CSCI-442 Project 3: Parallel Zip

> ***Important***
>
> - You'll want to read this **entire document** before beginning the project. Please ask any questions you have on the discussion board.
> - Finally, be sure to start early. If you wait until a few days before the due date, you are unlikely to finish in time.

See Canvas for the project due date.

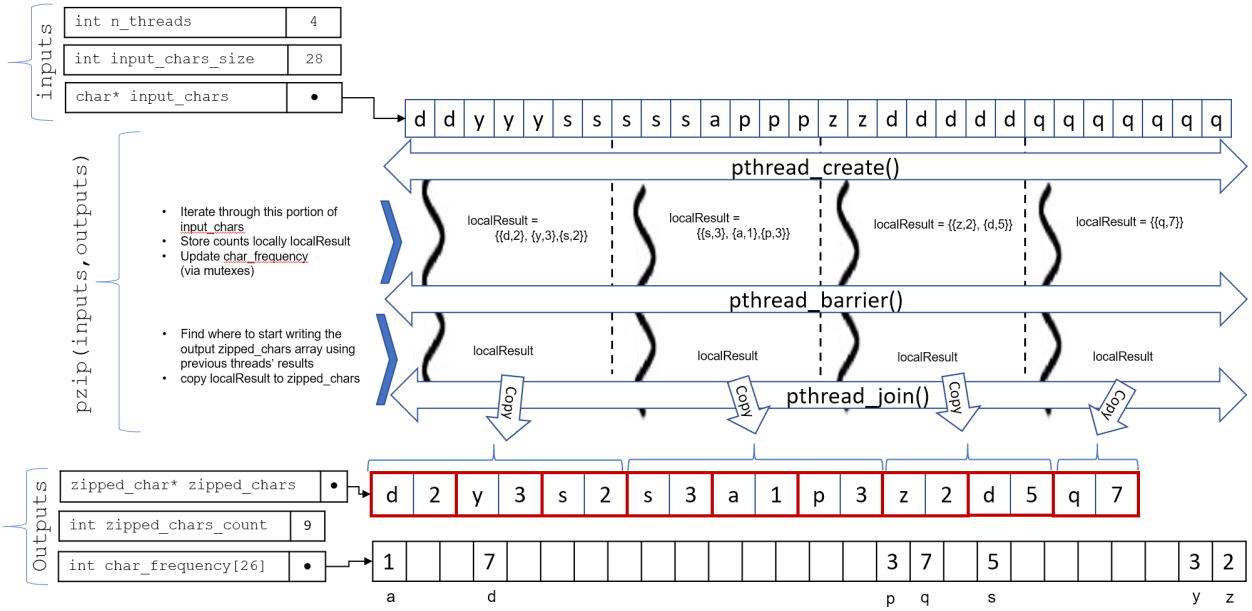
1) Project Desciption

1.1) Learning Objectives

- Understand how to use multiple threads to finish a computational task faster.
- Become familiar with running concurrent threads with POSIX-threads library (pthreads) and the basic usage of concurrent data structures among parallel threads.
- Sharpen systems programming skills by working on a practical project.

1.2) Summary

For this project, you will implement a parallel zip (pzip) program, using the C programming language and POSIX-threads. The pzip will read an input text file, which is composed of lowercase letters (a-z) only. As an output it will produce a binary file indicating the consecutive uses of each character. The pzip operation needs to be done in parallel using pthreads library.



The figure above shows an overview of the inputs, outputs and your program will flow. You are expected to implement the middle portion indicated by `pzip(inputs,outputs)`. Within this function, there are three major steps that you should follow to establish the parallel threads logic:

1. Call `pthread_create()` to launch parallel threads: Once threads are created they will iterate through an equal and dedicated portion of `input_chars`, store the consecutive occurrence results locally, and also update a global `char_frequency` array that holds the total/global frequency of the occurrences of each character.
2. Call `pthread_barrier()` to synchronize the pthreads, without destroying them: This barrier is required to make sure that each thread has finished locally counting their portions of characters. Threads need to synchronize because each thread needs to know how many `zipped_char` structs they have in their `localResult` arrays, so that they can calculate the exact index of `zipped_char` array that they need to copy their local results into.
3. Call `pthread_join()` to finish the parallel execution and synchronize the pthreads again.

Please note that your code is expected **ONLY** to operate on the input and output data structures provided in the figure.

> ***Warning***
>
> `zipped_chars` array holds consecutive occurrences, whereas `char_frequency` array holds the total number of occurrences. A character may appear more than once in `zipped_chars` array, whereas `char_frequency` is populated on-the-go as threads encounter each character.

1.3) Functionality

After running `make`, you should have an executable program named `pzip` located in the root of your repository. The usage is as follows:

```
pzip INPUT_FILE OUTPUT_FILE N_THREADS [--debug]
```

- `INPUT_FILE`: The input file name which is contains only lowercase letters (a-z). The format is explained below.
- `OUTPUT_FILE`: The output file name which will be the result of your program output. The format is explained below.
- `N_THREADS`: The number of parallel threads that will be used during pzip.
- `[--debug]`: Optional parameter to display the content of input/output variables. When this option is set, the output becomes a human readable text file. Otherwise, it is a binary file. We will test your program without this parameter.

1.4) Assumptions

- The number of threads is assumed to be greater than or equal to zero.
- The number of characters in the input file is assumed to be a positive multiple of the number of threads.
- Each thread is expected to process an equal portion of the input characters.
- **IMPORTANT:** If the consecutive occurrence of a character spans two threads, you **SHOULD NOT** merge these two occurrences and **SHOULD** report them separately. For example, in the image above, the character s appears twice for the first thread and, again, the same letter appears three times in the portion processed by the second thread. In the final output, rather than counting s for 5 times, we have two counts of it, which are 2 and 3, respectively.

- The number of the same character in a row will not exceed 255 (i.e., the maximum value of a `uint8_t`)

1.5) What is implemented for you?

- main() function in `main/pzip.c` (**DO NOT TOUCH THIS FILE**)
- Program parameter handling
- Input file reading, parsing and coversion (i.e., mapping) to `input_chars` array
- Output file formatting, writing and coversion (i.e., mapping) from `zipped_chars` array
- Debug output

To ease your implementation and to make grading fairer, using the starter code *is a requirement of this project*. You are **NOT ALLOWED** to make any modification to the `main/pzip.c` file. You may add new structs or functions to the `main/pzip.h` file. However, you **SHOULD NOT** change/delete/modify existing functions/variables/headers/structs in `main/pzip.h` file.

1.6) What are you expected to do?

- **START FROM HERE**: Implement body of the `pzip()` function in `src/pzip.c`
- In the same file, also create a callback pthread function that will be called by `pthread_create()` within `pzip()` function.
- By the end of pzip() function, properly populate output pointers to `zipped_chars`, `zipped_chars_count` and `char_frequency`. Please note that the memory for these arrays and variables will be allocated and freed for you by main().
- Implement other functions and global/local variables as you need.

1.7) Input File Format

The input file is already parsed into an `input_chars` array for you and the total number of chars in this file is provided to you in the `input_chars_size` variable. The input file is simply a text file that contains nothing but the 26 lower case letters (i.e., a-z). There are no white spaces, line breaks, return characters or other characters. While you don't need to parse the input file, you need to know the format so that you can prepare your own test inputs. Example input:

- Input file content: `aaeeooooooeee`
- `int input_chars_size = 12;`
- `char* input = {'a','a','e','e','o','o','o','o','o','e','e','e'};`

You should generate inputs of any size of NUM by using the following script in your repository root:

```
$ ./generate_chars.py NUM > test_input
```

> ***Warning***
>
> **DO NOT GENERATE FILES MANUALLY USING A TEXT EDITOR** The input files are only lower case letters with nothing else. The use of text editors may accidentally inject a newline into your input file. Please use the provided python script or run the command below to clean your files of any newlines. $ cat FILENAME | tr -d 'n' > NEWFILENAME

1.8) Output Format

There are two output formats used by the program. Both of these formats are generated by the starter code using the `zipped_chars` array.

1. Binary Output (Default)

If the `--debug` option is not provided, the output of pzip is a binary file. This file is automatically generated using `zipped_chars` array and `zipped_chars_count` variable, which indicates the size of the array.

2. Text Output (--debug mode)

If the `--debug` option is provided in the program arguments, the contents of the `zipped_chars` array will be written as human readable text file. Each line of the output file will consist of a charracter and the number of consecutive occurences of that character.

Examples

Example 1

- Input file content: `aaeeooooooeee`
- `int input_chars_size = 12;`
- `char* input = {'a','a','e','e','o','o','o','o','o','e','e','e'};`
- `int n_threads = 2;`
- `struct zipped_char\* zipped_chars = {{'a',2}, {'e','2'},{'o',2},{'o',3}, {'e',3}};`
- Binary output file (in hexa-decimal):

  ```
  61 02 65 02 6f 02 6f 03 65 03
  ```

- Text (--debug) output file (in plain text, new lines are ommitted): a 2 e 2 o 2 o 3 e 3

Example 2

- Input file content:

  ```
  aaeeooooooeeeeeeeeeeaaaaaaaaaaddddddddddsssssssslssssssyyyyyywwwwww
  ```

- `int input_chars_size = 64;`
- `char* input = {`a`,`a`,`e`,`e`,`o`,`o`,`o`,`o`,`o`,`e`,`e`,`e`,`e`,`e`, `e`,`e`,`e`,`e`,`e`,`a`,`a`,`a`,`a`,`a`,`a`,`a`,`a`,`a`,`a`,`d`,`d`,`d`, `d`,`d`,`d`,`d`,`d`,`d`,`s`,`s`,`s`,`s`,`s`,`s`,`s`,`l`,`s`,`s`,`s`,`s`, `s`,`s`,`y`,`y`,`y`,`y`,`y`,`y`,`w`,`w`,`w`,`w`,`w`,`w`};`
- `int n_threads = 4;`
- `struct zipped_char\* zipped_chars = {{'a',2}, {'e','2'},{'o',5},{'e',7}, {'e',3},{'a',10},{'d',3},{'d',6},{'s',7},{'l',1},{'s',2},{'s',4},{'y',6}, {'w',6}};`
- Binary output file (in hexa-decimal):

  ```
  61 02 65 02 6f 05 65 07 65 03 61 0a 64 03 64 06 73 07 6c 01 73 02 73 04 79
  06 77 06
  ```

- Text (--debug) output file (in plain text, new lines are ommitted):

  a 2 e 2 o 5 e 7 e 3 a 10 d 3 d 6 s 7 l 1 s 2 s 4 y 6 w 6

Example 3

- Input file content: `aaaaaaaaaaaa`
- `int input_chars_size = 12;`
- `char* input = {'a','a','a','a','a','a','a','a','a','a','a','a'};`
- `int n_threads = 4;`
- `struct zipped_char\* zipped_chars = {{'a',3}, {'a','3'},{'a',3},{'a',3}};`
- Binary output file (in hexa-decimal):

  `61 03 61 03 61 03 61 03`
- Text (--debug) output file (in plain text, new lines are ommitted): a 3 a 3 a 3 a 3

> ***Warning***
>
> Note that the zipped output file is not fully compressed. In example 1, the zipped chars could have been determined as `{{'a',2}, {'e','2'},{'o',5},{'e',3}};` . However, for the sake of simplicity for the project, we do not ask parallel threads to talk to each other and merge their output. You are not asked to implement this functionality, and your program may not pass our automated tests if you implement this optimization.

2) Evaluation and Grading

2.1) Grading

We will be grading your code based on:

- **Functionality and accuracy:**

  Your program should produce the output as explained above. Please note that, due to the simplifications we have made, the output may change depending on the number of threads being used, if character sequences span thread boundaries, as in the example given in the figure. Your submitted code should have the same `mains/pzip.c` file as in the starter code.
- **Parallelism and performance:**

  Most of pzip, including input/output and reading/writing, will operate in parallel. Your program should operate as shown in the Figure above. You should **NOT** do the counting of characters serially. Serial creation and joining of threads is OK.
- **General requirements:**

  Your program should follow the non-project-specific general requirements indicated below.

> ***Warning***
>
> You will **NOT** receive performance points if your code is not correct. Slow, but correct programs are **always** more valuable than fast, incorrect programs, and this is reflected in the grading of this project. But also keep in mind that the autograder has a 5 minute timeout as none of the test cases should take longer than that to complete (even input-huge).
>
> Additionally, we will be using `diff` to verify correctness. This means you will *not* get partial credit on a within-test basis (i.e., you will either pass, or fail, each individual test. There is no in-between)

Lastly, if your program crashes during execution, it will be considered "incorrect", *regardless of whether it produces the correct output file*. Due to this it is **VITAL** that all memory issues are taken care of as these can cause code to crash. It is possible to have memory issues even when you don't call malloc especially in this project where memory is being divided up for the threads!

2.2) Performance Measurement

- To test whether your program properly AND efficiently use threads, we will run your program with large test files (e.g. `test/input_large`). We will use the following formula to evaluate the 'parallel efficiency', i.e., `PE`, of your code:

  `PE = ((CPU_TIME_USER+CPU_TIME_SYS)/WALL_TIME) / N_THREADS`

- In a perfectly parallel program, `PE` should be equal to 1.0, however this is never possible. Actual parallel efficiency will be less than 1.0. To measure the `PE` of your program via the `measure.py` script we provided, you may execute the following command:

```
$ ./measure.py ./pzip /tmp/CSCI-442--DO-NOT-DELETE/input_huge ./out 8
```

2.3) Performance Criteria

- On Isengard, our ideal solution for the parameters in the above command runs under 1 second (`WALL_TIME < 1`) with a `PE` greater than 0.75.
- Your program is expected to run the command above on Isengard under 1.5 seconds (`WALL_TIME < 1.5`) with a pe greater than 0.5 (`PE > 0.5`).
    - Please note that these values are valid only for the input file referenced above (`/tmp/CSCI-442--DO-NOT-DELETE/input_huge`) and with `N_THREADS=8` on Isengard.
    - Your first run may be slower due to internal page caching. Within the grading script, your code will be run three times and only the fastest one will be used for grading.
    - Note: If you have a `WALL_TIME < 0.25`, then your PE does not need to meet the requirement of `> 0.5`.
- If your code fails to meet the performance criteria above, you will get a partial grade, depending on how fast and efficient your code is.
    - Reminder: you will receive **NO** performance points if your program does not produce the correct output or crashes.
- **The top three fastest and correct submissions will be given +3, +2 and +1 extra points, respectively.**

2.4) Testing Input Huge

To validate if your output for input_huge is correct, we provide the hash of the correct input_huge output below. output_huge_8t solution hash:

$ 2f7c59a2ff08217dd0ac35fa4a437e92

You can check the hash of your output via the command below

$ md5sum FILENAME

Where FILENAME is the name of your output file.

If your hash does not match up, then there is an error in your code. Do make sure to run it multiple to make sure your code is not outputing the correct solution only some of the time. Memory issues are able to make your code output the correct output on some runs but incorrect output on other runs.

3) Submission Information

Submission of your project will be handled via **Gradescope**.

1. Create the submission file using the provided `make-submission` script:

   ```
   prompt> ./make-submission
   ```

2. This will create a `.zip` file named `$USER-submission` (e.g., for me, this would be named `lhenke-submission.zip`).
3. Submit this `.zip` file to Gradescope. You will get a confirmation email if you did this correctly.

> ***Warning***
>
> You are **REQUIRED** to use `make-submission` to form the `.zip` file. Failure to do so may cause your program to not compile on Gradescope.

4) General Requirements

- You are **REQUIRED** to use Isengard to develop and test this project.
- You should handle errors gracefully. All system calls can fail: if this occurs print a relavent and descriptive error to `stderr` (*not* `stdout`) and exit. Your program should have a non-zero exit status if any errors are encountered. (Make sure to add 'n's to those errors too!)
- Your program should have a zero exit status if no errors are encountered.
- Your project must be written in the C programming language, and execute on Isengard.
- You should follow Linux Kernel coding style, a common style guide for open-source C projects.
- Your project must not execute external programs or use network resources.
- Your project should be memory safe. For example, if your program is susceptible to buffer-overflow based on certain inputs, it is not memory safe. As a corollary to this, you should not use any of the following functions: `strcat`, `strcpy`, or `sprintf`.
- You should `free` any memory that you heap-allocate, and `close` (or `closedir`) any files that you open.
- To compile your code, the grader should be able to `cd` into the root directory of your repository and run `make` using the provided `Makefile`.

5) Resources

You will be using some or all of the following pthread library calls:

- `pthread_create`
- `pthread_join`
- `pthread_mutex_init`

- `pthread_barrier_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_barrier_wait`

Please refer to http://lemuria.cis.vtc.edu/~pchapin/TutorialPthread/pthread-Tutorial.pdf and https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html for tutorials on how to use pthreads.

6) Reference Executables

Provided for you is a reference executable file called `pzip_instructor`. It is a working version of the project that scored 100% in the autograder. You may use it to help understand the behavior of a working project as well as double check any of your outputs. It is included in this template repository and can be run with `./pzip_instructor`. Things to keep in mind about the reference executable:

- It was developed on Isengard and is only guaranteed to work on Isengard.
- This solution is not the ideal solution. If you are running it on the input_huge, out of three runs, it will have a WALL TIME < 1.5 and a PE > .75 which is expected of your submission as well.
- It is an instructor version and you may not execute it from within your own code. You will recieve a zero if you do!

7) Collaboration Policy

This is an **individual project**. All code you submit should be written by yourself. You should not share your code with others.

Please see the syllabus for the full collaboration policy.

> **WARNING: Plagarism will be punished harshly!**

8) Access to Isengard

Remote access to Isengard is quite similar to ALAMODE, but the hostname is `isengard.mines.edu`.

For example, to `ssh` into the machine with your campus MultiPass login, use this command:

```
$ ssh username@isengard.mines.edu
```

A tutorial has been linked in the discussion board to `ssh` via Visual Studio Code.

Note: you need to be on the campus network or VPN for this to work. If you are working from home, use either the VPN or hop thru `jumpbox.mines.edu` first.