Stanford CS149: Parallel Computing
Written Assignment 3

Problem 1: Miscellaneous Short Problems (25 pts)

A. (8 pts) A key idea in this course is the difference between *abstraction* and *implementation*. Consider two abstractions we've studied: ISPC's `foreach` and Cilk's spawn construct. **Briefly describe how these two abstractions have similar semantics.** (Hint: what do the constructs declare about the associated loop iterations? Be precise!). **Then briefly describe how their implementations are quite different** (Hint: consider their mapping to modern CPUs). As a reminder, we give you two syntax examples below:

```
ISPC foreach:
============================

foreach (i = 0 ... 100) {
  x[i] = y[i];
}
```

```
Cilk:
============================
void f(int i, float* x, float* y) {
  x[i] = y[i]
}

for (int i=0; i<100; i++) {
  cilk_spawn f(i, x, y);
}
```

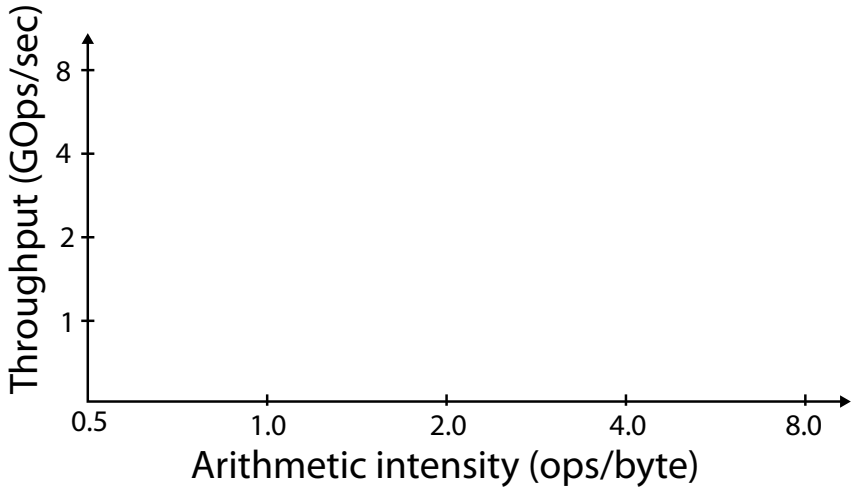
B. (7 pts) You have a simple message passing program that uses **blocking sends** to communicate with remote processor P1. Pseudocode is below.

```
int my_buffer[32];

// initialize contents of my_buffer with msg 1 here ..

// send second message to P2
send(P1, 32, my_buffer);

// write contents of msg 2 into my_buffer here ..

// send second message to P2
send(P1, 32, my_buffer);
```

Your friend remembers the conversation about deadlock when using blocking sends from class, and changes to code so that asynchronous sends are used instead of blocking sends are used. The program immediately starts giving incorrect results. Why? (A full credit answer will briefly describe how to modify the code so that it is both correct and maintains asynchronous communication of both messages.)

C. (10 pts) In class we described the usefulness of making roofline graphs, which plots the instruction throughput of a machine (gigaops/sec) as a function of a program's arithmetic intensity (ops performed per byte transferred from memory). Consider the roofline plot below. Please plot the roofline curve for a machine featuring a **1 GHz dual-core processor. Each core can execute one 4-wide SIMD instruction per clock**. This processor is connected to a memory system providing 4 GB/sec of bandwidth. *Hint: what is the peak throughput of this processor? What are its bandwidth requirements when running a piece of code with a specified arithmetic intensity?*

Plot the expected throughput of the processor when running code at each arithmetic intensity on the X axis, and draw a line between the points.



Problem 2: Data-Parallel Thinking (35 pts)

Assume you are given a library that can execute a bulk launch of N independent invocations of an application-provided function using the following CUDA-like syntax:

```
my_function<<<N>>>(arg1, arg2, arg3...);
```

For example the following code would output: (`id` is a built-in id for the current function invocation)

```
void foo(int* x) {
   printf("Instance %d : %d\n", id, x[id]);
}
int A[] = {10,20,30}
foo<<<3>>>(A);

"Instance 0 : 10"
"Instance 1 : 20"
"Instance 2 : 30"
```

The library also provides the data-parallel function `exclusive_scan` (using the + operator) that works as discussed in class.

```
exclusive_scan(N, in, out);

Example usage:
N    = 6
in   = {1, 2, 3, 4, 5, 6}
=====================================
out  = {0, 1, 3, 6, 10, 15}
```

In this problem, we'd like you to design a data-parallel implementation of `largest_segment_size()`, which, given an array of flags that denotes a partitioning of an array into segments, computes the size of the longest segment in the array.

```
int largest_segment_size(int N, int* flags);
```

The function takes as input an array of N flags (`flags`) (with 1's denoting the start of segments), and returns the size of the largest segment. The first element of flags will always be 1. For example, the following flags array describes five segments of lengths 4, 2, 2, 1, and 1.

```
N       = 10
flags   = {1,  0,  0,  0,  1,  0, 1,  0, 1,  1}
=============================================
result:  = 4
```

Questions on next page...

A. (20 pts) The first step in your implementation should be to compute the size of each segment. Please use the provided library functions (bulk launch of a function of your choice + `exclusive_scan` to implement the function `segment_sizes()` below. *Hint: We recommend that you get a basic solution done first, then consider the edge cases like how to compute the size of the last segment.*

```
// Example output of segment_sizes(N, flags, num_segs, sizes):
//   N        =  8
//   flags    =  {1, 0, 1, 0, 0, 0, 1, 0}
//   ==================================
//   num_segs = 3
//   sizes    = {2, 4, 2}


// you may wish to define functions used in bulk launches here









// You can allocate any required intermediate arrays in this function
// You may assume that 'seg_sizes' is pre-allocated to hold N elements,
// which is enough storage for the worse case where the flags array
// is all 1's.
void segment_sizes(int N, int* flags, int* num_segs, int* seg_sizes) {









}
```

B. (15 pts) Now implement `largest_segment_size()` using `segment_sizes()` as a subroutine. **NOTE: this problem can be answered even without a valid answer to Part A.** Your implementation may assume that the number of segments described by `flags` is always a power of two. A full credit implementation will maximize parallelism and minimize work when computing the maximum segment size from an array of segment sizes. *Hint: we are looking for solutions with `lg2(num_segs)` span.*

```
// you may want to implement helper functions here that are called via bulk launch
```

```
int largest_segment_size(int N, int* flags) {

  int num_segs;
  int seg_sizes[N];
  segment_sizes(N, flags, &num_segs, seg_sizes);

}
```

Problem 3: Implementing CS149 Spark (40 pts)

In this problem we want you to implement a *very simple* version of Spark, called CS149Spark, that supports only a few operators. You will implement CS149Spark as a simple C++ library consisting of a base class `RDD` as well as subclasses for all CS149Spark transforms.

```
  class RDD {
    public:
    virtual bool hasMoreElements() = 0;   // all RDDs must implement this
    virtual string next() = 0;            // all RDDs must implement this

    int count() {                  // returns number of elements in the RDD
      int count = 0;
      while (hasMoreElements()) {
         string el = next();
         count++:
      }
      return count;
    }

    vector<string> collect() {     // returns STL vector representing RDD
      vector<string> data;
      while (hasMoreElements()) {
         data.append(next());
      }
      return data;
    }
};

class RDDFromTextFile : public RDD {
  ifstream inputFile;             // regular C++ file IO object
  public:
  RDDFromTextFile(string filename) {
    inputFile.open(filename);     // prepares file for reading
  }

  bool hasMoreElements() {
    return !inputFile.eof();      // .eof() returns true if no more data to read
  }

  string next() {
    return inputFile.readLine();  // reads next line from file
  }
};
```

For example, given the two definitions above, a simple program that counts the lines in a text file can be written as such.

```
  RDDFromTextFile r("myfile.txt'');   // creates an RDD where each element is a string
                                      // corresponding to a line from the text file

  printf("The RDD has length %d\n", r.count());
```

A. (10 pts) Now consider adding a `l33tify` RDD transform to CS149Spark, which returns a new RDD where all instances of the character 'e' in string elements of the source RDD are converted to the character '3'. For example, the following code sequence creates an RDD (`r1`) whose elements are lines from a text file. The RDD `r2` contains a l33tified version of these strings. This data is collected into a regular C++ vector at the end of the program using the call to `collect()`.

```
RDDFromTextFile r1("myfile.txt'');    // creates an RDD where each element is a string
                                      // corresponding to a line from the text file

RDDL33tify r2(r1);                    // l33tify all elements for r1
vector<string> lines = r2.collect();  // lines from the file, but in l33t form
```

Implement the functions `hasMoreElements()` and `next()` for the `l33tify` RDD transformation below. **A full credit solution will use minimal memory footprint and never recompute (compute more than once) any elements of any RDD.**

```
////////////////////////////////////////////////////////////
class RDDL33tify : public RDD {

  RDD parent;


  RDDL33tify(RDD parentRDD) {
    parent = parentRDD;



  }

  bool hasMoreElements() {









  }

  string next() {









  }
};
```

B. (10 pts) Now consider a transformation `FilterLongWords` that filters out all elements of the input RDD that are strings of greater than 32 characters.

Again, we want you to implement `hasMoreElements()` and `next()`.

You may declare any member variables you wish and assume `.length()` exists on strings. **Careful: `hasMoreElements()` is trickier now! Again a full credit solution will use minimal memory footprint and never recompute any elements of any RDD.**

A sample program using the `FilterLongWords` RDD transformation is below:

```
  RDDFromTextFile r1("myfile.txt'');   // creates an RDD where each element is a string
                                       // corresponding to a line from the text file

  RDDL33tify r2(r1);                   // converts elements to l33t form
  RDDFilterLongWords r3(r2);           // removes strings that are greater than 32 characters
  print("RDD r3 has length %d\n", r3.count());

////////////////////////////////////////////////////////////////////
class RDDFilterLongWords : public RDD {

  RDD parent;


  public:
  RDDFilterLongWords(RDD parentRDD) {
    parent = parentRDD;


  }

  bool hasMoreElements() {









  }

  string next() {









  }
};
```

C. (10 pts) Finally, implement a `groupByFirstWord` transformation which is like Spark's `groupByKey`, but instead (1) it uses the first word of the input string as a key, and (2) instead of building a list of all elements with the same key, concatenates all strings with the same key into a long string.

For example, `groupByFirstWord` on the RDD ["hello world", "hello cs149", "good luck", "parallelism is fun", "good afternoon"] would produce the RDD ["hello world hello cs149", "good luck good afternoon", "parallelism is fun"].

Your implementation can be rough pseudocode, and may assume the existence of a dictionary data structure (mapping strings to strings) to actually perform the grouping, an iterator over the dictionaries keys, and useful string functions like: `.first()` to get the first word of a string, and `.append(string)` to append one string to another.

Rough pseudocode is fine, but your solution should make it clear how you are tracking the next element to return in `next()`. A full credit solution will use minimal memory footprint and never recompute any elements of any RDD.

```
class RDDGroupByFirstWord : public RDD {
  RDD parent;
  Dictionary<string, string> dict;       // assume dict["hello''] returns the string
                                         // associated with key "hello''



  public:
  RDDGroupByFirstWord(RDD parentRDD) {
    parent = parentRDD;


  }

  bool hasMoreElements() {









  }

  string next() {









  }
};
```

D. (5 pts) Describe why the RDD transformations `L33tify`, `FilterLongWords`, and RDD construction from a file, as well as the action `count()` can all execute efficiently on very large files (consider TB-sized files) on a machine with a small amount of memory (1 GB of RAM).

E. (5 pts) Describe why the transformation `GroupByFirstWord` differs from the other transformations in terms of how much memory footprint it requires to implement.

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

Practice Problem 1: Bringing Locality Back

Justin Timberlake and Kanye West hear that Spark is all the rage and decide they are going to code up their own implementation to compete against that of the Apache project. Justin's first test runs the following Spark program, which creates four RDDs. The program takes Justin'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. Understanding that the Spark RDD abstraction affords many possible implementations, Justin 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.**

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

B. With things looking good, Kanye runs off to write a new single "All of the Nodes" to use in the marketing for their product. At that moment, Taylor calls back, and says "Actually, Justin, 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 Justin schedules his Spark computations to improve memory efficiency and performance.

C. After hacking all night, the next day, Justin, 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.**

Practice Problem 2: Implementing a Barrier

In class we talked about the `barrier()` synchronization primitive. No thread proceeds past a barrier until all threads in the system have reached the barrier. (In other words, the call to `barrier()` will not return to the caller until its known that all threads have called `barrier()`. Consider implementing a barrier in the context of a message passing program that is only allowed to communicate via **blocking sends and receives**. Using only the helper functions defined below, implement a barrier. Your solution should make no assumptions about the number of threads in the system. **Keep in mind that all threads in a message passing program execute in their own address space—there are no shared variables.**

```
// send msg with id msgId and contents msgValue to thread dstThread
void blockingSend(int dstThread, int msgId, int value);

// recv message from srcThread. Upon return, msgId and msgValue are populated
void blockingRecv(int srcThread, int* msgId, int* msgValue);

// returns the id of the calling thread
int getThreadId();

// returns the number of threads in the program
int getNumThreads();
```

Practice Problem 3: Data Parallel Graphs

Consider a representation for a directed graph much like the one we used your programming assignment. A graph of N vertices is represented by an array of per-vertex `values` and an array of `incoming_edges` which lists the vertex id of all incoming edges, and `incoming_starts`, which denotes the start and end position of vertex i's incoming edge (in the array `incoming_edges`). In other words, the j'th incoming edge to vertex i is given by:
`incoming_edges[incoming_starts[i] + j]`.

C code for this definition is here. Followed by an example to refresh your memory:

```
struct Graph {
  int    N;
  float* values[N];              // per vertex values
  int*   incoming_edges[NUM_EDGES];
  int*   incoming_starts[N+1];    // this is a length N+1 array, the last element
                                  // is always equal to the total number of edges
};

Example, the graph defined below:
N = 4;
incoming_edges =  {1, 2, 0, 3, 3, 0, 1, 2};    // 8 edges
incoming_starts = {0, 1, 4, 6, 8};

Has edges:
(0,1)
(1,2), (1,0), (1,3)
(2,3), (2,0)
(3,1), (3,2)
```

(Question continues on next page...)

Assume you are given a library of data parallel functions on arrays. We've provided pseudocode below so it is clear what operations these functions perform. However, please make no assumption about their actual implementation. (they could be parallelized, etc.) **We are using `T` as a generic type in the code below, assume it can be floats or ints.**

```
void init(int n, T val, T* out) {      // initialize all array elements to val
  for (int i=0; i<n; i++)
     out[i] = val;
}

void gather(int n, T* in, int* index, T* out) {   // gather from in into out
  for (int i=0; i<n; i++)
     out[i] = in[index[i]];
}

void scatter(int n, T* in, int* index, T* out) {  // scatter from in to out
   out[index[i]] = in[i];
}

void subtract(int n, T* in1, T* in2, T* out) {    // elementwise subtract in2 from in1
  for (int i=0; i<n; i++)
     out[i] = in1[i] - in2[i];
}

void compare(int n, T* in, T val, int* out) {     // compare elements of in to value
  for (int i=0; i<n; i++)
     out[i] = (in[i] > val) ? 1 : 0;
}

void shift_left(int n, T* in, T* out) {           // note: in is of size n, out of size n-1
  for (int i=0; i<n-1; i++)
     out[i] = in[i+1];
}

void segmented_prefixsum(int n, T* in, int* segments, T* out) {

  // This is an inclusive segmented scan involving an add operator, but it
  // is performed per segment. 'segments' has the value 1 at the start
  // of all segments
     // Example:
     // if:   in = {1,2,3,4,5,6,7,8}, and segments = {1,0,0,1,0,1,0,0}
     // then out = {1,3,6,  4,9,   6,13,21}
}
```

(Question continues on next page...)

A. Given the library functions on the previous page, implement an algorithm that computes an array `big_change_mask`, where `big_change_mask[i]=1` only if vertex i's value (`g.values[i]`) is THRESHOLD units greater than **the sum the values of the vertices that have edges pointing to it** (and 0 otherwise.) Be precise with your pseudocode, but we will not be picky about potential edge cases for the last vertex.

```
Graph g; // assume graph is initialized.
         // Number of vertices is N, number of edges is NUM_EDGES

// your solution should write 0's and 1's to this array
// you may allocate any temporal arrays as necessary
int* big_change_mask = alloc(int, N);

//
// Hint: build segments array
//

int* ones = alloc(int, N);
int* segments = alloc(int, NUM_EDGES);
init(N, 1, ones);









//
// Hint: gather vertices and perform per-vertex sums
//
```

B. (THIS QUESTION CAN BE ANSWERED INDEPENDENTLY OF A CORRECT SOLUTION TO PART A.) Now consider a version of the library that uses various techniques discussed in class to **compress key data structures in the problem, such as the list of incoming edges, and boolean masks.** Give one reason why using compressed in-memory data structures might be a bad idea if the code was to be run on a *single thread* of a large 64-core machine. Give one reason why compressing datra structures might be a good idea if the code was run on all 64-cores of the machine.