Assignment 3 Revised

This is a revision of the third assignment. It has been changed to make it easier to implement.

Please read the document, ***Rules Regarding Programming Assignments for Everyone***, posted on the course website and make sure you follow it when you do this assignment. This assignment has two parts. The first part is your first MPI program, which is not difficult as a parallel algorithm but has technical challenges. The second part is a statement about the performance of the program.

1 Details

Part 1

You are to write an MPI program to implement the command described here. The program, when given two command line arguments, the first of which is a string called the ***key***, and the second of which is the path name of a text file, searches for all occurrences of the key in the text file. For each ***position*** in which the key occurs, ***it prints that position as a character offset from the beginning of the file*** on the standard output stream. ***The positions should be printed in ascending order***. For example, if the name of the program is `find_matches`, then the command

```
find_matches  “parallel algorithm” ~/cs493.65/lecture_notes
```

should print every place in the file `~/cs493.65/lecture_notes` at which the string “parallel algorithm” begins and if that occurs at positions 3, 745, and 930, then the three numbers should appear in the order: 3 then 745 then 930.

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

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

or alternatively

```
find_matches “Lincoln is on a \$5 bill” denominations
```

whereas to search for the key “Lincoln’s on a $5 bill” in denominations, you have to enter

```
find_matches “Lincoln\’s on a \$5 bill” denominations
```

A key can not contain embedded newline characters.

Error Handling

If one or more arguments are missing or if the file can not be opened for reading, the program should print an error message on standard error 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.

2 Program Design and Implementation Requirements

1. Only the root process can perform input and output and error reporting. It is an error if any other process does so.
2. The input file should be read just once, and it should be distributed to the remaining processes. Your program can use the code provided in the demos directory if you choose, but you must document how you use it. In particular, to get the size of the file, the program must not read the entire file. See how to do this with either `stat()` or `lseek()`.
3. The program must produce correct results regardless of the size of the key or the input file. There is no upper bound on the lengths of either of these, up to the limits of the physical hardware.
4. The program must not use a library function such as `strstr()` to check for the key in the text. A process needs to check whether the key is in the text must use any one of the various string search algorithms that exist, such as *brute-force*, *Knuth-Morris-Pratt*, *Boyer-Moore*, or *Rabin-Karp*.
5. The program must work correctly regardless of how many tasks are run.
6. The program must be documented and written to comply with the requirements stated in the ***General Requirements Programming Assignments*** referred to above.

3 Suggestions

3.1 Getting File Size

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

3.1.1 Method 1: Using `stat()`

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

```c
#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 ( stat(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;
}
```

3.1.2 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>`:

```c
#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;
}
```

3.2 Returning the Matched Positions

Assume that a task has a piece of the input file that it searches 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.

Part 2

Analysize the performance of your solution, i.e., the running time of this program. Assume that the amount of time to check whether one string is contained in another string is a constant times the sum of the lengths of the two strings, i.e., it is $c \cdot (m + n)$, where m and n are the lengths of the two strings, and c is some arbitrary constant. Assume that the communication parameters are L and B, for the latency and bandwidth respectively. (This way you do not have to type Greek letters.) Write the running time using big-Theta ($\Theta($ $)$) notation as a comment in the program source code. Justify the answer if you hope to receive full credit.

4 Grading Rubric

The program will be graded based on the following rubric out of 80 points:

- The program must compile and run on any `cslab` host. If it does not compile and link on any `cslab` host, it loses 60 points.
- **Correctness** (40 points)
  - The program should print exactly the byte offsets of the key, one per line, and nothing else.
  - The program should work correctly no matter how many processes it is given.
  - It should handle errors correctly (10 points).
- **Performance** (20 points)

  If this program is distributed among p processors, the speedup compared to a sequential implementation that checks the lines in sequence one by one, should be roughly proportional to p.

  Some lines may be very short and others very long. A good program will be load balanced so that every process does about the same amount of work. Programs that just assign a sequence of lines to each process will not be well balanced in general.
- **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.

The performance result is worth 20% of the grade. For full credit, it must be justified with an analysis.

5 Submitting the Homework

In these instructions, assume your program file is named `myprog.c`. ***The project must be submitted by October 27 at 19:00.***

1. You will use the `submithwk_cs49365` command to submit this assignment. **To submit your file, type the command**[1]

   ```
   $ /data/biocs/b/student.accounts/cs493.65/bin/submithwk_cs49365  -t 3  myprog.c
   ```

   The program will copy your `program` into the directory

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

   and if it is successful, it will name it `hwk3_username.c` and display the message, "`File hwk3_`*`username`*`.c successfully submitted.`" where `username` is your actual 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 typing the command

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

   and making sure you see a non-empty file named `hwk3_`*`username`*`.c`.

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

[1] If you have modified your PATH variable to include the directory `/data/bioc/b/student.accounts/cs493.65/bin`, then you can just type `submithwk_cs49365`.