Home assignment 1

Emanuel Paraschiv & Mohammad Asfour
Group 45

February 2025

Introduction

In this report we will present the implementation of the tasks solved for home assignment 1.

Matrix

Starting from the original matrixSum.c file that performs the sum of matrix's elements, the first feature we implemented was to find the minimum and maximum in the given matrix.

In order to do that, we followed the base principle, namely each thread/ worker will work on its assigned strip where it will find the 'local' minimum and maximum along with their positions (one index for the row and one for the column).

In order to ensure termination of all threads, Barrier() function is used. Once this step is completed, the main thread find the global maximum and minimum from all candidates (strips) along with their position.

For part b, we changed the logic for finding the min and max. Now, if a value appears bigger that what is stored the global max variable, for example, then this variable is locked in order to obtain mutual exclusion and the new value is written to it. Same mechanism is applied for minimum.

For part c where we had to implement a bag of tasks, the main idea is that each thread will pick a row from the 'bag' and process it. In order to process it, it first acquires a lock to change the count of rows in the bag and then unlocks it. Therefore in each row the thread finds the sum, min and max.

```
pthread_mutex_lock(&lockBag);
row = rowBag;
rowBag++;
pthread_mutex_unlock(&lockBag);
```

Quicksort

The starting point of developing parallel mutithreaded program in C was to first implement a standard sequential version of this algorithm.

It had a wrapper function *quicksort(array, length)* that would call the recursive algorithm which worked by splitting the array is smaller parts and swapping elements around a given pivot until sorted.

In order to make the execution depend on multiple thread, the quicksort() helper was transformed into a worker function that uses specific mechanisms.

This function takes as argument a pointer to a struct that contains the old parameters and splits the array around the pivot value.

```
typedef struct {
    int *arr;
    int min;
    int max;
} Args;
```

The main idea is that the program creates threads for each split, for the left and right sub-arrays. Each thread then calls the function recursively and thus divides it further:

```
pthread_create(&leftT, NULL, Worker, &leftArgs);
pthread_create(&rightT, NULL, Worker, &rightArgs);
```

In the case in which the number of threads created is bigger than the allowed number, then the program continues its execution sequentially on the same number of threads:

```
Args leftArgs = {arr, min, pivot - 1};
        Args rightArgs = {arr, pivot + 1, max};
        Worker(&leftArgs);
        Worker(&rightArgs);
```

Tee command

The purpose of this command is to take a file as parameter and write a given input to both the standard output and to the file.

In order to implement the tee command, 3 threads are used:

- One for Standard input
- Another one for Standard output
- And a third one for Writing to the given file

We assume that the input writes data to a shared buffer. While this process takes place, the buffer is locked, and once the end of input is finished, we announce it through a flag.

In order to avoid inconsistencies, the output function is blocked by a condition variable until the data is processed concurrently by the file writing thread and vice-versa. In order to write to output, the 'putchar' function is used:

```
for (int i = 0; i < buff_count; i++) {
        putchar(buffer[i]);
    }
```

The writing thread simply takes the characters from the buffer and continues to write until it reaches the end. When a thread finishes execution, it gets back to the main function to join, and when all threads are completed the file is closed.

Palindrome

For this task we are required to find palindromes and semordnilaps given a dictionary file.

In the main function, the dictionary is read and all words are stored in a 'words[]' array.

```
    while (fgets(lineBuff, MAXWORDLENGTH, fp_in) != NULL && word_count < MAXWORDS) {
      size_t wordLength = strlen(lineBuff);
      if (wordLength > 0 && lineBuff[wordLength - 1] == '\n') {
        lineBuff[wordLength - 1] = '\0';
      }
     //Alocate space for the word
      words[word_count] = (char *) malloc((strlen(lineBuff) + 1) * sizeof(char));
      //Move word in buffer to allocated space
      strcpy(words[word_count], lineBuff);
      word_count++;
}
fclose(fp_in);
```

In the beginning, 2 helper functions (return boolean) are created `isPalindrome` and `isSemor`, that ensure that a given word is either a palindrome (by reversing it and comparing to the original) or a semordnilap (by reversing it and checking its existence in the dictionary). A binary seach algorithm was also used (since the file was ordered) to find the reversed word in the file for Semordnilaps.

In the worker function `Palindrom`, the word array is divided between the number of threads, which is determined by an input in the command line (if

no number is specified, 10 worker threads will be created). In this worker function, arrays (`PalCount[], SemorCount[]`) are used to track how many palindromes and semordnilaps were found by each thread, and mutexes are used to ensure that shared data is updated safely:

```
pthread_mutex_lock(&lockPal);
Paltotal++;
pthread_mutex_unlock(&lockPal);
```

In the end, the results are written to the file `outputFile`.

The following table displays the runtime of the program for different number of threads:

nb of threads	**runtime (ms)**
1 (sequential)	5.1
2	3.2
4	2.3
6	2
8	1.7
10	1.7

Table 1: Runtime of the program as a function of number of threads used

As we would expect, the runtime decreases as the number of threads increase.