Parallel Computer Architecture and Programming
Written Assignment 4 SOLUTIONS

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
   Lock  binLocks[NUM_BINS];            // lock per bin
};

int   hashFunction(string str);         // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);  // return true if 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);
   bool onlyOne = false;

   // be careful to avoid deadlock due to (1) creating a circular wait or
   // (2) due to the same thread taking the same lock twice
   if (bin1 < bin2) {
     lock(binLocks[bin1];
     lock(binLocks[bin2];
   } else if (bin1 > bin2) {
     lock(binLocks[bin2];
     lock(binLocks[bin1];
   } else {
     lock(binlocks[bin1];
     onlyOne = true;
   }

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

     unlock(binLocks[bin1]);
     if (!onlyOne)
       unlock(binLocks[bin2];
```

```
      return true;
    }
    unlock(binLocks[bin1]);
    if (!onlyOne)
      unlock(binLocks[bin2];
    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?

Solution: It is $\texttt{FILTER\_SIZE}^2/2$*, since each input and output pixel are read exactly once, and each* `convolve` *operation performs* $\texttt{FILTER\_SIZE}^2$ *operations per pixel. We also accepted* $\texttt{FILTER\_SIZE}^2$ *for full credit since the question referred to "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.

Solution: `(CHUNK_SIZE+FILTER_SIZE-1)`^{2}*. We also accepted* `(CHUNK_SIZE+FILTER_SIZE)`^{2} *for full credit.*

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.

Solution: Need $12 \times 12 = 144$ *elements of* `temp` $= 5 \times 5 \times 144 = 3600$ *operations. Producing 64 elements of* `output` *is another* $64 \times 25 = 1600$ *operations. So there are now* $\frac{3600+1600}{64} = \frac{5200}{64} \approx 81$ *operations per pixel, compared to* $2 \times 25 = 50$ *operations per pixel in part A.*

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?

Solution: It will hurt performance since it increases the number of arithmetic operations that need to be performed, and the program is already compute bound. Note that a fair number of students said that the problem is was ***arithmetic intensity was increased****, hence the slowdown. Increasing arithmetic intensity of a compute-bound program will not change its runtime if the total amount of work stays the same (it just reduces memory traffic). The essence of the answer here is that more work is being done.*

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?

Solution: Since the energy cost of data transfer to/from DRAM is significantly higher than the cost of performing an arithmetic operation, reducing the amount of data transfer is likely to reduce the energy cost of running the program. Some students mentioned that reduced memory footprint was also a nice property of the transformed program (it doesn't have to allocate `temp`*), particular since DRAM sizes are smaller on mobile devices. We also accepted this answer for credit.*

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× her original performance. Why is this the case?

Solution: The program creates four RDDs, and since Katy's implementation elects to evaluate and materialize the contents RDD's immediately, the implementation will require up to 4 TB of memory to store all of these structures. There is only 1 TB of memory in aggregate across the 10 nodes of the cluster, so they will need to be stored and loaded from disk to implement each operation. The computation will be disk I/O limited. By increasing the cluster size to 40 nodes, there is now 4 TB of memory across the cluster, and the RDDs can be stored in memory and not out on disk. This will yield a significant performance improvement. Note: Even if the RDD were freed immediately after use, the system would need about 2 TB of memory and still require on-disk storage.

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.

Solution: Taylor is pointing out that the semantics of Spark RDDs permit a much more efficient implementation. The computation can be reordered to that instead of performing one operation on all elements of an RDD before proceeding to the new, the entire pipeline of operations can be performed on a chunk of the dataset all at once, with different chunks running in parallel on all the nodes. For example, an implementation could load 1 MB of lines of the input file from disk, run the entire sequence of RDD operations on this chunk (storing intermediate data in cache). This would be exceptionally memory efficient, and take advantage of the producer-consumer locality in the problem.

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.**

Solution: Assuming the elements of the RDDs are partitioned equally across the cluster, losing one node results in a loss of 1/10th of the output. Since the system partitioned the data across the cluster, the system knows exactly which partition of elements where lost. Given the lineage of the RDD `misspelled`*, this partition can be recomputed in parallel across all remaining 9 nodes. If one node took 60 minutes to compute this partition, 9 nodes would so the work in 60/9 = 6.6 minutes. We also accepted 6 minutes as a perfectly valid answer, since if the failed node was replaced then the lost partition could be recomputed in 60/10=6 minutes.*

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;

};
```

NOTE: This solution is an aggressive solution that only takes a lock on `cur` when it is absolutely neccessary. Other hand-over-hand solutions are possible.

```
// 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 (the node we look at for value comparison)
   Node* prev = NULL;                    // comes before the cur node (this is the node we hold the lock on)


   // note: no lock on cur at this stage, but that's okay as long as we don't access
   // cur->left or cur->right
   while (cur) {                         // while node we are trying delete is not not found

      if (value == cur->value) {         // found node to delete!
         lock(cur->lock);                // gaurantees cur->left or cur->right cannot be deleted
         // 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) {
            if (prev != NULL) {
               *prev_link = NULL;
               unlock(prev);
            }
            unlock(cur);
            free(cur);
            return;
         } else if (cur->left == NULL) {
            // Case 2: Node has one child. Make parent node point to this child
            if (prev != NULL) {
               *prev_link = cur->right;
               unlock(prev);
            }
            unlock(cur);
            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 (smallest node in the right-subtree).  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 right subtree
            // We will need to perform hand-over-hand locking while traversing the subtree since there
            // could be a thread trying to delete a node in the subtree
            if (prev != NULL) unlock(prev->lock);
            cur->value = delete_helper(cur->right, &cur->right);
            unlock(cur->lock);
            return;
         }

      } else if (value < cur->value) {
         // still searching, traverse left
         lock(cur->lock);  // careful: grab cur lock BEFORE letting go of prev (hand over hand)
         if (prev != NULL) unlock(prev->lock);
         prev_link = &cur->left;
         prev = cur;
         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
// There is NO LOCK on n at this time, but prev is LOCKED
int delete_helper(Node* n, Node* prev) {

   Node** prev_link = &prev->right;  // link to this subtree from node to delete
   Node* cur = n;
   Node* prev = NULL;   // a little odd this can be NULL when prev_link is not, but use
                        // if (prev != NULL) checks below to avoid releasing lock on node to
                        // delete (which is prev here)

   // at this stage, we just have a lock on the node we are about to delete and cur is
   // the right child of that node. Need a lock on cur to proceed, since otherwise
   // cur->left/right might be modified out from under us
   lock(cur->lock);

   // search for the smallest value in the tree by always traversing left
   while (cur->left) {
      if (prev != NULL) unlock(prev->lock);
      prev_link = &cur->left;
      prev = cur;
      cur = cur->left;
      lock(cur->lock);
   }

   // this is the smallest value.  We are holding a lock on the node we are about to delete.
   // prev_link is the address of the link to NULL
   int value = cur->value;

   // remove the node with the smallest value
   if (cur->right == NULL) {
      // Case 1: Node is a leaf
      *prev_link = NULL;
   } else {
      // Case 2: Node has a right child
      *prev_link = cur->right;
   }

   if (prev != NULL) unlock(prev->lock);
   unlock(cur->lock);
   free(cur);
   return value;
}
```