CS 314 Principles of Programming Languages OpenMP Parallel Programming Project Due Date : Wednesday, Dec 10 2014, 11:59pm

In this project, you will implement a parallelized version of Project 2 (i.e. spell checker that is based on hash functions).

You are given a sequential C implementation of the version and your job is to parallelize it using OpenMP. The following two sections will explain the Sequential C implementation and clarify what kind of parallelism you should be restricted to during this project

1. C Sequential Implementation

This section present 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_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 (program file).

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

1.3.1 Creating the word list (internal representation of the dictionary)

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

- 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_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 100000000. 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 that are stored in 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; /* modulo operation */
                bv[hash] = 1;
        }
}
```

1.3.3 Using the bit vector to spell check the input word

Having created the bit vector , the sequential spell checker program takes a word as its 1st command line argument as input and checks if the words is spelled correctly using the created bit vector bv and the different hash functions.

```
/* do the spell checking */
misspelled = 0;
for (j = 0; j < num_hf; j++) {
        hash = hf[j] (word);
        hash %= bv_size; /* modulo operation */
        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.

You will need to submit four OpenMP versions of the spell checker, named spell_t4_singleloop.c, spell_t4_fastest.c, spell_t8_singleloop.c, and spell_t8_fastest.c. Do not change the bit vector size or the applied hash functions.

1. spell_t4_singleloop.c :A version that uses exactly 4 threads and exploits parallelism in only a single loop level.
2. spell_t4_fastest.c : A version that uses exactly 4 thread and runs as fast a possible (feel free to parallelize as many loop levels as you believe are beneficial to improve the program's performance.
3. spell_t8_singleloop.c: A version that uses 8 threads and exploits parallelism in only a single loop level.
4. spell_t8_fastest.c : A version that uses 8 threads and runs as fast as possible.

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

3. Project Template and Compilation

You are given a project3_template 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_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.
4. **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.
5. **spell_t8_singleloop.c**: This file is just a copy of spell_seq.c with number of threads set for you as 8(using omp_set_num_threads(8)). In this version, you should implement a version that uses exactly 8 threads (already set for you) and exploits parallelism in only a single loop level.
6. **spell_t8_fastest.c**: This file is just a copy of spell_seq.c with number of threads set for you as 8(using omp_set_num_threads(8)). In this version, you should implement a version that uses exactly 8 thread (already set for you) and runs as fast a possible

Compilation

Generally to compile a file with openMp, you just need to add –fopenmp as an argument in gcc . For instance, **spell_t4_singleloop.c** version could be compiled as follows. **gcc spell_t4_singleloop.c hash.c word_list.c --O0 --ggdb --Wall --fopenmp --lrt --lm --o spell_t4_singleloop**

A Make file was written for you to compile any of the 5 versions of the spell checkers. The instructions follow for the make commands.

make spell_seq : compiles only the sequential version (spell_seq).
make spell_t4_singleloop : compiles only spell_t4_singleloop.
make spell_t4_fastest : compiles only spell_t4_fastest.
make spell_t8_singleloop : compiles only spell_t8_singleloop.
make spell_t8_fastest : compiles only spell_t8_fastest.

make all or make: compiles all the 5 versions (i.e. spell_seq, spell_t4_singleloop, spell_t4_fastest, spell_t8_singleloop, spell_t8_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.

Go to http://report.rutgers.edu/mrtg/index.html, and select choose Instruction lab, you will have the list of ILAB machines that you can use. You better choose Intel Machines with as many cores as possible to get benefit of the number of cores (e.g. Quadcore machines, Core i5 machines).

5. 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 much more important than performance.

6. Project Questions

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