Parallel Computer Architecture and Programming
Written Assignment 4

30 points total + 5 points EC. Due Monday, July 24 at the start of class.

Problem 1: Hash Table Parallelization (10 points)

Below is an implementation of a simple hash table. (The hash table uses a linked list to store elements that hash to each bin). The hash table has a function called `tableInsert` that takes two strings, and inserts both strings into the table only if **neither string already exists in the table**. Please implement `tableInsert` below in a manner that enables maximum concurrency. You may add locks wherever you wish. (And update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual per-bin list (notice we did not give you implementations for `findInList` and `insertInList`). **Careful, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. (HINT: deadlock!)**

```
struct Node {
  string value;
  Node* next;

};

struct HashTable {
   Node* bins[NUM_BINS];                // each bin is a singly-linked list

};

int   hashFunction(string str);          // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);   // return true is str is in the list
void  insertInList(Node* n, string str); // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int bin1 = hashFunction(s1);
   int bin2 = hashFunction(s2);



   if (!findInList(table->bins[bin1], s1) &&
       !findInList(table->bins[bin2], s2)) {



     insertToList(table->bins[bin1], s1);



     insertToList(table->bins[bin2], s2);


     return true;
   }



   return false;
}
```

Problem 2: Two Box Blurs are Better Than One (10 pts)

An interesting fact is that repeatedly convolving an image with a box filter (a filter kernel with equal weights, such as the one often used in class) is equivalent to convolving the image with a Gaussian filter. Consider the program below, which runs two iterations of box blur.

```
float input[HEIGHT][WIDTH];
float temp[HEIGHT][WIDTH];
float output[HEIGHT][WIDTH];

float weight;   // assume initialized to (1/FILTER_SIZE)^2

void convolve(float output[HEIGHT][WIDTH], float input[HEIGHT][WIDTH], float weight) {

  for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
      float accum = 0.f;
      for (int jj=0; jj<FILTER_SIZE; jj++) {
        for (int ii=0; ii<FILTER_SIZE; ii++) {

           // ignore out-of-bounds accesses (assume indexing off the end of image is
           // handled by special case boundary code (not shown)

           // count as one math op (one multiply add)
           accum += weight * input[j-FILTER_SIZE/2+jj][i-FILTER_SIZE/2+ii];
        }
      }
      output[j][i] = accum;
    }
  }
}

convolve(temp, input, weight);
convolve(output, temp, weight);
```

A. (2 pts) Assume the code above is run on a processor that can comfortably store `FILTER_SIZE*WIDTH` elements of an image in cache, so that when executing `convolve` each element in the input array is loaded from memory exactly once. What is the arithmetic intensity of the program, in units of math operations per element load?

Many times in class Prof. Kayvon emphasized the need to increase arithmetic intensity by exploiting producer-consumer locality. But sometimes it is tricky to do so. Consider an implementation that attempts to double arithmetic intensity of the program above by producing 2D chunks of output at a time. Specifically the loop nest would be changed to the following, **which now evaluates BOTH CONVOLUTIONS.**

```
for (int j=0; j<HEIGHT; j+=CHUNK_SIZE) {
  for (int i=0; i<WIDTH; i+=CHUNK_SIZE) {

    float temp[..][..];  // you must compute the size of this allocation in 6B

    // compute required elements of temp here (via convolution on region of input)

    // Note how elements in the range temp[0][0] -- temp[FILTER_SIZE-1][FILTER_SIZE-1] are the temp
    // inputs needed to compute the top-left corner pixel of this chunk

    for (int chunkj=0; chunkj<CHUNK_SIZE; chunkj++) {
      for (int chunki=0; chunki<CHUNK_SIZE; chunki++) {
        int iidx = i + chunki;
        int jidx = j + chunkj;
        float accum = 0.f;
        for (int jj=0; jj<FILTER_SIZE; jj++) {
          for (int ii=0; ii<FILTER_SIZE; ii++) {
            accum += weight * temp[chunkj+jj][chunki+ii];
          }
        }
        output[jidx][iidx] = accum;
      }
    }
  }
}
```

B. (2 pts) Give an expression for the number of elements in the `temp` allocation.

C. (2 pts) Assuming `CHUNK_SIZE` is 8 and `FILTER_SIZE` is 5, give an expression of the **total amount of arithmetic performed per pixel of output** in the code above. You do not need to reduce the expression to a numeric value.

D. (2 pts) Will the transformation given above improve or hurt performance if the original program from part A was *compute bound* for this `FILTER_SIZE`? Why?

E. (2 pts) Why might the chunking transformation described above be a useful transformation in a mobile processing setting regardless of whether or not it impacts performance?

Problem 3: Bringing Locality Back (10 pts)

The musicians Katy Perry and Kanye West hear that Spark is very popular and decide they are going to code up their own implementation of Spark to compete against that of the Apache project. Katy's first test runs the following Spark program, which creates four RDDs. The program takes Katy's lengthy (1 TB!) list of dancing tips and finds all misspelled words.

```
var lines = spark.textFile("hdfs://mydancetips.txt'');     // 1 TB file
var lower = lines.map( x => x.toLower() );                 // convert lines to lower case
var words = lower.flatmap( x => x.split(`` '') );          // convert RDD of lines to RDD of
                                                           // individual words
var misspelled = words.filter( x => !x.isInDictionary() ); // filter to find misspellings

print misspelled.count();   // print number of misspelled words
```

A. (3 pts) Understanding that the Spark RDD abstraction affords many possible implementations, Katy decides to keep things simple and implements his Spark runtime such that each RDD is implemented by a fully allocated array. This array is stored either in memory or on disk depending on the size of the RDD and available RAM. **The array is allocated and populated at the time the RDD is created — as a result of executing the appropriate operator (map, flatmap, filter, etc.) on the input RDD.**

   Katy runs her program on a cluster with 10 computers, each of which has 100 GB of memory. The program gets correct results, but Katy is upset because the program runs *very slow*. She calls her friend Taylor Swift, ready to give up on the venture. Encouragingly, Taylor says, "shake it off Katy", just run your code on 40 computers. Katy does this and observes a speedup much greater than $4\times$ her original performance. Why is this the case?

B. (4 pts) With things looking good, Kanye runs off to write a new song "All of the Nodes" to use in the marketing for their product. At that moment, Taylor calls back, and says "Actually, Katy, I think you can schedule the computations much more efficiently and get very good performance with less memory and far fewer nodes." Describe how you would change how Katy schedules her Spark computations to improve memory efficiency and performance.

C. (3 pts) After hacking all night, the next day, Katy, Kanye, and Taylor run the optimized program on 10 nodes. The program runs for 1 hour, and then right before `misspelled.count()` returns, node 6 crashes. Kanye is irate! He runs onto the machine room floor, pushing Taylor aside and says, "Taylor, I have a single to release, and I don't have time to deal with rerunning your programs from scratch. Geez, I already made you famous." Taylor gives Kayne a stink eye and says, "Don't worry, it will be complete in just a few minutes." Approximately how long will it take after the crash for the program to complete? You should assume the `.count()` operation is essentially free. But please **clearly state any assumptions about how the computation is scheduled in justifying your answer.**

Problem 4: Deletion from a Binary Tree (OPTIONAL PROBLEM: 5 POINTS EXTRA CREDIT)

The code below, and continuing on the following two pages implements node deletion from a binary search tree. We have left space in the code for you to insert locks to ensure thread-safe access and high concurrency. You may assume functions `lock(x)` and `unlock(x)` exist (where `x` has type `Lock`). **Your solution NEED ONLY consider the delete operation.**

```
// opaque definition of a lock.  You can call 'lock' and 'unlock' on
// instances of this type
struct Lock;


// definition of a binary search tree node.  You may edit this structure.
struct Node {

   int value;
   Node* left;
   Node* right;

   Lock lock;

};
```

```
// Delete node containing value given by 'value'
// Note: For simplicity, assume that the value to be deleted is not the root node!!!
void delete_node(Node* root, int value) {

   Node** prev_link = NULL;              // pointer to link to current node
   Node* cur = root;                     // current node during traversal

   while (cur) {                         // while node not found
      if (value == cur->value) {         // found node to delete!

         // Case 1: Node is leaf, so just remove it, and update the parent node's pointer to
         // this node to point to NULL
         if (cur->left == NULL && cur->right == NULL) {

            *prev_link = NULL;


            free(cur);
            return;
         } else if ( cur->left == NULL ) {
            // Case 2: Node has one child. Make parent node point to this child

            *prev_link = cur->right;

            free(cur);
            return;
         } else if ( cur->right == NULL) {
            // *** ignore this case, symmetric with previous case ***
         } else {
            // Case 3: Node has two children.  Move the next larger value in the tree into this
            // position.  The subroutine delete_helper returns this value and executes the swap
            // by removing the node containing the next larger value. SEE NEXT PAGE


            /* We need to hold cur->lock the entire time we're in the tree subtree */
            cur->value = delete_helper(cur->right, &cur->right);

            return;
         }
      } else if (value < cur->value) {
         // still searching, traverse left

         prev_link = &cur->left;
         cur = cur->left;
      } else {
         // still searching traverse right

         // *** ignore this case, symmetric with previous case ***
      }
   }
}
```

```
// The helper method removes the smallest node in the tree rooted at node n.
// It returns the value of the smallest node.
int delete_helper(Node* n, Node** prev) {

   Node** prev_link = prev;
   NODE* cur = n;

   // search for the smallest value in the tree by always traversing left
   while (cur->left) {

      prev_link = &cur->left;

      cur = cur->left;

   }

   // this is the smallest value
   int value = cur->value;

   // remove the node with the smallest value
   if (cur->right == NULL) {
      // Case 1: Node is a leaf

      *prev_link = null;

      free(cur);
   } else {
      // Case 2: Node has a right child

      *prev_link = cur->right;

      free(cur);
   }

   return value;
}
```