Assignment 3

1 Synopsis

Please read the document, ***General Requirements Programming Assignments***, posted on the course website and make sure you adhere to those requirements when you do this assignment. In this assignment you'll write your second MPI program, to solve the ***string matching problem***.

This version of the string matching problem is to find ***all*** occurrences of a particular string, which we will call the ***pattern***, in a second, usually much longer, string which we will call the ***text***. Your task is to design and write a parallel MPI program that will solve this problem efficiently and with speedup nearly proportional to p, the number of tasks. In this context "efficiently", means that the efficiency of the program is high, i.e., the total *work* done by the parallel program is not much more than the work done by a sequential program that solves the same problem.

2 Program Invocation, Input, and Output

The program expects two command line arguments: the string to be matched, henceforth called the ***pattern***, and the name of a text file that contains the string to be searched. If the program name is search, then correct usage is

```
search <pattern>  <file>
```

If the pattern contains characters special to the shell, or blanks, it should be enclosed in single or double quotes, depending on which characters it contains. For example, to search for the pattern "Lincoln is on a $5 bill" in the file `denominations` you would enter

```
search ’Lincoln is on a $5 bill’ denominations
```

because `$` is a shell special character that retains its meaning inside a double-quoted string, or alternatively

```
search ‘‘Lincoln is on a \$5 bill’’ denominations
```

whereas to search for the pattern "Lincoln's on a $5 bill" in `denominations`, you have to enter

```
search ‘‘Lincoln\’s on a \$5 bill’’ denominations
```

The text string can contain any ASCII characters except control characters. This includes newlines and all white space characters. The file has no size limit.

2.1 Valid Usage

With valid arguments, the program will output the positions of the starting character of each occurrence of the pattern in the text in ascending order, one per line, if there are any matches at all. If there are no matches, it should produce no output at all. Positions are zero-based, so if the pattern matches the beginning of the text, the first number should be 0. For example, if `file` contains the text

```
cccagtagtcccgggcccc
```

then the call

```
search ccc file
```

should output

```
0
9
15
16
```

and absolutely nothing else. Notice that the pattern overlaps itself in the text and that's why both 15 and 16 are the start of it in the text. If the file contains

```
aaaaaaaaaa
```

and the pattern is just "`a`", then it outputs `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` one number per line. This shows that the number of starting points can be equal to the length of the text in the worst case. This has major ramifications in terms of how the program is designed.

2.2 Error Handling

If one or more arguments are missing, if the file can not be opened for reading, or if the string contains invalid characters, the program should print an error message on standard output that includes how to use it correctly and it should exit. If during processing the program results in an error such as being unable to allocate memory or other programmatic failures, it should print a message on standard error that it failed and it should clean up all MPI processes and exit.

3 Program Implementation Requirements

1. Only the root process can perform input, output, processing the command line, and error handling. It is an error if any other process does so.
2. The program must produce correct results regardless of the size of the pattern or the input file. There is no upper bound on the lengths of either of these, up to the limits defined by the Unix implementation. Look at the file, */usr/include/limits.h*.
3. The program must not use a library function such as `strstr()` to check for the pattern in the text. A process needs to check whether the pattern is in the text using any one of the various string search algorithms that exist, such as *brute-force*, *Knuth-Morris-Pratt, Boyer-Moore,* or *Rabin-Karp*.
4. The program must work correctly regardless of how many tasks are run.
5. The program must be documented and written to comply with the requirements stated in the ***General Requirements Programming Assignments*** referred to above.

4 Program Design Considerations

The program will be assessed on how well it is designed. The root process must get the command-line arguments, process them, and distribute data to the other tasks. Each task should work on a portion of the text string. When each process has finished its searching, it must send the matching positions, if any, to the root task. The root task must print the positions in ascending order on the standard output stream. You must decide how to do this efficiently. Passing redundant data increases running time and overhead and reduces efficiency.

4.1 Getting File Size

There are two ways to get the size of a file without reading the file.

4.1.1 Method 1: Using `lstat()`

If you need to determine the file size without opening the file, use the `lstat()` system call (only in POSIX-compliant systems):

```
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>

int main( int argc, char* argv[])
{
    struct stat statbuf;

    /* Check usage */
    if ( 2 > argc ) {
        printf("Usage: %s filename\n", argv[0]);
        exit(1);
    }

    /* Call stat() to fill statbuf struct */)
    if ( lstat(argv[1], &statbuf) == -1 ) {
        printf("Could not stat file %s\n", argv[1]);
        exit(1);
    }
    /* Print size of file. intmax_t is a type that holds big numbers */
    printf(" %8jd\n", (intmax_t)statbuf.st_size);
    return 0;
}
```

4.1.2 Method 2: Using `lseek()`

You can also use `lseek()` to get the file size, but this requires opening the file using the `open()` function defined in `<fcntl.h>`:

```
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>

int main( int argc, char* argv[])
{
    int       in_fd;
    intmax_t filesize;

    /* Check usage */
    if ( 2 > argc ) {
        printf("Usage: %s filename\n", argv[0]);
        exit(1);
    }

    /* open file     */
    if ( (in_fd = open(argv[1], O_RDONLY)) == -1 ) {
        fprintf( stderr, "could not open %s for reading\n", argv[1]);
        exit(1);
    }

    filesize = lseek(in_fd, 0, SEEK_END);

    close(in_fd);

    /* Print size of file. intmax_t is a type that holds big numbers */
    printf(" %8jd\n", (intmax_t)filesize);
    return 0;
}
```

4.2 Returning the Matched Positions

Assume that a task has a piece of the input file that it searches for matches with the pattern and that it finds several matching positions. It needs to send these positions back to the root process. One problem is that the total number of matches is not known in advance and therefore the storage for it can only be allocated at runtime. The number of matching positions cannot exceed the number of character positions in its piece of the text of course. I suggest that it send back an array of integers containing the positions, of size equal to the actual number of integers in that array. When you think about how `MPI_Send` works, this will be clear.

5 Testing

The program should be tested with very large files and very large and very small patterns. Does it work with the example from above? If the number of processes is very large, does it still work? If the pattern matches overlap, does it work? If the pattern is larger than the file does it work? If the pattern does not match at all does it work? These are all tests you should run.

6 Instructions for Submitting the Assignment

1. Use the `submithwk_cs49365` program to submit your program. It can be run on any `cslab` host, so login to a `cslab` host when you're ready to submit. To submit, if your file is named `myhwk.c`, you'd enter

   ```
   $ cd ~
   $ submithwk_cs49365  -t 3  myhwk.c
   ```

   The program will try to copy `myhwk.c` into the directory

   ```
   /data/biocs/b/student.accounts/cs493.65/hwks/hwk3/
   ```

   and if it is successful, it will display the message, "`File hwk3_`*`username`*`.c successfully submitted.`" where *`username`* is your username. You will not be able to read this file, nor will anyone else except for me. But you can double-check that the command succeeded by entering the command

   ```
   ls -l /data/biocs/b/student.accounts/cs493.65/hwks/hwk3
   ```

   and making sure you see a nonempty file named `hwk3_`*`username`*`.c` where *`username`* is your user name and whose date of last modification is the time at which you ran the command.

2. ***You can do step 1 as many times as you want. Newer versions of the file will overwrite older ones.***

7 Deadline

You have one week to do this assignment. It must be submitted before its ***deadline***, which is ***Monday, March 25, at 7:00 PM. After that, you will not receive credit for completing it.***

8 Program Grading Rubric

This assignment is **20%** of your final grade. The output must be correct and the program must conform to the rules written in the ***Programming Rules*** document on the class's webpage. It must be thoroughly documented.

The program will be graded based on the following rubric out of 100 points:

- The program must compile and run on across any set of `cslab` hosts that are available at the time. If it does not compile and link on any `cslab` host, it loses 80 points.
- **Correctness** (60 points)
  - The program should do exactly what is described above. Incorrect output, incorrectly formatted output, missing output, or output containing other characters are all errors.
  - It must process patterns and text files of arbitrary size.
  - The program should produce the same output no matter how many processes it is given, except for the elapsed time.
  - It should handle errors correctly.
- **Performance** (10 points)

  When run with successively greater numbers of processes, the elapsed time should decrease, except that if the number gets too large (larger than the available processors generally), it will start to increase again. Check that this behavior occurs on average. On any one run, it may not be exactly like this, but over many runs it should behave like this.
- **Design and clarity** (10 points)

  Is the program organized clearly? Are functions and variables designed in an appropriate way? Are the tasks divided appropriately? If not, the program will lose points.
- **Compliance with the Programming Rules** (20 points)

  Are all of the rules stated in that document observed? Programs that violate them will lose points accordingly.