CS 314 OpenMP Parallel Programming Project
Due Date: Friday, December 15, 2017, 11:59pm

You are given a sequential C implementation of a (probabilistic) spell checker based on Bloom filters, and your job is to parallelize it using OpenMP for C. The following, we will explain the Sequential C implementation and clarify what kind of parallelism you should exploit in this project.

1. C Sequential Implementation

This section presents some helper in subsections 1.1 and 1.2 that includes the implementation of the hash functions and some helper functions used in the sequential implementation. Subsection 1.3 explains the sequential implementation that uses the helper functions and the hash functions, defined in subsection 1.1, 1.2. You can compile this version by simply writing **make** command in the project directory.

1.1 hash.h(declaration), hash.c (Implementation)

These files include implementation of 15 hash functions given to you. You do not have to implement any hash functions in this project. The following code the header file code that illustrates how to use these functions. Note here that **`HashFunction`** is a function pointer that can point to any of the 15 functions that follows it in declaration below (e.g. **`RSHash, JSHash,…, hash_mult_900`**).

```
#ifndef HASH_H
#define HAS_H

typedef unsigned int (*HashFunction) (const char *str);

unsigned int RSHash(const char *str);
unsigned int JSHash(const char *str);
unsigned int ELFHash(const char *str);
unsigned int BKDRHash(const char *str);
unsigned int SDBMHash(const char *str);
unsigned int DJBHash(const char *str);
unsigned int DEKHash(const char *str);
unsigned int BPHash(const char *str);
unsigned int FNVHash(const char *str);
unsigned int APHash(const char *str);

unsigned int hash_div_701(const char *str);
unsigned int hash_div_899(const char *str);
unsigned int hash_mult_699(const char *str);
unsigned int hash_mult_700(const char *str);
unsigned int hash_mult_900(const char *str);
#endif
```

1.2 word_list.h (declaration), word_list.c(Implementation)

These files include little helper functions that are already used in the sequential program that is given to you. `create_word_list(const char *path);` reads the word list of the dictionary from the dictionary file (i.e. path) on disk and stores it in `word_list` struct. `get_num_words(word_list * wl)` returns the number of words in the dictionary stored in `word_list` struct. `get_word(word_list * wl, size_t index)` returns the word at location `index` in the word list.

```
typedef struct {
     char **words;
     size_t num_words;
} word_list;

word_list *create_word_list(const char *path);
const char *get_word(word_list * wl, size_t index);
size_t get_num_words(word_list * wl);
void destroy_word_list(word_list * wl);
```

1.3 spell_seq.c (original program file).

This is the sequential implementation of the spell checker. We present it here into three subsections .

1.3.1 Loading the dictionary file stored in `"word_list.txt"` by calling `create_word_list` function and get number of read words using `get_num_words` function.

```
word_list *wl;
wl = create_word_list("word_list.txt");
if (!wl) {
    fprintf(stderr, "Could not read word list\n");
    exit(EXIT_FAILURE);
}
wl_size = get_num_words(wl);
```

1.3.2 Creating the bit vector (The main portion to parallelize in your project)

- In this part you have hf array already declared for you as follows. hf is an array function pointer. Each element in this list points to one of the already implemented hash functions as follows:

```
HashFunction hf[] = {RSHash, JSHash, ELFHash, BKDRHash, SDBMHash,
      DJBHash, DEKHash, BPHash, FNVHash, APHash, hash_div_701,
      hash_div_899, hash_mult_699, hash_mult_700, hash_mult_900};
```

- The code to allocate the bit vector in C is shown below. For our particular spell checker, we created a bit vector of size 100,000,000. Do not modify the size.

```
/* allocate the bit vector (bv)*/
char *bv;
bv_size = 100000000;
num_hf = sizeof(hf) / sizeof(HashFunction);
bv = calloc(bv_size, sizeof(char));
if (!bv) {
      destroy_word_list(wl);
      exit(EXIT_FAILURE);
}
```

- The code to set the entries in the bitvector.

  Here, there are two nested **for** loops that fill the bit vector (bv) . For each word in the dictionary, each of 14 hash functions stored in the **hf** array is called and the corresponding location in the bit vector is set to 1.

```
/* create the bit vector entries */
for (i = 0; i < wl_size; i++) {
      for (j = 0; j < num_hf; j++) {
            hash = hf[j] (get_word(wl, i));
            hash %= bv_size;
            bv[hash] = 1;
      }
}
```

1.3.3

Having created the bit vector, this sequential spell checker program takes a word to check as the 1st command line argument and checks if the word is spelled correctly using the created bit vector **`bv`.**

```
/* do the spell checking */
misspelled = 0;
for (j = 0; j < num_hf; j++) {
    hash = hf[j] (word);
    hash %= bv_size;
    if (bv[hash] == 0)
        misspelled = 1;
}
```

2. Parallelization Task

In this project, you are asked to only exploit loop-level parallelism in the given sequential program. You will express loop-level parallelism through OpenMP pagmas, i.e. `#pragma omp parallel variations`. You are allowed to perform two loop level transformations, namely loop interchange and loop distributions to reshape loops in order to expose more exploitable loop-level parallelism in OpenMP. Other transformations are not allowed, or using other forms of parallel constructs.

You will need to submit four OpenMP versions of the spell_seq.c code, named spell_t2_singleloop.c, spell_t2_fastest.c, spell_t4_singleloop.c, and spell_t4_fastest.c. **Do not change the bit vector size or the applied hash functions.**

1. **spell_t2_singleloop.c** : A version that uses exactly 2 cores or threads and exploits parallelism in only a single loop level. In other words, you can only add a single #pragma to the code.
2. **spell_t2_fastest.c**: A version that uses exactly 2 cores or thread and runs as fast a possible (feel free to parallelize as many loops and loop levels as you believe are beneficial to improve the program's performance.
3. **spell_t4_singleloop.c** : A version that uses exactly 4 cores or threads and exploits parallelism in only a single loop level. In other words, you can only add a single #pragma to the code.
4. **spell_t4_fastest.c**: A version that uses exactly 4 cores or thread and runs as fast a possible (feel free to parallelize as many loops and loop levels as you believe are beneficial to improve the program's performance.

Note that all four versions could be identical. Loop-level parallelism has its overhead, so choosing the right loop level(s) to parallelize can be crucial to achieve the best possible performance.

3. Project Template and Compilation

You are given **a project3 student** directory that contains the following files

1. Helper files (i.e. word_list.c, word_list.h, hash.c, hash.h): **DO NOT CHANGE THESE FILES.**
2. spell_seq.c : This file contains the sequential implementation of spell checker. When you run this program you will be able to see the time taken on the sequential version to create the spell checker. **DO NOT CHANGE THIS FILE.**
3. **spell_t2_singleloop.c:** This file is just a copy of spell_seq.c with number of threads set for you as 2 (using omp_set_num_threads(2)). In this version, you should ***implement*** a version that uses exactly 2 threads (already set for you) and exploits parallelism in only a single loop level.

4. **spell_t2_fastest.c:** This file is just a copy of spell_seq.c with number of threads set for you as 2 (using omp_set_num_threads(2)). In this version, you should ***implement*** a version that uses exactly 2 thread (already set for you) and runs as fast a possible.
5. **spell_t4_singleloop.c:** This file is just a copy of spell_seq.c with number of threads set for you as 4(using omp_set_num_threads(4)). In this version, you should ***implement*** a version that uses exactly 4 threads (already set for you) and exploits parallelism in only a single loop level.
6. **spell_t4_fastest.c:** This file is just a copy of spell_seq.c with number of threads set for you as 4(using omp_set_num_threads(4)). In this version, you should ***implement*** a version that uses exactly 4 thread (already set for you) and runs as fast a possible

3. How to Get Started?

Compilation

Generally, to compile a file with openMp, you just need to add –fopenmp as an argument in gcc . For instance, **spell_t2_singleloop.c** version could be compiled as follows. **gcc spell_t2_singleloop.c hash.c word_list.c -O0 -ggdb -Wall -fopenmp -lrt -lm -o spell_t2_singleloop.**

A Make file was written for you to compile any of the 5 versions of the spell checkers, or all of them at once. The instructions follow for the make commands.

make spell_seq	: compiles only the sequential version (spell_seq).
make spell_t2_singleloop	**:** compiles only **spell_t2_singleloop.** .
make spell_t2_fastest	**:** compiles only **spell_t2_fastest**.
make spell_t4_singleloop	**:** compiles only **spell_t4_singleloop.**
make spell_t4_fastest	**:** compiles only **spell_t4_fastest**.

make all or make: compiles all the 5 versions (i.e. spell_seq, spell_t2_singleloop.c, spell_t2_fastest, spell_t4_singleloop.c, spell_t4_fastest)

make clean : deletes the executable and the obj files

4. What Parallel Machine to Use?

You will need to implement your codes on the ilab machines. There is an incomplete list of ilab machines with their number of cores / parallel threads.
You may want to choose an ilab machine with many cores to get benefit of parallel execution (e.g. Quadcore machines, Core i5 machines). We will publish a list of these machines later.

4. Grading

You will be graded based on (a) **correctness** and (b) **performance**, i.e., how close your four versions were with respect to our sample parallel program versions**. You are not allowed to change the original source code with the exception of adding OpenMP loop-level pragmas and performing loop interchange and/or loop distribution. For example, you are not allowed to add any printfs in your submitted program versions.**

At a later time, we will post the performance improvement requirements for your code versions in order to get full credit. **Correctness is more important than performance.**

5. Project Questions

All project related questions should be posted on our sakai project 3 piazza forum. Thanks.