Prof. Jingke Li (lij@pdx.edu); Classes/Labs: TR 1400-1550 @FAB 150, 88-10; Office Hours: TR 1300-1400 @FAB 120-06

Assignment 3: Programming with Chapel (Due Thursday, 5/23/24)

This assignment aims at practicing programming in the emerging parallel language, Chapel. You are going to write programs for two problems, oddeven sort and producer-consumer, each with two versions. For this assignment, all students do the same work.

1 Odd-Even Sort

The file `oddeven.c` contains an implement of the odd-even sort algorithm. Read and understand this program. If you are not familiar with this algorithm, find info online to refresh yourself, or wait for next week's lecture.

Compile the program with the `DEBUG` switch on, so that you can see the step-by-step actions of this algorithm. Run the program several times. Do you observe any case where the array being sorted before all iterations are executed?

Compile the program with the `WORST` switch on. Now the program sorts a specific array, which holds `1..N` in the reverse order. This is the worst case for odd-even sort. Verify that it takes all the iterations to sort this array.

Here are your tasks:

1. Convert the C program to a data parallel version in Chapel, `oddeven1.chpl`. This conversion is mostly straightforward, similar to what you did in this week's lab. Make sure you represent parallelizable loops with `forall`. The following are the specific requirements:

   - Faithfully convert most of the functions, but replace `print_array` with a single `writeln` statement and replace `swap` with a swap statement. (For the latter, you may find an example in the Chapel lecture.)
   - Define three configurable constants `DEBUG`, `WORST`, and `N`, to correspond to the three items with the same name.

   A skeleton program file `oddeven1.chpl` is provided. You may copy it over to `oddeven1.chpl` and use it as your starting version.

2. Write a second version, `oddeven2.chpl`, to add early termination to the program – the program should terminate as soon as there is no swaps in either odd or even phases. (*Hint:* This can be done through updating a global flag whenever a swap happens.) A new requirement:

   - Make the `oddeven_sort` function return the number of iterations it takes to sort. If this number is less than `(N+1)/2`, the `main` function prints out a message:

     ```
     "Early terminiation saved k rounds!"
     ```

     with a proper `k` value.

   (*Hint:* You may need to run the program with a large `N` value to see early termination happen.)

2 Producer-Consumer

The file `cqueue.chpl` contains a representation of circular queue data structure. The queue items are stored in a buffer array; when the end of the buffer is reached, it continues back from the beginning. Read and understand this program. Pay special attention to the `sync` variable declarations, especially the buffer array, which means every array element is a self-sync item, allowing only alternating reads and writes.

1. The file `prodcons1.chpl` contains a skeleton of a producer-consumer program with one producer and one consumer. The total number of items to add and remove is represented by a configurable constant `numItems`, with a default value of 32. Your first task is to complete this program by defining both the producer and the consumer routines. The two routines should print out a line for each item added or removed:

   ```
   Producer added 28 from buf[7]
   consumer rem'd 21 from buf[0]
   ```

2. Your second task is to implement an extended version of the program in `prodcons2.chpl`. In this version you are to modify the previous version to allow multiple copies of the consumer routine to be created and run concurrently. Additional requirements are:

   - The consumer function now takes an integer parameter, serving as an ID to allow differentiation among multiple copies of the same function. So messages from this program will look like

     ```
     Consumer[1] rem'd item 21 to buf[7]
     Consumer[2] rem'd item 24 to buf[0]
     ```

   - The number of consumers is represented by a configurable constant, `numCons`, which has a default value of 2.
   - All copies of the consumer function compete to remove items from the queue, until all items are removed.
   - There is a new challenge: each consumer needs to know when to terminate. (*Hint:* Use a global `sync` variable to count total removed items.)
   - The producer and all the consumers should run concurrently.

   Test your programs with different parameter values.

Summary and Submission

Write a summary covering (1) status of your programs; (2) observations on your programs' behaviors (e.g. Does the early-termination oddeven sort terminates early? How early?); and (3) your experience with this assignment. Make a zip file containing all your programs and your write-up. Submit it through the "Assignment 3" folder on Canvas.

Appendix: Sample output from oddeven sort programs

```
linux> ./oddeven1
Init: 32 119 100 70 184 154 10 163 8 54
t=1:  32 70 119 100 154 10 184 8 163 54
t=2:  32 70 100 10 119 8 154 54 184 163
t=3:  32 10 70 8 100 54 119 154 163 184
t=4:  10 8 32 54 70 100 119 154 163 184
t=5:  8 10 32 54 70 100 119 154 163 184
Array is sorted.

linux> ./oddeven1 --WORST=true
Init: 10 9 8 7 6 5 4 3 2 1
t=1:  9 7 10 5 8 3 6 1 4 2
t=2:  7 5 9 3 10 1 8 2 6 4
t=3:  5 3 7 1 9 2 10 4 8 6
t=4:  3 1 5 2 7 4 9 6 10 8
t=5:  1 2 3 4 5 6 7 8 9 10
10 element array is sorted.

linux> ./oddeven1 --DEBUG=false
10 element array is sorted.
```

```
linux> ./oddeven1
Init: 251 73 195 1 106 51 23 102 230 59
t=1:  73 1 251 51 195 23 106 59 102 230
t=2:  1 51 73 23 251 59 195 102 106 230
FAILED: a[3]=73, a[4]=23

linux> ./oddeven2
Init: 21 0 132 91 28 194 195 233 255 50
t=1:  0 21 91 28 132 194 195 50 233 255
t=2:  0 21 28 91 132 50 194 195 233 255
t=3:  0 21 28 50 91 132 194 195 233 255
t=4:  0 21 28 50 91 132 194 195 233 255
Early terminiation saved 1 rounds!
10 element array is sorted.
```