Chapter 6

Sample Programs

This chapter presents three programming examples in C*. If you have C* installed at your site, these programs may be available in the C* examples directory. Check with your system administrator for the location of this directory.

6.1 Julia

This program displays a Julia set. A Julia set is a kind of fractal, obtained by iteratively applying a function at points in a complex plane until the function goes off to infinity. The number of iterations at each point determines how the point is displayed.

Note these points about the program:

- In the program, the shape **`julia`** is the 1024-by-1024 plane in which the set is to be displayed. The points in the plane are obtained by initializing parallel variables using **`pcoord`**.
- The **`where`** statement makes a position inactive when the function goes off to infinity for the complex number at that position; in cases where this does not happen, the **`for`** loop stops the iterations at 500.
- The program requires no communication among parallel variables.
- The CM-200 version includes calls to the CM graphics library to display the Julia set. When compiling the program, you must link to the **`cmsr`** and **`X11`** libraries; the comments at the beginning of the program show how to do this. The program uses the CM Generic Display Interface to prompt the user for the device on which the Julia set is to be displayed:

a framebuffer or an X Window System. **NOTE**: You must use Version 2.0 of the graphics library for this program to compile correctly.

- The CM-5 version uses calls to CMX11. Use these commands to compile and link:

```
% cs -vu julia.cs -lcmx_cm5_vu_sp -lXm -lXt -lXext \
 -lX11
```

CM-200 Version

Here is the CM-200 version:

```
/*
 *
 * This program computes a Julia set (a fractal) in the square region of
 * the complex plane defined by the point (LEFT_SIDE,TOP_SIDE) and the
 * side length SI.
 *
 * The fractal is computed on an N x N grid, with each pixel being
 * handled by a separate virtual processor on the CM. Each pixel will be
 * assigned a value in the range 0 to RES.
 *
 *
 * The fractal is computed by iterating the complex function
 * f(Z) = Z*Z + C for each coordinate in the above region of
 * the complex plane and counting the number of iterations
 * required to cause the function's value to become unbounded
 * at each coordinate. The set of iteration counts is then used as
 * the set of values that is displayed on the output device.
 *
 * NOTE: Complex numbers are handled as pairs of real numbers.
 *
 * To compile this example use the following command:
 *
 *          cs -o julia julia.cs  -lcmsr -lX11
 *
 */

#include <stdio.h>
#include <cm/cmsr.h>
#include <cm/display.h>

/*
```

```
 * parameters defining the size, position, and resolution of the image
 * to be computed
 */

#define N           512
#define RES         500
#define SI          3.0f
#define TOP_SIDE    -1.53f
#define LEFT_SIDE   -1.50f

/* the components of the complex constant, C */

#define CR      0.320f
#define CI      0.043f

main()
{
   shape [N][N]julia ;

   float:julia zr,  zi  ;        /* real and imaginary components of z */
   float:julia zrs, zis ;        /* squares of same                  */
   int:julia   ittr     ;        /* number of cycles before cell done  */

   float cell_size ;
   int   i ;

   /* Set up output device */

   initialize_display() ;

   /* for all pixels simultaneously */

   with ( julia )
   {
      ittr = 0     ;
      zrs  = 0.0f   ;
      zis  = 0.0f   ;

      cell_size = SI / N ;

      /* Give each processor the coordinate in the complex plane that
         it should use. */

      zr = pcoord(1) * cell_size + TOP_SIDE;
      zi = pcoord(0) * cell_size + LEFT_SIDE;

      for(i=0; i<RES; i++)
      {
         /* Perform the iteration for those points in the plane where
            the iteration has not produced an unbounded value. There
```

```
            is a result from the theory of complex numbers that states
            that the values produced by the iteration will remain
            bounded so long as the magnitude of the function's value
            does not exceed 2. */

         where ( zrs+zis <= 4.0f )
         {
            /* iterate f(Z) = Z*Z + C */

            zrs = zr * zr ;
            zis = zi * zi ;
            zi  = 2.0f * zr * zi + CI ;
            zr  = zrs - zis + CR ;

            /* Update the iteration count for those processors that
               are still handling unbounded iterations. */

            ittr = i ;
         }
      }

      /* Done. Display the resulting fractal. */

      CMSR_write_to_display(&ittr);
   }

   if (CMSR_display_type()==CMSR_x_display)
   {  printf("Type return to quit");
      getchar();
   }
}

initialize_display()
{
   int zoom ;

   CMSR_select_display_menu(8, N, N);

   if (CMSR_display_type()==CMSR_cmfb_display)
   {
      zoom = (1024/N) - 1;
      CMFB_set_zoom (CMSR_cmfb_display_display_id(), zoom, zoom, 0);
      CMSR_set_display_offset(128/(zoom+1), 0);
   }
}
```

CM-5 Version

Here is the CM-5 version:

```
/*
 *
 * This program computes a Julia set (a fractal) in the square region of
 * the complex plane defined by the point (LEFT_SIDE,TOP_SIDE) and the
 * side length SI.
 *
 * The fractal is computed on an N x N grid, with each pixel being
 * handled by a separate virtual processor on the CM. Each pixel will be
 * assigned a value in the range 0 to RES.
 *
 *
 * The fractal is computed by iterating the complex function
 * f(Z) = Z*Z + C for each coordinate in the above region of
 * the complex plane and counting the number of iterations
 * required to cause the function's value to become unbounded
 * at each coordinate. The set of iteration counts is then used as
 * the set of values that is displayed on the output device.
 *
 * NOTE: Complex numbers are handled as pairs of real numbers.
 */

#include <stdio.h>
#include <cm/cmxdisplay.h>
#include <cm/cmx-cs.h>

/*
 * parameters defining the size, position and resolution of the image
 * to be computed
 */

#define N          512
#define RES        256
#define SI         3.0f
#define TOP_SIDE   -1.53f
#define LEFT_SIDE  -1.50f

/* the components of the complex constant, C */

#define CR      0.320f
#define CI      0.043f

main(argc, argv)
{
    shape [N][N]julia ;
```

```
  float:julia zr,  zi  ;      /* real and imaginary components of z */
  float:julia zrs, zis ;      /* squares of same                    */
  int:julia   ittr     ;      /* number of cycles before cell done  */
  CMX_display_t display;
  Widget widget;
  int depth, mask;
  float cell_size ;
  int   i ;

  /* Set up X display */
  CMXSetArgcArgv(argc, argv);      /* Enable X command-line options */
  CMXSetXWindowTitle("Julia Set");
  display = CMXCreateSimpleDisplay(256, N, N);
  widget = CMXGetWidget(display);
  depth = CMXGetDepth(widget);
  depth = (depth < 8) ? 1 : 8;  /* CMXPutImage supports only 1, 8 */
  mask = (1 << depth) - 1;

  /* For all pixels simultaneously */
 with ( julia )
  {
    ittr = 0      ;
    zrs  = 0.0f   ;
    zis  = 0.0f   ;

    cell_size = SI / N ;

    /* give each processor the coordinate in the complex plane that
       it should use */

    zr = pcoord(1) * cell_size + TOP_SIDE;
    zi = pcoord(0) * cell_size + LEFT_SIDE;

    printf("Iteration ");
    for(i=0; i<RES; i++)
    {
     /* Perform the iteration for those points in the plane where
        the iteration has not produced an unbounded value. There
        is a result from the theory of complex numbers that states
        that the values produced by the iteration will remain
        bounded so long as the magnitude of the function's value
        does not exceed two. */

        printf("%d ", i);
        fflush(stdout);
        where ( zrs+zis <= 4.0f )
        {
          /* iterate f(Z) = Z*Z + C */

          zrs = zr * zr ;
```

```
                zis = zi * zi ;
                zi  = 2.0f * zr * zi + CI ;
                zr  = zrs - zis + CR ;

            /* update the iteration count for those processors that are
               still handling unbounded iterations */

                ittr = i ;
             }
         /* done. display the low <depth> bits of the iteration count. */
            CMXPutImage(CMXGetDisplay(display), CMXGetDrawable(display),
            CMXGetGC(display), ittr & mask, depth, 0, 0, 0, 0,
            CMXGetWidth(widget), CMXGetHeight(widget));
        }
        printf("\n");

        }

        printf("Type return to quit");
        getchar();
}
```

Output

Figure 17 shows what the output looks like.

Figure 17. Output of the Julia program.

6.2 Prime Number Sieve

This program finds the prime numbers in a set of numbers. Note these points about the program:

- The function **find_primes** uses the keyword **current** for its parallel variables so that it can be used with any current shape.
- The program uses the **<?=** reduction operator and **pcoord** to obtain the lowest coordinate among active positions of the current shape.
- It uses the parallel conditional expression to initialize the parallel variable **is_candidate**.
- It contains nested **where** statements. Inside the first **where** statement, the program finds the minimum value still active; the second **where** statement makes all multiples of this minimum value active and removes them as candidates to be prime numbers.
- It uses scalar left indexing to select individual elements of parallel variables.

Here is the code:

```
#define MAXIMUM_PRIME 16384

#define FALSE 0
#define TRUE 1
#define FIRST_PRIME 2
/*                                                              */
/*             Function to find prime numbers                   */
/*                                                              */
/*  Parameters:                                                 */
/*                                                              */
/*     A pointer, "is_prime_p," to a one-dimensional parallel   */
/*     char that will have non-zero elements in all positions   */
/*      where the index is a prime number                       */
/*                                                              */
/*  Side effects:                                               */
/*                                                              */
/*     find_primes alters the one-dimensional parallel bool     */
/*      that is pointed to by "is_prime_p."                     */
/*                                                              */
/*  Calling constraints:                                        */
/*                                                              */
/*     The shape of the parallel bool pointed to by             */
/*     "is_prime_p" must be the current shape and all           */
```

```
/*      positions must be active                                  */
/*                                                                */
/*  Algorithm:                                                    */
/*                                                                */
/*      This function will use the Sieve of Eratosthenes to       */
/*      find the prime numbers.  That is, it will iterate         */
/*      through all numbers which are indices to the one-         */
/*       dimensional parallel bool                                */
/*                                                                */
void find_primes(bool:current *is_prime_p) {
   bool:current is_candidate;
   int minimum_prime;

   *is_prime_p = FALSE;

   is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;

   do
      where(is_candidate) {
         minimum_prime = <?= pcoord(0);
         where(!(pcoord(0) % minimum_prime))
            is_candidate = FALSE;
         [minimum_prime]*is_prime_p = TRUE;
         }
   while(|= is_candidate);
}

main() {
   shape [MAXIMUM_PRIME]s;

   bool:s is_prime;
   int i;

   with(s)
      find_primes(&is_prime);
   for(i=0; i<MAXIMUM_PRIME; i++)
      if([i]is_prime)
         printf("The next prime number is %d\n", i);
}
```

The program's output begins:

```
The next prime number is 2
The next prime number is 3
The next prime number is 5
The next prime number is 7
```

... and so on.

6.3 Shuffle

This program provides a perfect shuffle of a deck of cards — that is, the deck is split into two halves, and the cards from each half are alternated. The program continues to shuffle the deck until the original order is obtained again.

Note these points about the program:

- The program uses general communication to rearrange the deck into its new order after each shuffle.
- The CM-200 version uses the shape **physical** — this is a predeclared shape name in C*. The shape **physical** is always of rank 1; its number of positions is the number of physical processors to which the program is attached when it runs on a CM. The program then restricts the context to those positions whose coordinates are less than or equal to **DECK_SIZE**. This is a typical way of changing the number of positions in your shape in CM-200 C* programs, if you don't want a power of 2. Note that the CM-5 version doesn't require a **where** statement, since it can use a shape of 52 positions.

CM-200 Version

Here is the version for CM-200 C*:

```
#include <stdio.h>
#define DECK_SIZE 52
/*                                                                  */
/*                  Function to print a deck of cards               */
/*                                                                  */
/*  Parameters:                                                     */
/*                                                                  */
/*      A parallel int, "deck," of physical shape, the first        */
/*      DECK_SIZE entries of which contain card numbers.            */
/*                                                                  */
/*  Side effects:                                                   */
/*                                                                  */
/*      The contents of "deck" are printed (allowing three          */
/*      columns per int) followed by a new line.                    */
/*                                                                  */
/*  Calling constraints:                                            */
/*                                                                  */
/*      The first DECK_SIZE entries of physical shape should        */
/*      be active (because "deck" is passed by value) and           */
```

```
/*      DECK_SIZE should be less than or equal to               */
/*      dimof(physical, 0).                                     */
/*                                                              */
/*  Algorithm:                                                  */
/*                                                              */
/*      Self-evident                                            */
/*                                                              */
void print_deck(int:physical deck)
{
   int i;

   for(i = 0; i < DECK_SIZE; i++)
      printf("%3d", [i]deck);
   printf("\n\n");
}

/*                                                              */
/*             Main function to shuffle a deck of cards         */
/*                                                              */
/*  Parameters:                                                 */
/*                                                              */
/*      None                                                    */
/*                                                              */
/*  Description:                                                */
/*                                                              */
/*      This program takes a pseudo deck of cards and           */
/*      repeatedly performs the perfect shuffle transformation  */
/*      on the deck until it is back in its original order.     */
/*                                                              */
/*      The perfect shuffle is performed by cutting the deck    */
/*      in the middle and then interleaving cards from the      */
/*      two half decks.  For example, if the original deck      */
/*      contained the cards 0, 1, 2, 3, 4, and 5, the first cut */
/*      deck would contain 0, 1, and 2, and the second cut      */
/*      deck would contain 3, 4, and 5.  Interleaving these     */
/*      two decks results in 0, 3, 1, 4, 2, and 5.              */
/*                                                              */
/*  Side effects:                                               */
/*                                                              */
/*      The program performs output.                            */
/*                                                              */
/*  Program constraints:                                        */
/*                                                              */
/*      The number of cards in the deck, DECK_SIZE, should be   */
/*      less than or equal to dimof(physical, 0).               */
/*                                                              */
/*  Algorithm:                                                  */
/*                                                              */
/*       A "send" is used to perform the shuffle.               */
/*                                                              */
```

```
main()
{
   int:physical original_deck, deck, shuffling_order;

   /* offset is the half-way point in the deck (for cutting purposes) */
   int offset = (DECK_SIZE+1)/2, number_shuffles = 0;

with(physical)
   /* only positions in the deck are left active */
   where((deck = original_deck = pcoord(0)) < DECK_SIZE) {
      printf(”original deck:”);
      print_deck(original_deck);

      /* generate the perfect shuffle transformation: positions in the
      first half of the deck are to be sent
      to consecutive even positions
      in the shuffled deck; whereas positions in the second half of the
      deck are to be sent to consecutive odd positions in
      the shuffled deck*/

      shuffling_order = (2*deck < DECK_SIZE) ? (2*deck)
                          : (2*(deck-offset)+1);

      printf(”shuffle order:”);
      print_deck(shuffling_order);

      do {
         /* perform the shuffle */
         [shuffling_order]deck = deck;

         /* print the shuffled deck and an incremented
         sequence number */
         printf(”%3d:”, ++number_shuffles);
         print_deck(deck);
      /* continue to shuffle until the deck is in its original order */
      } while(|=(deck != original_deck));

      /* print the number of shuffles required */
      printf(”\nNumber of shuffles = %d\n”, number_shuffles);
   }
}
```

CM-5 Version

Here is the version for CM-5 C*:

```
#include <stdio.h>
#define DECK_SIZE 52
shape [DECK_SIZE]deck_shp;

/*                                                                  */
/*                Function to print a deck of cards                 */
/*                                                                  */
/*  Parameters:                                                     */
/*                                                                  */
/*      A parallel int, "deck," of physical shape, the first        */
/*      DECK_SIZE entries of which contain card numbers.            */
/*                                                                  */
/*  Side effects:                                                   */
/*                                                                  */
/*      The contents of "deck" are printed (allowing three          */
/*      columns per int) followed by a new line.                    */
/*                                                                  */
/*  Calling constraints:                                            */
/*                                                                  */
/*      The first DECK_SIZE entries of physical shape should        */
/*      be active (because "deck" is passed by value) and           */
/*      DECK_SIZE should be less than or equal to                   */
/*      dimof(physical, 0).                                         */
/*                                                                  */
/*  Algorithm:                                                      */
/*                                                                  */
/*      Self-evident                                                */
/*                                                                  */
void print_deck(int:deck_shp deck)
{
   int i;

   for(i = 0; i < DECK_SIZE; i++)
      printf("%3d", [i]deck);
   printf("\n\n");
}

/*                                                                  */
/*             Main function to shuffle a deck of cards             */
/*                                                                  */
/*  Parameters:                                                     */
/*                                                                  */
/*      None                                                        */
/*                                                                  */
/*  Description:                                                    */
```

```
/*                                                                  */
/*      This program takes a pseudo deck of cards and               */
/*      repeatedly performs the perfect shuffle transformation      */
/*      on the deck until it is back in its original order.        */
/*                                                                  */
/*      The perfect shuffle is performed by cutting the deck        */
/*      in the middle and then interleaving cards from the          */
/*      two half decks.  For example, if the original deck          */
/*      contained the cards 0, 1, 2, 3, 4, and 5, the first cut     */
/*      deck would contain 0, 1, and 2, and the second cut          */
/*      deck would contain 3, 4, and 5.  Interleaving these         */
/*      two decks results in 0, 3, 1, 4, 2, and 5.                  */
/*                                                                  */
/*  Side effects:                                                   */
/*                                                                  */
/*      The program performs output.                                */
/*                                                                  */
/*  Program constraints:                                            */
/*                                                                  */
/*      The number of cards in the deck, DECK_SIZE, should be       */
/*      less than or equal to dimof(physical, 0).                   */
/*                                                                  */
/*  Algorithm:                                                      */
/*                                                                  */
/*       A "send" is used to perform the shuffle.                   */
/*                                                                  */

main()
{
   int:deck_shp original_deck, deck, shuffling_order;

   /* offset is the half-way point in the deck (for cutting purposes) */
   int offset = (DECK_SIZE+1)/2, number_shuffles = 0;

with(deck_shp) {
   /* only positions in the deck are left active */
   deck = original_deck = pcoord(0)
   printf("original deck:");
   print_deck(original_deck);

   /* generate the perfect shuffle transformation: positions in the
      first half of the deck are to be sent
      to consecutive even positions
      in the shuffled deck; whereas positions in the second half of the
      deck are to be sent to consecutive odd positions in
      the shuffled deck */

   shuffling_order = (2*deck < DECK_SIZE) ? (2*deck)
                        : (2*(deck-offset)+1);
```

```
   printf("shuffle order:");
   print_deck(shuffling_order);

   do {
      /* perform the shuffle */
      [shuffling_order]deck = deck;

      /* print the shuffled deck and an incremented
         sequence number */
      printf("%3d:", ++number_shuffles);
      print_deck(deck);
   /* continue to shuffle until the deck is in its original order */
   } while(|=(deck != original_deck));

   /* print the number of shuffles required */
   printf("\nNumber of shuffles = %d\n", number_shuffles);
   }
}
```

Output

Here is the output from the program:

```
original deck:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51

shuffle order:  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36
38 40 42 44 46 48 50  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33
35 37 39 41 43 45 47 49 51
  1:  0 26  1 27  2 28  3 29  4 30  5 31  6 32  7 33  8 34  9 35 10 36 11
37 12 38 13 39 14 40 15 41 16 42 17 43 18 44 19 45 20 46 21 47 22 48 23
49 24 50 25 51

  2:  0 13 26 39  1 14 27 40  2 15 28 41  3 16 29 42  4 17 30 43  5 18 31
44  6 19 32 45  7 20 33 46  8 21 34 47  9 22 35 48 10 23 36 49 11 24 37
50 12 25 38 51

  3:  0 32 13 45 26  7 39 20  1 33 14 46 27  8 40 21  2 34 15 47 28  9 41
22  3 35 16 48 29 10 42 23  4 36 17 49 30 11 43 24  5 37 18 50 31 12 44
25  6 38 19 51

  4:  0 16 32 48 13 29 45 10 26 42  7 23 39  4 20 36  1 17 33 49 14 30 46
11 27 43  8 24 40  5 21 37  2 18 34 50 15 31 47 12 28 44  9 25 41  6 22
38  3 19 35 51
```

```
  5:  0  8 16 24 32 40 48  5 13 21 29 37 45  2 10 18 26 34 42 50  7 15 23
31 39 47  4 12 20 28 36 44  1  9 17 25 33 41 49  6 14 22 30 38 46  3 11
19 27 35 43 51

  6:  0  4  8 12 16 20 24 28 32 36 40 44 48  1  5  9 13 17 21 25 29 33 37
41 45 49  2  6 10 14 18 22 26 30 34 38 42 46 50  3  7 11 15 19 23 27 31
35 39 43 47 51

  7:  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44
46 48 50  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41
43 45 47 49 51

  8:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51

Number of shuffles = 8
```