Message Descrambling

Dr. Christopher Marcotte — *Durham University*

Message descrambling

The fundamental principle of coded messages is the required use of external information in order to turn something superficially information-sparse into something information-dense. One of the most basic ciphers is called the Caesar cipher, which circularly shifts a letter in the original message in the alphabet by a fixed amount.

The following is a scrambled message:

> *Zpv!uijol!zpvs!qbjo!boe!zpvs!ifbsucsfbl!bsf!voqsfdfefoufe!jo!uif!ijtupsz!pg!uif!xpsme-!cvu!uifo!zpv!sfbe/*

And ***encoder.c*** was used to scramble it:

encoder.c

```c
#include <stdio.h>  // Import the printf function
#include <omp.h>    // Import the OpenMP library functions

// Start of program
int main(){

   // This is a "character array", consisting of 105 letters in byte form.
   char mymessage[] = "This is a placeholder of the original message";

   // The message is encoded by shifting all letters by one, i.e.
   // A becomes B, c becomes d and so forth. This shift is easy to implement
   // in byte form as all you need to do is to add 1 to a character.
   // To printf a single character, use e.g.
   //       printf("%c\n", mymessage[0]);
   // To printf the whole array do e.g.   (subtle difference between %c and %s)
   //       printf("%s\n", mymessage);

   /* MISSING CODE: for loop over the 105 characters and adding the value 1 to each entry; */

   printf("%s\n", mymessage); // This prints a character array to screen

   return 0; // Exit program
}
```

We will use this encoder to investigate parallel decryption of the secret message.

Exercise

Fix and finish the serial encoder. Include a way to determine the length of `char mymessage[]` programmatically.

Solution

First, `char mymessage[]` is not 105 characters long, so we *need* to find how long it is first. One way is to increment the number of characters in the string until we hit the terminator:

```c
int n;
for (n = 0; mymessage[n] != '\0'; ++n);
```

Alternatively, use `strlen()` in `#include <string.h>` or measure the memory allocation of the string and normalize by the memory for storing a single `char`.

Then we need to loop over the message, modifying each character:

```
for (int i=0;i<n;i++){
  mymessage[i] +=1;
  printf("Thread [%d] sees %c\n", omp_get_thread_num(), mymessage[i]);
}
```

Exercise

Write a serial decoder which undoes the action of the above program. What is the decoded message? *Read through the comments in the code*!

Solution

We need to loop over the encoded message, modifying each character:

```
for (int i=0;i<105;i++){
  mymessage[i] -=1;
  printf("Thread [%d] sees %c\n", omp_get_thread_num(), mymessage[i]);
}
```

Hint

You may find it easier to finish this exercise *after* completing the OpenMP Thread Test exercise.

Exercise

Parallelize the decoder using OpenMP. Use suitable pragmas and run the decryption with multiple threads. Insert a statement in the for-loop that prints the current thread number and decryption result as the decoder progresses. Run the code repeatedly and vary the number of threads (`OMP_NUM_THREADS`); What do you observe?

Solution

Obviously we need to include the OpenMP header, `#include <omp.h>`. We just need one `#pragma` to parallelise the decoding. Just before the for loop of the serial decoder:

```
#pragma omp parallel for
```

The correct descrambled message is:

You think your pain and your heartbreak are unprecedented in the history of the world, but then you read.

Exercise

Write a new encoder/decoder pair which only produces the correct message if the number of threads used by the sender (the encoder) is the same as the number used by the receiver (decoder).

Solution

The easiest way to do this is by having the offset in the loop be `omp_get_num_threads()`. Alternatively, you can try `omp_get_thread_num()` if you can guarantee that a particular thread will handle a particular index.

Exercise

You may notice that such a short text is not sufficient to really tax even a single thread, nevermind several OpenMP threads. Encode the plain text version of *Frankenstein; Or, The Modern Prometheus* by Mary Wollstonecraft Shelley. Time the encoding with several threads, and produce a plot showing the speedup from using multiple threads.

Warning

If you are using a Windows terminal without Unicode support, or a font lacking unicode support, you *may* need to normalize the encoding of the text file to ASCII, which is a subset of UTF-8, the nigh-universal encoding for text, now. The unfortunate thing about UTF-8 for us is that it is variable-width, 1 – 3 bytes per character, which will make our lives more difficult for no good pedagogical reason. You can create an ASCII-encoded version of the Frankenstein file with the command:

```
iconv -f UTF-8 -t ASCII Frankenstein.txt > ASCII-Frankenstein.txt
```

which may solve some subtle issues around encoding characters coherently.

Additionally, there is no portable method to determine the length of a text file in C – meaning to get an accurate character count, we would need `fstat` by invoking `#include <sys/stat.h>`. However, you may not *need* to do this to complete the exercise – a quick and (technically incorrect) approach will likely work fine, if not robustly, using `fseek` and `ftell`.

Solution

One might plausibly use a sequence of commands like*,

*Following the W3 Schools code here.

```
FILE *fptr;
// Open a file in read mode
fptr = fopen("filename.txt", "r");
// Store the content of the file
char myString[N];
fgets(myString, N, fptr);
```

to load chunks of size *N* from the file to work on. I suspect that one will not find much of a benefit of the approach simply because the disk access is going to be a limiting factor. Alternatively, one could try to fit the entire book into the `char` array – this might require either unlimiting the stack size with `ulimit -s unlimited`, or using a heap allocated buffer with `malloc`.

A complete implementation of the encoder can be found in ***`frank_encoder.c`***:

`frank_encoder.c`

```c
#include <stdio.h>  // Import the printf function
#include <omp.h>    // Import the OpenMP library functions
#include <stdlib.h>

int main(int argc, char *argv[]){
   if (argc != 4) {
     printf("Incorrect number of inputs\n");
     return 0;
   }

   int nthreads = atoi(argv[1]);
   omp_set_num_threads(nthreads);

   FILE *fp;
   fp = fopen(argv[2],"r");
   if (!fp) {
     printf("Failed to open the file specified\n");
     return 0;
   }

   fseek(fp,0,SEEK_END);
   int size = ftell(fp);
   fseek(fp,0,SEEK_SET);
   printf("%d\n",size);

   char msg[size];
   fread(msg, 1, size, fp);
   fclose(fp);

   double t0 = omp_get_wtime();
   #pragma omp parallel for default(none) shared(msg, size, nthreads)
   for (int i=0; i<size; i++) {
      msg[i] += omp_get_num_threads();
   }
   double t1 = omp_get_wtime();
   printf("%d: %2.16f s\n", nthreads, t1-t0);

   //printf("%s\n", msg);

   FILE *fw;
   fw = fopen(argv[3], "w");
```

```
    fwrite(msg, 1, size, fw);
    fclose(fw);
    return 0; // Exit program
}
```

I won't bother with the timing results – it's much slower for any number of threads greater than 1 on my machine.

Aims

- Review of some basic *C* programming patterns.
- Familiarity with simple usage of OpenMP library functions.
- A gentle introduction to simple loop parallelism in a novel setting.