



**INDIAN INSTITUTE OF TECHNOLOGY DELHI**

**COP290  
DESIGN PRACTICES**

**REPORT**

**LAB-2**

**Team Name - Team\_Akatsuki**

**Rajan Gupta - 2018ME20710 Token- 30/30**

**Ayush Singh - 2020CS10331 Token-0/30**

**Vibhanshu Lodhi - 2020CS10409 Token- 0/30**

**Contributions:-**

**Rajan Gupta - All parts of the submission.**

**Ayush Singh - No contribution.**

**Vibhanshu Lodhi - No contribution.**

## Part 1- Storing/restoring contexts

### Task:

The main function has an array of numbers. Now we have to check if a given number is less than 40 and not a prime number and returns the square of that number. It should be done without using loops in code.

The idea here is to store a context (i.e, CPU registers) inside the conjecture method and whenever an assertion fails, restore the stored context and continue the process with next element of the array. Only the first value that satisfies all the assertions should get fully processed and the program ends.

### Code:

```
static ucontext_t ctx;

static void conjecture(int len, void* options, int sz, void fn(void*)); // Create context and start traversal

void assert(bool b); // Restore context if condition fails

static void conjecture(int len, void* options, int sz, void fn(void*)){
    // Create context and start traversal
    int i= -1;
    getcontext(&ctx);
    i++;
    if(i==len) exit(0);
    fn(options+ (i*sz));
}

void assert(bool b){ // Restore context if condition fails
    if(!b){
        // static ucontext_t current_ctx;
        // getcontext(&current_ctx);
        // swapcontext(&current_ctx, &ctx);
        setcontext( &ctx );
    }
}
```

**Approach of Testing:-** The approach of testing was based on checking all the possible cases i.e. several testcases.

### Testcase:-1 Empty Array

```
58
59 int main(void) {
60     int mynums[] = {};
61     // We have to ensure that conjecture lives in the bottom
62     // If the conjecture frame is popped, we will never be a
63     conjecture(0, (void*) mynums, sizeof(int), &app);
64 }
65
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignme
● 10331_2020CS10409/part1/src/output$ cd "/home/rajan/Desktop/COP290/
018ME20710_2020CS10331_2020CS10409/part1/src/output"
./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignme
● 10331_2020CS10409/part1/src/output$ ./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignme
○ 10331_2020CS10409/part1/src/output$
```

Program outputs nothings.

### Testcase:-2

```
59 int main(void) {
60     int mynums[] = {11, 23, 42, 39, 55};
61     // We have to ensure that conjecture lives in the bot
62     // If the conjecture frame is popped, we will never b
63     conjecture(5, (void*) mynums, sizeof(int), &app);
64 }
65
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assi
● 10331_2020CS10409/part1/src/output$ cd "/home/rajan/Desktop/COP
018ME20710_2020CS10331_2020CS10409/part1/src/output"
./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assi
● 10331_2020CS10409/part1/src/output$ ./"main"
1521
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assi
○ 10331_2020CS10409/part1/src/output$
```

Program Outputs  $39 \times 39 = 1521$  (As expected)

### Testcase:-3

```
59 int main(void) {
60     int mynums[] = {11, 23, 49, 85, 25};
61     // We have to ensure that conjecture lives in the bot
62     // If the conjecture frame is popped, we will never b
63     conjecture(5, (void*) mynums, sizeof(int), &app);
64 }
65
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assi
● 10331_2020CS10409/part1/src/output$ cd "/home/rajan/Desktop/COP
018ME20710_2020CS10331_2020CS10409/part1/src/output"
./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assi
● 10331_2020CS10409/part1/src/output$ ./"main"
625
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assi
○ 10331_2020CS10409/part1/src/output$
```

Program Outputs  $25 \times 25 = 625$  (As expected)

#### Testcase 4:- Array of size 1 lakh with all numbers greater than equal to 40

```
59 int main(void) {
60     int mynums[100000];
61     for(int i = 0; i < 100000; i++) {
62         mynums[i] = 40+i;
63     }
64     // We have to ensure that conjecture lives in the bottom
65     // If the conjecture frame is popped, we will never be a
66     conjecture(100000, (void*) mynums, sizeof(int), &app);
67 }
68
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignme
10331_2020CS10409/part1/src/output$ cd "/home/rajan/Desktop/COP290/
018ME20710_2020CS10331_2020CS10409/part1/src/output"
./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignme
10331_2020CS10409/part1/src/output$ ./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignme
10331_2020CS10409/part1/src/output$
```

Program Outputs nothing (As expected)

#### Testcase 5:- Array of size 1 lakh like testcase 4, but two fully processable values

```
58
59 int main(void) {
60     int mynums[100000];
61     for(int i = 0; i < 100000; i++) {
62         mynums[i] = 40+i;
63     }
64     mynums[50000] = 25;
65     mynums[100000-1] = 39;
66     // We have to ensure that conjecture lives in the bottom of th
67     // If the conjecture frame is popped, we will never be able to
68     conjecture(100000, (void*) mynums, sizeof(int), &app);
69 }
70
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignment2-CO
10331_2020CS10409/part1/src/output$ cd "/home/rajan/Desktop/COP290/COP_La
018ME20710_2020CS10331_2020CS10409/part1/src/output"
./"main"
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignment2-CO
10331_2020CS10409/part1/src/output$ ./"main"
625
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_Lab2/Assignment2-CO
10331_2020CS10409/part1/src/output$
```

Program processes only 25 and gives 625.



## Part 2- Concurrency

**Task:** Given N text files, you need to find the count of every word summed across all the files. This must be accomplished using “multithreading” and all the results should be stored in a hashmap.

### Checkpoint 1- hashmap

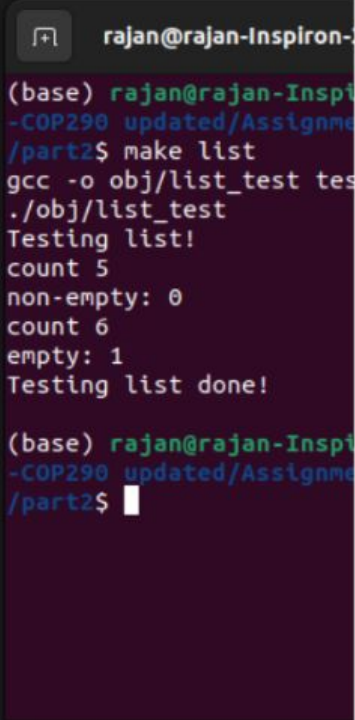
#### Approach:-

First a linked list has been coded in list.c file and then in order to implement hashmap, the approach called as Hashing with Chaining as been used, i.e. whenever two keys collide in same table slot, they are simply inserted in the linked list whose pointer is stored in the table slot. In order to hash the key, polynomial hashing using the ASCII value of each char in string has been implemented.

#### Outputs:-

##### list\_test.c output:-

```
4  #include <string.h>
5
6  int main(int argc, char** argv){
7      printf("Testing list!\n");
8      struct list* l = list_new();
9
10     int v = 5;
11     struct listentry* n = list_add(l, &v);
12     int* d = (int *) n->data;
13     printf("count %d\n", *d);
14     printf("non-empty: %d\n", is_empty(l));
15
16     int v1 = 6;
17     struct listentry* n1 = list_add(l, &v1);
18     int * d1 = (int*) n1->data;
19     printf("count %d\n", *d1);
20
21     list_rm(l, n);
22     list_rm(l, n1);
23
24     printf("empty: %d\n", is_empty(l));
25     printf("Testing list done!\n\n");
26     return 0;
27 }
```



The screenshot shows the terminal output of the list\_test.c program. The output is as follows:

```
(base) rajan@rajan-Inspiron-7560:~/COP290 updated/Assignments/Part2$ make list
gcc -o obj/list_test test.c
./obj/list_test
Testing list!
count 5
non-empty: 0
count 6
empty: 1
Testing list done!

(base) rajan@rajan-Inspiron-7560:~/COP290 updated/Assignments/Part2$
```

The output is as expected.

When one node was in, is\_empty() returned 0

When both nodes were removed, is\_empty() returned 1

### Hashmap\_test.c output:-

part2 > test > C hashmap\_test.c > ...

```
10
17 int main(int argc, char** argv) {
18     hashmap_create(&hashmap);
19     printf("Testing hashmap!\n");
20     printf("Iterate empty hashmap!\n");
21     hashmap_iterator(&hashmap, printer);
22     printf("Done iterate empty hashmap!\n");
23
24     char* key1 = "hello\0";
25     int* c1 = (int*) malloc(sizeof(int));
26     *c1 = 23;
27     hashmap_put(&hashmap, key1, c1);
28     hashmap_iterator(&hashmap, printer);
29
30     char* key = "world\0";
31     int* c = (int*) malloc(sizeof(int));
32     *c = 24;
33     hashmap_put(&hashmap, key, c);
34
35     char* key2 = "hello\0";
36     int* c2 = (int*) malloc(sizeof(int));
37     *c2 = 2;
38     hashmap_put(&hashmap, key2, c2);
39     hashmap_iterator(&hashmap, printer);
40     printf("Testing hashmap done!\n\n");
41
42     return 0;
43 }
```

rajan@rajan-Inspiron-3576: ~/Desktop

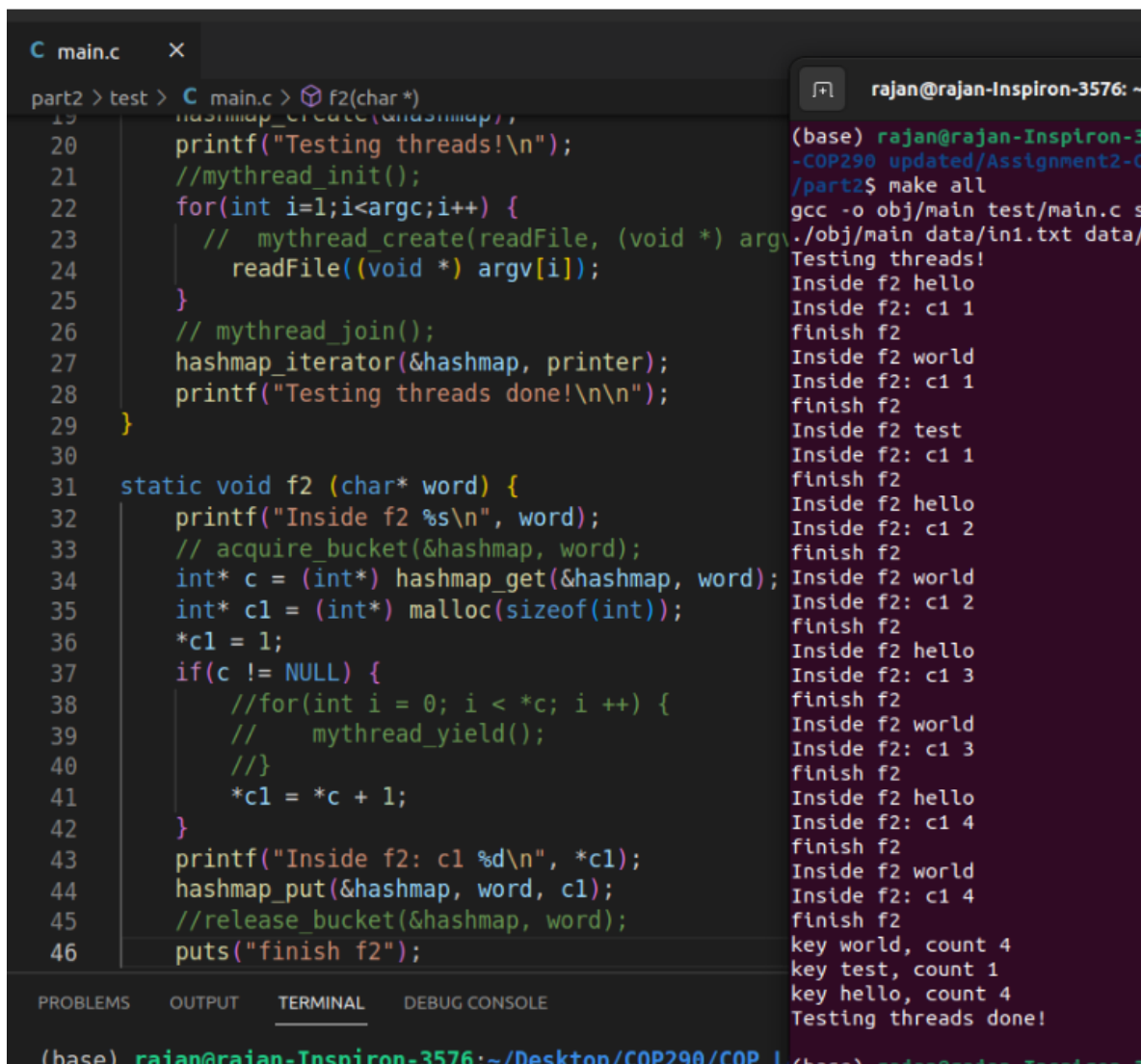
```
non-empty: 0
count 6
empty: 1
Testing list done!
```

```
(base) rajan@rajan-Inspiron-3576:~/D
-COP290 updated/Assignment2-COP290/2
/part2$ make hashmap
gcc -o obj/hashmap_test test/hashmap
thread.c
./obj/hashmap_test -I include/
Testing hashmap!
Iterate empty hashmap!
Done iterate empty hashmap!
key hello, count 23
key world, count 24
key hello, count 2
Testing hashmap done!
```

```
(base) rajan@rajan-Inspiron-3576:~/D
-COP290 updated/Assignment2-COP290/2
/part2$
```

The output is as expected.

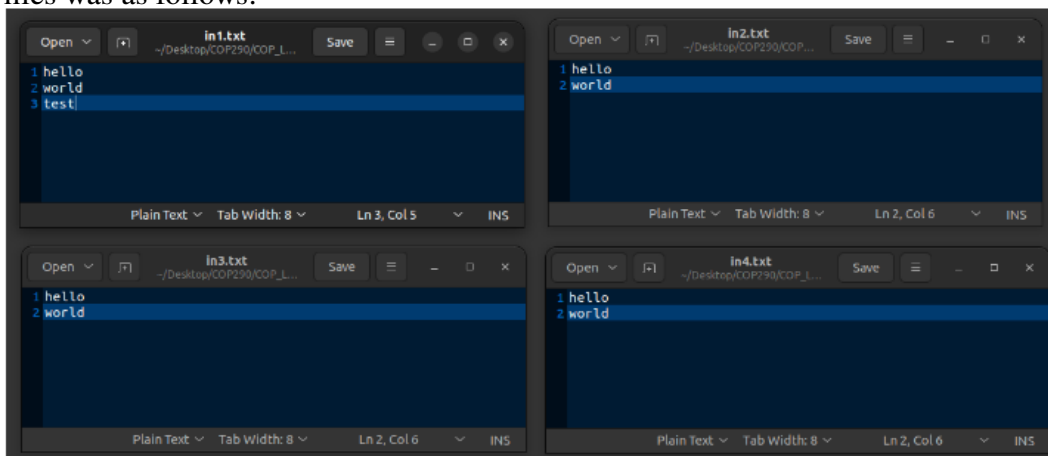
Now, all the thread part inside the main.c was commented, i.e. the main code runs on each text file (i.e. read 1 file completely at a time and go on next), to test the correctness of hashmap.



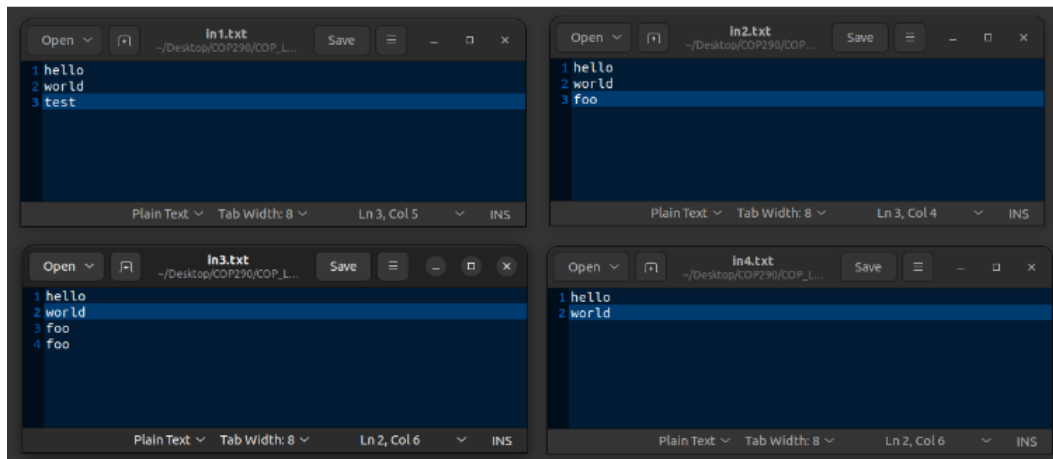
```
part2 > test > C main.c > f2(char *)
19  hashmap_create(&hashmap);
20  printf("Testing threads!\n");
21  //mythread_init();
22  for(int i=1;i<argc;i++) {
23      // mythread_create(readFile, (void *) argv[i], 0);
24      readFile((void *) argv[i]);
25  }
26  // mythread_join();
27  hashmap_iterator(&hashmap, printer);
28  printf("Testing threads done!\n\n");
29  }
30
31  static void f2 (char* word) {
32      printf("Inside f2 %s\n", word);
33      // acquire_bucket(&hashmap, word);
34      int* c = (int*) hashmap_get(&hashmap, word);
35      int* c1 = (int*) malloc(sizeof(int));
36      *c1 = 1;
37      if(c != NULL) {
38          //for(int i = 0; i < *c; i++) {
39              // mythread_yield();
40          //}
41          *c1 = *c + 1;
42      }
43      printf("Inside f2: c1 %d\n", *c1);
44      hashmap_put(&hashmap, word, c1);
45      //release_bucket(&hashmap, word);
46      puts("finish f2");
47  }
```

```
(base) rajan@rajan-Inspiron-3576: ~/Desktop/COP290/COP1
(part2) rajan@rajan-Inspiron-3576: ~/Desktop/COP290/COP1
-COP290 updated/Assignment2-COP1
/part2$ make all
gcc -o obj/main test/main.c s
./obj/main data/in1.txt data/
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 world
Inside f2: c1 1
finish f2
Inside f2 test
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
Inside f2 hello
Inside f2: c1 3
finish f2
Inside f2 world
Inside f2: c1 3
finish f2
Inside f2 hello
Inside f2: c1 4
finish f2
Inside f2 world
Inside f2: c1 4
finish f2
key world, count 4
key test, count 1
key hello, count 4
Testing threads done!
```

Again the output is as expected, since file contexts were read completely one by one and the content of 4 files was as follows:



As testcase-2, the contents were modified as follows and the code was run again.



```

C main.c x
part2 > test > C main.c > f2(char *)
19  hashmap_create(&hashmap, ,
20  printf("Testing threads!\n");
21  //mythread_init();
22  for(int i=1;i<argc;i++) {
23      // mythread_create(readFile, (void *) argv
24      readFile((void *) argv[i]);
25  }
26  // mythread_join();
27  hashmap_iterator(&hashmap, printer);
28  printf("Testing threads done!\n\n");
29  }
30
31  static void f2 (char* word) {
32      printf("Inside f2 %s\n", word);
33      // acquire_bucket(&hashmap, word);
34      int* c = (int*) hashmap_get(&hashmap, word);
35      int* c1 = (int*) malloc(sizeof(int));
36      *c1 = 1;
37      if(c != NULL) {
38          //for(int i = 0; i < *c; i++) {
39          //    mythread_yield();
40          //}
41          *c1 = *c + 1;
42      }
43      printf("Inside f2: c1 %d\n", *c1);
44      hashmap_put(&hashmap, word, c1);
45      //release_bucket(&hashmap, word);
46      puts("finish f2");

/part2$ make all
gcc -o obj/main test/main.
./obj/main data/in1.txt da
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 world
Inside f2: c1 1
finish f2
Inside f2 test
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
Inside f2 foo
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2: c1 3
finish f2
Inside f2 world
Inside f2: c1 3
finish f2
Inside f2 foo
Inside f2: c1 2
finish f2
Inside f2 foo
Inside f2: c1 3
finish f2
Inside f2 hello
Inside f2: c1 4
finish f2
Inside f2 world
Inside f2: c1 4
finish f2
key world, count 4
key test, count 1
key hello, count 4
key foo, count 3
Testing threads done!
(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/COP_L
• 10331_2020CS10409/part1/src/output$ cd "/home/rajan/Des
018ME20710_2020CS10331_2020CS10409/part1/src/output"
./"main"

```

The output of the program is again as expected.



## Checkpoint2:- threads

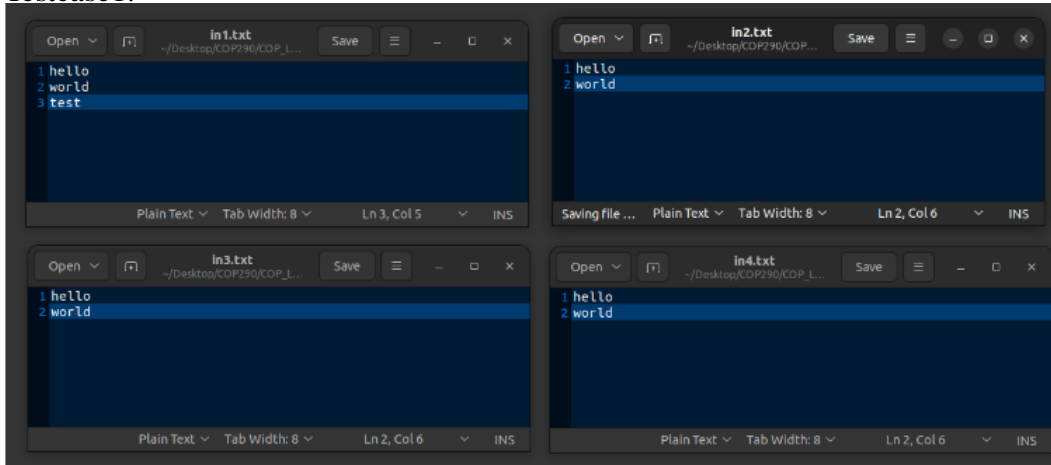
### Approach:-

In this part, implementation of functions like creating threads(stacks/contexts) and storing those contexts and in each context, a task like function call was stored, and on call to join the threads, start running each context made earlier one by one.

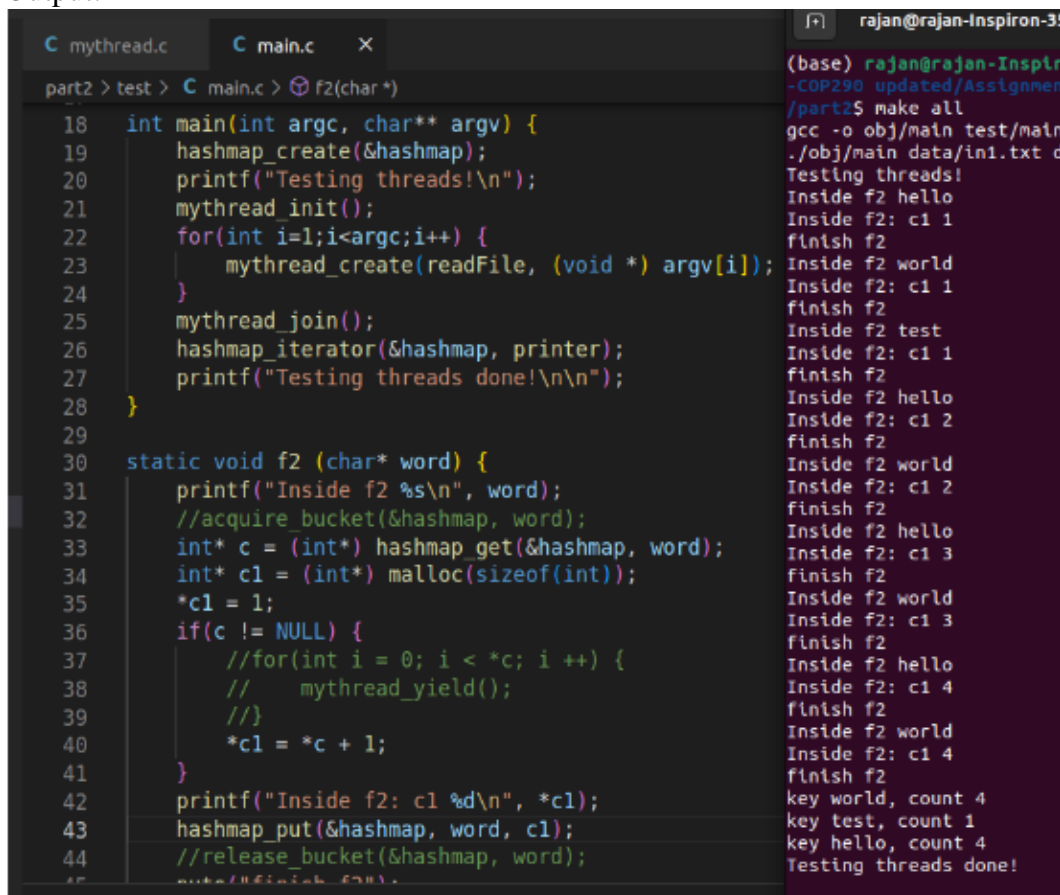
### Outputs:-

To test this part, only thread yielding and acquire\_bucket, release\_bucket part was commented from main code.

Testcase1:-

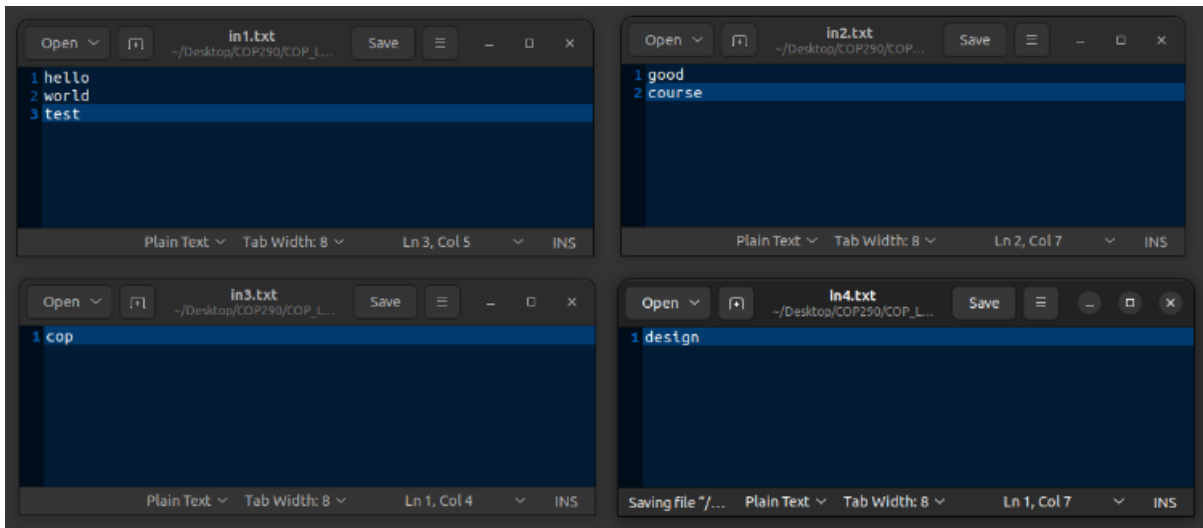


### Output:-



The program is as expected.

## Testcase2:-



## Output:-

```
> test > C main.c > f2(char *)
int main(int argc, char** argv) {
    hashmap_create(&hashmap);
    printf("Testing threads!\n");
    mythread_init();
    for(int i=1;i<argc;i++) {
        mythread_create(readFile, (void *) argv[i]);
    }
    mythread_join();
    hashmap_iterator(&hashmap, printer);
    printf("Testing threads done!\n\n");
}

static void f2 (char* word) {
    printf("Inside f2 %s\n", word);
    //acquire_bucket(&hashmap, word);
    int* c = (int*) hashmap_get(&hashmap, word);
    int* c1 = (int*) malloc(sizeof(int));
    *c1 = 1;
    if(c != NULL) {
        //for(int i = 0; i < *c; i++) {
        //    mythread_yield();
        //}
        *c1 = *c + 1;
    }
    printf("Inside f2: c1 %d\n", *c1);
    hashmap_put(&hashmap, word, c1);
    //release_bucket(&hashmap, word);
    printf("finish f2\n");
}

(base) rajan@rajan-Inspiron-3
-COP290 updated/Assignment2-C
/part2$ make all
gcc -o obj/main test/main.c s
./obj/main data/in1.txt data/
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 world
Inside f2: c1 1
finish f2
Inside f2 test
Inside f2: c1 1
finish f2
Inside f2 good
Inside f2: c1 1
finish f2
Inside f2 course
Inside f2: c1 1
finish f2
Inside f2 cop
Inside f2: c1 1
finish f2
Inside f2 design
Inside f2: c1 1
finish f2
key world, count 1
key design, count 1
key cop, count 1
key good, count 1
key test, count 1
key hello, count 1
key course, count 1
Testing threads done!
```

The output is again as expected.

### Checkpoint3:- locks

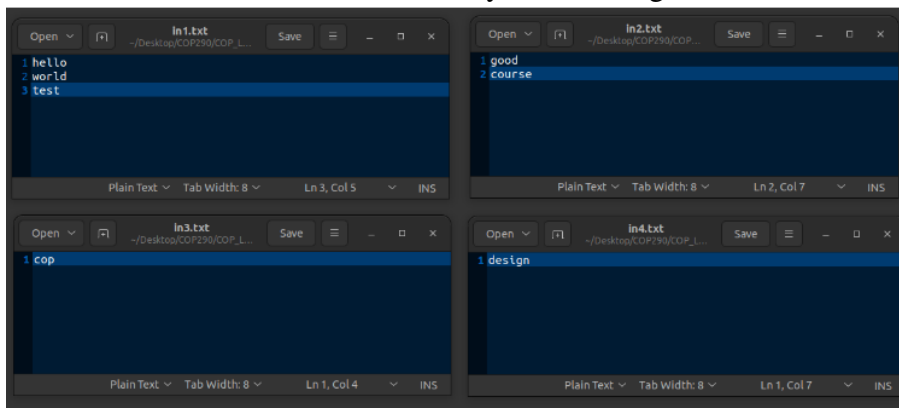
#### Approach:-

Implementation of yield function was also done in thread library. Whenever a context calls yield, it stores itself and go to next stored context and start running it. For e.g., while file1 reading was going on, and let program was somewhere in middle of file1, on call of yield, it will save itself, and go to context of file2. On other call of yield from the later context, program will come back to file1, and restore the context and start running from where it had left.

#### Outputs:-

In order to run it, only the acquire bucket and release bucket part was commented from the main code.

**Testcase 1:** Each word occurs only once among all files.



#### Output:-

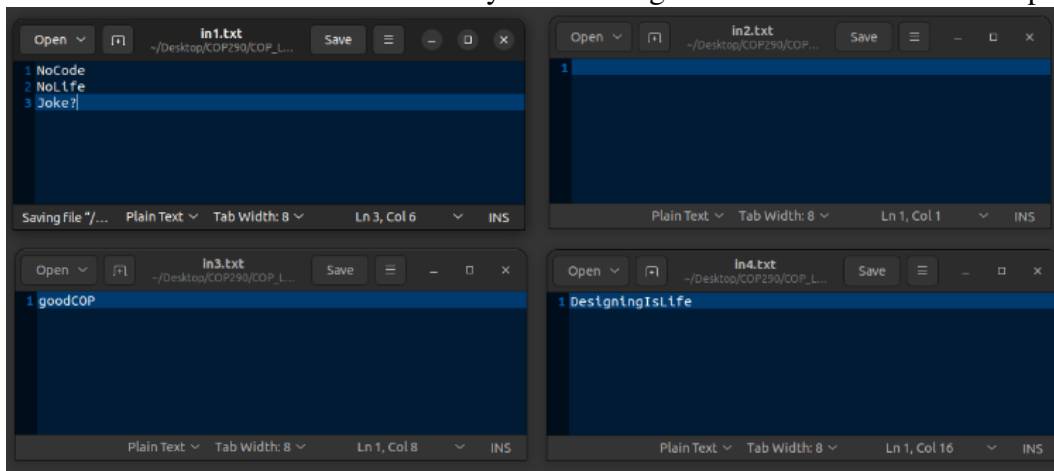
```
> test > C main.c > f2(char *)
int main(int argc, char** argv) {
    hashmap_create(&hashmap);
    printf("Testing threads!\n");
    mythread_init();
    for(int i=1;i<argc;i++) {
        mythread_create(readFile, (void *) argv[i]);
    }
    mythread_join();
    hashmap_iterator(&hashmap, printer);
    printf("Testing threads done!\n\n");
}

static void f2 (char* word) {
    printf("Inside f2 %s\n", word);
    //acquire_bucket(&hashmap, word);
    int* c = (int*) hashmap_get(&hashmap, word);
    int* c1 = (int*) malloc(sizeof(int));
    *c1 = 1;
    if(c != NULL) {
        for(int i = 0; i < *c; i++) {
            mythread_yield();
        }
        *c1 = *c + 1;
    }
    printf("Inside f2: c1 %d\n", *c1);
    hashmap_put(&hashmap, word, c1);
    //release_bucket(&hashmap, word);
    puts("finish f2");
}
```

```
(base) rajan@rajan-Inspiron-3570:~/Desktop/COP290/Assignment2$ make all
gcc -o obj/main test/main.c
./obj/main data/in1.txt data/in2.txt data/in3.txt data/in4.txt
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 world
Inside f2: c1 1
finish f2
Inside f2 test
Inside f2: c1 1
finish f2
Inside f2 good
Inside f2: c1 1
finish f2
Inside f2 course
Inside f2: c1 1
finish f2
Inside f2 cop
Inside f2: c1 1
finish f2
Inside f2 design
Inside f2: c1 1
finish f2
key world, count 1
key design, count 1
key cop, count 1
key good, count 1
key test, count 1
key hello, count 1
key course, count 1
Testing threads done!
```

The output of program is as expected.

**Testcase 2:** Each word occurs only once among all files and blank files also present.



**Output:-**

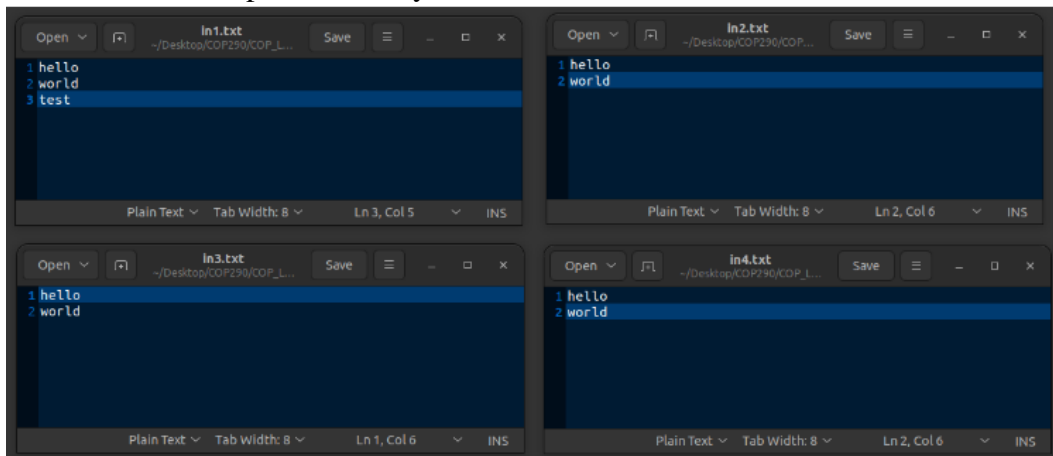
```
int main(int argc, char** argv) {
    hashmap_create(&hashmap);
    printf("Testing threads!\n");
    mythread_init();
    for(int i=1;i<argc;i++) {
        mythread_create(readFile, (void *) argv[i]);
    }
    mythread_join();
    hashmap_iterator(&hashmap, printer);
    printf("Testing threads done!\n\n");
}

static void f2 (char* word) {
    printf("Inside f2 %s\n", word);
    //acquire_bucket(&hashmap, word);
    int* c = (int*) hashmap_get(&hashmap, word);
    int* c1 = (int*) malloc(sizeof(int));
    *c1 = 1;
    if(c != NULL) {
        for(int i = 0; i < *c; i++) {
            mythread_yield();
        }
        *c1 = *c + 1;
    }
    printf("Inside f2: c1 %d\n", *c1);
    hashmap_put(&hashmap, word, c1);
    //release_bucket(&hashmap, word);
    puts("finish f2");
}
```

```
(base) rajan@rajan-Inspiron-3576: ~/Desktop/COP290 updated/Assignment2-COP290
/part2$ make all
gcc -o obj/main test/main.c src/libhashmap.c
./obj/main data/in1.txt data/in2.txt data/in3.txt data/in4.txt
Testing threads!
Inside f2 NoCode
Inside f2: c1 1
finish f2
Inside f2 NoLife
Inside f2: c1 1
finish f2
Inside f2 Joke?
Inside f2: c1 1
finish f2
Inside f2 goodCOP
Inside f2: c1 1
finish f2
Inside f2 DesigningIsLife
Inside f2: c1 1
finish f2
key Joke?, count 1
key DesigningIsLife, count 1
key NoCode, count 1
key goodCOP, count 1
key NoLife, count 1
Testing threads done!
```

The output of the program is as expected.

### Testcase 3: Repetition in keys



### Output:-

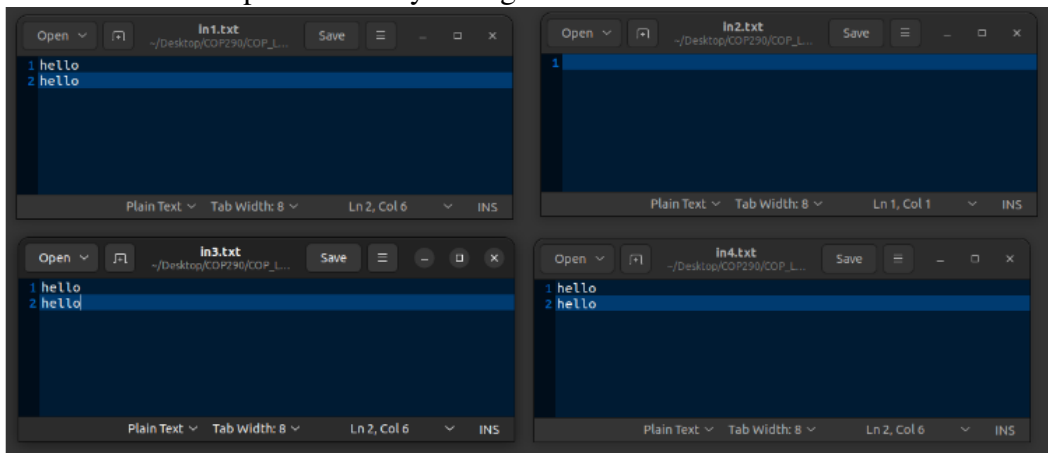
```
C mythread.c C main.c X
part2 > test > C main.c > f2(char *)
18 int main(int argc, char** argv) {
19     hashmap_create(&hashmap);
20     printf("Testing threads!\n");
21     mythread_init();
22     for(int i=1;i<argc;i++) {
23         mythread_create(readFile, (void *) argv[i]);
24     }
25     mythread_join();
26     hashmap_iterator(&hashmap, printer);
27     printf("Testing threads done!\n\n");
28 }
29
30 static void f2 (char* word) {
31     printf("Inside f2 %s\n", word);
32     //acquire_bucket(&hashmap, word);
33     int* c = (int*) hashmap_get(&hashmap, word);
34     int* c1 = (int*) malloc(sizeof(int));
35     *c1 = 1;
36     if(c != NULL) {
37         for(int i = 0; i < *c; i++) {
38             mythread_yield();
39         }
40         *c1 = *c + 1;
41     }
42     printf("Inside f2: c1 %d\n", *c1);
43     hashmap_put(&hashmap, word, c1);
44     //release_bucket(&hashmap, word);
45     puts("finish f2");
}

(base) rajan@rajan-Inspiron-3576:~/Desktop/COP290/Assignment2-COP290
-COP290 updated/Assignment2-COP290
/part2$ make all
gcc -o obj/main test/main.c src/libhashmap.c -I./obj/main data/in1.txt data/in2.txt
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 world
Inside f2: c1 1
finish f2
Inside f2 test
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2 hello
Inside f2 hello
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
key world, count 2
key test, count 1
key hello, count 2
Testing threads done!
```

The counts in output of the program is lesser than correct values!



#### Testcase 4: Repetition in keys along with blank files



Output:-

```
C mythread.c  C main.c  X
part2 > test > C main.c > f2(char *)
19  hashmap_create(&hashmap);
20  printf("Testing threads!\n");
21  mythread_init();
22  for(int i=1;i<argc;i++) {
23      mythread_create(readFile, (void *) argv[i]);
24  }
25  mythread_join();
26  hashmap_iterator(&hashmap, printer);
27  printf("Testing threads done!\n\n");
28  }
29
30  static void f2 (char* word) {
31      printf("Inside f2 %s\n", word);
32      //acquire_bucket(&hashmap, word);
33      int* c = (int*) hashmap_get(&hashmap, word);
34      int* c1 = (int*) malloc(sizeof(int));
35      *c1 = 1;
36      if(c != NULL) {
37          for(int i = 0; i < *c; i++) {
38              mythread_yield();
39          }
40          *c1 = *c + 1;
41      }
42      printf("Inside f2: c1 %d\n", *c1);
43      hashmap_put(&hashmap, word, c1);
44      //release_bucket(&hashmap, word);
45      puts("finish f2");
46  }
```

```
(base) rajan@rajan-Inspiron-3576:~$
-COP290 updated/Assignment2-
/part2$ make all
gcc -o obj/main test/main.c
./obj/main data/in1.txt data
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2: c1 2
finish f2
Inside f2 hello
Inside f2: c1 3
finish f2
Inside f2 hello
Inside f2: c1 3
finish f2
Inside f2 hello
Inside f2: c1 4
finish f2
key hello, count 4
Testing threads done!
(base) rajan@rajan-Inspiron-3576:~$
-COP290 updated/Assignment2-
/part2$
```

The count in program output is lesser than correct output!

### **Reasoning for testcase 3 and 4:-**

In Testcase 3 and Testcase 4, the output of the code has varied from what the correct counts should have been. It is because of the race condition. A race condition is an undesirable situation that occurs when a system attempts to perform two or more operations at same time, whereas because of nature of system, the operations should had been done in proper sequence to do the task correctly.

The reason behind it can be understood via following explanation of test case 4 as follows:

First, it reads the file1 and reads hello. Since previous count of hello was 0, it doesnt yield and puts (hello,1) in hashmap.

Now it reads hello from file 1 and yields to file 2, which was blank file and the context ends directly and we restore the file1 context.

Now we do put (hello,2) in hashmap and context of file 1 also ends.

Now we have two contexts of file 3 and file 4 left in our thread and we go two file 3 and read hello with count 2 and store this "2" in variable c and yields.

Now we go in context4, read hello, stores its count "2" in its c variable, and yields.

It goes to context3 and since yield was in for-loop of size 2, it yields again

It goes to context4 and since here also yield was in the for-loop of size 2, it yields again.

Now in context3, it put the hello with count "3" and read hello again and yields

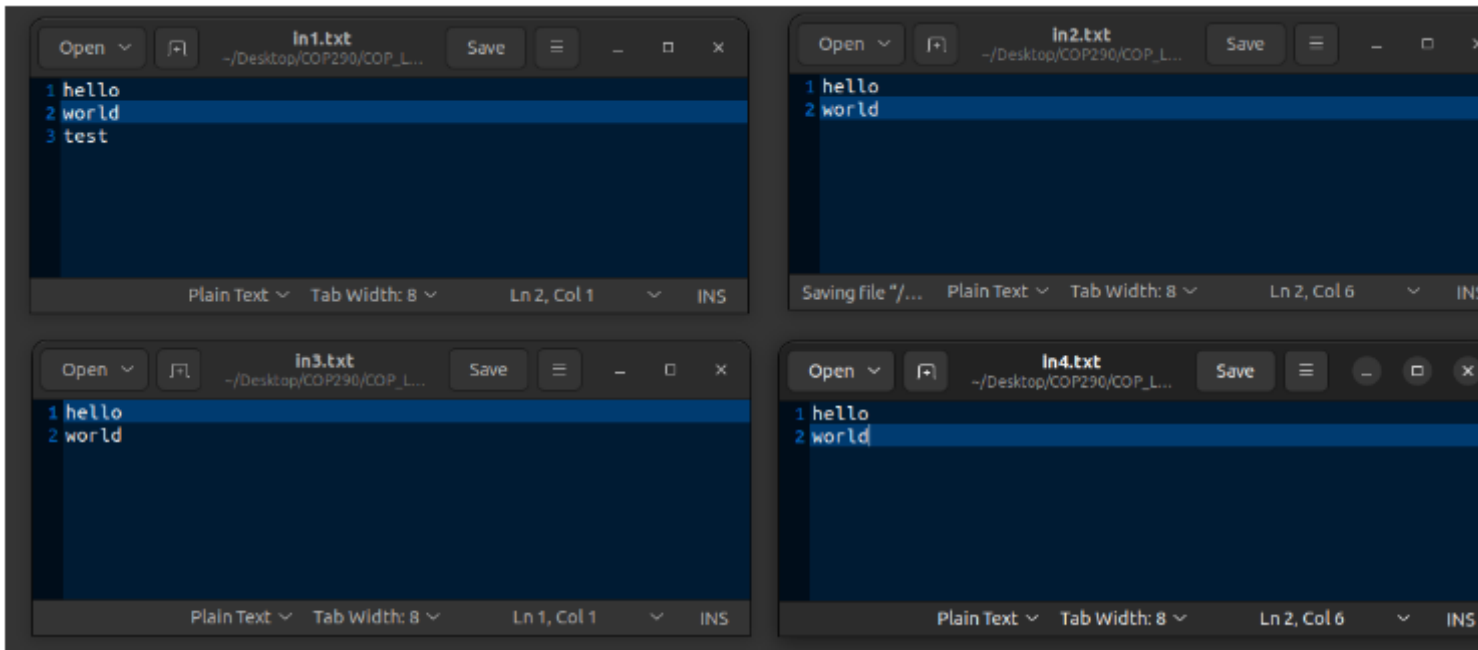
Now in context4, since c variable stored count as "2" , it puts (hello,2+1) in hashmap, whereas context3 already updated the count to 3 and it should have been (hello,4) by the context4 for correctness.

Hence due to the flaw in code occurring due to race conditions, the count we get is less than what is expected from a correct program.

In order to resolve the race conditions, locks are implemented and locks gives a guarantee that only 1 context will be able to enter the lock. For e.g. when context3 in above code has read hello and yielded, Now context 4 will read hello and won't be able to acquire the same lock and yield back to context3. Now context3 will first put it as 3 and free the lock, and then context4 will be able to acquire the same lock and will put (hello,3+1) which will be correct.

In this way, we won't be encountering the above race condition and our program will give correct outputs.

### Testcase 3:- Repetition in keys (With both yield and locks)



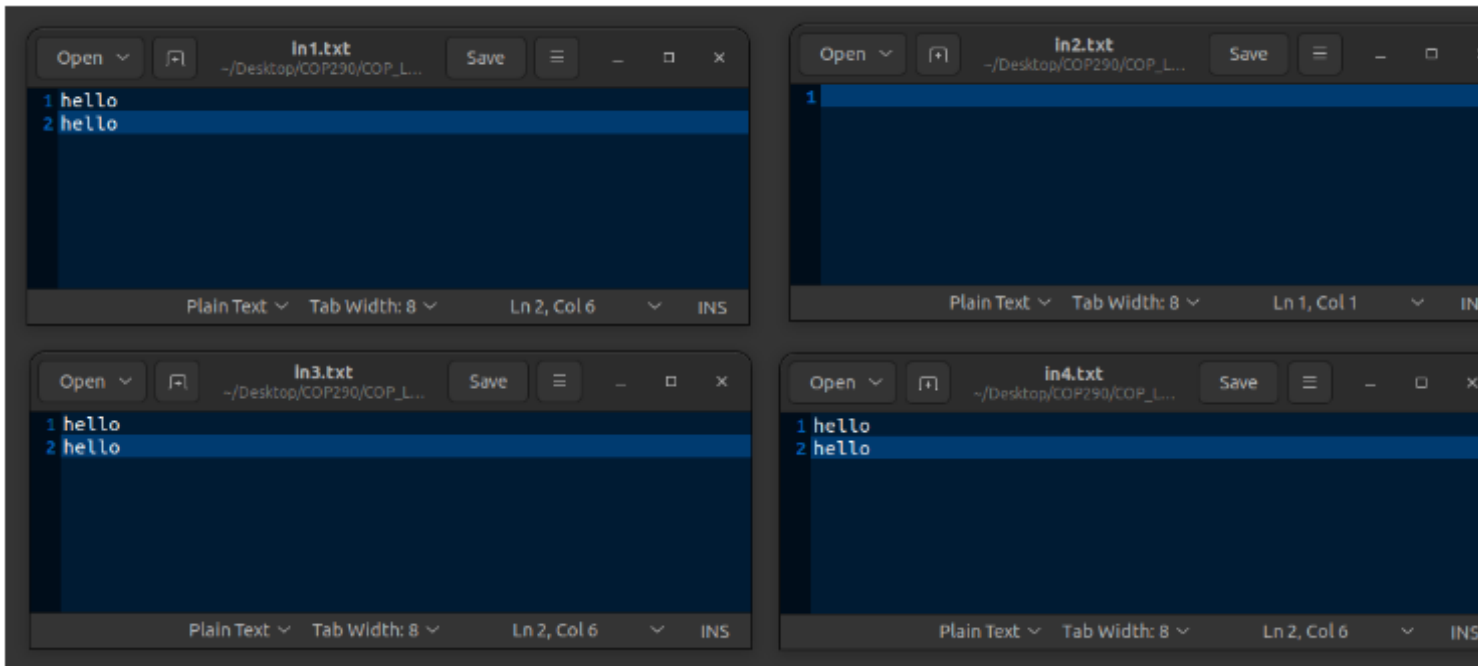
### Output:-

```
C mythread.c  C main.c  X
part2 > test > C main.c > f2(char *)
20     printf("Testing threads!\n");
21     mythread_init();
22     for(int i=1;i<argc;i++) {
23         mythread_create(readFile, (void *) argv[i]);
24     }
25     mythread_join();
26     hashmap_iterator(&hashmap, printer);
27     printf("Testing threads done!\n\n");
28 }
29
30 static void f2 (char* word) {
31     printf("Inside f2 %s\n", word);
32     acquire_bucket(&hashmap, word);
33     int* c = (int*) hashmap_get(&hashmap, word);
34     int* c1 = (int*) malloc(sizeof(int));
35     *c1 = 1;
36     if(c != NULL) {
37         for(int i = 0; i < *c; i++) {
38             mythread_yield();
39         }
40         *c1 = *c + 1;
41     }
42     printf("Inside f2: c1 %d\n", *c1);
43     hashmap_put(&hashmap, word, c1);
44     release_bucket(&hashmap, word);
45     puts("finish f2");
46 }
47

(base) rajan@rajan-Inspiro:~/COP290 updated/Assignment/part2$ make all
gcc -o obj/main test/main.o
./obj/main data/in1.txt data/in2.txt data/in3.txt data/in4.txt
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 world
Inside f2: c1 1
finish f2
Inside f2 test
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2 hello
Inside f2 hello
Inside f2: c1 2
finish f2
Inside f2 world
Inside f2: c1 2
finish f2
Inside f2: c1 3
finish f2
Inside f2 world
Inside f2: c1 3
finish f2
Inside f2: c1 4
finish f2
Inside f2 world
Inside f2: c1 4
finish f2
key world, count 4
key test, count 1
key hello, count 4
Testing threads done!
```

The output of the program is as expected.

#### Testcase 4: Repetition in keys along with blank files (With both yield and locks)



Output:-

```
mythread.c  main.c  X
part2 > test > C main.c > f2(char *)
20     printf("Testing threads!\n");
21     pthread_init();
22     for(int i=1;i<argc;i++) {
23         pthread_create(&readFile, (void *) argv[i]);
24     }
25     pthread_join();
26     hashmap_iterator(&hashmap, printer);
27     printf("Testing threads done!\n\n");
28 }
29
30 static void f2 (char* word) {
31     printf("Inside f2 %s\n", word);
32     acquire_bucket(&hashmap, word);
33     int* c = (int*) hashmap_get(&hashmap, word);
34     int* c1 = (int*) malloc(sizeof(int));
35     *c1 = 1;
36     if(c != NULL) {
37         for(int i = 0; i < *c; i++) {
38             pthread_yield();
39         }
40         *c1 = *c + 1;
41     }
42     printf("Inside f2: c1 %d\n", *c1);
43     hashmap_put(&hashmap, word, c1);
44     release_bucket(&hashmap, word);
45     puts("finish f2");
46 }
47
```

```
(base) rajan@rajan-Inspiron-3576:
-COP290 updated/Assignment2-
/part2$ make all
gcc -o obj/main test/main.c
./obj/main data/in1.txt data
Testing threads!
Inside f2 hello
Inside f2: c1 1
finish f2
Inside f2 hello
Inside f2: c1 2
finish f2
Inside f2 hello
Inside f2 hello
Inside f2: c1 3
finish f2
Inside f2 hello
Inside f2: c1 4
finish f2
Inside f2: c1 5
finish f2
Inside f2 hello
Inside f2: c1 6
finish f2
key hello, count 6
Testing threads done!

(base) rajan@rajan-Inspiron-
-COP290 updated/Assignment2-
/part2$
```

The output of the program is as expected.

## Part 3- pThreads

### Problem statement:

Now take the code from part 2 and place your files inside part 3 folder. Ensure that you are using mythread.c provided by us rather than your own mythread.c file.

Do a comparative analysis between the runtimes of part 2 and part 3. What happens when you change the number of input files? Size of input files? What if each file has the same word repeated many times? Generate a documentation for your code using doxygen.

Write a report using LaTeX where you would plot graphs (Time vs File Size) and also generate tables for the same.

### Aim:

The aim of the study has been to compare the performance of Concurrency i.e. running single thread at a time vs the performance offered by parallelism i.e. running more than 1 thread in parallel.

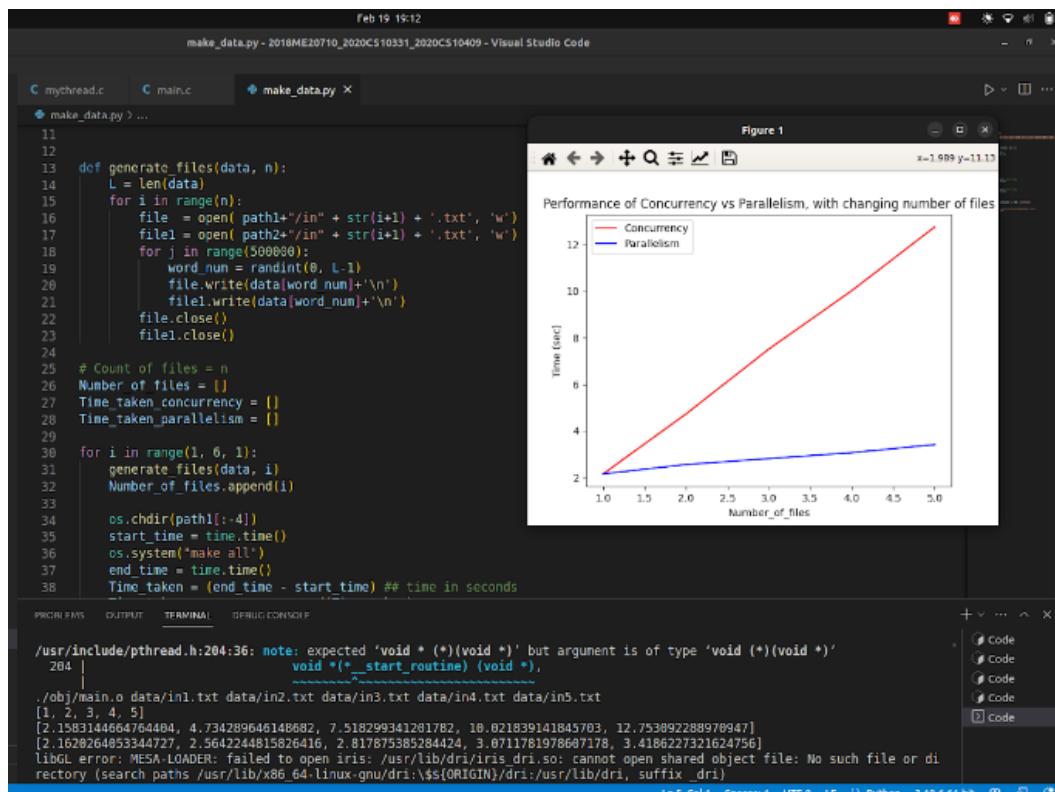
### Graph 1:

First we will see how both the threads performs when we start increasing the number of files.

All the print statements in the code of both the testing scripts have been commented.

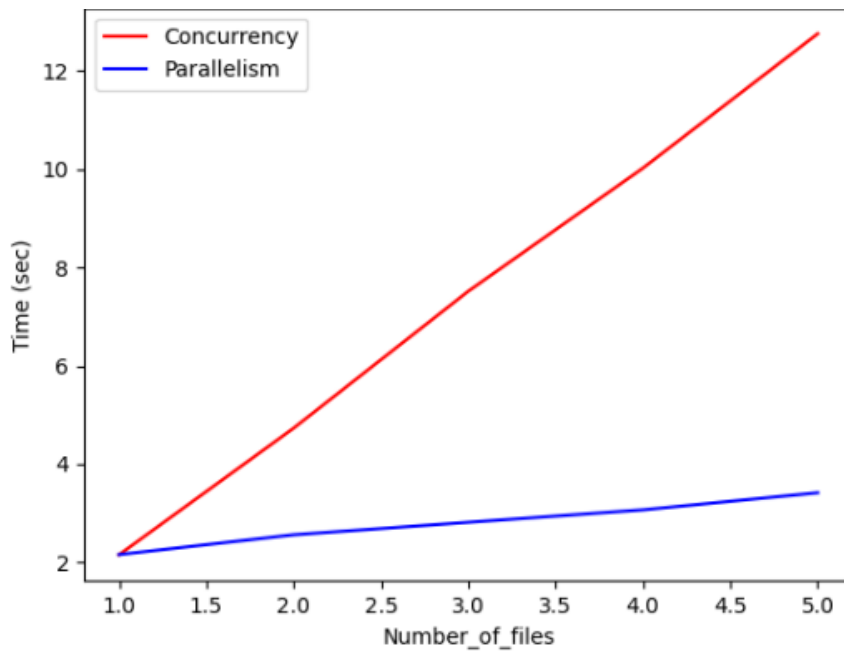
For this, we are reading a list of 250k unique words and generating files with each having 5 lakh randomly selected words of length between 8 to 12 characters making each file 5.5 Mb. Hence we can claim that within a file, there shouldn't be too much repetition.

First we compared with **yield as off**



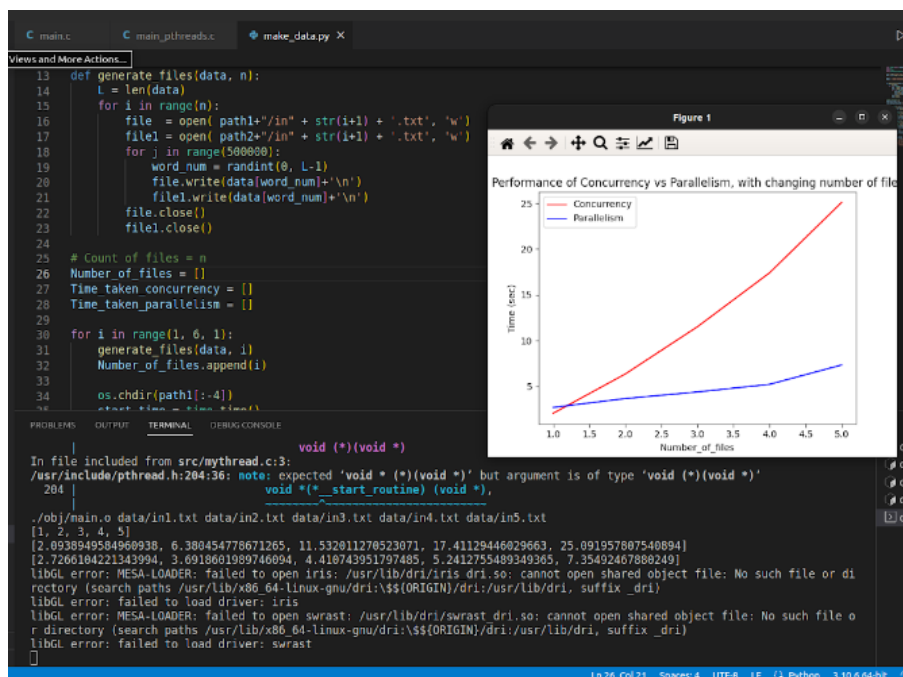
A python script was used to make `i` files when `i` range from 1 to 5 and time for make run (concurrency) and make test(parallelism) was recorded and plotted.

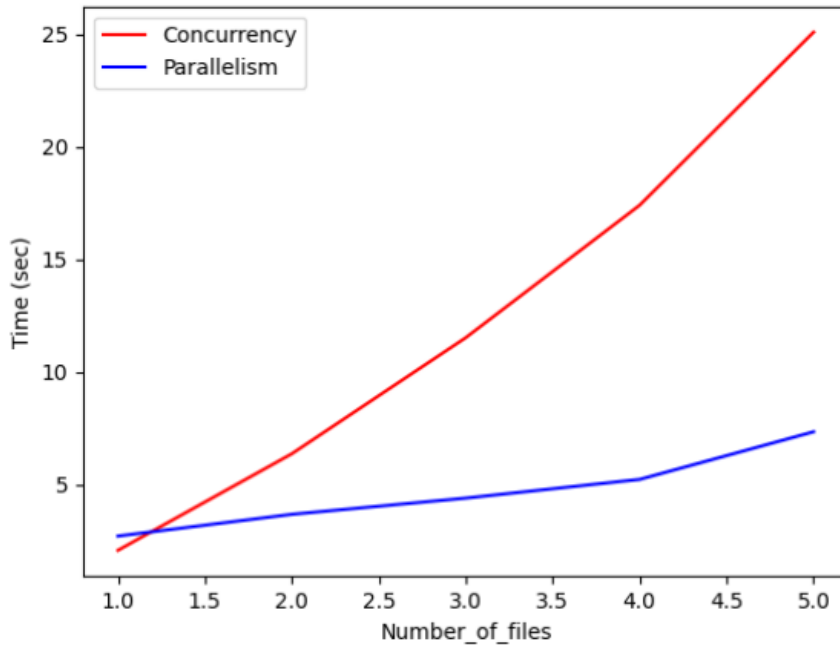




The trend is visible that the performance of parallelism is better than performance of concurrency, as the number of files increases.

Now plotting the results for when **yield** was on keeping other parameters same, we get the following results





Again, its visible that parallelism is solving the problem faster as compared to concurrency. Also as compared to yield off, program now taking more time, which was as expected.

The table for above graphs where x-axis column denotes **number of input files** and the time taken by the program in **seconds**.

X - Axis	Yield ON		Yield OFF	
	Concurrency	Parallelism	Concurrency	Parallelism
1	2.09389	2.72661	2.15831	2.16202
2	6.38045	3.69186	4.73428	2.56422
3	11.53201	4.41074	7.51829	2.81787
4	17.41129	5.24127	10.02183	3.07117
5	25.09195	7.35492	12.75309	3.41862

## Graph 2:

Now, in order to see the performance of concurrency vs parallelism with respect to file size, 250000 unique words were generated from the following python script to create input testfiles for experiments.

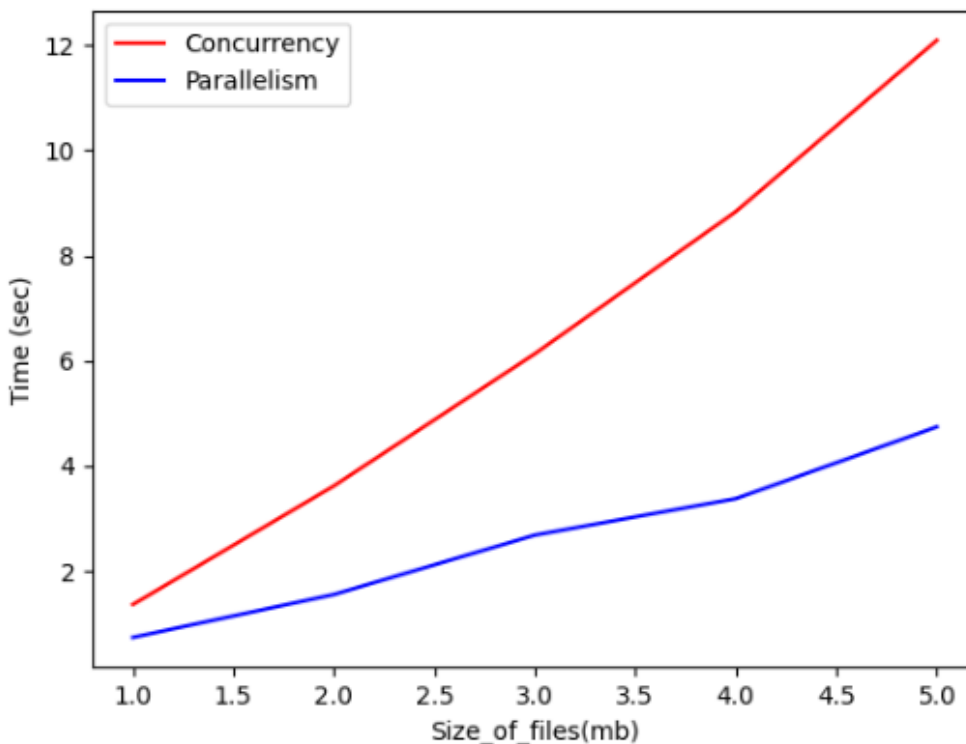
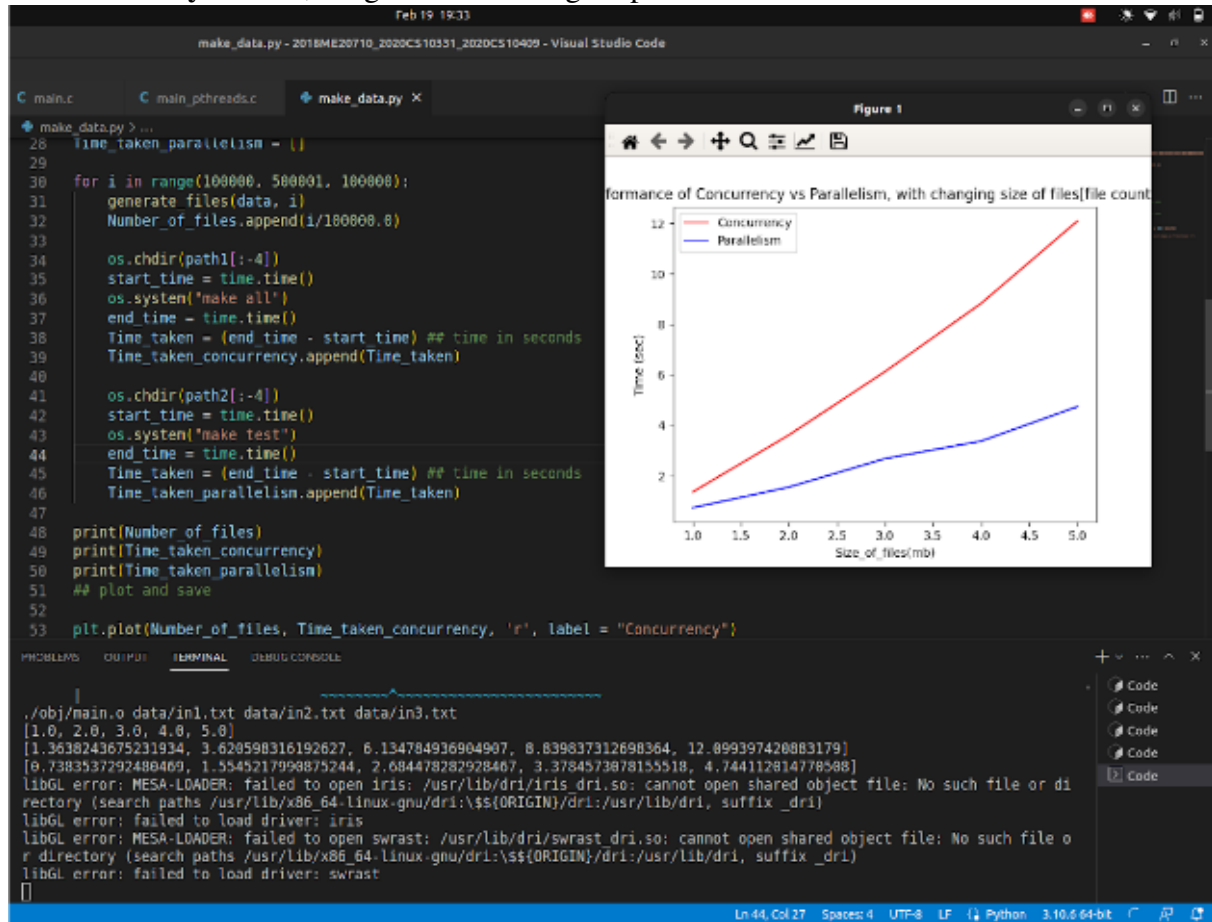
```

C mythread.c  C main.c  C main_pthreads.c  + make_data.py  + make_words.py x
make_words.py > ...
1  import random
2  import string
3
4  # Set the number of words to generate
5  num_words = 25000
6
7  # Generate a list of unique random words
8  words = set()
9  while len(words) < num_words:
10     word = ''.join(random.choice(string.ascii_lowercase) for i in range(random.randint(8, 12)))
11     words.add(word)
12
13 # Write the words to a text file
14 with open('unique_words.txt', 'w') as file:
15     for word in words:
16         file.write(word + '\n')

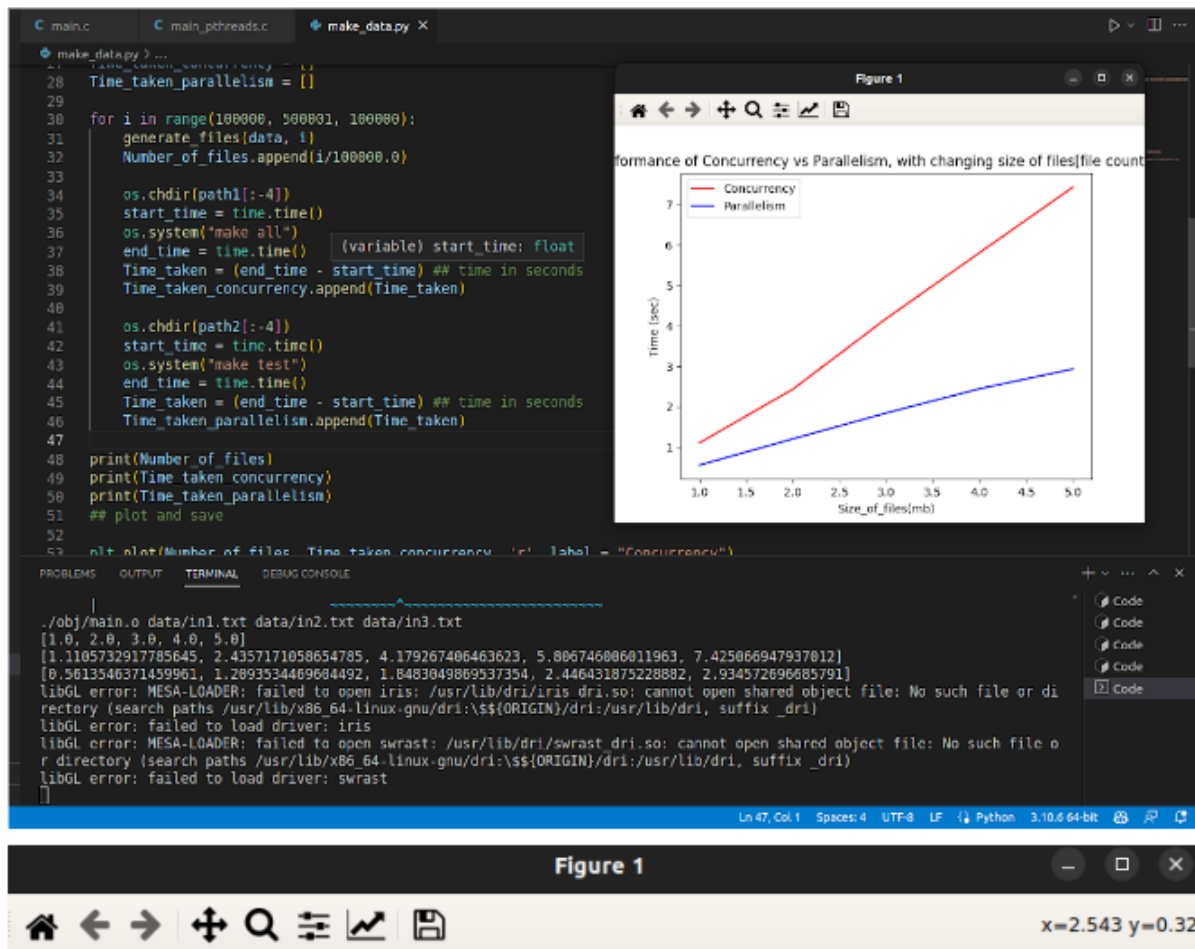
```

Then number of data files in data folder was fixed to be 3 and a loop was run on concurrency and parallelism that, number of words in each file is increasing in each iteration. In each experiment we kept same number of words in each file, ranging from 1 lakh to 5 lakh with step size 1 lakh words.

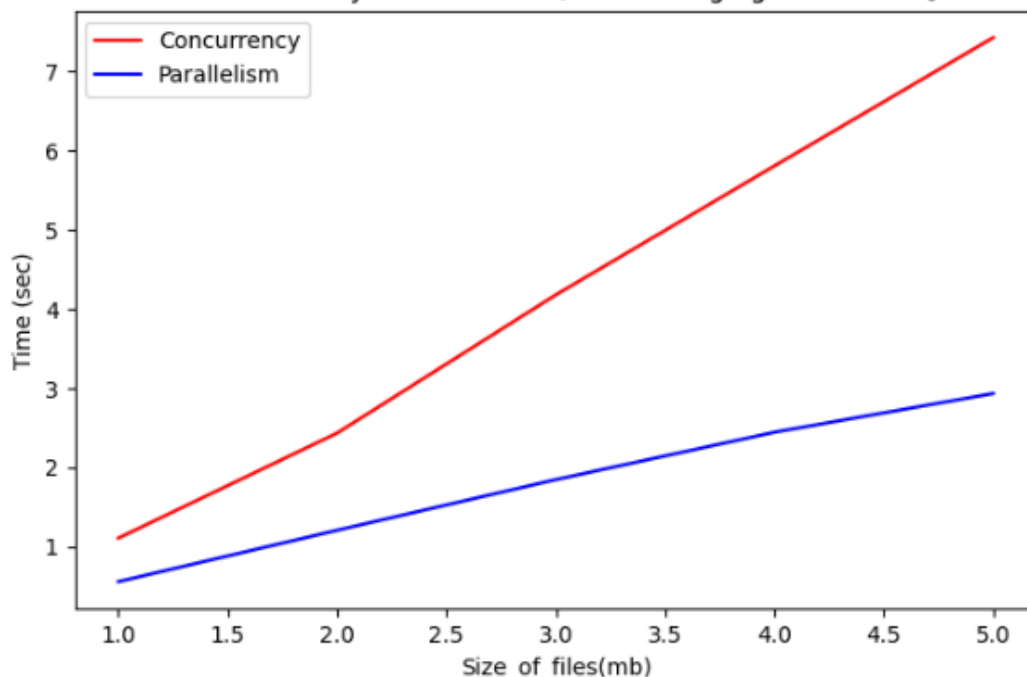
Now with **yield On**, we got the following outputs



Now with **yield as off** in both code keeping parameters same, we gets



Performance of Concurrency vs Parallelism, with changing size of files[file count = 3]



It is clearly noticeable that the time of yield off is less as compared to yield on the case, as well as concurrency took more time than parallelism.

The table for above graphs where x-axis column denotes **size of input files** and the time taken by the program in **seconds**.

X - Axis	Yield ON		Yield OFF	
	Concurrency	Parallelism	Concurrency	Parallelism
1	1.3638	0.7835	1.1105	0.56135
2	3.6205	1.5545	2.4357	1.20935
3	6.1347	2.6844	4.1792	1.8483
4	8.8398	3.3784	5.8067	2.4464
5	12.0993	4.7441	7.42506	2.9354

### Graph 3:

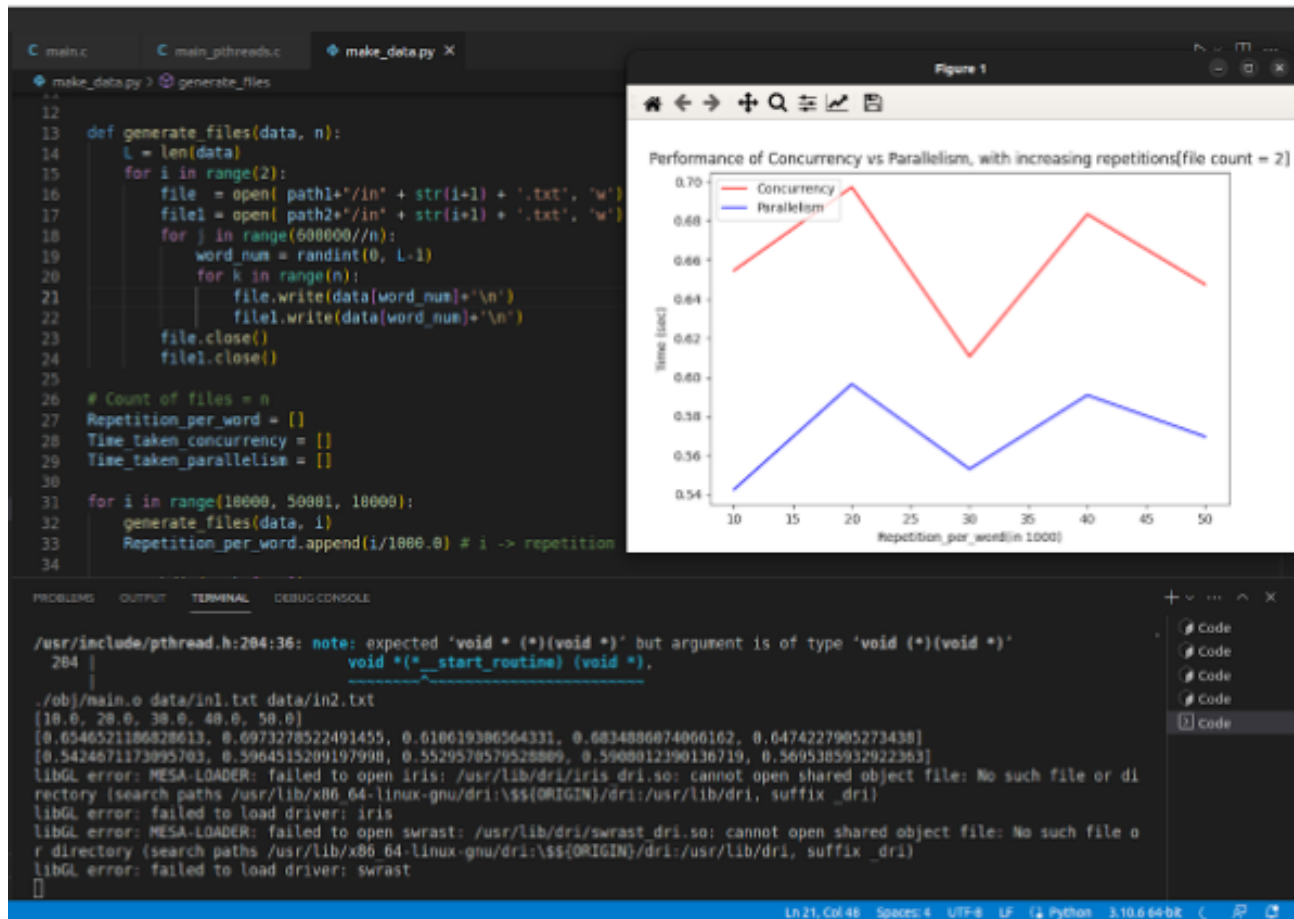
In order to see, how both performs when number of repetitions is significant in the file.

In data folder, only 2 file is taken with **total** 6 lakh words each.

Now, whenever a word was written to a file, it was repeatedly written x times. Experiment was done for x = 10k, 20k, 30k, 40k and 50k.

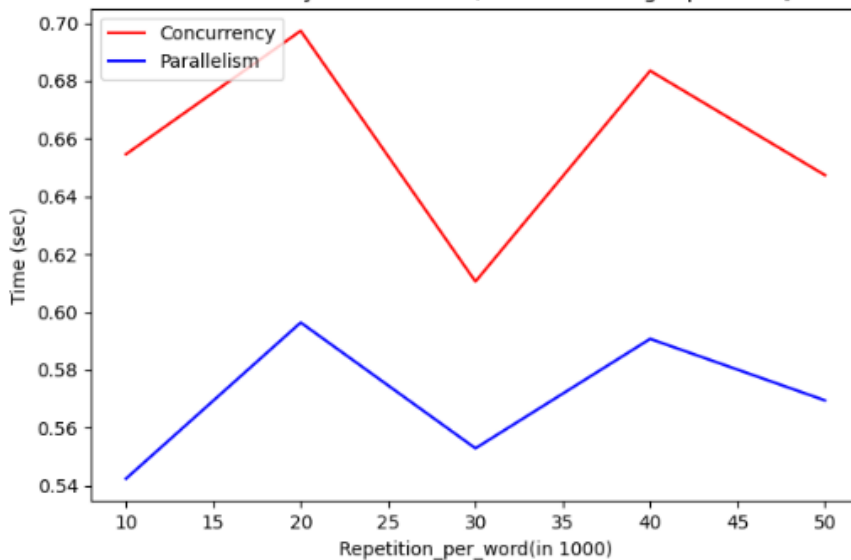
Again the word was selected from pool of 250k words generated in above cases.

Outputs with **yield off**:-



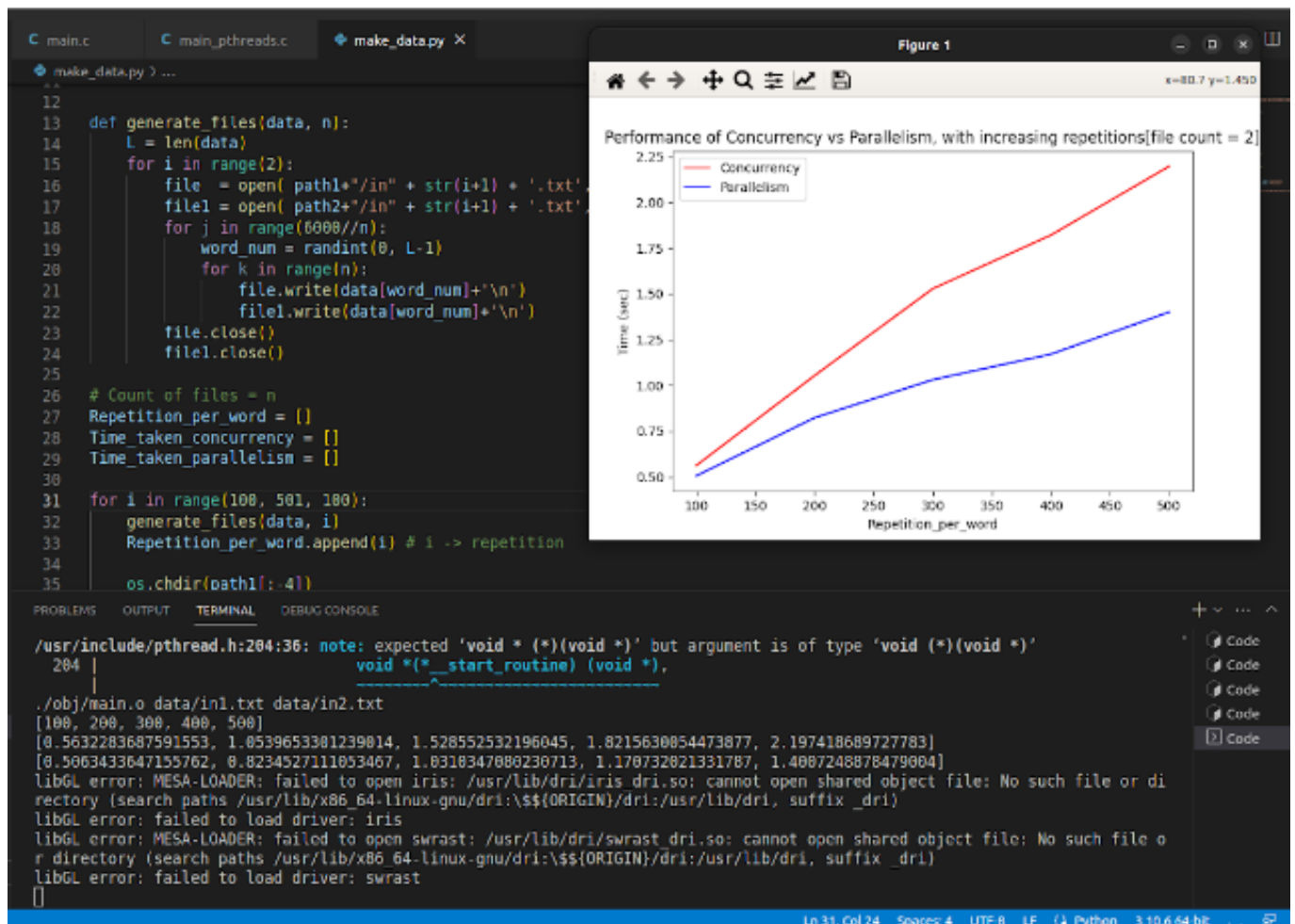


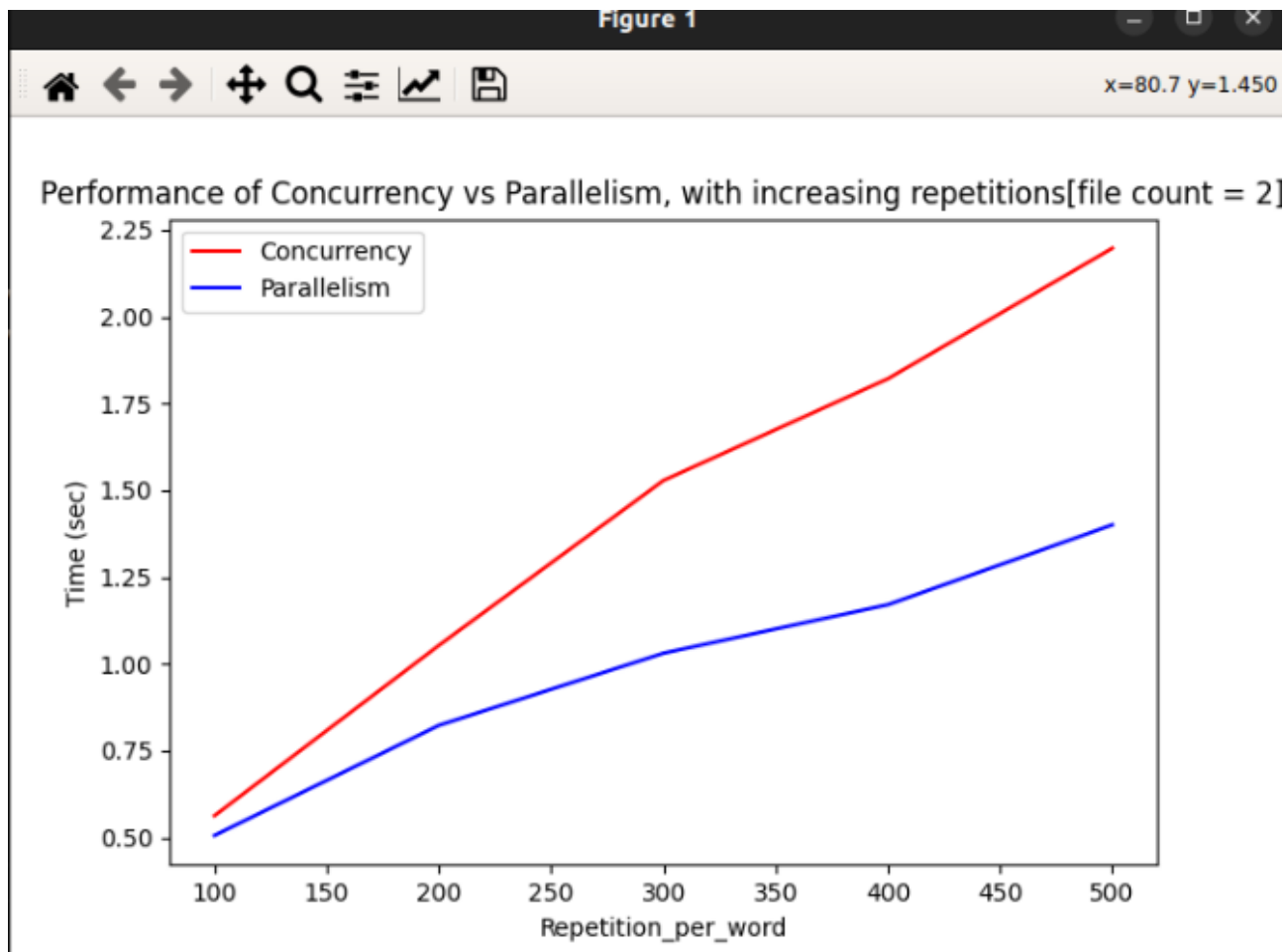
Performance of Concurrency vs Parallelism, with increasing repetitions[file count = 2]



### Outputs with yield on:-

Since, yielding was going too long to run with the above parameters, 6000 words each file with x = 100,200,300,400,500 was chosen as **new parameters** for this part





The table for above graphs where x-axis column denotes **number of repetitions** and the time taken by the program in **seconds** for **different parameters** as specified above.

X - Axis	Yield ON		Yield OFF	
	Concurrency	Parallelism	Concurrency	Parallelism
10	0.5632	0.5063	0.6546	0.5426
20	1.0539	0.8234	0.6973	0.5964
30	1.5285	1.031	0.6106	0.5529
40	1.8215	1.1707	0.6834	0.5908
50	2.1974	1.4007	0.6474	0.5695

## **My Key Learnings:-**

I learnt a lot of things from lab-2 including-

C

contexts

stack memory

heap memory

git

concurrency

race condition

locking

latex

doxygen etc.