For: dwight
Printed on: Fri, Jul 16, 1993 09:52:18
From book: manual
Document: appb
Last saved on: Wed, Jul 8, 1992 14:19:36

Appendix B: ABI Example Routines

This appendix contains the complete code for the ABI example, parts of which were shown in Chapter 2.

```
/* cxrd.h : state structure for complex row distribute */
struct state_cxrd
{
  int n;
  int cnt;
};
```

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

/* cxrd.c :
 * Input and Output functions for Complex
 * Row Distribute data node.
 */

in_cxrd(inp, sp)
complex *inp;
struct state_cxrd *sp;
{
  int i;

  for (i=0; i<sp->n; i++)
    {
      inp[i].r = (float)i + sp->cnt*100;
      inp[i].i = inp[i].r;
    }
  sp->cnt++; /* Count the number of columns */
}

out_cxrd(outp, sp)
complex *outp;
struct state_cxrd *sp;
{
  int i;

  printf("ROW %d:\t (%6.1f,%6.1f) (%6.1f,%6.1f) ... (%6.1f,%6.1f)
(%6.1f,%6.1f)\n", sp->cnt++,
        outp[0].r,outp[0].i, outp[1].r,outp[1].i,

outp[sp->n-2].r,outp[sp->n-2].i,outp[sp->n-1].r,outp[sp->n-1].i);
}
```

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

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

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

```
/* cxcc.h : state structure for
 * complex column collecct.
 */
struct state_cxcc
{
  int n;
  int cnt;
};
```

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

/* cxcc.c :
 * Input and output functions for Complex Column
 * Collect data node. These are defined, but
 * may not ever be used */

in_cxcc(inp, sp)
complex *inp;
struct state_cxcc *sp;
{
  int i;

  for (i=0; i<sp->n; i++)
    {
      inp[i].r = (float)i;
      inp[i].i = inp[i].r;
    }
  sp->cnt++;
}

out_cxcc(outp, sp)
complex *outp;
struct state_cxcc *sp;
{
  int i;

  printf("ROW %d:\t (%6.1f,%6.1f) (%6.1f,%6.1f) ... (%6.1f,%6.1f)
(%6.1f,%6.1f)\n", sp->cnt++,
        outp[0].r,outp[0].i, outp[1].r,outp[1].i,

outp[sp->n-2].r,outp[sp->n-2].i,outp[sp->n-1].r,outp[sp->n-1].i);
```

```
}
/* Distribute and Collect data nodes have no primitives */
```

```
/* state structure for power.c */
struct state_power
{
   /* Size of the vector to be operated on */
  int size;
};
```

```
#include <assign/assign.h>
#include <math.h>
#include ”power.h”

/* power.c : primitive execution for power function.
 * The ”power” operator: power = SQRT( (real)^2 + (imaginary)^2 )
 */
void
prim_power(inp, outp, sp)
     complex *inp;
     float *outp;
     struct state_power *sp;
{
  int i;

  for (i=0; i < sp->size; i++)
    {
     outp[i] = (float)sqrt( (inp[i].r*inp[i].r) + (inp[i].i*inp[i].i)
);
    }
}
```

```
/* prrred.h : state structure for prred.c */
struct state_prred
{
  /* Size of the output vector, and each
   * input vector to be reduced.
   */
  int n;
};
```

```
#include "prred.h"

/*
 * input routine for a prred node
 */
in_prred(inp, sp)
float *inp;
struct state_prred *sp;
{
  int i;

  for (i=0; i<sp->n; i++)
    inp[i] = (float)i;
}

/*
 * output routine for a prred node
 */
out_prred(outp, sp)
float *outp;
struct state_prred *sp;
{
  int i;

  /* Just print every element of the output vector */
  for (i=0; i<sp->n; i++)
    printf("output[%2d] : %12.6f\n", i, outp[i]);
}

/*
 * plus reduce of two input vectors of floats.
 */
prim_prred(inp0, inp1, outp, sp)
float *inp0, *inp1, *outp;
struct state_prred *sp;
{
  int i;

  /* Look in the state structure for the number of
   * elements in each input vector.
   */
  for (i=0; i < sp->n; i++)
    outp[i] = inp0[i] + inp1[i];
}
```

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

/*
 * ngf_cxrd_node(): parallel row distribute of complex
 *  data.  This node will distribute ROW vectors of
 *  COL elements each, where ROW is the number
 *  of rows in the input matrix, and COL is the number
 *  of columns.
 */
NODE *ngf_cxrd_node(gp, name, ROW, COL)
GRAPH *gp;
char *name;
int ROW, COL;
{
  int i;
  NODE *np;
  char port_name[50], init[50];

  /* Create a new node in graph "gp" with name "name".
   * The node will be a Parallel Row Distribute
   * (PRD_NODE) type.
   */
  np = add_dnode(gp, name, PRD_NODE);

  /* A distribute node has one input port, of type
   * COMPLEX and a consume amount equal to the size
   * of the input matrix (ROW * COL)
   */
  add_in_port(np, "in", COMPLEX, ROW*COL);

  /* Row distribute will need an output port for every
   * row in the matrix.  Each row vector will be have
   * COL elements.
   */
  for (i=0; i < ROW; i++)
    {
      /* every output port will need a unique name */
      (void)sprintf(port_name,"out%d", i);
      /* produce amount of COL elements of type COMPLEX */
      add_out_port(np, port_name, COMPLEX, COL);
    }

  /* Associate a state structure called "state_cxrd" and
   * initialize its size.  "" means there is no dynamic
   * initialization function.
   */
  (void)sprintf(init, "%d,0", COL);
  add_state(np, "state_cxrd", init, "");

  /* We can still associate IO functions, even if this will
```

```
   * never be a source or sink node. */
  add_io(np, "in_cxrd", "out_cxrd");

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

```
#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. */
  add_in_port(np, "in", COMPLEX, n);
  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 */
}
```

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

/*
 * ngf_cxcc_node(): parallel column collect.
 *  This node will collect COL vectors made up of ROW
 *  elements each.  Each vector will become a column
```

```
 *  in the final matrix.  This resulting matrix will
 *  therefore be ROW x COL in size.
 *  The type of data used is COMPLEX.
 */
NODE *ngf_cxcc_node(gp, name, ROW,COL)
GRAPH *gp;
char *name;
int ROW, COL;
{
  int i;
  NODE *np;
  char port_name[50], init[50];

  /* Create a new node in graph "gp" with name "name".
   * The node will be a Parallel Column Collect
   * (PCC_NODE) type.
   */
  np = add_cnode(gp, name, PCC_NODE);

  /* Add an input port for every input vector */
  for (i=0; i < COL; i++)
    {
      /* Give the new port a unique name */
      (void)sprintf(port_name, "in%d", i);
      /* The input vector consumes ROW elements of type COMPLEX */
      add_in_port(np, port_name, COMPLEX, ROW);
    }

  /* There is one output port, for the entire collected
   * matrix of size ROW*COL, also of type COMPLEX */
  add_out_port(np, "out", COMPLEX, ROW*COL);

  /* Associate a state structure of name "state_cxcc" with
   * this node.  Initialize the state struct to "init"
   * which is the number of elements in each row of the final
   * matrix ( == # of columns == COL) and a counter to keep
   * track of the number of columns in the final matrix.
   */
  (void)sprintf(init, "%d,0", COL);
  add_state(np, "state_cxcc", init, "");

  /* We can still associate IO functions, even if this will
   * never be a source or sink node.
   */
  add_io(np, "in_cxcc", "out_cxcc");

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

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

/*
 * ngf_power_node() : Invoke a ready-to-use
 * node which performs the operation in ”prim_power”.
 */
NODE *
ngf_power_node(gp, name,  size)
     GRAPH *gp;
     char *name;
     int size;
{
  NODE *np;
  char ilist[50];

  /* Create the new node, and use the function called
   * ”prim_power” to perform the run-time computation.
   * size*400 is the estimated number of clocks for
   * one invocation of the function.
   */
  np = add_node(gp, name, ”prim_power”, size*500);

  /* This node will ahve one input and one output port.
   * The input data is of type COMPLEX and the output
   * is of type FLOAT. */
  add_in_port(np, ”in”, COMPLEX, size);
  add_out_port(np, ”out”, FLOAT, size);

  /* Associate a state structure and initialize it to be
   * the size of the input vector. */
  (void)sprintf(ilist, ”%d”, size);
  add_state(np, ”state_power”, ilist, ””);

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

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

/*
 * ngf_prred_node(): parallel plus reduce of ROW vectors
 * of COL elements each.  (A summing row reduction).
 * Similar to a row collect, except an operation is
 * performed on the vectors to yield one result vector.
 */
NODE *ngf_prred_node(gp, name, ROW, COL)
GRAPH *gp;
char *name;
int ROW, COL;
```

```
{
  int i;
  NODE *np;
  char port_name[50], init[50];

  /* Create a new node in graph "gp" with name "name".
   * The node will be a Parallel Reduce
   * (PR_NODE) type.   The reduction operation is
   * given in a function called "prim_prred".
   * The estimated number of clocks for this primitive
   * is COL*10.
   */
  np = add_rnode(gp, name, PR_NODE, "prim_prred", COL*10);

  /* Like the row collect, an input port is needed for
   * each vector.  Each will consume COL elements
   * of type FLOAT.
   */
  for (i=0; i < ROW; i++)
    {
      /* Each input port will need a unique name */
      (void)sprintf(port_name, "in%d", i);
      /* Every input vector must be of the same size */
      add_in_port(np, port_name, FLOAT, COL);
    }

  /* The reduce node has one output, which MUST be the
   * same size as an input vector. */
  add_out_port(np, "out", FLOAT, COL);

  /* Associate a state structure called "state_prred"
   * an initialize its size. */
  (void)sprintf(init, "%d", COL);
  add_state(np, "state_prred", init, "");

  /* We can still associate IO functions, even if this will
   * never be a source or sink node. */
  add_io(np, "in_prred", "out_prred");

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

```
/*
 * 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);
#else
  /* Or emit the intermediate representation to the opened file */
  emit_graph(abi_graph(”abi”, row, col), out_file);
#endif

  return(0);
}
```

```
/*
 * 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);
}
```

```
/*
 * rcfan_graph - build a parallel fan of "row" incr
 * nodes 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 colection 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 of 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 */
}
```