CS 314 OpenMP Parallel Programming Project
Due Date: Monday, May 1, 2023, 11:59pm

You are given a sequential C implementation of a (probabilistic) spell checker based on Bloom filters very similar to the one you implemented in Project 2. Your job is to parallelize the sequential code using loop level parallelism, i.e., parallel loops, and loop transformations in OpenMP for C. You will also submit a report that discusses your experimental evaluation of your parallelized code versions.

Please note: In contrast to Project 2, each time you invoke the spell checker, the program recomputes the entire bitvector used for the Bloom filter.

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.

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.

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);
```

Creating the bit vector

- 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 15 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;
   }
}
```

Using the bit vector

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 and loop scheduling strategy can be crucial to achieve the best possible performance.

3. How to Get Started?

You need to copy proj3.tar into your local directory on ilab. As with the other projects, you must read protect your project directory in order to avoid cheating. The project tar file is available on canvas and contains a Makefile, files related to a wordlist, the five versions of the spell checker, and four sample solutions as executables.

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 as 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 as possible.
7. <name>-sol are sample solutions for the four parallelized spell checkers. Your solutions should achieve similar or better execution times than these sample solutions.

You can generate the different versions of the code by saying "make". Please see the provided Makefile for details.

4. Project Code and Report

You will submit a project report together with the source code of your four parallelized code versions. The report must include execution times for at least 10 experiments for each of your four codes. The experimental results should be reported in the form of a table, with a row for each code version and columns for each code execution. The execution times must be reported in milliseconds. You also should report the average and median execution times. Reporting the standard deviation would also be useful.

You should run your experiments on an ilab machine that is not too loaded. Your report should list the name of the particular ilab machine that you are using for your experiments. For a fair comparison across all code versions, you should stick with a single ilab machine for all experiments.

Your report must be in pdf format and should not be longer than a single page. Your project report must include the experimental results and their discussion. Use latex or Microsoft word to generate the report.

5. Project Submission and Grading

Please submit a single tar file with your project report (report.pdf), the four source code versions, and an optional ReadMe file. **DO NOT submit any executables or object files**. You will be graded based on (a) correctness and (b) performance of your four parallel code versions, i.e., how close your four versions were with respect to our sample solutions. We will recompile your code which means that if your code does not compile and run on the ilab cluster, you will not receive any credit.

Your project report is 30% of your grade. If you code does not compile or run on ilab, you will receive no credit for your report.

6. Project Questions

All project related questions should be posted on piazza. Thanks.