Assignment 2
Breaking encryption with MPI
Due date: Tuesday, November 2 11:59 pm

Overview

In this assignment, you will write a program which has a serial section which encrypts a text string and a parallel component which uses MPI to decrypt the string.

You will create two programs:
- A serial program that will encrypt the plaintext - passed as a command line argument - using a simple substitution cipher and place it in a file called `ciphertext.txt`
- A parallel program that will use brute-force decryption and parallelize it using MPI. It will read the encrypted text from `ciphertext.txt` and display all valid decrypted results. You will use the system dictionary to check the results are correct - i.e. verify that even work in the decrypted string is a valid English word.

Encryption

The encryption algorithm has the following steps:
- store each unique letter from the string in an input dictionary
- create a second dictionary with the order of the letters mixed, called the encryption dictionary
- for each non-space character in the input string:
  - match the character to the input dictionary
  - find the corresponding character in the encryption dictionary
  - replace the input character with the corresponding character from the encryption dictionary
- Note that the program will be tested using only lower case characters and spaces.

Example:

input string: `the cat`

Take the letters in the input string and store them in an input dictionary. Store each letter only once. Do not store blank characters in the dictionaries. You can use any kind of data structure to build the dictionaries you wish, but they do not need to be complicated.

input dictionary: `theca`

Notice the spaces and repeated characters are removed. The create the encryption dictionary by rearranging the characters in the input dictionary. You can do this by copying the contents of the input dictionary and then randomly swapping the characters in the string. Pick a large enough number of swaps to ensure that the letters are thoroughly mixed.

encryption dictionary: `cehat`

Take the original string and use the two dictionaries to encrypted the string. For each input character, match it to the input dictionary and replace it with the dictionary character from the same location.

input dictionary: `theca`
encryption dictionary: `cehat`

The letter `t` in the original string is replaced with `c`, the letter `h` is replaced with `e`, and so on.

The input string "`the cat`" will become "`ceh atc`". Notice that blank characters pass through the algorithm without being changed.

The encryption algorithm is performed in serial and does not require any MPI code. It must ask the user to enter a string and should not require any other arguments. It must place the encrypted string into the file called `ciphertext.txt`.

Strings containing a lot of different letters will be too slow to decrypt, so when testing your code, don't use long input strings containing many different letters.

Decryption

The decryption algorithm performs the encryption operation in reverse. Since the decryption does not have access to the encryption dictionary, it will rely on brute force. You will speed up decryption by using parallel computations.

It will read the encrypted string from `ciphertext.txt` and the corresponding list of letters that occur in the string. Then create a decryption list and exhaustively reorder the decryption dictionary until you find correct text.

From the previous examples, the parameters passed to each MPI function would be the following two strings (only the part listed after the colon).

input string to the MPI functions: "`ceh atc`"
list of letters in the string: "`cehat`"

Each function will need to create a distinct decryption dictionary and exhaustively test it with the input string to see if it produces correct text. An easy way to divide the work is to have each process start the decryption dictionary with a unique letter. This means there will be one process for each unique letter in the input.

For example:

function rank 0
  decryption dictionary: `cehat`
function rank 1
  decryption dictionary: `echat`
function rank 2
  decryption dictionary: `hceat`
function rank 3
  decryption dictionary: `aehct`

function rank 4
  decryption dictionary: `tceha`

In the above example each function has a different first letter in the decryption dictionary. The first letter won't change during the execution of that function. The remaining letters will be reordered in an attempt to find the correct sequence to decrypt the string.

For example, function rank 0 starts with the dictionary "`cehat`". It will need to create all possible orderings of the string with the first letter always being `c`. It will need to test:

```
cehat
cehta
cetha
cetah
ceaht
ceath
...
```

In the above case there should be 24 different lists that need to be created and tested on processor rank 0. For processor rank 1 there would be 24 different strings starting with the letter `e`. For processor 2 there would 24 strings starting with `h`, and so on.

You can create the different orderings for the dictionary any way you wish. Two possible ways are to either use a recursive algorithm or to use a counter for each character and modulus arithmetic to loop through each possible value. Don't bother testing any dictionaries which contain duplicate characters.

When the program is tested there can be any number of characters in the string and dictionary.

To test the validity of a decryption dictionary:
- use the decryption dictionary and the original list of characters to decode the string
- test the words created against the system dictionary, if they are words that exist in the dictionary then you have found a solution

For example, the following information is passed to function rank 0.

  input string to the MPI functions: "`ceh atc`"
  list of letters in the string: "`cehat`"

The function searches all possible permutations of the letters and creates all possible decryption dictionaries for that function. One example would be:

  decryption dictionary: `ceaht`

Use the two lists of letters to convert the string:
  input dictionary: `cehat`
  decryption dictionary: `ceaht`

The string "ceh atc" becomes "cea htc". We check these "words" against the system dictionary, and they don't match anything. In this case the decryption dictionary "ceaht" is not correct.

If instead the decryption dictionary was "theca" (which would be generated by function rank 4) then the decoding would work like this:

input dictionary: cehat
decryption dictionary: theca

The string "ceh atc" becomes "the cat" which is correct.

Once you find a valid solution you can print the results. For example:

rank 4: the cat

Note that it is possible to have more than one valid result when you decrypt the string - it is possible that more that one dictionary will result in a string of valid English words. You must display all valid results.

You can pass a list of the letters in the string to each process if you wish.

Note that you do not have access to the encoding dictionary in the decryption program. Keep in mind that we might use our own ciphertext.txt when grading your code. As a result, this assignment requires that you write an exhaustive search for the solution. This iproblem s computationally expensive - and nicely data-parallel - so it is a natural candidate for speedup through parallel computations.

Assume there will be one processor assigned for each unique letter in the input string. The program would be started with five processes (-n 5) for the string "the cat".

You can use the system dictionary in /usr/share/dict/words to check the decryption results. You can use system commands (such as grep) and regular expressions to search the file for decrypted strings. You will have difficulty finding a single result using grep if you don't use regular expressions to identify the start and end of a line.

Documentation

Include a short report called A2Report.pdf. It must contain timing information that demonstrates how long the program took to decrypt alphabets of three unique characters and longer. Indicate at what size does the alphabet become "too large" to decrypt using this method. For the purposes of A2, if the decryption code takes longer than a minute to decrypt a string, the alphabet is "too large".

In the interests of time and minimizing the server load, you do **not** need to repeat each run 10 times - you can do it once per alphabet length: so 1 run for strings of only 3 letters, 1 run - for 4 letters, etc..

Submission and Evaluation

Submit the assignment using Moodle. Submit only the source code and the makefile. Bundle the code in a zip file.

The assignments will be marked on the `linux.socs.uoguelph.ca` server. If you develop your code on a different platform then it is a good idea to put the include files in an #ifdef for that system so they will still compile on the server. Test your program on the SoCS Linux server before you submit it.

The TA will unpack your code and type "`make`". It must produce tow executable:

- `a2encrypt`, which implements the serial encryption.
- `a2decrypt`, which implements the parallelized decryption. Remember that you cannot assume that the code was encrypted using your code - we will test it using our own encrypted data

If the makefile is missing, the make command does not work, or the program does not execute then you will lose a substantial number of marks.

It is always a good idea to unpack and test the file you are submitting to be sure that what you submit actually compiles.