Prof. Jingke Li (FAB 120-06, lij@pdx.edu); Classes: M 16:40-18:30, W 16:40-17:20, Labs: W 17:25-18:55, all @ FAB 88-10.

Assignment 3: Programming with Chapel

(Due Wednesday, 2/26/20)

This assignment is to practice programming in the emerging parallel programming language, Chapel. You are going to re-implement several familiar parallel programs in Chapel, i.e. prime-finding, quicksort, and producer-consumer. Before starting on this assignment, you should complete Lab 6 first. As usual, CS515 students are required to complete one extra program (see details below).

1 Prime-Finding (`prime1.chpl`)

The file `prime.cpp` contains a sequential implementation of the sieve of Eratosthenes prime-finding algorithm. Your task is to implement a Chapel version. The only requirement is that the inner loop of the main loop nest is a `forall` loop.

Note: The Chapel program should look simpler than the C++ program, since there is no need to process command-line arguments; you may simply use configurible constants.

2 Quicksort (`qsort1.chpl`)

The file `qsort.cpp` contains a sequential implementation of the quicksort algorithm. Your task is to implement a Chapel version. The only requirement is to parallelize the recursive calls:

```
cobegin {
  quickSort(low, middle-1);
  quickSort(middle+1, high);
}
```

Again, you may use Chapel's language features to simply the program:

- No need to define a `swap` function; Chapel has a swap operator, `<=>`.
- No need to define a `printArray`; just use `writeln(a);` to print array `a`.

3 Producer-Consumer (`prodcons[123].chpl`)

Your task is to write the same three versions of produce-consumer programs, as in Assignment 1.

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.

Note: You should use this program as is, and not to modify anything.

3.1 Base Version (`prodcons1.chpl`)

In this version you are to write a producer routine to add items to a queue, and a consumer routine to remove items from the same queue. The specific requirements are:

- The total number of items to add and remove is represented by a configurable constant `numItems`, with a default value of 32.

- Both routines print out a line for each item added or removed, in the following form:

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

- The producer and the consumer routines must run concurrently, which can be achieved by write the `main()` function as follows:

```
proc main() {
  begin producer();
  consumer();
}
```

3.2 Extended Version (`prodcons2.chpl`)

In this version you are to modify the previous version to allow multiple copies of consumer routines 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.

3.3 Full Version (`prodcons3.chpl`)

(Optional for CS415 students, 10% extra points.) In this version you are going to further enhance the program by allowing multiple producers. Additional requirements are:

- The producer 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

```
Producer[1] added item 12 to buf[2]
Producer[2] added item 21 to buf[0]
```

- The number of producers is represented by a configurable constant, `numProd`, which has a default value of 2.
- The workload of adding `numItems` items to the queue is evenly partitioned among the producers; each handles `numItems/numProd` items. In the case `numProd` does not evenly divide `numItems`, the last producer will handle the extra items.
- All the producers and the consumers should run concurrently.

Test your programs with different parameter values.

Summary and Submission

Write a short (one-page) summary (in plain text or pdf) covering your experience with this assignment. Make a zip file containing all your programs and your write-up. Submit it through the “Assignment 3” dropbox on D2L.