2 The Assign Flowgraph

As discussed in Chapter 1, the flowgraph is the underlying programming model for Assign. A flowgraph is a multidigraph consisting of a collection of nodes and directed edges. A multidigraph is a directed graph where multiple edges are allowed between the same pair of nodes. Nodes represent computations and edges represent FIFO queues.

Each node iterates an infinite number of times. For each iteration, the node consumes data from each of its input edges, performs a computation on the inputs, and produces a vector of data items on each of its output edges.

NOTE

For the Assign flowgraph model, cyclic graphs are not allowed, since data can only flow in one direction.

Flowgraph Nodes

There are three types of nodes in an Assign flowgraph:

- atom nodes
- subgraph nodes
- data nodes

An atom node corresponds to a primitive task, such as an FFT, with optional input and output functions.

A subgraph node is part of a hierarchical flowgraph. A subgraph node is a complete flowgraph which is included inside another flowgraph. The subgraph is used as if it were a single node performing the function of the nested flowgraph.

A data node is a special node that distributes, collects, or reduces data used in parallel computations. Descriptions of distribute, collect, and reduce nodes are included in the section *Adding Nodes to the Graph*, later in this chapter.

Each flowgraph also has a *source* node and a *sink* node. A source node serves as the first node in the flowgraph. It is responsible for supplying the initial data to the flowgraph. A source node has no input edges. A flowgraph can have only one source node.

A sink node serves as the last node in the flowgraph. It is responsible for collecting and outputting the final results of the flowgraph. A sink node has no output edges. A flowgraph can have only one sink node.

Nodes are connected to one another by edges via input and output ports on the nodes. An edge binds an output port from one node to the input port of another node. Each edge corresponds to a FIFO queue with exactly one writer and one reader. Each port in a node is bound to at most one edge. The two ports bound to an edge must transfer data of the same type.

Each input port has a name, a scalar type, and an integer *consume amount*. The consume amount is the length of the input vector to the port. Each output port has a name, a scalar type, and an integer *produce amount*. the produce amount is the length of the output vector going out on the port.

NOTE

> In the current implementation of Assign, the consume amount and the produce amount must be the same for the input and output port of a node.

Node Functions

Typically, the first step in building an Assign application is writing the appropriate node functions. This includes the following:

- primitives
- state structures
- input and output functions
- dynamic initializers

The following sections describe the different elements of node functions.

Primitives

A primitive is a function that describes the node's execution. It is compiled and executed on one or more iWarp cells. The parameters passed to or from the primitive should reflect the number of input or output ports on the corresponding node. If there is one input port and two output ports, then you should pass a pointer to an input vector and two pointers to output vectors. You can also pass a pointer to the state structure, so relevant state information can be used during primitive execution. This allows you to make a more generalized primitive which you can use in a variety of situations by modifying the values in the state structure. The following section describes state structures.

NOTE

> User primitives may be written in C or FORTRAN. The primitive is compiled into an object file before it is used. Therefore, use of FORTRAN will require the iWarp FORTRAN compiler (iwf77), and C will require the iWarp C compiler (iwcc).

The order in which parameters are passed to the primitive is also important. The argument list should contain all (if any) of the pointers to input vectors, followed by all (if any) of the pointers to output vectors, followed by an optional pointer to the state structure. The order of the input vectors within the list should correspond to the order in which the input ports were added using the function add_in_port(). Similarly, the output vectors should be listed in the same order they were declared using the function add_out_port(). These functions are described in the section, *Writing the Flowgraph Generator.*

State Structures

A state structure is an optional C structure that contains any variables you use to make the primitives more generalized. When you create a graph, nodes using the same primitive will all have their own state structure. If these state structures are initialized to different values, the primitives may all function differently, even though they are the same function. Currently, the state structure is defined in a separate .h (header) file, which must be included in the primitive's .c file. An example of how to define and use the state structure is shown in the section, *Node Function Example*.

The state structure defines the internal state of a node. A pointer to the state structure is passed to a primitive each time the primitive is invoked. The primitive can inspect and modify the state structure. State structures can be initialized at compile time with a *static initializer*, and/or at run time with a *dynamic initializer.* The following section describes initializers.

Initializers

State structures can be initialized at compile time with a static initializer, and/or at run time with a dynamic initializer. A static initializer is an arbitrary C structure initializer list. A dynamic initializer is a C function that sets the values of the variables in the state structure. This function is executed at run time, before the primitive is executed. The dynamic initializer can use statically-assigned values, or simply overwrite them. The dynamic initializer function is optional, as is static initialization.

Input and Output Functions

If a primitive function is going to be a source or sink node, there must be input or output functions, respectively, associated with the primitive. When it is uncertain if a primitive will be used as a source or sink, you can still write the input or output functions. If the primitive is not used as a source or sink node, the input and output functions are ignored.

The input function of an atom node is invoked whenever the node is a source node. The output function of an atom node is invoked whenever the node is a sink node. An input function is a C function which returns an input vector of length c_k (consume amount for port k) for each input port k. An output function is a C function which emits a vector of length p_k (produce amount for port k) for each input port k. An example of input and output functions is shown in the section, *Node Function Example*.

NOTE

The number of input parameters must be equal to the number of input ports on the node, and the number of output parameters must be equal to the number of output ports on the node. Also, the order of the parameters should correspond to the order in which they are passed to the primitive, as discussed in the section *Primitives*.

Flowgraph Execution Rules

Each atom executes according to the following rules: Initially, the atom calls an optional dynamic initializer. Then the following steps occur one or more times:

1. Remove c_k data items from each input port k. If this is a source node, the data for each input port is received via an invocation of the input function. Otherwise, the data for each port k comes from input queue k.
2. Invoke the corresponding primitive using the above c_k data items.
3. Write p_k data items to each output port k. If the atom is a sink node, the data written to the port is passed to the output function, which sends the data to the host computer. Otherwise, data written to output port k is deposited in output queue k.

Each atom executes independently according to the atom execution rules. Each subgraph node executes the corresponding subgraph. This is accomplished by binding the input edges of the subgraph node to the input ports of the subgraph's source node. Accordingly, the output edges of the subgraph node are bound to the output ports of the subgraph's sink node.

Data Types and Definitions

Definitions of types, macros, and values can all be found in the file assign.h. This file must be included with a #include statement as follows in order to use the definitions in your Assign C program input:

```
#include <assign/assign.h>
```

Data Types

Data used in a flowgraph can be any of the following standard C types: int, float, and double. There are also special complex variables which you can use: complex_int, complex, and complex_double. Complex types use float-sized fields. The complex data types are defined in assign.h as follows:

```
typedef struct {int r, i;} complex_int;
typedef struct {float r, i;} complex;
typedef struct {double r, i;} complex_double;
```

Therefore a variable of type complex can access the real part of a complex number with variable.r, and the imaginary part with variable.i.

Another structure you will use is GRAPH. This holds all necessary information about the entire flowgraph. You must declare variables of this type (one for each graph or subgraph) for use in building the flowgraph. The most useful way of declaring GRAPH variables is as follows:

```
GRAPH *gp; /* gp == "Graph Pointer" */
```

Note that the variable `gp` is a pointer to the GRAPH structure. This is because the variable is passed to all pre-defined functions as a pointer.

Another necessary structure is NODE. This structure contains all information about a node in the flowgraph. An example declaration follows:

```
NODE *np; /* np == "Node Pointer" */
```

This variable is also passed to pre-defined functions as a pointer, hence the declaration.

Useful #define Values.

Some useful names have been pre-defined in assign.h. You should use these in function calls instead of meaningless integers.

```
/* type of data on each edge */

#define INT    0
#define FLOAT  1
#define DOUBLE 2
#define COMPLEX_INT 3
#define COMPLEX 4
#define COMPLEX_FLOAT 4
#define COMPLEX_DOUBLE 5
```

Node Function Example

The following example illustrates parts of a node function. The example is a complex increment node that increments the elements of a vector by one. Since each element is complex type, the real and imaginary parts are each incremented. The input and output functions use standard input and standard output.

The state structure definition is placed in a header file called cxincr.h. All other functions are placed in a file called cxincr.c which must include cxincr.h. The cxincr.c file is compiled into object code and left in a file called cxincr.o. This is later linked with other primitives and a main function for execution on the hardware cells. Refer to Chapter 3, *Compiling and Executing Your Program*, for details on this process.

The following primitives are part of a broader example that is modeled after an adaptive beam interpolation (ABI) application. Note that although we refer to the example as ABI, it is not an actual adaptive beam interpolation application. The graph generator for this example is included in the section *An Example Flowgraph Generator*, later in this chapter. All of the code for the example is contained in Appendix B.

```
/* cxincr.h : state structure for cxincr.c */
struct state_cxincr
{
   /* Size of the vector to be incremented */
  int size;
};
```

The following is the cxincr.c file.

```
/*
 * cxincr.c :
 * increment elements of a complex vector of size sp->size
 */

#include "cxincr.h"
#include <assign/assign.h>

void
prim_cxincr(inp, outp, sp)  /* The primitive execution function */
complex *inp, *outp;
struct state_cxincr *sp;
{
```

```
  int i;

  for (i=0; i < sp->size; i++)
    {
      outp[i].r = inp[i].r + 1.0;
      outp[i].i = inp[i].i + 1.0;
    }
}
/*
 * input routine for a cxincr node
 */
in_cxincr(inp, sp)
complex *inp;
struct state_cxincr *sp;
{
  int i;

  for (i=0; i<sp->size; i++) {
    inp[i].r = (float)i;
    inp[i].i = (float)i;
  }
}
/*
 * cxincr_out - output routine for a cxincr node
 */
out_cxincr(outp, sp)
complex *outp;
struct state_cxincr *sp;
{
  int i;

  for (i=0; i<sp->size; i++) {
   printf("(%.2f,%.2f)\n", outp[i].r, outp[i].i);
  }
}
```

Since the names of all the functions and the structure are used in creating the flowgraph, there is a simple naming convention you should follow. If the name of the primitive function is foo, you should use the following names:

state_foo	name of state structure
prim_foo	name of primitive execution function
init_foo	name of dynamic initializer function
in_foo	name of input function
out_foo	name of output function

This convention is useful for using a primitive where only the name is known. Then the other names will be known while writing the graph generator.

Writing the Flowgraph Generator

Once the node functions are written, the next step is to create the flowgraph with a graph generator. This is done by writing a C program which instantiates the primitives as nodes and connects the ports with edges. The C program uses pre-defined functions that can be found in a library called libassign.a. You must also include the file assign.h to access the necessary pre-defined functions.

This method of graph generation exploits the tremendous expressive power of C, including iteration, functions, and recursion. This expressive power is extremely helpful for generating large graphs with hundreds or thousands of nodes. Some advantages include:

- you can use functions as an abstraction mechanism to create subgraphs
- you can use iteration to create iterative constructs such as pipelines
- you can use recursive functions to create recursive constructs such as fan-out fan-in trees.

Another advantage is the use of command line arguments and manifest constants which you can use to parameterize the flowgraph generator. This means you can use one graph generator to produce an entire family of graphs by simply changing some constants and recompiling the new graph.

The functions used to build the flowgraph are described in the following sections. An example flowgraph generator is provided in the section that follows the function descriptions.

Creating an Empty Graph

```
GRAPH *alloc_graph(name)
char *name;
```

alloc_graph() allocates a new graph. It is given the name *name* (a string). The function returns a pointer to this new graph structure. You should store the pointe in a variable for later function calls. Even though the pointer actually points to a data structure, it is more convenient to picture the pointer as pointing to the entire physical graph (nodes connected with edges, etc.).

Adding Primitives to the Graph

```
void add_obj_file(graph, filename)
GRAPH *graph;
char *filename;
```

add_obj_file() tells the Assign compiler to use the functions in *filename*.o as primitives. This function can be called as many times as necessary (at least once for each primitive object file). If the primitive .o files are defined in a linked library, you do not need to call add_obj_file(). Appendix A describes how to use Assign libraries.

```
void add_inc_file(graph, filename)
GRAPH *gp;
char *filename;
```

add_inc_file() is similar to add_obj_file() except that *filename* is the name of the primitive header file that contains the optional state structure definition.

Adding Nodes to the Graph

```
NODE *add_node(graph, name, primitive, weight)
GRAPH *graph;
char *name, *primitive;
int weight;
```

add_node() adds atom *name*, with primitive function *primitive*, and execution weight *weight*, to graph *graph* (the pointer generated by **alloc_graph()**). The weight is an estimated number of cycles consumed by one iteration of the primitive. This does not affect the semantics of the flowgraph. It is a hint you supply to help Assign balance the workload across the cells of the iWarp array. **add_node()** returns a pointer to the node structure which is used in subsequent calls to add ports to the node. Again, it is more convenient to think of the returned pointer as pointing to a node in the graph and not to a C structure.

```
NODE *add_gnode(graph, subgraph)
GRAPH *graph, *subgraph;
```

add_gnode() hierarchically adds a subgraph to another graph. The subgraph is then treated as a single node in the outer graph. The subgraph node gets the name of subgraph *subgraph*. The subgraph node also gets the input ports of *subgraph*'s source node and the output ports of *subgraph*'s sink node.

In addition to the two node addition routines described, there are three special node types used for data manipulation: distribute nodes, collect nodes, and reduce nodes. Distribute nodes distribute data to the sections of the flowgraph that operate on the corresponding data in parallel. You can then use a collection node to collect the results back into one vector or matrix. A reduce node accepts parallel data (like a collection node) and performs some operation on the data vectors to reduce them into one vector. Descriptions of these routines follow.

```
NODE *add_dnode(graph, name, type)
GRAPH *graph;
char *name;
int type;
```

add_dnode() adds a distribute node with name *name* and type *type* to flowgraph *graph*. The *type* parameter tells the node how to distribute the data. Possible values for *type* (predefined in assign.h) are PRD_NODE (Parallel Row Distribute) and PCD_NODE (Parallel Column Distribute). For an i by j size matrix, the input to the distribute node is the matrix as one long vector of the matrix rows. If row distribution is used, each row (i sets of j data elements from the input) is passed to the corresponding ith parallel execution portion of the flowgraph. If column distribution is used, the jth element from every row is passed to the jth parallel execution of the flowgraph.

The easiest way to imagine these types of distribution is as follows: The row distribute will divide an input matrix into vectors of rows and pass each of these to the corresponding execution portion of the flowgraph. The column distribute will break the matrix into vectors of columns, and pass each column to the corresponding parallel execution portion of the flowgraph.

```
NODE *add_cnode(graph, name, type)
GRAPH *graph;
char *name;
int type;
```

add_cnode() adds a collect node with name *name* and type *type* to flowgraph *graph*. The *type* parameter tells the node how to collect the data. Possible values for *type* (predefined in assign.h) are PRC_NODE (Parallel Row Collect) and PCC_NODE (Parallel Column Collect). For an i by j size matrix, the input to the collect node will be vectors of either the j columns (column collect) or the i rows (row collect) of the output matrix. The output is created by combining the rows or the columns into the final matrix (in vector form).

Since distribute and collect nodes perform no operation on the data, no primitive execution routine is needed. Input or output routines may be needed if the corresponding data node is a source or sink node. The ABI example provided in Chapter 2 and Appendix B provides a good illustration of data distribution and collection nodes.

```
NODE *add_rnode(graph, name, type, primitive, weight)
GRAPH *graph;
char *name;
int type;
char *primitive;
int weight;
```

add_rnode() adds a reduce node with name *name* and type *type* to flowgraph *graph*. The execution performed in reducing the data is defined in the *primitive*. The *weight* is an estimate of the number of clocks needed to perform one execution of the primitive. The inputs to a reduce node are the set of vectors on which the reduce function operates. All input vectors must be the same size. The reduction function takes two vectors at a time and produces one vector, which is reduced by the reduction function. This is repeated on all the input vectors until one final result vector is obtained. This is the final output of the reduction node, and the output will be the same size as the input vectors.

Connecting the Nodes

```
add_in_port(node, name, type, consume)
NODE *node;
char *name;
int type, consume;
```

add_in_port() adds input port *name* of type *type* to node *node*. The *consume* amount (c_k from the flowgraph description) tells the port how many data items to read in at each node iteration.

```
add_out_port(node, name, type, produce)
NODE *node;
char *name;
int type, produce;
```

add_out_port() adds output port *name* of type *type* to node *node*. The *produce* amount (p_k from the flowgraph description) tells the port how many data items to write out at each node iteration.

```
add_edge(graph, out_node, out_port, in_node, in_port, initq)
GRAPH *graph;
NODE *out_node, *in_node;
char *out_port, *in_port;
int initq;
```

add_edge() adds an edge to graph *graph* that connects output port *out_port* of node *out_node* to input port *in_port* on node *in_node*. The queue corresponding to the edge is initialized with *initq* zeros. (This feature is not currently implemented, and the value passed to initq must be 0.)

Adding a State to a Node

```
add_state(node, state, init_list, init_func)
NODE *node;
char *state, *init_list, *init_func;
```

add_state() associates a state structure with node *node*. The structure is defined by the structure name *state*. The state structure is initialized at compile time by the static initializer list *init_list*. If *init_list* is NULL, Assign assumes there is no static initializer list. The state structure is initialized at run time by the dynamic initializer function *init_func*. If *init_func* is NULL, Assign assumes there is no dynamic initialization routine.

Adding I/O to a Node

```
add_io(node, input, output)
NODE *node;
char *input, *output;
```

add_io() associates the input function *input* and the output function *output* with node *node*. If *input* is NULL, Assign assumes there is no input function and the node can not be used as a source node. If *output* is NULL, Assign assumes there is no output function and the node can not be used as a sink node. I/O functions can be defined and added , even if a node is not used as a source or sink node. Input and output functions are described in the section, *Input and Output Functions*, earlier in this chapter.

Flowgraph Generator Output

```
print_graph(graph, level, outfile)
GRAPH *graph;
int level; /* 0 = all subgraph levels, else print level levels */
FILE *outfile
```

print_graph() sends an ASCII representation of the flowgraph *graph* to the file *outfile*. It lists each node with its input and output ports, and its attributes. The *level* argument tells how many hierarchical subgraph levels to include in the listing. A value of 0 specifies all levels. This feature is very helpful when debugging the final flowgraph. You can send the output to stdout or stderr by passing stdout or stderr as the *outfile* argument to print_graph().

```
emit_graph(graph, outfile)
GRAPH *graph;
FILE *outfile;
```

emit_graph() generates the intermediate representation, which is the input to the assign0 pass of the compiler. This function call must be the final call in the graph generator. The intermediate representation is left in a file pointed to by *outfile*. It is the user's responsibility to open the file (using fopen()) with the prefix name, followed by a .graph suffix. The example flowgraph generator in the following section illustrates one way of doing this.

Deallocating a Graph

```
dealloc_graph(graph)
GRAPH *graph
```

dealloc_graph() frees the dynamically-allocated memory used to build the flowgraph. Using dealloc_graph() with a GRAPH pointer empties the graph and frees everything but the main GRAPH structure. To free the main GRAPH structure, call free(*graph*) immediately following the call to dealloc_graph(). Using dealloc_graph() is optional, and after its use the flowgraph pointed to by *graph* is empty, and therefore, is not valid.

An Example Flowgraph Generator

The C code in this section comprises a graph generator which describes an adaptive beam interpolation (ABI) model as shown in Figure 2-1.

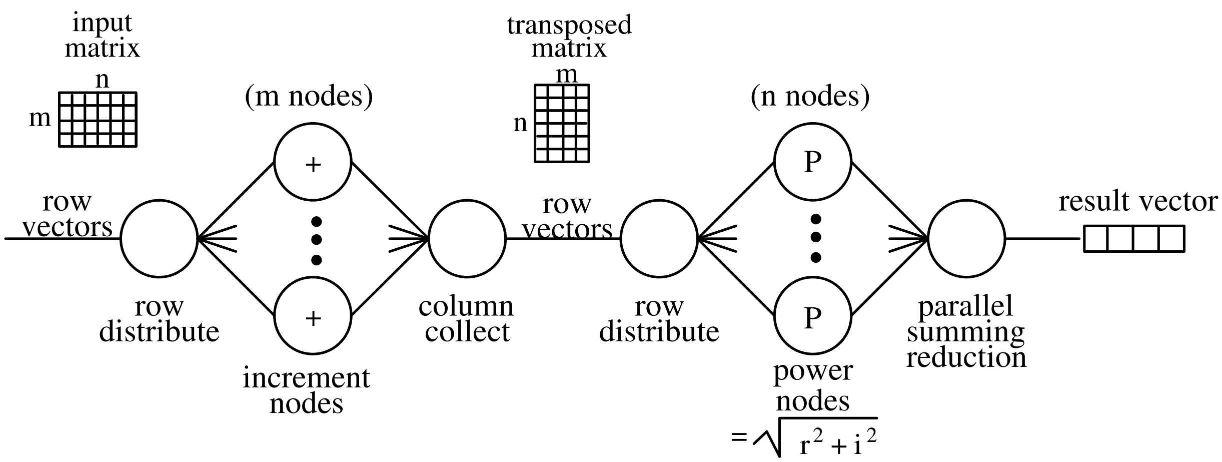

Figure 2-1: The ABI Graph

The example assumes that the primitives power, cxincr, cxrd, cxcc, and prred have already been created using the instructions and conventions previously described in this chapter. The code for all of these primitives is included in Appendix B. This example illustrates the power of Assign, as shown in the following characteristics:

- Use of data distribution and collection
- Matrix transpose or ‘‘corner turning’’ between the rcfan and prfan is done on-the-fly with no data buffering required
- Use of parallel data reduction
- Use of both complex and floating-point data in the same flowgraph
- Use of subgraphs to construct a larger graph

The following function is called to create a ready-to-use cxincr node with ports, state, and I/O functions. This type of function is called a node generation function (NGF). These functions are used at graph build time, but are commonly written with node primitives so correct names are used.

This NGF is passed the graph pointer, the name of the node to be created, its execution weight, and the input sample size. the function returns a pointer to the newly created cxincr node. It is called from the abi function.

```
#include <assign/assign.h>
#include <stdio.h>
/*
 * ngf_cxincr_node() : create a node which increments
 * each element of a complex vector by one. The size
 * of the vector is n.
 */
NODE *ngf_cxincr_node(gp, name, n)
GRAPH *gp;
char *name;
int n;
{
  NODE *np;
  char ilist[50];

  /* Add the node to graph "gp" with name "name".
   * The primitive function which performs the run-time
   * addition is called "prim_cxincr".  The estimated
   * number of clocks for this operation is 20 * size.
   */
  np = add_node(gp, name, "prim_cxincr", n*20);

  /* There is one in and one out port. Both operate on
   * an "n" element vector of type COMPLEX. */
  (void)add_in_port(np, "in", COMPLEX, n);
  (void)add_out_port(np, "out", COMPLEX, n);

  /* Add a state structure and initialize the size field
   * to the size of the vector. */
  (void)sprintf(ilist, "%d", n);
  add_state(np, "state_cxincr", ilist, "");

  return(np);   /* return a ready-to-use node */
}
```

The following functions, rcfan_graph and prfan_graph, build the row increment and power compute fans shown in Figure 2-1. These functions illustrate how iteration can be used to create a repetitive graph without a great deal of code.

```
/*
 * rcfan_graph - build a parallel fan of "row" incr
 * nodes with data size "col"
 */

GRAPH *rcfan_graph(name, row, col)
char *name;
int row, col;
{
  int i;
  GRAPH *gp;
  NODE *cxrd_np, *cxcc_np, *cxincr_np;
  char temp[50];

  /* First thing is to allocate space for the new graph */
  gp = alloc_graph(name);
```

```
  /* Add object files which contain run-time primitives */
  add_obj_file(gp, "cxincr.o");
  add_obj_file(gp, "cxrd.o");
  add_obj_file(gp, "cxcc.o");

  /* Add header files which contain state structure info for
   * the run-time primitives. */
  add_inc_file(gp, "cxincr.h");
  add_inc_file(gp, "cxrd.h");
  add_inc_file(gp, "cxcc.h");

  /* Invoke the data distribution and data collection nodes */
  cxrd_np = ngf_cxrd_node(gp, "rd", row, col);
  /* NOTE: The final matrix is transposed, so tell the collect
   * node the real dimensions of the final matrix, which is
   * the reverse of the dimensions on the input matrix.
   */
  cxcc_np = ngf_cxcc_node(gp, "cc", col, row);

  /* Generate the compute nodes in the fan and connect
   * them to the distribute / collect nodes.
   */
  for (i=0; i < row; i++)
    {
      /* Make a unique name for every complex incr node */
      (void)sprintf(temp, "cxincr%d", i);
      cxincr_np = ngf_cxincr_node(gp, temp, col);

      /* Add input and output edges, using the correct names of ports
       * on the compute node and the data nodes */
      (void)sprintf(temp, "out%d", i);
      add_edge(gp, cxrd_np, temp, cxincr_np, "in", 0);
      (void)sprintf(temp, "in%d", i);
      add_edge(gp, cxincr_np, "out", cxcc_np, temp, 0);
    }

  return(gp); /* return the rcfan graph */
}

/*
 * prfan_graph - build a parallel fan of "row" power nodes,
 * data size "col"
 */
GRAPH *prfan_graph(name, row, col)
char *name;
int row, col;
{
  int i;
  GRAPH *gp;
  NODE *cxrd_np,  *pr_np, *power_np;
  char temp[50];

  /* Allocate space for the new graph */
  gp = alloc_graph(name);
```

```
  /* Add object files which contain run-time primitives */
  add_obj_file(gp, "power.o");
  add_obj_file(gp, "prred.o");
  add_obj_file(gp, "cxrd.o");

  /* Add header files which contain state structure info for
   * the run-time primitives. */
  add_inc_file(gp, "power.h");
  add_inc_file(gp, "prred.h");
  add_inc_file(gp, "cxrd.h");

  /* Invoke the data distribution and data colection nodes */
  /* NOTE: Row distribute -> row collect does no transpose */
  cxrd_np = ngf_cxrd_node(gp, "dist", row, col);
  pr_np = ngf_prred_node(gp, "prred", row, col);

  /* Create a power compute node for every row in the matrix.
   * Connect it to the named ports on the distribute and
   * collect nodes.
   */
  for (i=0; i < row ; i++)
    {
     (void)sprintf(temp, "power%d", i);
     power_np = ngf_power_node(gp, temp, col);
     (void)sprintf(temp, "out%d", i);
     add_edge(gp, cxrd_np, temp, power_np, "in", 0);
     (void)sprintf(temp, "in%d", i);
     add_edge(gp, power_np, "out", pr_np, temp, 0);
    }

  return(gp); /* Return the prfan graph */
}
```

The following function calls the previously described functions and builds the ABI. The row and col parameters specify the size of the input matrix and therefore, the size of the flowgraph. You specify these numbers at runtime. The function is called by main() and returns a pointer to the final graph.

```
/*
 * abi_graph() : Generate an Adaptive Beam Interpolation graph by
 * combining a row distribute & column collect complex incr fan,
 * with a row distribute , parallel row reduce power fan.
 */

GRAPH *
abi_graph(name, row, col)
     char *name;
     int row;
     int col;
{
  GRAPH *gp;
  NODE *n1, *n2;
```

```
/* allocate space for the main graph */
  gp = alloc_graph(name);

  /* Add the two graphs as sub-graph NODE's.
   * The input ports of the source node become the input
   * ports of the subgraph node pointer. Also, the output
   * ports of the sink node become the output ports of the
   * subgraph node.
   */
  n1 = add_gnode(gp, rcfan_graph("rcfan", row, col));

  /* NOTE: The output from the first subgraph is a TRANSPOSED
   * matrix. Therefore, the matrix sizes to the second subgraph
   * must be reversed.
   */
  n2 = add_gnode(gp, prfan_graph("prfan", col, row));

  /* connect the two subgraph nodes with an edge */
  add_edge(gp, n1, "out" , n2, "in", 0);

  /* return the main graph */
  return(gp);
}
```

The main program follows. It checks the user run-time input (size of the input matrix). The program then calls the abi_graph() function to create the graph. It then either prints a readable graph on standard out or emits the intermediate representation, depending on the DEBUG setting. It also opens the file for output and passes it to emit_graph().

```
/*
 * abi-gen.c : The flowgraph generator file for Adaptive
 * Beam Interpolation.  This file runs on the host to
 * create the flowgraph for input into the Assign compiler.
 * The Node Generation Functions are used to create the graph
 * one node at a time. These functions are found in files
 * starting with "ngf_".
 */

#include <stdio.h>
#include <strings.h>
#include <malloc.h>
#include <assign/assign.h>

extern void exit();

#define OUT_NAME_LEN \
  (strlen(argv[0])-strlen("-gen")+strlen(".graph")+1)
#define PRINT_LEVEL 0
#define DEBUG 0

/* Forward declarations of any future functions */
GRAPH *abi_graph();
GRAPH *rcfan_graph();
GRAPH *prfan_graph();

NODE *ngf_cxrd_node();
NODE *ngf_cxcc_node();
```

```
NODE *ngf_cxincr_node();
NODE *ngf_prred_node();
NODE *ngf_power_node();

main(argc, argv)
int argc;
char **argv;
{
  int row,col;
  FILE *out_file;
  char *out_name;

  /* Error checking on the input parameters */
  if (argc != 3)
    {
      (void)fprintf(stderr, "usage: abi-gen <fan size> <data
   size>\n");
      exit(1);
    }
  if ((row = atoi(argv[1])) < 1)
    {
      (void)fprintf(stderr, "abi-gen:bad size (matrix col size)\n");
      exit(2);
    }
  if ((col = atoi(argv[2])) < 1)
    {
      (void)fprintf(stderr, "abi-gen:bad size (matrix row size)\n");
      exit(3);
    }

  /* Open a file for the intermediate representation. It must end in
   * ".graph"
   */
  out_name=(char *)malloc((unsigned)OUT_NAME_LEN);
  out_name=(char *)getpref(argv[0]);
  out_name=(char *)strcat(out_name,".graph");
  out_file=fopen(out_name,"w");
  if (out_file==(FILE *)NULL)
    {
      (void)fprintf(stderr, "Unable to open output file '%s' for writing
output\n",out_name);
      exit(4);
    }

#if DEBUG
  /* Print a readable output to stdout */
  print_graph(abi_graph("abi",row,col), PRINT_LEVEL, stdout);
#else
  /* Or emit the intermediate representation to the opened file */
  emit_graph(abi_graph("abi", row, col), out_file);
#endif
```

```
    return(0);
  }
```

Now that the flowgraph generator and the primitives are defined, the input can be compiled and executed. Chapter 3 describes how to compile and execute your application using the Assign driver.