CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 4 SOLUTIONS

Problem 1: Miscellaneous

A. (3 pts) You and your business partner buy a **single-core computer** to host your startup's web site. Your site receives exactly one type of request: it involves a disk read with latency 900 ms (during which the processor is idle, and waits) followed by 600 ms of processing. Thus, its response latency is 1.5 sec.

As your business grows you start receiving requests at steady rate of one request every 100 ms. Your partner claims, "Let's buy 14 more machines so we can keep up!" You claim, "Wait a minute, we can save money and handle this load without all those machines!" How many machines do you actually need to service 10 requests per second, and how would you thread your web server to achieve this throughput? **Assume that disk I/O bandwidth is infinite, multiple I/O operations can be outstanding at once, and that all requests are unique and so optimizations like caching are not possible.**

Solution: It is possible to achieve throughput of 10 requests per second using six machines by overlapping processing with disk I/O. A solution would use a pool of three worker threads per machine. During the 900 ms one thread must wait for its I/O to complete, the other two threads perform their 2×600 ms of CPU work. This results in 100% CPU utilization and a throughput of $1000/600 = 1.67$ requests/sec per machine. So six machines are needed to achieve throughput of 10 requests/sec. Saving server cost with efficient use of machines machine just improved your startup's bottom line!

B. (2 pts) At 8:30am, you go to grab a coffee at the Gates 3rd floor cafe. There is no line, so you immediately walk up and order. Later that day, you go to get your daily double-creme, super-duper, organic soy latte between classes at 12:50pm to take to 15-418/618, and the line is 15 students long. You tell the manager, "you really need to hire more workers,". She responds, "I can't, because it would be cost too much to double my staff for the entire workday". Using the words "throughput" and "elasticity" in your answer, make a suggestion on how the manager might run her business.

Solution: The manager could hire more staff to work during peak cafe hours. In the early morning, when few students are around, the cafe need only provide low order processing throughput, since the arrival rate of the orders is low. However, it is helpful to add more help during peak hours to sustain high order processing throughput. Note that just like our discussion of servers on an e-commerce website, the cafe is maximally using its available resources in peak hours (everyone is working hard). As the line builds, some customers decide it's not worth it to wait in line (or wait for a slow web site), and the cafe misses out on potential sales. A manager has to decide whether it's worth it to chase those sales by providing higher throughput in those periods, at the risk of overprovising the cafe's resources during downtime. Elasticity provides an efficient (for the manager) solution. Of course, when talking about real human resources, and not virtual machines, it can be difficult for workers to be offered only a small number of (some times unpredictable) hours!

C. (2 pts) Consider a parallel version of the 2D grid solver problem from class. The implementation uses a 2D tiled assignment of grid cells to processors. (Recall the grid solver updates all the red cells of the grid based on the value of each cell's neighbors, then all the black cells). Since the grid solver requires communication between processors, you choose to buy a computer with a cross-bar interconnect. Your friend observes your purchase and suggests there there is another network topology that would have provided the same performance at a lower cost. What is it? (Why is the performance the same?)

Solution: In this application, each processor need only communicate with the processor that responsible for its left, right, top, bottom grid neighbors. The mesh topology provides all of these connections as point-to-point links, so the app will run just as well on a mesh as on a fully-connected crossbar. The mesh has O(N) cost, where the crossbar has O(N^2), where N is the number of processors. A 2D torus would also be a valid answer.

D. (3 pts) Recall a basic test and test-and-set lock, written below using compare and swap (`atomicCAS`) Keep in mind that `atomicCAS` is atomic although it is written in C-code below.

```
int atomicCAS(int* addr, int compare, int val) {
  int old = *addr;
  *addr = (old == compare) ? val : old;
  return old;
}

void LOCK(int* lock) {
   while (1) {
      if (*lock == 0)
         if (atomicCAS(lock, 0, 1) == 0)
            return;
   }
}
```

Imagine a program that uses this lock to synchronize access to a variable shared by 32 threads. This program is run on a processor with **four cores**, each of which are **eight-way multi-threaded** (32 total execution contexts across the machine.) You can assume that **the lock is highly contended, the cost of lock acquisition on lock release is insignificant and that the size of the critical section is large (say, 100,000's of cycles)**. You can also assume there are no other programs running on the machine.

Your friend (correctly) points there is a performance issue with your implementation and it might be advisable to not use a spin lock in this case, and instead use a lock that de-schedules a thread off an execution context instead of busy waiting. You look at him, puzzled, mentioning that the test-and-test-and-set lock means all the waiting threads are just spinning on a local copy of the lock in their cache, so they generate no memory traffic. What is the problem he is referring to? **(A full credit answer will mention a fix that specifically mentions what waiting threads to deschedule.)**

Solution: The problem here is that the thread with the lock must share the core with seven other threads that are also mapped to the core's execution slots (7/8 of the core's instruction processing ability is spent spinning!). It would be ideal to deschedule the seven threads mapped to the same core as the thread holding the lock, so the thread in the critical section could have all the core's execution resources and complete the critical section as quickly as possible.

Problem 2: Hash Table Parallelization (10 points)

Below you will find an implementation of a hash table (a linked list per 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. (Update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't 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 idx1 = hashFunction(s1);
   int idx2 = 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 (idx1 < idx2) {
     lock(binLocks[idx1];
     lock(binLocks[idx2];
   } else if (idx1 > idx2) {
     lock(binLocks[idx2];
     lock(binLocks[idx1];
   } else {
     lock(binlocks[idx1];
     onlyOne = true;
   }

   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {
     insertToList(table->bins[idx1], s1);
     insertToList(table->bins[idx2], s2);

     unlock(binLocks[idx1]);
     if (!onlyOne)
       unlock(binLocks[idx2];
     return true;
   }
   unlock(binLocks[idx1]);
   if (!onlyOne)
     unlock(binLocks[idx2];
   return false;
}
```