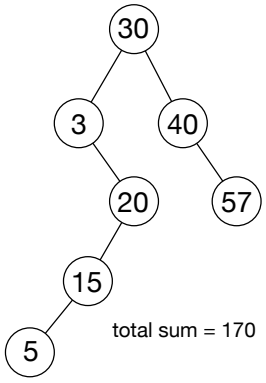
Stanford CS149: Parallel Computing
Written Assignment 5

Problem 1: Transactions on Trees (30 pts)

Consider the binary search tree illustrated below.



The operations `insert` (insert value into tree, assuming no duplicates) and `sum` (return the sum of all elements in the tree) are implemented as transactional operations on the tree as shown below.

```
struct Node {
   Node *left, *right;
   int value;
};
Node* root;  // root of tree, assume non-null

void insertNode(Node* n, int value) {
  if (value < n->value) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else if (value > n->value) {
    if (n->right == NULL)
       n->right = createNode(value);
    else
       insertNode(n->right, value);
  }  // insert won't be called with a duplicate element, so there's no else case
}

int sumNode(Node* n) {
   if (n == null) return 0;
   int total = n->value;
   total += sumNode(n->left);
   total += sumNode(n->right);
   return total;
}

void insert(int value) {  atomic { insertNode(root, value); }   }
int sum()              {  atomic { return sumNode(root); )      }
```

Consider the following four operations are executed against the tree in parallel by different threads.

```
insert(10);
insert(25);
insert(24);
int x = sum();
```

A. (5 pts) Consider different orderings of how these four operations could be evaluated. Please draw all possible trees that may result from execution of these four transactions. (Note: it's fine to draw only subtrees rooted at node 20 since that's the only part of the tree that's effected.)

B. (5 pts) Please list all possible values that may be returned by `sum()`.

C. (5 pts) Do your answers to parts A or B change depending on whether the implementation of transactions is optimistic or pessimistic? Why or why not?

D. (5 pts) Consider an implementation of **lazy, optimistic** transactional memory that manages transactions at the granularity of tree nodes (the read and writes sets are lists of nodes). Assume that the transaction `insert(10)` commits when `insert(24)` and `insert(25)` are currently at node 20, and `sum()` is at node 40. Which of the four transactions (if any) are aborted? **Please describe why.**

E. (5 pts) Assume that the transaction `insert(25)` commits when `insert(10)` is at node 15, `insert(24)` has already modified the tree but not yet committed , and `sum()` is at node 3. Which transactions (if any) are aborted? **Again, please describe why.**

F. (5 pts) Now consider a transactional implementation that is **pessimistic** with respect to writes (check for conflict on write) and **optimistic** with respect to reads. The implementation also employs a "writer wins" conflict management scheme – meaning that the transaction issuing a conflicting write will not be aborted (the other conflicting transaction will). Describe how a **livelock problem** could occur in this code.

Problem 2: Implementing Transactions (25 pts)

In this problem we will explore the implementation of an optimistic read, optimistic write, lazy versioning software TM (STM) like TL2. The STM operates over 32-bit values.

In your implementation, each transaction is encapsulated by a `Txn` object that maintains a local timestamp for the transaction as well as the transaction's read and write sets. Your implementation should have the following properties:

1. A global timestamp and a single global lock to protect commits.
2. A transaction's local timestamp is the value of the global timestamp when the transaction starts.
3. A table that maps memory locations to a version number.
4. Writes are stored to a write log `wset` as `(address, value)` pairs.
5. The version of committed writes is the current global timestamp; committing also increments the global timestamp.
6. The read set is validated on commit; if any read location has a version number greater than the local timestamp the transaction retries.

The skeleton code for the transactional memory system is given on the next page. You should write your answers in the space provided on the page after that. Code for `__begin` and `write` is provided, and you should provide code for `read` and `commit`. Don't get hung up on syntax; we don't expect you to pen down flawless, compilable C code - some pseudocode is acceptable as long as its meaning is clear.

```
// setjmp stores a snapshot of the registers (stack pointer, instruction pointer, etc.) into
// a buffer (t.rollback). A future call to longjmp restores the saved register values and thus
// restarts control flow at the point when setjmp was called.
#define TXN_BEGIN(t)    \ // TXN_BEGIN is called to begin a transaction.
        setjmp(t.rollback); \
        t.__begin();

typedef uint64_t timestamp_t;

class Txn {
public:
  Txn() {}
  virtual ~Txn() {}
  void retry() { longjmp(rollback, 1); } // return control flow to context saved by setjmp.
  void __begin();
  void write(uint32_t* p, uint32_t v);

  uint32_t  read(uint32_t* p); // Implement this method on the next page!
  void      commit(); // Implement this method on the next page!

  jmp_buf       rollback;
private:
  #define TABLE_SZ    4096
  timestamp_t   local_timestamp;

  static timestamp_t get_version(uint32_t* p) {
    return versions[(((intptr_t)(p)) / 4) % TABLE_SZ];
  }
  static void set_version(uint32_t* p, timestamp_t t) {
    versions[(((intptr_t)(p)) / 4) % TABLE_SZ] = t;
  }

  // Used to store uncommited writes to memory locations
  typedef std::map<uint32_t*, uint32_t> write_set_t;
  write_set_t   wset;

  // Used to keep track of reads that this transaction has made
  typedef std::set<uint32_t*> read_set_t;
  read_set_t    rset;

  // Used to map memory addresses to a timestamp (e.g. to indicate most recent use)
  static timestamp_t  versions[TABLE_SZ];
  static timestamp_t  global_timestamp;
  static mutex_t      commit_lock;
};

/////implementation file///////////
timestamp_t  Txn::global_timestamp = 0;
mutex_t      Txn::commit_lock = MUTEX_INITIALIZER;
timestamp_t  Txn::versions[TABLE_SZ];

void Txn::__begin(void) {
  wset.clear();
  rset.clear();
  local_timestamp = global_timestamp;
}
void Txn::write(uint32_t* p, uint32_t v) {
  wset[p] = v;
}
```

```
uint32_t Txn::read(uint32_t* p) {
   // YOUR CODE HERE!









}

void Txn::commit() {
  mutex_lock(&commit_lock);
  // YOUR CODE HERE!









  mutex_unlock(&commit_lock);
}
```

Problem 3: Two Box Blurs are Better Than One (25 pts)

Consider the program below, which runs two iterations of box blur on an input image. (We discussed in class how convolution with a box filter "blurs" an image.)

```
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. (5 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 we've 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. (5 pts) Give an expression for the number of elements in the `temp` allocation.

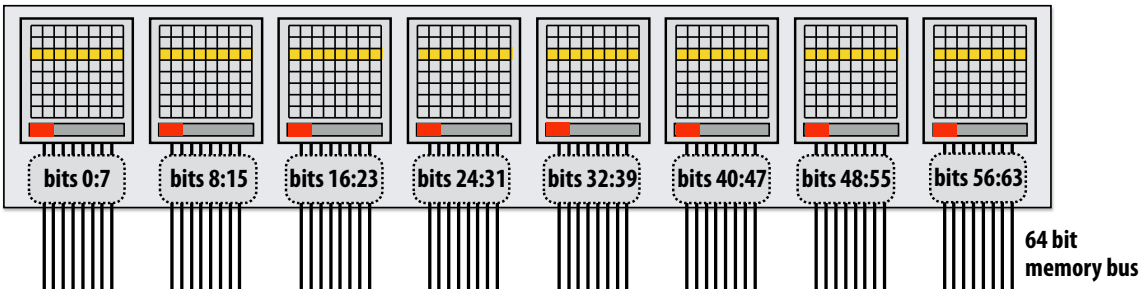
C. (5 pts) Assuming `CHUNK_SIZE` is 8 and `FILTER_SIZE` is 5, give an expression of the **total amount of arithmetic performed per pixel of output** in the code above. You do not need to reduce the expression to a numeric value.

D. (5 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. (5 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? (hint: consider the energy cost of DRAM access)

Problem 4: Controlling DRAM (20 pts)

Consider a DRAM DIMM with 8 chips (8-bit interface per chip) just like what we talked about in class. Physical memory addresses are strided across the chips as in the figure below, so that 64 consecutive bits from the address space can be read in a single clock over the bus. The DRAM row size is 2 kilobits (256 bytes). There is only a single bank per chip. (We ignore banking in this problem.)



The memory controller processes requests with the following logic:

```
int active_row;   // stores active row

handle_64bit_request(void* addr) {

   int row, col;

   compute_row_col(addr, &row, &col);   // compute row/col from addr (0 cycles)

   if (row != active_row)
      activate_row(row);     // this operation takes 15 cycles

   transfer_column(col);     // this operation takes 1 cycle
}
```

Questions are on the next page...

Now consider the following C-program, which executes using 2 pthreads on a dual-core processor with a single shared cache.

```
struct ThreadArg {
  int threadId;
  double sum;  // thread-local variable
  int N;       // assume this is very large
  double* A;   // pointer to shared array
};

// each thread processes one half of array A
void myfunc(void* targ) {
  ThreadArg* arg = (ThreadArg*)targ;
  arg->sum = 0.f;
  int offset = arg->threadId * arg->N / 2;
  for (int i=0; i<arg->N / 2; i++)
    arg->sum += arg->A[offset + i];
}

/* main code */

ThreadArg args[2];
args[0].threadId = 0; args[1].threadId = 1;
args[0].A = args[1].A = new double[N];

// initialize args[].sum, args[].N, args[].A, and launch two pthreads here that run myfunc
// Then wait for threads to complete

print("%f\n", args[0].sum + args[1].sum);
```

A. (5 pts) Assume that the two threads run at approximately the same speed, so the memory controller receives requests from the two threads in interleaved order: thread0_req0, thread1_req0, thread0_req1, thread1_req1, etc. Given this stream, what is the effective bandwidth of the memory system as observed by the processor (the rate at which it receives data)? Assume that:

- The program is bandwidth bound so that the memory system always has a deep queue of requests to process.
- The granularity of transfer between the memory controller and the cache is 64 bits. (e.g., 8-byte cache line size)
- Note that array elements are DOUBLES (8 bytes).

B. (5 pts) Modify the program code to significantly improve the effective memory system bandwidth. What is the new bandwidth you observe?

C. (5 pts) Return to the original code given in this assignment (ignore your solution to part B), and assume that requests now arrive at the memory controller every ten cycles. For example...

```
cycle 0: thread 0 req 0
cycle 10: thread 1 req 0
cycle 20: thread 0 req 1
cycle 30: thread 1 req 1
cycle 40: thread 0 req 2
cycle 50: thread 1 req 2
cycle 60: thread 0 req 3
...
```

Write (rough) pseudocode for a memory request scheduling algorithm that allowed the memory system to keep up with this request stream. **Your implementation can assume there is an incoming request buffer called `request_buf` that holds up to 4 requests.** (The processor stalls if the request buffer is full.)

D. (5 pts) You add hardware multi-threading to your dual-core processor (2 threads-per core) and spawn four threads in your code. You assign contiguous blocks of the input array to each pthread. Assuming the memory request arrival rate stays the same (but now requests from four threads, rather than two, are interleaved), how would you change your solution in part C to keep up with the request stream? (you may modify the buffer size if need be). Is overall memory latency higher or lower than in part C? Why?

The following problems are PRACTICE PROBLEMS and will not be graded.

Practice Problem 1: Paparazzi Camera

You are designing a heterogeneous multi-core processor to perform real-time "celebrity detection" on a future camera. The camera will continuously process low-resolution live video and snap a high-resolution picture whenever it identifies a subject in a database of 150 celebrities. Pseudocode for its behavior is below:

```
void process_video_frame(Image input_frame) {
  Image face_image = detect_face(input_frame);
  for (int i=0; i<150; i++)
     if (match_face(face_image, database_face[i]))
        take_high_res_photo();
}
```

In order to not miss the shot, the camera MUST call `take_high_res_photo` within 500 ms of the start of the original call to `process_video_frame`! To keep things simple:

- Assume the code loops through all 150 database images regardless of whether a match is found (e.g., we want to find all matches).
- The system has plenty of bandwidth for any number of cores.

Two types of cores are available to use in your chip. One is a fixed-function unit that accelerates `detect_face`, the other is a general-purpose processor. The cost (in units of chip resources) of the cores and their performance (in ms) executing important functions in the pseudocode are given below:

Operation	Core Type	
	C1 (fixed-function)	C2 (Programmable)
Resource Cost	1	1
Perf (ms): `detect_face`	50	250
Perf (ms): `match_face`	N/A	25

A. Assume a video frame arrives exactly every 500 ms. **If you only use cores of type C2**, what is the minimum number of cores do you need to meet the performance requirement for the video stream? (You cannot change the algorithm, and please justify your answer).

B. Your team has just built a multi-core processor that contains a large number of cores of type C2. It achieves 15.2× speedup on the camera workload discussed above. Amdahl's Law says that the maximum speedup of the camera pipeline in this problem should be $(250 + 150 \times 25)/250 = 16\times$, so your team is happy. They are shocked when your boss demands a speedup of 20×. Your team is on the verge of quitting due "unreasonable demands". How do you argue to them that the goal is reasonable one if they consider all the possibilities in the above table? (Hint: What assumption are they making in their Amdahl's Law calculations, and why does it not hold?)

.

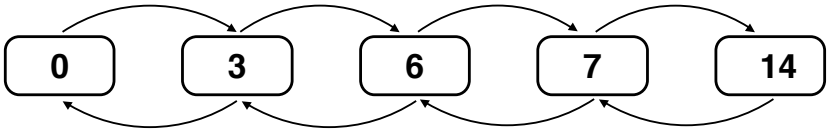
C. Now assume you can use both cores of types C1 and C2 in your design. How many of each core do you choose to minimize resource usage, while still meeting the same latency requirements as in part A? Does your new chip use more or fewer total resources than your solution in part A (by how much)?

D. Beyonce is about to drop a new album, and your paparazzi customers want to follow her around all day. They request a camera that is more energy efficient. Energy efficiency is so important they are willing to relax their performance requirement and allow high-res photos to be taken within 2500 ms (not 500 ms) of a video frame arriving. You find that nearly all the power in your application is used by loading database faces from memory. Describe how you would change the pseudocode to **approximately triple** the energy efficiency of the camera while still meeting the performance requirements. You should assume you use the same processor design as in part C (or part A for that matter), and that your processor has a cache that holds up to four database images. Hint: we are expecting you to reduce the number of times `database_face[i]` is loaded from memory by accessing data resident in cache.

Practice Problem 2: Transactions on a Doubly Linked List

Consider a **SORTED doubly-linked list** that supports the following operations. Yes, this is the same data structure as used in Written Assignment 4.

- `insert_front`, which traverses the list from the front.
- `delete_front`, which deletes a node by traversing from the front
- `insert_back`, which traverses the list **backwards from the end** to insert a node in the opposite order as `insert_front`.



In this problem, assume that the entire body of each function `insert_front`, `delete_front`, and `insert_back` is placed in its own atomic block, and the code is run on a system **supporting optimistic (for reads and writes) transactional memory**.

A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - Test 1: `insert_front(2)`, `delete_front(14)`
   - Test 2: `insert_front(12)`, `delete_front(6)`
   - Test 3: `insert_front(13)`, `insert_back(4)`

   Assuming all unit tests start with the list in the state shown above, is the code correct? (By correct, we mean there are no race conditions and so all operations will modify the data structure according to their specification.) Why or why not?

B. Consider two transactions performing `insert_front(4)` and `delete_front(14)`. Assume both transactions start at the same time on different cores and the transaction for `insert_front(4)` proceeds to commit while the `delete_front(14)` transaction has just iterated to the node with value 7. Must either of the two transactions abort in this situation? Why? **(Remember this is an optimistic transactional memory system!)**

C. Must either transaction abort if the transaction for `delete_front(14)` proceeds to commit before the transaction for `insert_front(4)` does? Why? **Please assume that at the time of the attempted commit, `insert_front(4)` has iterated to node 3, but has not begun to modify the list.**

D. Must either transaction abort if the situation in part C is changed so that `delete_front(14)` attempts to commit first, but by this time `insert_front(4)` has made updates to the list (although not yet initiated its commit)? Why?