Programming for Performance (ECE459): Final

April 22, 2014

This open-book final has 12 non-cover pages and 6 questions, worth 25 points each. You may consult any printed material (books, notes, etc).

1 Nonblocking I/O

This C code sends a file over a socket.

```c
// http://www.kegel.com/dkftpbench/nonblocking.html
#include <fcntl.h>

#define BUFSIZE 1024

void sendFile(const char *filename, int socket) {
    int fd, nread, nwrite, i;
    char buf[BUFSIZE];

    fd = open(filename, O_RDONLY);
    /* Send the file, one chunk at a time */
    do {
        /* Get one chunk of the file from disk */
        nread = read(fd, buf, BUFSIZE);
        if (nread == 0) {
            /* All done; close the file and the socket. */
            close(fd); close(socket);
            break;
        }
        /* Send the chunk */
        for (i=0; i<nread; i += nwrite) {
            /* write might not take it all in one call,
             * so we have to try until it's all written */
            nwrite = write(socket, buf + i, nread - i);
        }
    } while (1);
}
```

Part 1 (5 points). Identify the lines in `sendFile()` which perform blocking I/O.

Next, you'll convert `sendFile()` to use nonblocking I/O. Here is some state that you'll want to maintain.

```
struct sendfile_state {
    int fd;             /* file being sent */
    int socket;         /* socket to send file over */
    char buf[BUFSIZE]; /* current chunk of file */
    int buf_len;        /* bytes in buffer; <= BUFSIZE */
    int buf_used;       /* bytes from buffer sent so far; <= m_buf_len */
    enum { IDLE, SENDING } state; /* what we're doing */
};
```

Create two functions: a new `sendFile()` function, which opens the file and sets up the state in the `sendfile_state` struct; and a `handle_io()` function, which may read a chunk from the file and sends data over the socket. The `handle_io()` function should return and 0 when work remains and a nonzero value when the file has been completely sent over the network (The read may be blocking; it is only the socket I/O that needs to be nonblocking.)

Part 2 (20 points). Describe how the main function would call the functions I've described. Provide implementations for `sendFile()` and `handle_io()`. You may assume the existence of function `setNonblocking()`, which will set the mode of the socket to non-blocking mode. You may omit error-handling code.

Answer.

2 Dependencies and OpenMP

Assume that `A` and `C` are properly allocated arrays. After the completion of `slow`, the program reads the values of `C`.

```
#include <vector>

int foo(int i);
int bar(int i);
int baz(int i);

void work(int * A, int * C, unsigned size)
{
  for (unsigned i = 1; i < size; ++i) {
    A[i] = foo(A[i-1]);
  }
  std::vector<int> B(size * 2); // vector of (size * 2) elements
  for (unsigned i = 0; i < size; ++i) {
    B[i*2] = bar(A[i]);
    B[i*2+1] = baz(A[i]);
  }
  for (unsigned i = 0; i < size; ++i) {
    if (B[i*2] > B[i*2+1]) {
      C[i] = foo(A[i] * B[i*2]);
    }
    else {
      C[i] = foo(A[i] - B[i*2+1]);
    }
  }
}

void swap(int * x, int * y);

void slow(unsigned iterations, int * A, int * C, unsigned size)
{
  for (unsigned i = 0; i < size; ++i) {
    A[i] = i;
  }
  for (unsigned i = 0; i < iterations; ++i) {
    if (i != 0) {
      swap(A, C);
    }
    work(A, C, size);
  }
}
```

Part 1 (10 points). For every `for` loop in the `work` function, identify the loop-carried dependencies. (Write your answers on the code itself.)

Part 2 (15 points). Assume you are only allowed to modify the `work` function. Use OpenMP and any reordering you wish to optimize the code. Argue your function preserves the behaviour of the original.

Answer.

3 OpenMPI and MapReduce

This OpenMPI code has a number of problems. One is sending too much information.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

struct histogramEntry {
    int zipcode; int num;
};

int initializeLogs(int ** logs) {
    // omitted initialization code...
}

struct histogramEntry * countZips(int * logs, int start, int count) {
    struct histogramEntry * results =
        malloc(100000*sizeof(struct histogramEntry));
    for (int i = start; i < count; i++) results[logs[i]].num++;
    return results;
}

int main (int argc, char * argv[]) {
    int rank, size, count;
    int * logs;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    if (rank == 0) {
        count = initializeLogs(&logs);
        for (int i = 0; i < size; i++) {
            MPI_Send(&count, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(logs, count, MPI_INT, i, 0, MPI_COMM_WORLD);
            // also, collate results returned from worker threads.
        }
    } else {
        MPI_Status stat;
        MPI_Recv(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat);
        MPI_Recv(logs, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat);
        int local_size = count / (size-1);
        struct histogramEntry * result =
          countZips(logs, (rank-1) * local_size, local_size);
        // also, return results to master
    }
    MPI_Finalize();
    return 0;
}
```

Reducing Communications Overhead (10 points). Indicate changes to the code which would reduce the communications overhead. For this part, you do not need to say anything about returning the results to the master thread, nor about combining the results.

MapReduce (15 points). Instead of using OpenMPI, we could reformulate this as a MapReduce problem. (For this part, imagine that the code actually did collate the results from the worker threads and produced a combined histogram.) Describe a suitable mapper function (explaining the format of its input, the algorithm, and the output) and reducer function.

Answer.

4 GPU Computation: Password Cracking

Cyclic Redundancy Checks (CRCs) are not an acceptable hash function for passwords. However, for the purposes of this question, we'll pretend that they are. You can think of a CRC as a checksum which is resilient against transposition.

```
typedef uint16_t crc; // 65,536 possible CRCs

#define WIDTH  (8 * sizeof(crc))
#define TOPBIT (1 << (WIDTH - 1))

crc  crcTable[256];

void crcInit(void) {
    crc  remainder;

    for (int dividend = 0; dividend < 256; ++dividend) {
        remainder = dividend << (WIDTH - 8);

        for (uint8_t bit = 8; bit > 0; --bit) {
            if (remainder & TOPBIT) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder = (remainder << 1);
            }
        }
        crcTable[dividend] = remainder;
    }
}

crc computeCRC(uint8_t const message[], int nBytes) {
    uint8_t data;
    crc remainder = 0;

    for (int byte = 0; byte < nBytes; ++byte) {
        data = message[byte] ^ (remainder >> (WIDTH - 8));
        remainder = crcTable[data] ^ (remainder << 8);
    }
    return remainder;
}
```

`[Source: http://www.barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code]`

Assume that you have a file containing CRCs of passwords and that you would like to recover the passwords using OpenCL. The passwords are all 6 characters from `[0-9A-Za-z]`, i.e. ASCII range (48-57, 65-90, 97-122). Your approach is going to be: using the GPU, compute all of the CRCs for 6-character passwords from the given range and put passwords into a buffer indexed by CRC. In your kernel(s), you may assume the existence of `decode_char()` which converts from an int in the range $(0, 62)$ to a character in `[0-9A-Za-z]` and the existence of function `atomic_table_write(buffer, index, c1, ..., c6)` which atomically writes the 6 characters of the password to the appropriate index in `buffer`.

Part 1 (6 marks). Explain all the buffers that you would declare in the host code and state whether each buffer should be read-only, write-only, or read/write.

Part 2 (4 marks). Describe the global range you'd pass to your `enqueueNDRangeKernel` call(s).

Part 3 (15 marks). Write relevant OpenCL kernel implementation(s). Include the important points; syntax and exact types aren't the most important thing, problem structure is. Argue that your implementation is thread-safe.

5 Compiler Optimizations

Consider the `visit` function below.

```
struct elem {
    int data;
    struct elem * next;
};

int getFifth(struct elem * e) {
    for (int i = 0; i < 5; i++) {
        if (e == NULL) return -1;
        e = e->next;
    }
    if (e != NULL) return e->data; else return -1;
}

void visit(struct elem * root) {
    printf("5th element is %d.\n", getFifth(root));
    printf("5th element is still %d.\n", getFifth(root));
}
```

Part 1 (5 marks)

Assume that `printf` has no cost and that `root` is properly initialized to a list of sufficient length. Identify an obvious, yet unsafe, transformation that would halve the work done by `visit`.

Transformation:

Part 2 (10 marks)

Now, assume that `visit` belongs to a larger multi-threaded program. Describe a potential action taken by another thread which makes the transformation unsafe.

Why the transformation is unsafe:

Part 3 (10 marks)

(a) What if `visit` belonged to a single-threaded program with no interrupts—would your proposed transformation be safe? Why or why not? (b) Would you expect a compiler to automatically carry out this transformation? What would stop it?

Answers:

6 Race Conditions/Thread Safety

Consider the following code:

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
```

For this question, assume the following global variables: `int a = 1, b = 2, c = 3.`

Part 1 (3 marks)

Let's say that we have two threads. One thread calls `swap(&b, &a)` while the other thread calls `swap(&b, &c)`. Write down all possible expected outputs of the values of the variables after the calls complete, assuming that `swap` is thread-safe.

Expected outputs:

Part 2 (6 marks)

Assume each line of code is atomic. Write down an interleaving of the code in two separate threads that shows that `swap` is not in fact thread-safe, along with the final output. That is, your output should not match either expected result from Part 1. You should only have code in one thread for each row of the table. (Use a scrap piece of paper and write your final answer in the table.)

Thread 1: `swap(&b, &a)`	**Thread 2:** `swap(&b, &c)`	**a**	**b**	**c**

Actual output:

Part 3 (4 marks)

Explain briefly (one sentence) what the `restrict` keyword does. Assume we label both parameters to `swap` with `restrict`. Would this make the function thread-safe? Explain why or why not. If it does make the function thread-safe, ignore the locking portion when completing Part 4.

Part 4 (12 marks)

Assume that in your code you create a mutex, `m`, and properly destroy it when the program ends. Write below where you would lock and unlock the mutex to make the code thread-safe. (Pseudocode like `lock(m)` is OK.) Use the finest grain locking possible with a single lock.

```
void swap(int *x, int *y) {

    int t = *x;

    *x = *y;

    *y = t;
}
```

Show what happens with the same interleaving (from Part 2) on your lock-enabled code. Verify that you get one of the expected outputs. Explain why the code is now thread-safe in all cases.

Thread 1: `swap(&b, &a)`	**Thread 2:** `swap(&b, &c)`	**a**	**b**	**c**

Actual output:

Explanation: