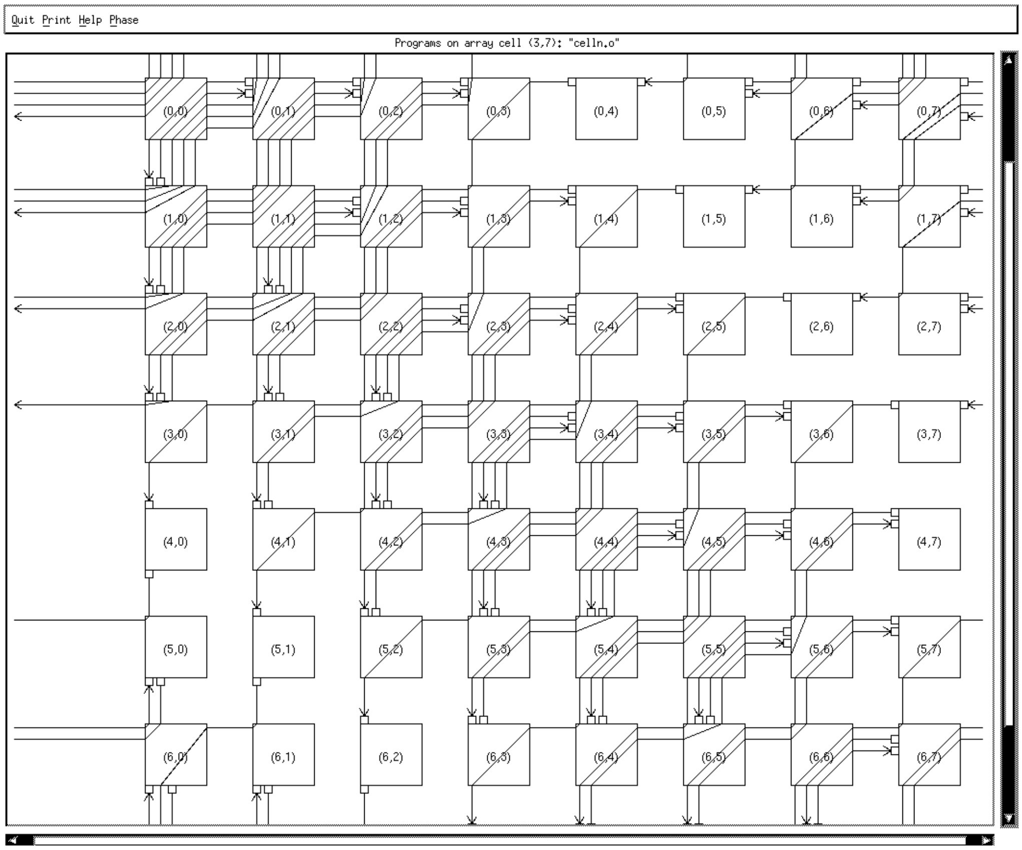


Figure 11: Display from the array module display program.

4 PCS experience

Development of PCS began in 1990 when the first iWarp systems arrived at Carnegie Mellon. C and PCS have been used to implement several parallel program generators developed at Carnegie Mellon including Adapt [Web92], Assign [O'H91], and Fx [SSOG93]. C and PCS are also directly used by system programmers for explicitly programming communication. . Starting in 1992, Intel included the PCS tool chain in its iWarp software distribution[Int92]. The Carnegie Mellon version of PCS has continued to evolve as we gain additional experience with iWarp[Hin93].

The network abstraction for long-lived connections has been very useful. For regular communication patterns, it is natural for the node program to be aware of the topology of the program. For instance, many hypercube algorithms make explicit use of the hypercube topology. These algorithms can be implemented with message passing, but the greater efficiency provided by long-lived connections does not add any conceptual complexity.

Compiling networks has been very effective. Creating long-lived connections at run time is quite error-prone, because it is easy for the programmer to encounter race or deadlock conditions while setting up arbitrary connections at run time. By compiling the networks, we were able to create programs that made sophisticated use of the communication system far sooner than we could have otherwise.

In general this implementation of the PCS tool chain has been very useful. As with any first generation system, we found limitations in the system after using it seriously. These limitations can be divided into three sets: conceptually simple implementation issues, poor interactions between the programming model and the physical system, and fundamental limitations in the programming model.

4.1 Simple implementation limitations

For any reasonably interesting set of networks, the network resource assignments will be different for most nodes. Originally, a separate network resource assignment table was compiled and linked for each node. The first implementation created up to N different network initialization files for each node on a N node system. Even for our initial 12 node system, this growth in compilation time was infeasible. The next implementation linked the network initialization information for all nodes with each node program. With only one initialization file, the compilation time was reduced, but the space requirements were infeasible for larger physical systems and complex array modules. The initialization phase of our current implementation explicitly loads and distributes the network information, so one network initialization file is created but only a portion of it is loaded on each node. This solution could also be performed by a sufficiently sophisticated program loader.

Name scoping was not sufficiently thought through. In the current implementation, the node program uses the port name to retrieve a reference to a port. If two ports both named “out” are defined in different network phases on the same node, the node program cannot unambiguously retrieve that port. The port name space requires more structure to avoid this ambiguity.

The current implementation only allows statically sized array modules. The size of the array module is defined at compile time. This static size is a bit constraining, but it enables greater run time efficiency. For run time flexibility, the array modules should be able to expand to use all available processors. A completely general virtual processor model would probably not be efficient enough, but a partially parameterized model may provide a reasonable trade-off between flexibility and efficiency.

4.2 Interaction limitations

An iWarp system (and most parallel systems) can have *distinguished* nodes. These nodes are different from the standard computing nodes because they have access to extra resources such as I/O boards or extra memory. The basic PCS model assumes that all nodes are created equal, so the introduction of distinguished nodes causes some conflicts between the programming model and the physical system.

On iWarp most distinguished nodes are not directly part of the torus mesh but instead are attached to the border of the torus via the torus back edges. Figure 12 shows a border node connected between nodes (1,0) and (1,3). For programs that do not use the border node, its presence means an extra link will be included in networks between (1,0) and (1,3).

A particular type of border node is not guaranteed to be in the same location on all machines. Originally, PCS tried to hide the real location of border nodes from the programmer, by forcing the programmer to

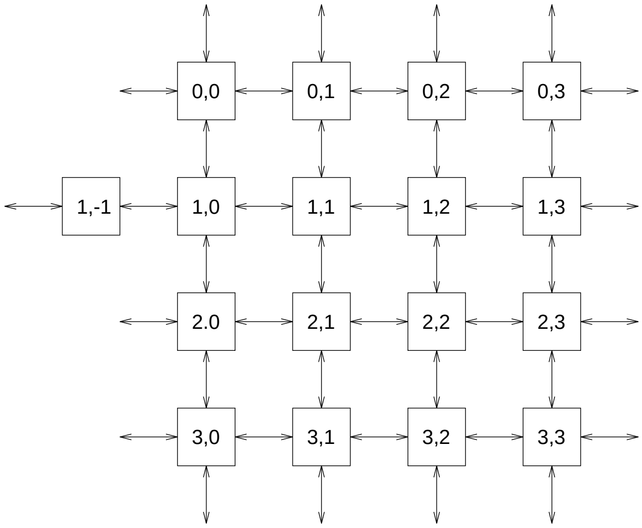


Figure 12: 4×4 physical array with a border node between nodes (1,0) and (1,3).

name the border nodes by ID's other than the location based grid ID and not allowing manual routing of networks that included border nodes. However, the direct routing control is vital in cases that try to optimize resource usage, so the current implementation no longer hides the location of border nodes. Instead it allows programmers who are willing to trade portability for performance to directly route to and from the border nodes.

Distinguished nodes also revealed limitations in the current implementation of automatic placement and routing. The current implementation is an all or nothing scheme. Either all nodes are automatically placed, or all nodes are manually placed by the programmer. In many cases, a programmer is concerned about placing one node program on a distinguished node but does not care exactly where the other node programs are placed. In this case, the system should allow the programmer to partially map the node programs to nodes. Then the system can take care of the remaining mapping and routing automatically. Pinning down the location of some node programs should be a simple extension to our current automatic placement algorithm.

Space sharing the machine through composing array modules destroys the torus model of the machine. If an array module is run alone, it can use all the network bandwidth. Similarly, if an array module has networks that are only routed across the face of the logical torus, it can use all the network bandwidth whether it is run by itself or it is composed with other array modules, but if an array module has networks routed across the back edges of the torus (as the example in Figure 4), those back edges may be routed through other array modules if the array module is composed.

Many applications require PCS communication to co-exist with other communication packages such as message passing and synchronization libraries that expect to directly control communication resources. Early implementations exposed the underlying resource reservation directly to the programmer. Programs that expected to use other packages had to explicitly reserve those resources in the array program. This forced the PCS programmer to understand which sets of resources all programs required. For example, the message passing library may require queues 0 to 3 and the synchronization library may require queues 7 to 10. This approach was unacceptable, because array programs were made incorrect by changes to other packages.

The current PCS implementation lets the programmer specify how many communication resources are required in the array program. This defaults to all available resources if nothing is specified. This requires less information from the programmer than keeping track of which resources are not available. Ideally, there should be a small run time routine that distributes communication resources and checks for obvious conflicts. Unfortunately, the current iWarp run time system does not have such a routine. Instead there are agreements between PCS and other packages about the order of resource allocation.

4.3 Model limitations

Array modules are static in the sense that they exist for the duration of the program. In some cases, it would be useful to run a series of array modules. For instance, the current implementation of tasking in the Fx parallel compiler uses array modules to represent tasks[SSOG93]. However, to support a general tasking scheme, array modules must be more dynamic. For instance, consider the task graph shown in Figure 13. To execute tasks 2 and 3 concurrently, the array is split into two groups. To executes tasks 5, 6, and 7, the array is split into three groups. Tasks 1 through 4 can be represented with one hierarchical array module, and tasks 4 through 8 can be represented with another array module, but these two array modules have different submodule divisions. The current PCS implementation can easily support tasks 1 through 4 or tasks 4 through 8, but the lack of time varying array modules makes it difficult to support tasks 1 through 8 directly. To support more general task graphs, the program must be able to run a series of array modules.

Time changing array modules can be supported by expanding the phase switching ideas described in Section 3.3. Instead of just switching between different sets of networks, the system also changes the currently executing array module structure. This structure includes the set of valid networks and ports, the ID of each node, and the size of each array module. To implement these time-dynamic modules, we need to define the *overlay* composition model in addition to the merging and nesting models defined in Section 3.2. The overlay model does not combine networks and divisions of the included array modules as the merging and nesting models do. The overlay model includes array modules to be instantiated at run time much as a library functions are linked into a single node program to be invoked at run time.

Figure 14 shows code segments to support this module switching. Module "mod1" is a hierarchical module that supports tasks 2 and 3. Module "mod2" is a hierarchical module that supports tasks 5, 6, and 7. The array program overlays these array modules, so they can be invoked at run time. The node program "main" is loaded on all the nodes. "main" implements the single unit tasks 1, 4, and 8. "main" is also responsible for moving data to the expected locations for modules "mod1" and "mod2" and calling functions in those array modules. `set_module` instantiates the networks like `set_phase`. In addition, it adjusts the node's ID and active array module information. After switching to an array module, functions defined in that array module will execute just as if the array module had been executed directly.

5 Related communication tools

The PCS tool chain is similar in spirit to many parallel *coordination languages*. Coordination languages are libraries or language additions that define communication structures and coordinate parallelism. The coordination language is generally orthogonal to the base computation language and can be incorporated with several different languages. Most implementations of these coordination languages embed the definition of the communication structure in the run time node program. Thus, without sophisticated compile time analysis, the communication structures must be defined at run time contributing to the run time communication overhead.

PVM[Sun90], Express[FKB91], and Linda[CG89] are three widely used coordination languages. They have been ported to a wide variety of architectures and are quite general. This generality means that the

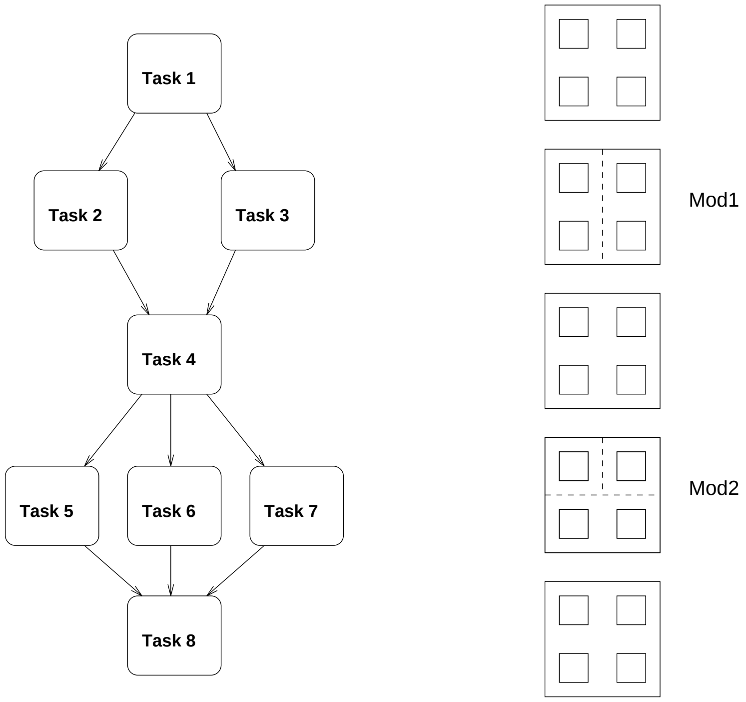


Figure 13: A program task graph and corresponding array modules.

systems must buffer messages and hide many of the features of the target system. While PVM and Express support the idea of creating "connections" between processing elements, the required buffering eliminates the possibility of supporting low latency communication using these tools. The communication structures are embedded in the node program, so are not executed until run time.

Fortran M [FC93] and Parallaxis [Bra92] are two examples of programming models that augment the base language to support communication pattern definitions. They can also be thought of as coordination languages except the communication definitions are language extensions not orthogonal additions. Fortran M augments Fortran 77 with support for connections and modules to support task parallelism. Similarly, Parallaxis augments Modula-2 with support for SIMD parallelism and user specified communication patterns. In both cases, the communication structures are defined in the run time program.

Transputer systems also incorporate the connection abstraction into their programming model via channels. Without additional routing switches the T800 systems do not support direct connections between nodes that are not physically adjacent. Channels can only connect adjacent processors, which simplifies allocating network resources.

The C_NET programming environment[ABT92] has been developed for the SuperNode architecture, a system of T800 processors and routing chips that enable reconfigurable connections between processors. The C_NET environment employs abstractions for long-lived connection and communication phases. In this environment, the user is responsible for writing three sets of programs: software components for processes similar to PCS node programs; phase programs that define a mapping between software components and processors and interprocessor connections, similar to the PCS array program; and topology definition programs that define how the routing switch is configured. The communication state is defined at compile time. The C_NET environment does not utilize the array module abstraction and does not allow array module composition.

MARC[BIK91] is another tool developed for a network of T800's. MARC is more directed towards automatically mapping and routing a set of communicating sequential processes than supporting a programmer

From array program

```
compose_module("mod1", "tset1");
compose_module("mod2", "tset2");

for (node.row = 0; node.row < num_rows; node.row++)
   for (node.col = 0; node.col < num_cols; node.col++)
      place(&node, "main.o");
```

From node program

```
/* Arrange data for task set 1 */
module_switch("tset1");
cur_inst = get_instance();
if (cur_inst == task2)
    task2();
else if (cur_inst == task3)
    task3();

/* Arrange data for task set 2 */
module_switch("tset2");
cur_inst = get_instance();
if (cur_inst == task5)
    task5();
else if (cur_inst == task6)
    task6();
else if (cur_inst == task7)
    task7();
```

Figure 14: Code to support time-dynamic array modules. By using the overlay model, this code implements the example in Figure 13.

using the communication system directly.

Newer transputer systems should provide greater opportunity for using long-lived connections[MTW93]. The T9000 processor provides virtual channel support, so multiple virtual channels can be mapping to the same physical channel. Therefore, multiple long-lived virtual channels can coexist. The C104 routing chip supports connections between processors that are not necessarily physically adjacent.

[SQH92] describes a tool to create connections between Data Parallel C modules. This is similar to the PCS support for external ports and hierarchically nesting array modules. However, this work only looks at one level of nesting.

6 Conclusions

Over the last four years, we have created and used PCS for programming iWarp. PCS enables programmers to access the communication hardware efficiently, and the network and array module abstractions enabled programmers to create interesting communication patterns quickly. PCS is not the ultimate parallel programming language, but it has been very useful for supporting higher level parallel program generators and system level programmers.

Like other parallel coordination languages, PCS separates communication support from computation support, and PCS is not intrinsically tied to any base computation language. Most use of PCS has been with C, but the Fx program generator uses Fortran node programs with PCS for communication definition.

Unlike other parallel coordination languages, PCS separates communication set up from communication execution. With this separation, PCS tools are able to allocate resources and define maps for the parallel program off line. This static communication set up is not added to the run time communication overhead and avoids many potential network creation error conditions. Since communication set up and execution are separated, using PCS for set up does not bind the user to any particular run time data movement package.

The communication abstractions are not specific to the iWarp hardware but are applicable to any system that supports a connection-based communication model either directly in hardware like the Transputer systems or through software on top of a different model like the Paragon. Networks and ports remove the programmer from the details of connection configuration and resource allocation on the target system. The array modules and communication phases provide program structure, enabling array module reuse and modular programming.

Finally, our implementation experiences and design evolution should be interesting to others working with a connection-based communication model. The details of our experiences will not be the same, but other developers will encounter many similar interaction and model issues.

References

[ABT92] J-M. Adamo, C. Bonello, and L. Trejo. The C_NET Programming Environment: An Overview. In *Parallel Processing: CONPAR92-VAPPAV The Second Joint International Conference on Vector and Parallel Processing*, Lyon, France, September 1992.

[B+88] S. Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[B+90] S. Borkar et al. Supporting Systolic and Memory Communication in iWarp. In *Proc. 17th Intl. Symposium on Computer Architecture*, pages 70–81. ACM, May 1990. A revised version has appeared as technical report CMU-CS-90-197.

[BIK91] J. E. Boillat, N. Iselin, and P. G. Kropf. MARC: A Tool for Automatic Configuration of Parallel Programs. In *Transputing '91*, 1991.

[Bra92] T. Braunl. Transparent Massively Parallel Programming with Parallaxis. *International Journal of Mini and Microcomputers*, 14(2):82–87, 1992.

[CG89] N. Carriero and D. Gelernter. Linda in Context. *Communication of the ACM*, 32(4):444–58, April 1989.

[DS87] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[FC93] I. T. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Technical Report ANL, Argonne National Laboratory, 1993.

[FKB91] J. Flower, A. Kolawa, and S. Bharadwaj. The Express Way to Distributed Processing. *Supercomputing Review*, pages 54–55, May 1991.

[HG92] S. Hinrichs and T. Gross. Utilizing New Communication Features in Compilation for Private-Memory Machines. In *Advances in Languages and Compilers for Parallel Processing*, July 1992.

[Hin93] S. Hinrichs. *Programmed Communcation Service Tool Chain User's Guide*, March 1993.

[Int92] Intel. *iWarp Programmer's Guide*, August 1992.

[MKS89] O. Menzilcioglu, H.T. Kung, and S. W. Song. Comprehensive Evaluation of a Two-Dimensional Configurable Array. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, pages 93–100, 1989.

[MTW93] M.D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers, and Transputers: Function, Performance, and Application*. IOS Press, Inc., Amsterdam, Netherlands, 1993.

[O'H91] D. O'Hallaron. The Assign Parallel Program Generator. In *Proceedings of the 5th Distributed Memory Computer Conference*, 1991.

[SQH92] B. Seevers, M. J. Quinn, and P. J. Hatcher. A Parallel Programming Environment Supporting Multiple Data-Parallel Modules. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, Boulder, CO, October 1992.

[SSOG93] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[Sun90] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–39, December 1990.

[Web92] J. A. Webb. Steps Toward Architecture-Independent Image Processing. *Computer*, 25(2):21–31, February 1992.