CS 314 Principles of Programming Languages- OpenMP Programming Project
Due Date : Monday, May 5 2014, 11:59pm

In this project, you will implement a parallelized version of Project 2 (i.e. Ceasar's cipher code breaker).

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 using to do this project.

1. C Sequential Implementation

This section explains the sequential implementation. You can compile this version by simply writing **make** command in the project directory. Do not modify the sequential implementation; you have to use it only as a baseline to calculate the sequential running time.

1.1) encode_decode_seq.c

This file contains the main function. The program takes in 2 command line arguments as input. The input is in the form

```
./encode_decode_seq <document_to_encode> <n>
```

Here "n" represents the amount by which the letters in the document are "shifted" up and 0 <= n <= 25. The main function calls function "encode_decode" twice. The first call encodes the document and then decodes it using the "Brute Force with Spell Checker" method used in project 2. The second call encodes the input document and decodes it using the "Frequency Analysis" method used in Project 2.

The "encode_decode" function takes 3 arguments namely the "file_name" which is the input document's name, "n" and a "decode_choice" variable to communicate which decoding method to use (brute force or frequency analysis). This function can be broken down into 4 parts.

1) Reads the file whose name is specified by the argument "file_name" into a three dimensional array called "document".

2) Encode the array by shifting each character by the value "n". The code that does this as given below.

```
for (i = 0; i < WORDS_PER_PARA; i++) {
  for (j = 0; j < PARAS_PER_DOC; j++) {
    for (k = 0; k < strlen (document[j][i]); k++) {
      document[j][i][k] = encrypt(document[j][i][k], n);
    }
  }
}
```

The encrypt function convert the character to a value (between 0 and 25), adds "n" to it, ensures that the result of the add is between 0 and 25 by applying modulus (%) to it and returns the character that corresponds to the result of modulus. Here is the code of the encrypt function.

```
char encrypt (char ch, int n) {
  return vtc((ctv(ch) + n) % 26);
}
```

3) Calls either the "decode_n" function or "decode_f" function depending on the decode_choice input argument. The decode_n function performs the decoding using the brute force method and returns "n_dash", which is the amount by which the encoded document needs to be "shifted" up in order to decode it. Similarly, the "decode_f" function performs the decoding using the frequency analysis method and returns "n_dash".

4) On obtaining "n_dash", the encoded document is decoded by adding n_dash to each encoded character in the "document" array. The code for this is the same as that in step 2.

1.2) decode_n_seq.c

This file includes the code to perform the decoding using the Brute Force spell checker method. This is done in 2 steps.

1) Load the dictionary file into an array. The dictionary is provided in the file named "dictionary.txt". This implementation expects the dictionary file to reside at the same folder where the source files exist. So do not move or modify the dictionary file.

2) Perform spell checking. For every word in the "document" array, this function returns the value of "n_dash" that resulted in the word being spelt correctly (at least as far as the given dictionary is concerned). This is done in the function "get_ndash". The code of the function is as given below.

```
for (i = 0; i < NUMBER_OF_CHARS; ++i) {
  strcpy(tmp_str, word);
  for(j = 0; j < strlen(tmp_str); ++j) {
    tmp_str[j] = vtc((ctv(tmp_str[j]) + i) % 26);
  }
  for(j = 0; j < DICT_SIZE; j++) {
    if (strcmp(tmp_str, dictionary[j]) == 0)
      n_dash = i;
  }
}
return n_dash;
```

"NUMBER_OF_CHARS" is defined to be 26 in the lib.h file. So for each "i" of the possible 26 values, this function shifts each letter of the input word by I and checks if the word belongs to the dictionary. If it does, then that is the value of "n_dash" that is returned.

This step is repeated for each word in the document and the value of n_dash for which most words are spelled correctly is assumed to be the decoding value.

1.3 decode_f_seq.c

This code breaker is based on the fact that in English some characters occur more often than others. The function "decode_f" performs the following 2 steps.

1) Gets the most frequently occurring character in the input encoded array.

2) Assuming 'e' is the most occurring character in English, returns the value "n_dash" which is the amount by which the encoded value needs to be shifted to obtain 'e'. As in detailed in project 2, this is based on the intuition that since 'e' was the most occurring character in the input document, the shift between the encoded latter and 'e' has to be the value of "n" that used to encode the input document. So all that needs to be computed is the value of "n_dash" that can be used to shift the most occurring character in the encoded document back to 'e'.

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 for 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 write four OpenMP parallel versions of the sequential implementation of encode_decode_seq.c, named encode_decode_t2_singleloop.c, encode_decode_t2_fastest.c, encode_decode _t4_singleloop.c, and encode_decode _t4_fastest.c. In addition, you are asked to parallelize the sequential code in files decode_n_seq.c and decode_f_seq.c. Functions defined in these files are called from encode_decode_seq.c. You will need to submit two parallelized versions of each of these two files. In total, you will need to submit eight files as part of this project.

Do not change any code beyond the two allowed loop level transformations (loop distribution and loop interchange), just add OpenMP directives.

1. **encode_decode_t2_single_loop.c**: A version that uses exactly 2 threads and exploits parallelism in only a single loop level. **This means that you can only use a single #pragma within the entire file. So you need to decide not only which loop to parallelize, but also which level of that loop to parallelize.**
2. **encode_decode_t2_fastest.c**: A version that uses exactly 2 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. **encode_decode_t4_single_loop.c:** A version that uses 4 threads and exploits parallelism in only a single loop level. **This means that you can only use a single #pragma within the entire file. So**

you need to decide not only which loop to parallelize, but also which level of that loop to parallelize.

4. **encode_decode_t4_fastest.c**: A version that uses 4 threads and runs as fast as possible.
5. **decode_n_single_loop**.c and **decode_f_single_loop**.c: This contains the code to perform the decoding using the brute-force method and the frequency analysis method. If you look at the makefile, these file are used by both the single loop versions, i.e encode_decode_t2_singleloop.c and encode_decode_t4_singleloop.c. You will have to parallelize both these files when you parallelize the single loop versions. **For the single version, you can pick at most a single loop level to parallelize in each of the files, i.e., you can use only a single #pragma.**
6. **decode_n_fastest**.c and **decode_f_fastest**.c: This contains the code to perform the decoding using the brute-force method and the frequency analysis method. If you look at the makefile, these file are used by both the fastest versions, i.e encode_decode_t2_fastest.c and encode_decode_t4_fastest.c. You will have to parallelize both these files when you parallelize the single loop versions. **For the fastest version, you may pick as many loop levels for parallelizations as you wish.**

For the single_loop version, you can choose to parallelize at most one loop level in each of the involved files, i.e., specify at most one #pragma in each of the decode_n_single_loop.c, decode_f_single_loop.c, and endoce_decode_?_single_loop.c files.

Totally there are 8 files that contain loops and you would need to parallelize loops in all the 8 files using the single_loop or the fastest methods. Create a tarball of the entire proj3 directory and submit it on sakai. You can use this command to create a tarball.

```
> tar -cvzf  proj3.tgz proj/
```

Note that the running times of 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 students** directory that contains the following files

1. Header files and lib file (i.e. lib.h, lib.c, decode_f.h, decode_n.h): **DO NOT CHANGE THESE FILES.**
2. **encode_decode_seq.c**: This file contains the sequential implementation of the encoder and decoder. When you run this program you will be able to see the time taken for the sequential execution. **DO NOT CHANGE THIS FILE.**
3. **decode_f_seq.c/decode_n_seq.c:** These files contain the code that performs decoding sequentially. **DO NOT CHANGE THIS FILE.**
4. **encode_decode_t2_singleloop.c:** This file is just a copy of encode_decode_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.

5. **encode_decode_t2_fastest.c:** This file is just a copy of encode_decode_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.
6. **encode_decode_t4_single_loop.c:** This file is just a copy of encode_decode_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.
7. **encode_decode_t4_fastest.c:** This file is just a copy of encode_decode_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.
8. **decode_f_single_loop.c/decode_n_single_loop.c:** These files are just copies of decode_f_seq.c and decode_n_seq.c. You should parallelize loops in these files too when you are parallelzing encode_decode_t2_single_loop.c or encode_t4_single_loop.c.
9. **decode_f_fastest.c/decode_n_fastest.c:** These files are just copies of decode_f_seq.c and decode_n_seq.c. You should parallelize loops in these files too when you are parallelzing encode_decode_t2_fastest.c or encode_decode_t4_fastest.c.

Compilation
Generally to compile a file with openMP, you just need to add –fopenmp as an argument in gcc. A Make file has been provided for you to compile any of the 5 versions. The instructions follow for the make commands.

make encode_decode_seq	: compiles only the sequential version (encode_decode_seq).
make encode_decode_t2_single_loop	: compiles only **encode_decode_t2_single_loop.**
make encode_decode_t2_fastest	: compiles only **encode_decode_t2_fastest**.
make encode_decode_t4_single_loop	: compiles only **encode_decode_t4_single_loop.**
make encode_decode_t4_fastest	: compiles only **encode_decode_t4_fastest**.
make all or make	: compiles all the 5 programs.
make clean	: deletes all the executables 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).

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 much more important than performance.

5. Project Questions

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