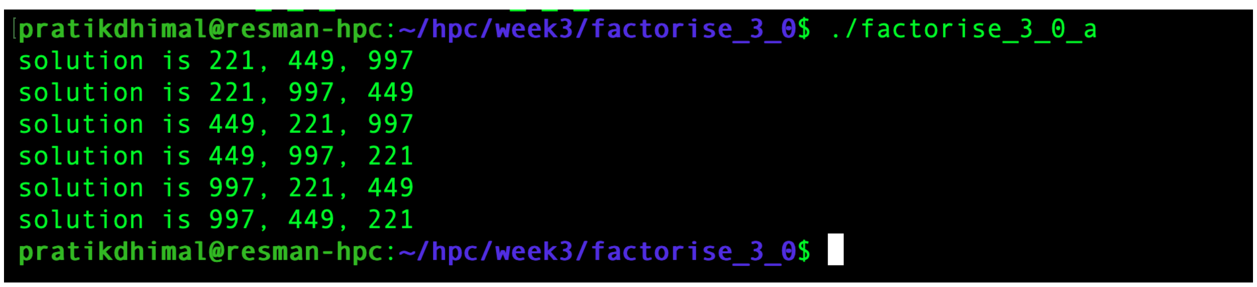
Pratik Dhimal
2407779
Workshop_Part1

Factorize_3_0

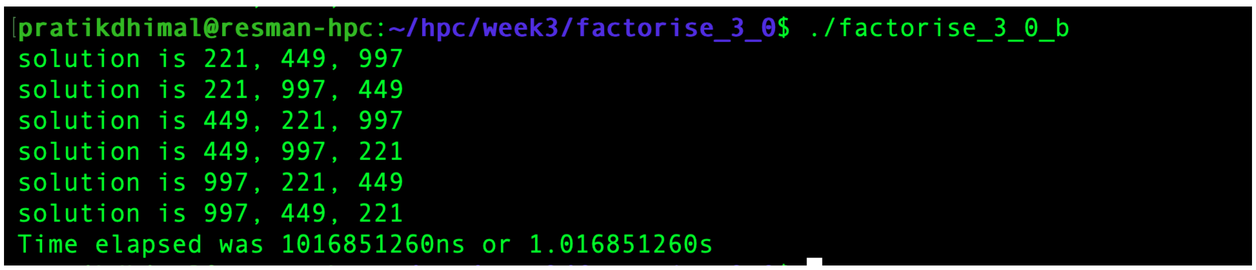
Factorize_3_0_a



```
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ ./factorise_3_0_a
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$
```

This program attempts to identify three integers (a, b, c) within the range 0–999 whose product equals 98,931,313. It uses three nested loops to exhaustively test every possible combination, and once a correct match is discovered, it outputs the values of a, b, and c.

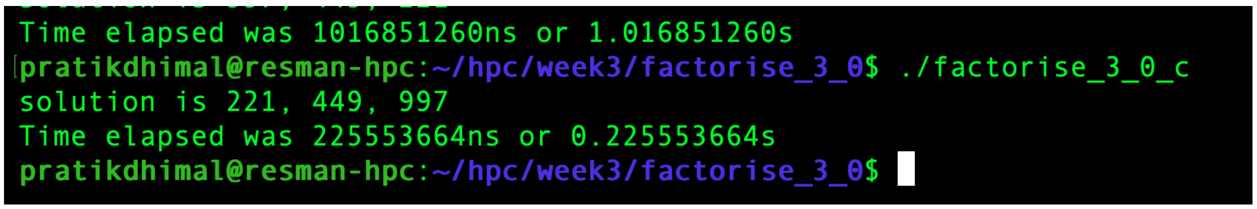
Factorize_3_0_b



```
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ ./factorise_3_0_b
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 1016851260ns or 1.016851260s
```

This version performs the same factor-search process, checking for three numbers whose product is 98,931,313. However, it also tracks how long the program runs using nanosecond-level timing. After execution, it displays both the discovered factors (if any) and the total runtime.

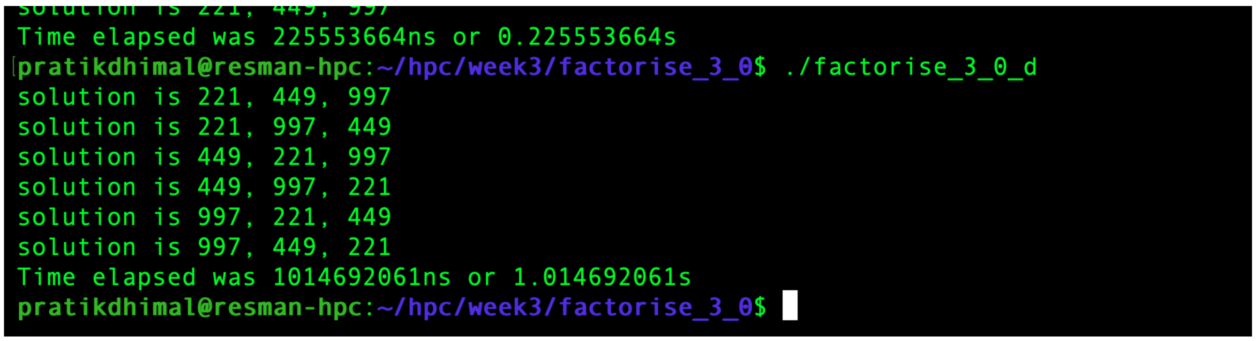
Factorize_3_0_c



```
Time elapsed was 1016851260ns or 1.016851260s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ ./factorise_3_0_c
solution is 221, 449, 997
Time elapsed was 225553664ns or 0.225553664s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ 
```

Like the earlier versions, this program searches for three values (a, b, c) that multiply to 98,931,313. The improvement here is that it stops as soon as the first valid solution is found, making it more efficient. It also measures and reports the total time required to obtain the result.

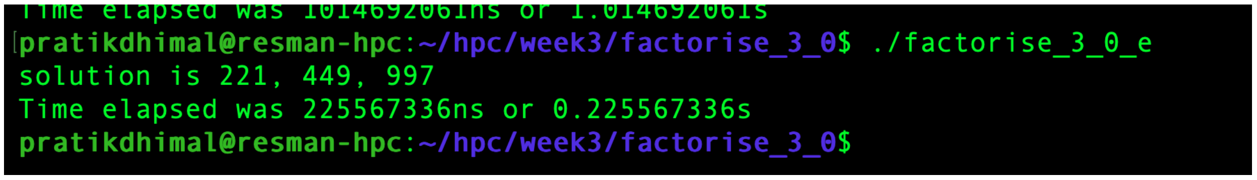
Factorize_3_0_d



```
solution is 221, 449, 997
Time elapsed was 225553664ns or 0.225553664s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ ./factorise_3_0_d
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 1014692061ns or 1.014692061s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ 
```

This version still computes three numbers whose product is 98,931,313, but now the loop boundaries are passed into the factorise() function through a struct. This approach leads to a cleaner, more adaptable design. It also records and outputs the time taken for the factorisation, similar to the previous implementations.

1.3 Factorize_3_0_e

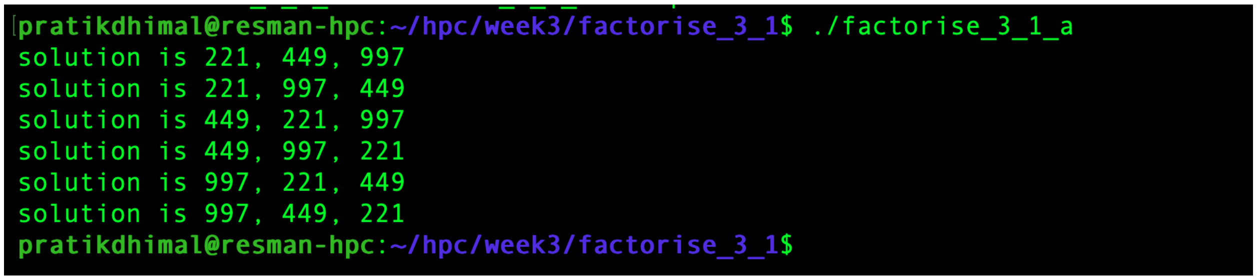


```
Time elapsed was 1014692061ns or 1.014692061s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$ ./factorise_3_0_e
solution is 221, 449, 997
Time elapsed was 225567336ns or 0.225567336s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_0$
```

This program again attempts to locate three integers (a, b, c) whose product equals 98,931,313. The enhancement here is that the same struct is used both to supply input parameters (such as starting point and block size) and to return the discovered factors to the main program. It also displays the result and the total search time, resulting in a more modular and organised code structure.

Factorize_3_1

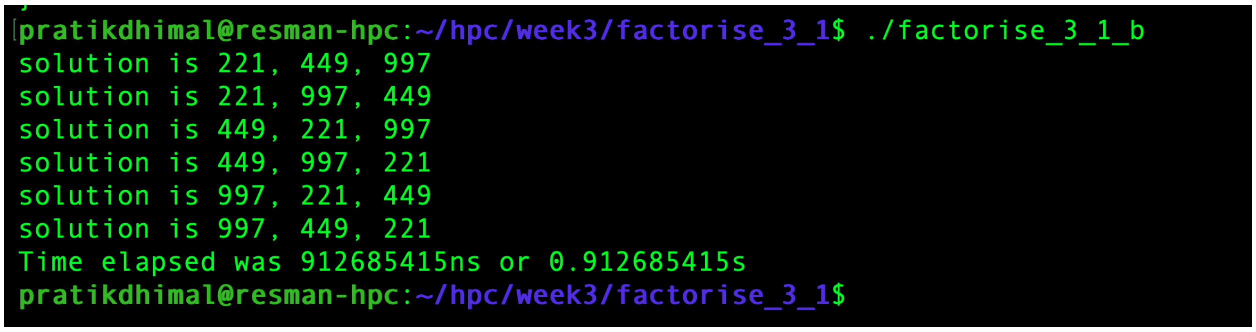
Factorize_3_1_a



```
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$ ./factorise_3_1_a
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$
```

This program uses a single POSIX thread to search for three integers (a, b, c) that multiply to 98,931,313. It launches one thread to perform the computation in parallel and waits for it to complete. Once the correct values are identified, the program prints them, demonstrating basic threading even with only one worker thread.

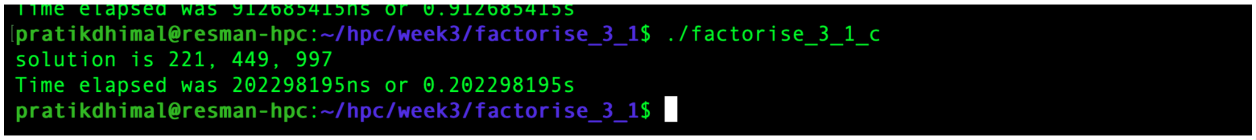
Factorize_3_1_b



```
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$ ./factorise_3_1_b
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 912685415ns or 0.912685415s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$
```

This version also relies on one thread to find three numbers whose product is 98,931,313, but it additionally records the runtime using nanosecond precision. After the thread completes its search, the program outputs both the discovered factors and the total time taken.

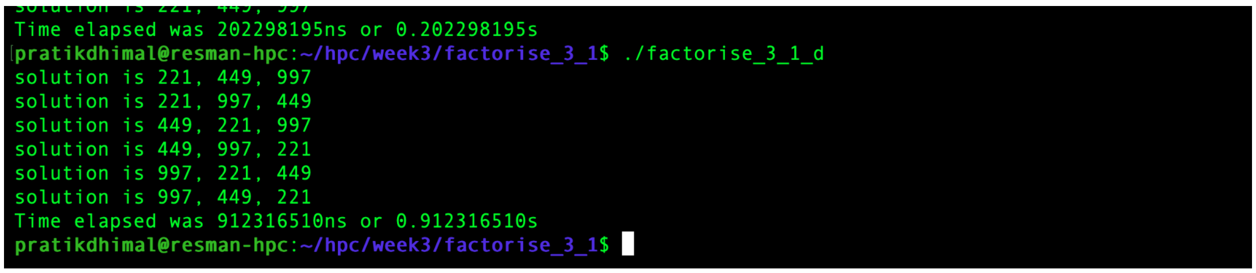
Factorize_3_1_c



```
Time elapsed was 912685415ns or 0.912685415s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$ ./factorise_3_1_c
solution is 221, 449, 997
Time elapsed was 202298195ns or 0.202298195s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$
```

In this implementation, a single thread systematically tests all possible combinations of (a, b, c) from 0 to 999 to determine which set produces 98,931,313. When the correct set is found, the thread prints the result and immediately stops. Meanwhile, the main function measures the total execution time and displays it afterward.

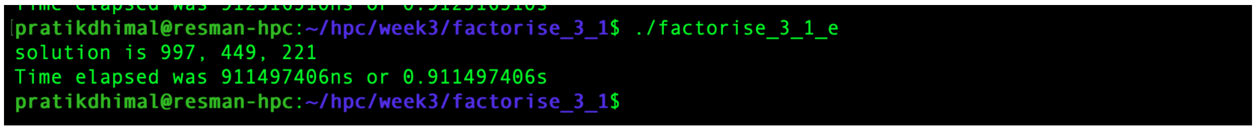
Factorize_3_1_d



```
Time elapsed was 202298195ns or 0.202298195s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$ ./factorise_3_1_d
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 912316510ns or 0.912316510s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$
```

This version passes a small struct to the thread, specifying the portion of the search space it should examine when looking for three values whose product is 98,931,313. The thread iterates through the assigned range, checks each combination of (a, b, c), and prints any valid result. The main function then reports how long the entire search took.

Factorize_3_1_e

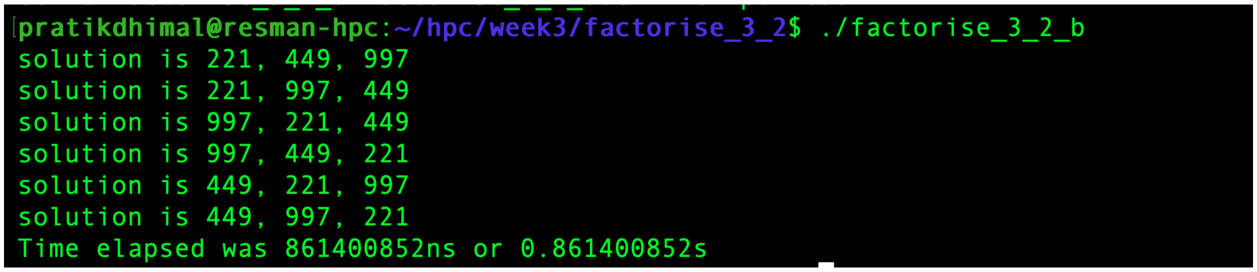


```
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$ ./factorise_3_1_e
solution is 997, 449, 221
Time elapsed was 911497406ns or 0.911497406s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_1$
```

In this improved version, the thread receives a struct that not only defines the search boundaries but also includes fields for storing any solution it finds. The thread examines all relevant (a, b, c) combinations, and once it discovers the correct triple, it writes the results back into the struct. After the thread finishes, the main program retrieves the stored values and prints the final answer along with the measured runtime.

Factorize_3_2

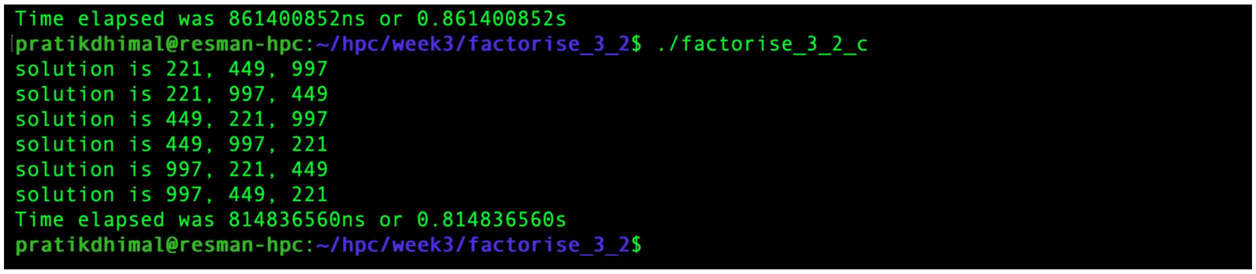
Factorize_3_2_b



```
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_2$ ./factorise_3_2_b
solution is 221, 449, 997
solution is 221, 997, 449
solution is 997, 221, 449
solution is 997, 449, 221
solution is 449, 221, 997
solution is 449, 997, 221
Time elapsed was 861400852ns or 0.861400852s
```

This program divides the search across two threads to identify three numbers whose product is 98,931,313. The workload is split by assigning one thread the range of a values from 0–499, while the second thread handles 500–999. Each thread tests all possible b and c values within its range. Both threads run at the same time, print any valid results, and the main function records how long the dual-threaded search takes.

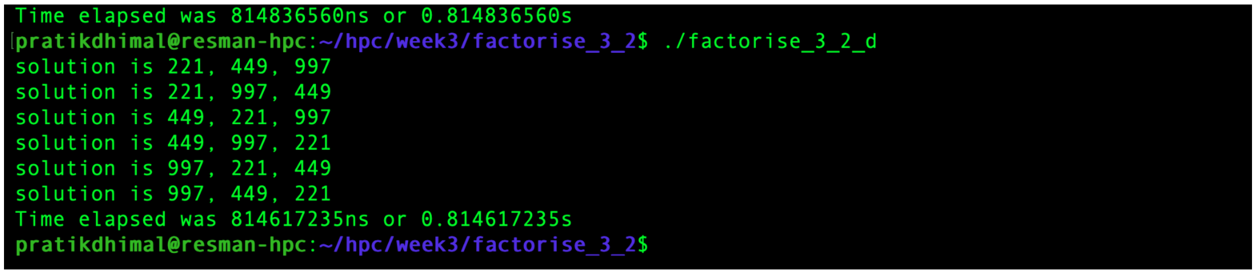
Factorize_3_2_c



```
Time elapsed was 861400852ns or 0.861400852s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_2$ ./factorise_3_2_c
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 814836560ns or 0.814836560s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_2$
```

In this version, two threads are again used, but each one receives a struct specifying its portion of the a range. Both threads execute a parameterised function that iterates through their assigned a values along with all combinations of b and c. Any valid triple is printed, and the main program measures the time required for this block-based, two-threaded search.

Factorize_3_2_d

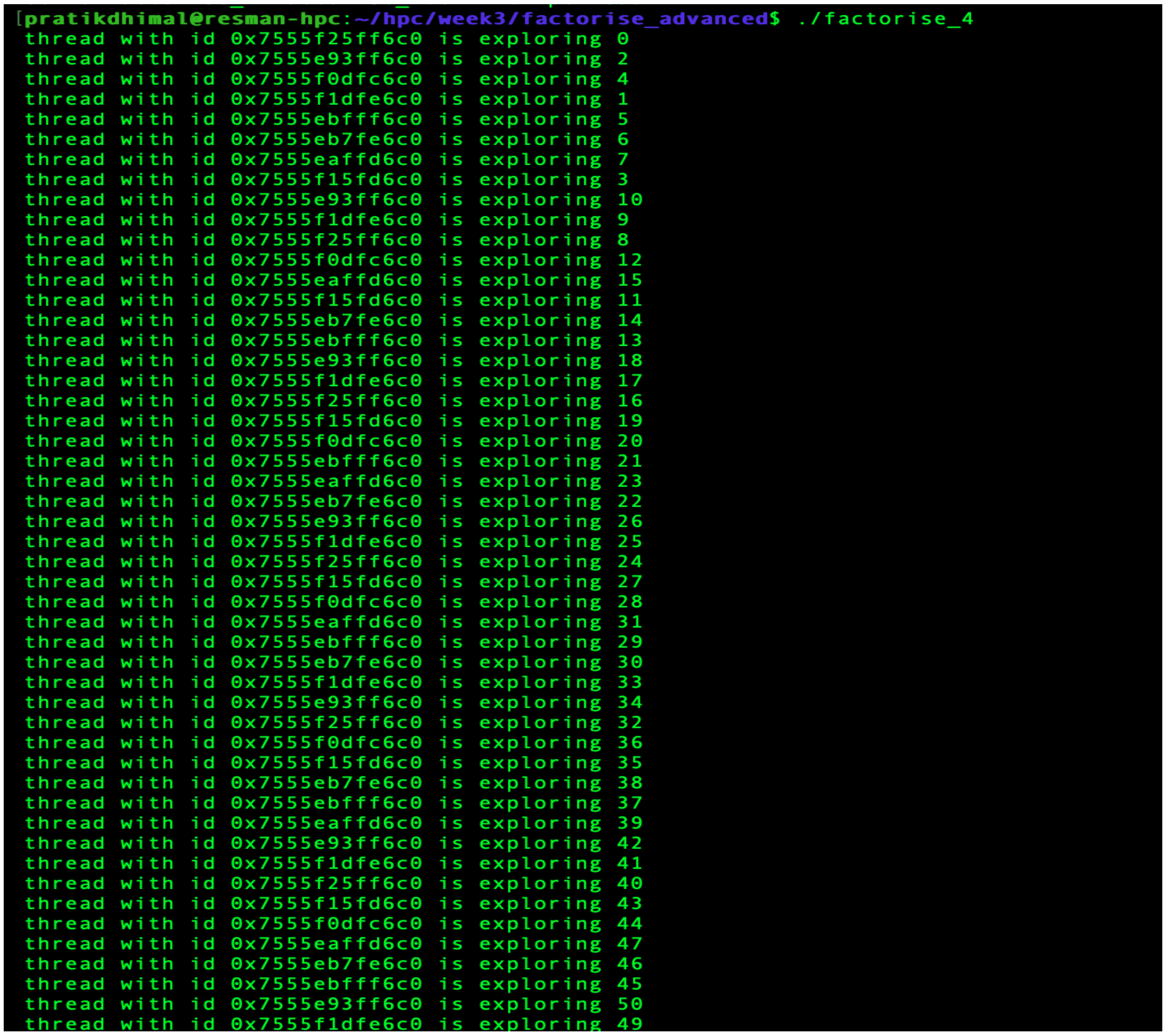


```
Time elapsed was 814836560ns or 0.814836560s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_2$ ./factorise_3_2_d
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 814617235ns or 0.814617235s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_3_2$
```

This approach also uses two threads, but the search is divided by alternating values: one thread handles even values of a (0, 2, 4, …), while the other processes odd values (1, 3, 5, …). Each thread receives a struct that defines its offset and the total number of threads so it can move through the search space in strides. They run concurrently, output any matching (a, b, c) combinations, and the main function reports how long the stride-based search took.

Factorize_advanced

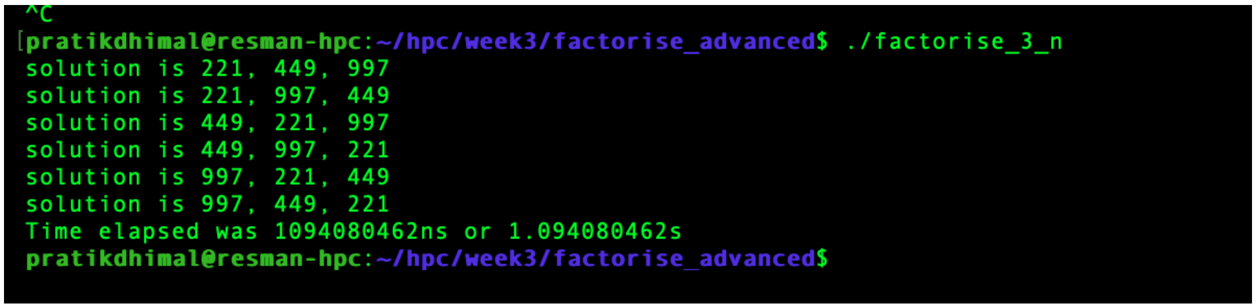
Factorize_4



```
[pratikdhimal@resman-hpc:~/hpc/week3/factorise_advanced$ ./factorise_4
 thread with id 0x7555f25ff6c0 is exploring 0
 thread with id 0x7555e93ff6c0 is exploring 2
 thread with id 0x7555f0dfc6c0 is exploring 4
 thread with id 0x7555f1dfe6c0 is exploring 1
 thread with id 0x7555ebfff6c0 is exploring 5
 thread with id 0x7555eb7fe6c0 is exploring 6
 thread with id 0x7555eaffd6c0 is exploring 7
 thread with id 0x7555f15fd6c0 is exploring 3
 thread with id 0x7555e93ff6c0 is exploring 10
 thread with id 0x7555f1dfe6c0 is exploring 9
 thread with id 0x7555f25ff6c0 is exploring 8
 thread with id 0x7555f0dfc6c0 is exploring 12
 thread with id 0x7555eaffd6c0 is exploring 15
 thread with id 0x7555f15fd6c0 is exploring 11
 thread with id 0x7555eb7fe6c0 is exploring 14
 thread with id 0x7555ebfff6c0 is exploring 13
 thread with id 0x7555e93ff6c0 is exploring 18
 thread with id 0x7555f1dfe6c0 is exploring 17
 thread with id 0x7555f25ff6c0 is exploring 16
 thread with id 0x7555f15fd6c0 is exploring 19
 thread with id 0x7555f0dfc6c0 is exploring 20
 thread with id 0x7555ebfff6c0 is exploring 21
 thread with id 0x7555eaffd6c0 is exploring 23
 thread with id 0x7555eb7fe6c0 is exploring 22
 thread with id 0x7555e93ff6c0 is exploring 26
 thread with id 0x7555f1dfe6c0 is exploring 25
 thread with id 0x7555f25ff6c0 is exploring 24
 thread with id 0x7555f15fd6c0 is exploring 27
 thread with id 0x7555f0dfc6c0 is exploring 28
 thread with id 0x7555eaffd6c0 is exploring 31
 thread with id 0x7555ebfff6c0 is exploring 29
 thread with id 0x7555eb7fe6c0 is exploring 30
 thread with id 0x7555f1dfe6c0 is exploring 33
 thread with id 0x7555e93ff6c0 is exploring 34
 thread with id 0x7555f25ff6c0 is exploring 32
 thread with id 0x7555f0dfc6c0 is exploring 36
 thread with id 0x7555f15fd6c0 is exploring 35
 thread with id 0x7555eb7fe6c0 is exploring 38
 thread with id 0x7555ebfff6c0 is exploring 37
 thread with id 0x7555eaffd6c0 is exploring 39
 thread with id 0x7555e93ff6c0 is exploring 42
 thread with id 0x7555f1dfe6c0 is exploring 41
 thread with id 0x7555f25ff6c0 is exploring 40
 thread with id 0x7555f15fd6c0 is exploring 43
 thread with id 0x7555f0dfc6c0 is exploring 44
 thread with id 0x7555eaffd6c0 is exploring 47
 thread with id 0x7555eb7fe6c0 is exploring 46
 thread with id 0x7555ebfff6c0 is exploring 45
 thread with id 0x7555e93ff6c0 is exploring 50
 thread with id 0x7555f1dfe6c0 is exploring 49
```

This program attempts to find four numbers whose product is 2,984,679,959 by distributing the work across several threads. Each thread begins at a different starting value of a and advances using a fixed stride. Every thread explores its assigned slice of the search area, checking all combinations of b, c, and d, and prints progress along with any solutions found. When all threads complete, the program reports the total time, illustrating how rapidly the complexity grows when more factors are involved.

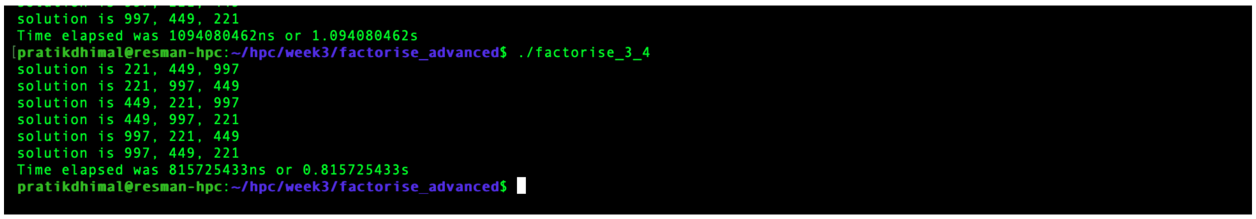
Factorize_3_n



```
^C
[pratikdhimal@resman-hpc:~/hpc/week3/factorise_advanced$ ./factorise_3_n
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 1094080462ns or 1.094080462s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_advanced$
```

Another program uses 16 threads to search for three integers whose product is 98,931,313. Each thread starts at a unique value of a and jumps forward using a stride equal to the total number of threads. Every thread independently evaluates all possible b and c values and prints any valid results. Once every thread has finished, the program prints the total execution time, showing a fully parallelised worst-case search.

Factorize_3_4

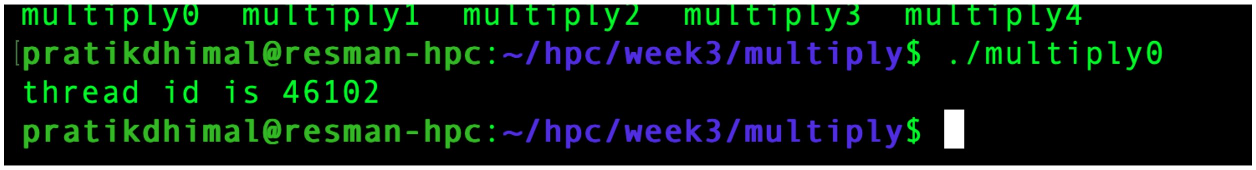


```
solution is 997, 449, 221
Time elapsed was 1094080462ns or 1.094080462s
[pratikdhimal@resman-hpc:~/hpc/week3/factorise_advanced$ ./factorise_3_4
solution is 221, 449, 997
solution is 221, 997, 449
solution is 449, 221, 997
solution is 449, 997, 221
solution is 997, 221, 449
solution is 997, 449, 221
Time elapsed was 815725433ns or 0.815725433s
pratikdhimal@resman-hpc:~/hpc/week3/factorise_advanced$
```

This program employs four threads to find three values that multiply to 98,931,313. Each thread processes every fourth value of a, starting from offsets 0, 1, 2, or 3, forming a striding pattern. For each assigned a, the thread iterates through all b and c combinations and prints any matching set. The main program waits for all threads to complete and then outputs the runtime for this four-thread search.

multiply

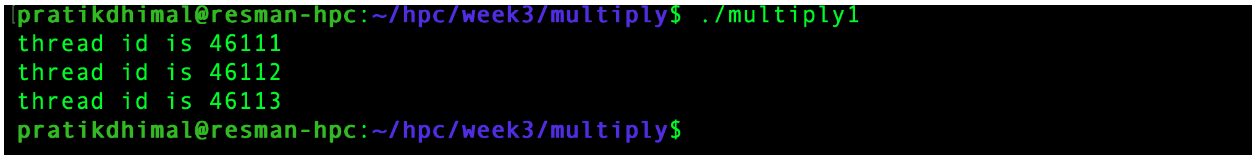
multiply0



```
multiply0  multiply1  multiply2  multiply3  multiply4
pratikdhimal@resman-hpc:~/hpc/week3/multiply$ ./multiply0
thread id is 46102
pratikdhimal@resman-hpc:~/hpc/week3/multiply$
```

Another program demonstrates simple threading by creating one thread that prints its own thread ID. The pthread_create function starts the thread, and pthread_join ensures the main program waits until it finishes. Inside the thread, a system call retrieves and displays the thread's unique ID before exiting.
The thread function uses a system call to get and print its thread ID before exiting

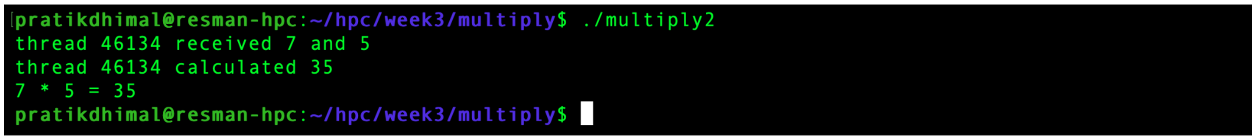
Multiply1



```
pratikdhimal@resman-hpc:~/hpc/week3/multiply$ ./multiply1
thread id is 46111
thread id is 46112
thread id is 46113
pratikdhimal@resman-hpc:~/hpc/week3/multiply$
```

This program extends the idea by creating three threads, all running the same function that prints their unique thread IDs. After launching all three threads with pthread_create, the main program waits for them using pthread_join. The output confirms that each thread has a distinct ID.

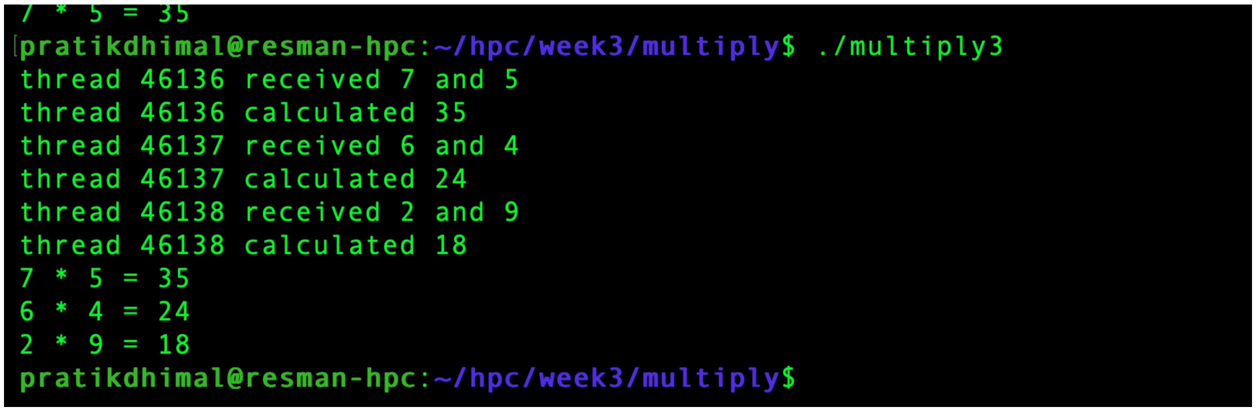
Multiply2



```
pratikdhimal@resman-hpc:~/hpc/week3/multiply$ ./multiply2
thread 46134 received 7 and 5
thread 46134 calculated 35
7 * 5 = 35
pratikdhimal@resman-hpc:~/hpc/week3/multiply$
```

In this program, a single thread multiplies two numbers stored in a struct and writes the result back to that struct. The main function sets the input values, starts the thread, and waits for it to complete. Afterward, it prints the result, showing how threads can return data through shared structures.

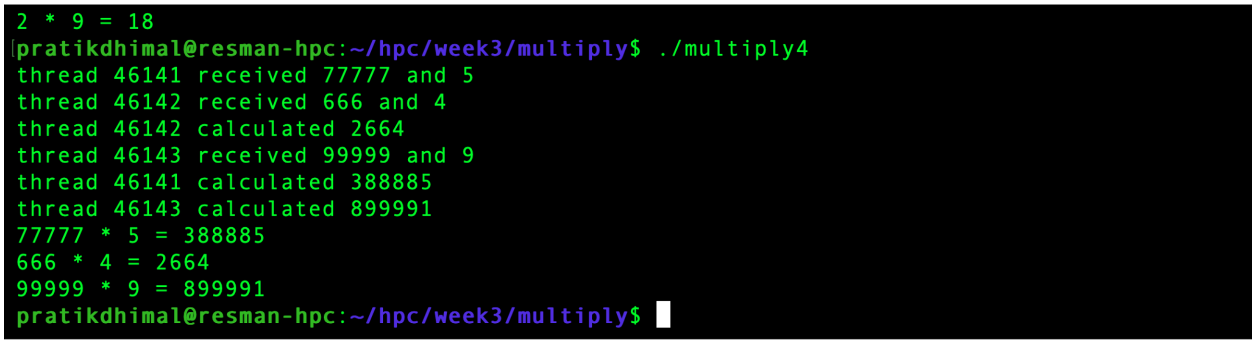
Multiply3



```
7 * 5 = 35
pratikdhimal@resman-hpc:~/hpc/week3/multiply$ ./multiply3
thread 46136 received 7 and 5
thread 46136 calculated 35
thread 46137 received 6 and 4
thread 46137 calculated 24
thread 46138 received 2 and 9
thread 46138 calculated 18
7 * 5 = 35
6 * 4 = 24
2 * 9 = 18
pratikdhimal@resman-hpc:~/hpc/week3/multiply$
```

This program creates three threads, each receiving a struct with two numbers to multiply and store the result in the struct.Each thread prints its ID, the numbers it received, and the result of the multiplication as it runs.After all threads finish, the main function prints the results from all three structs, showing how multiple threads can compute independently and return values via structs.This program creates three threads, each receiving a struct with two numbers to multiply and store the result in the struct.Each thread prints its ID, the numbers it received, and the result of the multiplication as it runs.After all threads finish, the main function prints the results from all three structs, showing how multiple threads can compute independently and return values via structs.

Multiply4

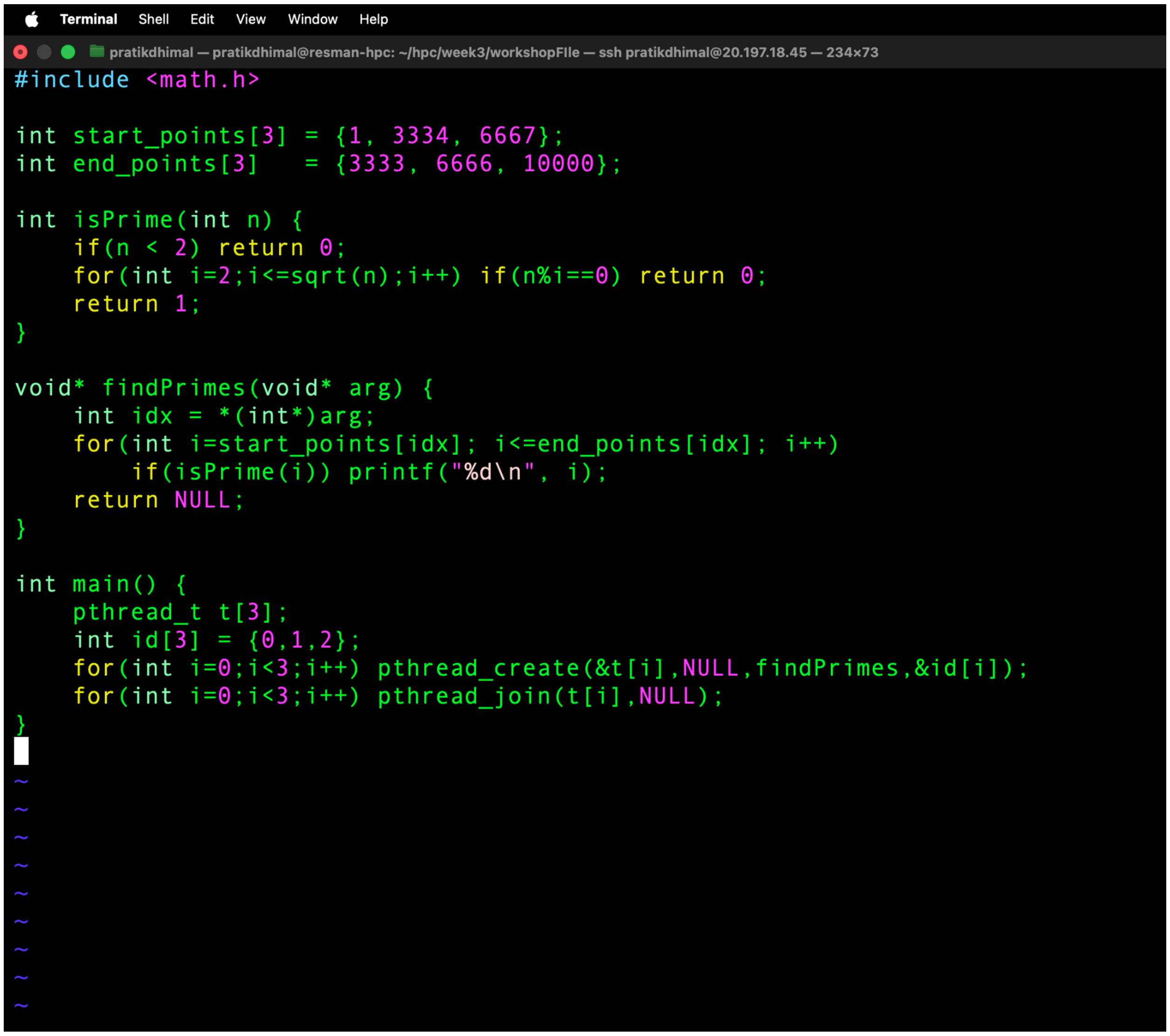

```
2 * 9 = 18
pratikdhimal@resman-hpc:~/hpc/week3/multiply$ ./multiply4
thread 46141 received 77777 and 5
thread 46142 received 666 and 4
thread 46142 calculated 2664
thread 46143 received 99999 and 9
thread 46141 calculated 388885
thread 46143 calculated 899991
77777 * 5 = 388885
666 * 4 = 2664
99999 * 9 = 899991
pratikdhimal@resman-hpc:~/hpc/week3/multiply$
```

In this final program, three threads are launched, each receiving a struct with two numbers to multiply, but the multiplication is performed through repeated addition instead of using the * operator. Each thread prints its ID, the numbers it processes, and the incremental results as it repeatedly adds. Once all threads finish, the main program outputs the final values, highlighting how threads can perform computations and communicate results through structs.

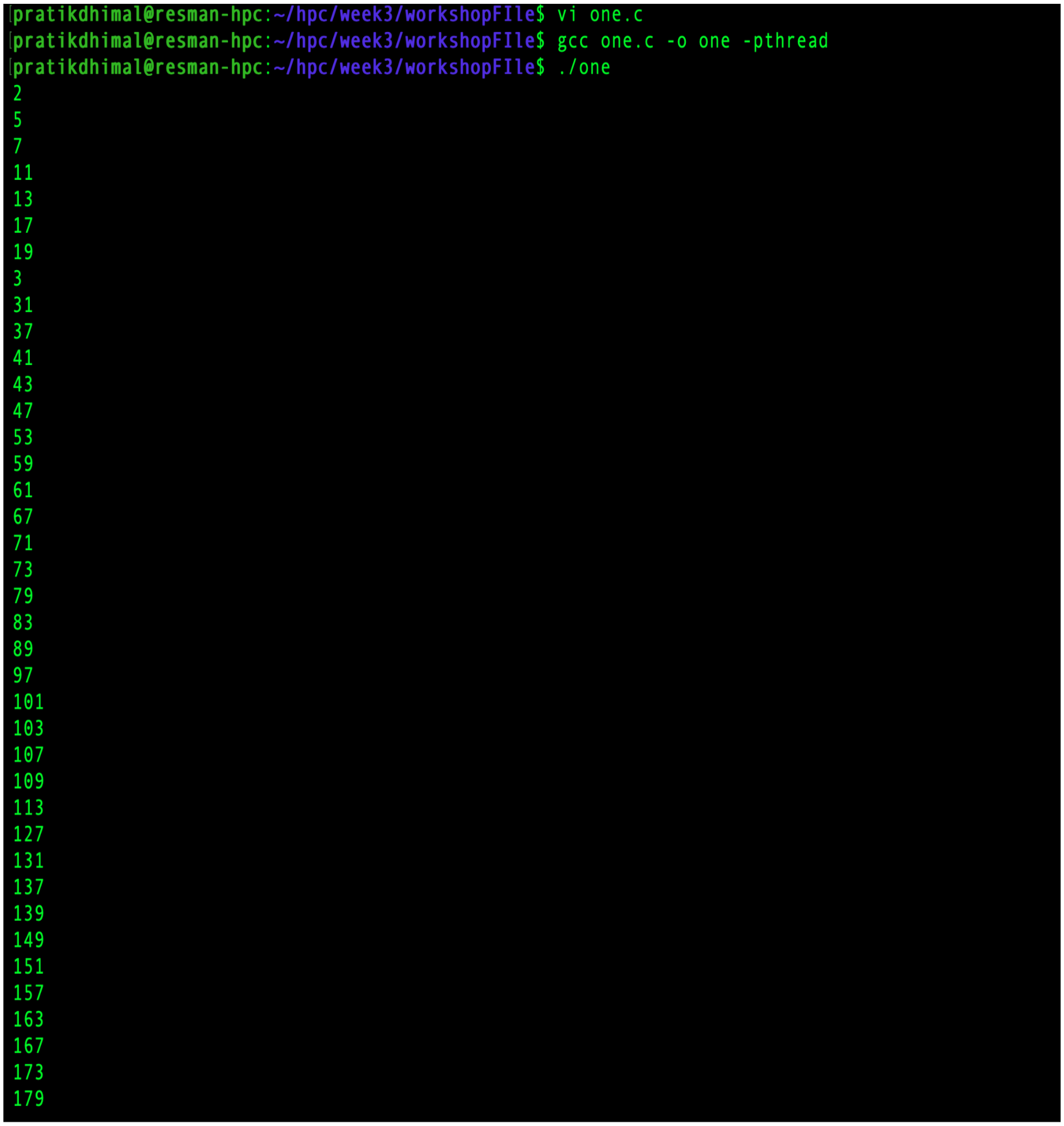
Workshop_Part2

ONE

This program uses three threads to search for all prime numbers between 1 and 10,000. Each thread determines its own segment of the range to process. It uses an is_prime() function that relies on square-root optimisation to test primality. All threads run concurrently and print the prime numbers they discover.


Terminal Shell Edit View Window Help
pratikdhimal — pratikdhimal@resman-hpc: ~/hpc/week3/workshopFIle — ssh pratikdhimal@20.197.18.45 — 234×73

```
#include <math.h>

int start_points[3] = {1, 3334, 6667};
int end_points[3]   = {3333, 6666, 10000};

int isPrime(int n) {
    if(n < 2) return 0;
    for(int i=2;i<=sqrt(n);i++) if(n%i==0) return 0;
    return 1;
}

void* findPrimes(void* arg) {
    int idx = *(int*)arg;
    for(int i=start_points[idx]; i<=end_points[idx]; i++)
        if(isPrime(i)) printf("%d\n", i);
    return NULL;
}

int main() {
    pthread_t t[3];
    int id[3] = {0,1,2};
    for(int i=0;i<3;i++) pthread_create(&t[i],NULL,findPrimes,&id[i]);
    for(int i=0;i<3;i++) pthread_join(t[i],NULL);
}

~
~
~
~
~
~
~
~
~
```

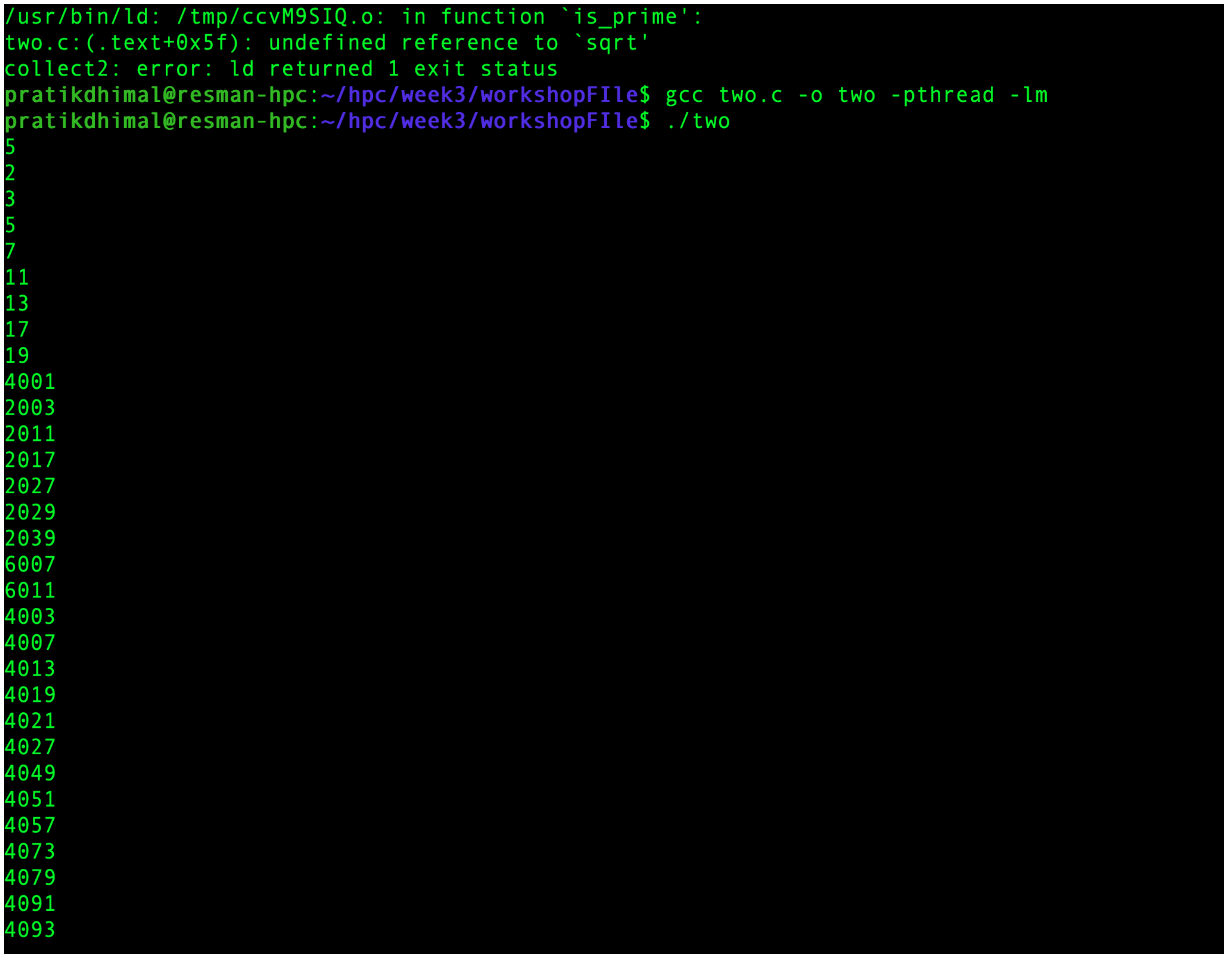


```
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ vi one.c
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ gcc one.c -o one -pthread
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ ./one
2
5
7
11
13
17
19
3
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
```

TWO

In this version, the user specifies how many threads to create. The program then splits the range 1–10,000 evenly based on that number. Every thread prints the primes it finds within its assigned block, and the main function waits for all threads to complete before finishing.

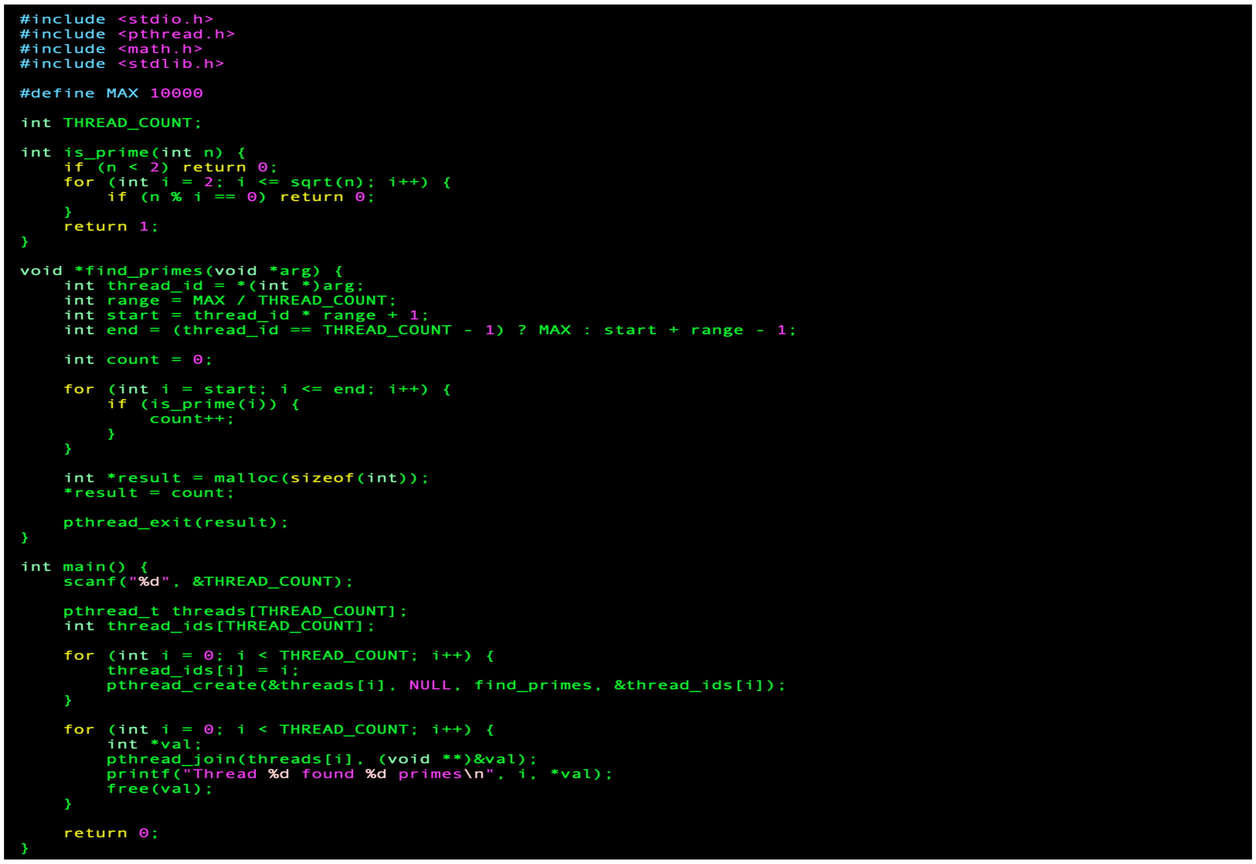
```c
#include <stdio.h>
#include <pthread.h>
#include <math.h>

#define MAX 10000

int THREAD_COUNT;

int is_prime(int n) {
    if (n < 2) return 0;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

void *find_primes(void *arg) {
    int thread_id = *(int *)arg;
    int range = MAX / THREAD_COUNT;
    int start = thread_id * range + 1;
    int end = (thread_id == THREAD_COUNT - 1) ? MAX : start + range - 1;

    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            printf("%d\n", i);
        }
    }

    return NULL;
}

int main() {
    scanf("%d", &THREAD_COUNT);

    pthread_t threads[THREAD_COUNT];
    int thread_ids[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, find_primes, &thread_ids[i]);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```



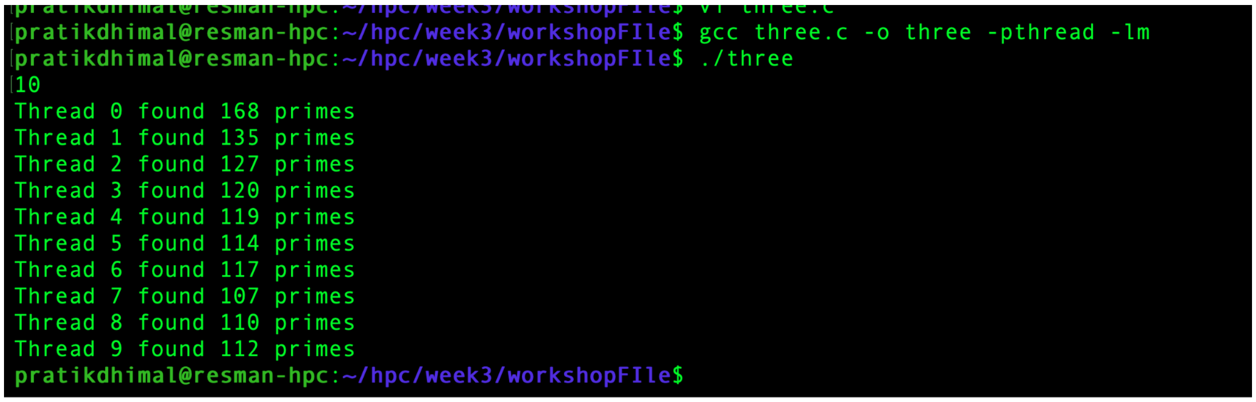
```
/usr/bin/ld: /tmp/ccvM9SIQ.o: in function `is_prime':
two.c:(.text+0x5f): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ gcc two.c -o two -pthread -lm
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ ./two
5
2
3
5
7
11
13
17
19
4001
2003
2011
2017
2027
2029
2039
6007
6011
4003
4007
4013
4019
4021
4027
4049
4051
4057
4073
4079
4091
4093
```

THREE

Another variant again asks the user for the number of threads. Instead of printing primes, each thread counts how many primes exist in its assigned portion of 1–10,000. These counts are stored in a shared thread_data array rather than being returned through pointers. Once all threads finish, the main program prints how many primes each thread discovered.



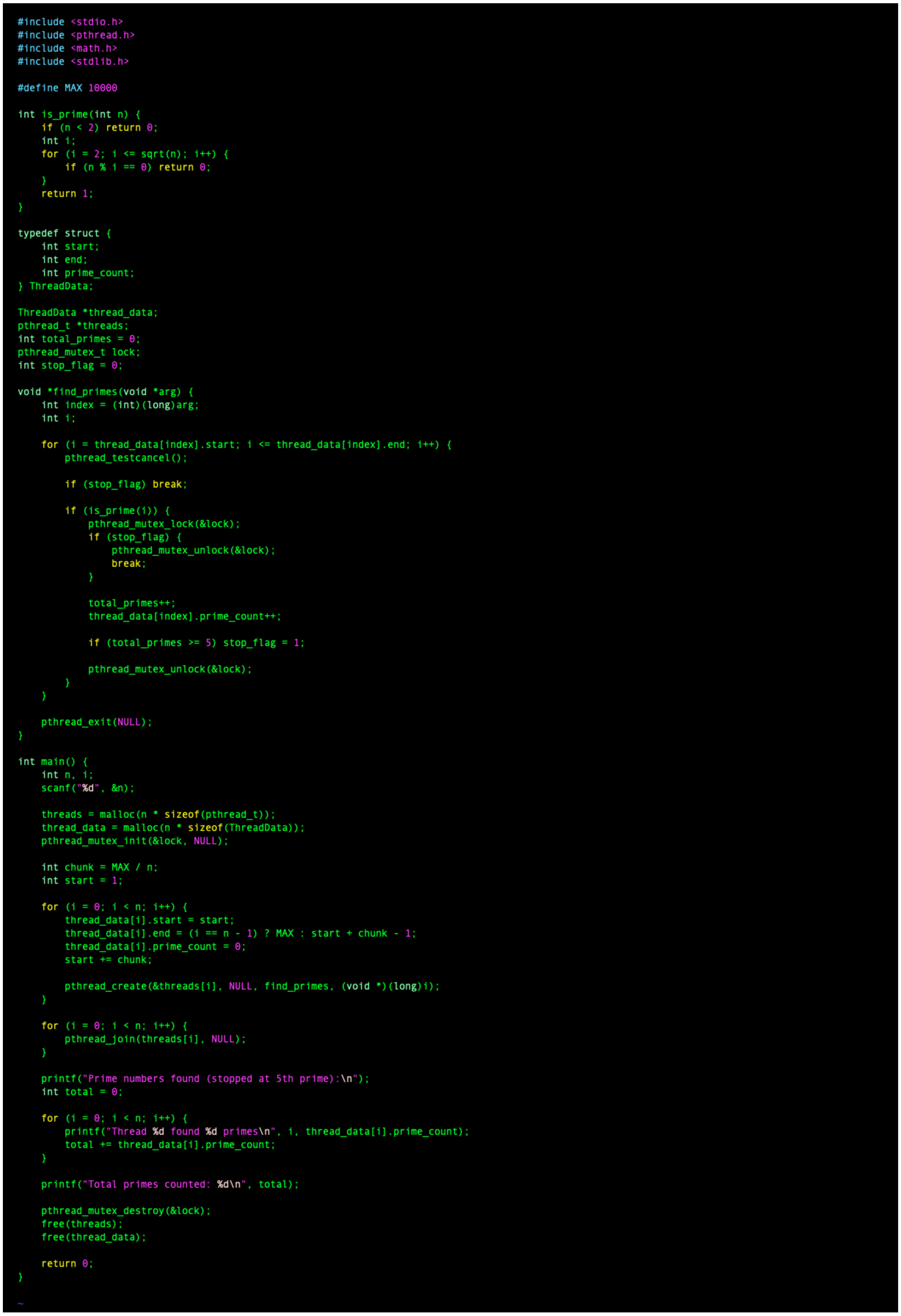
```c
#include <stdio.h>
#include <pthread.h>
#include <math.h>
#include <stdlib.h>

#define MAX 10000

int THREAD_COUNT;

int is_prime(int n) {
    if (n < 2) return 0;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

void *find_primes(void *arg) {
    int thread_id = *(int *)arg;
    int range = MAX / THREAD_COUNT;
    int start = thread_id * range + 1;
    int end = (thread_id == THREAD_COUNT - 1) ? MAX : start + range - 1;

    int count = 0;

    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            count++;
        }
    }

    int *result = malloc(sizeof(int));
    *result = count;

    pthread_exit(result);
}

int main() {
    scanf("%d", &THREAD_COUNT);

    pthread_t threads[THREAD_COUNT];
    int thread_ids[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, find_primes, &thread_ids[i]);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *val;
        pthread_join(threads[i], (void **)&val);
        printf("Thread %d found %d primes\n", i, *val);
        free(val);
    }

    return 0;
}
```



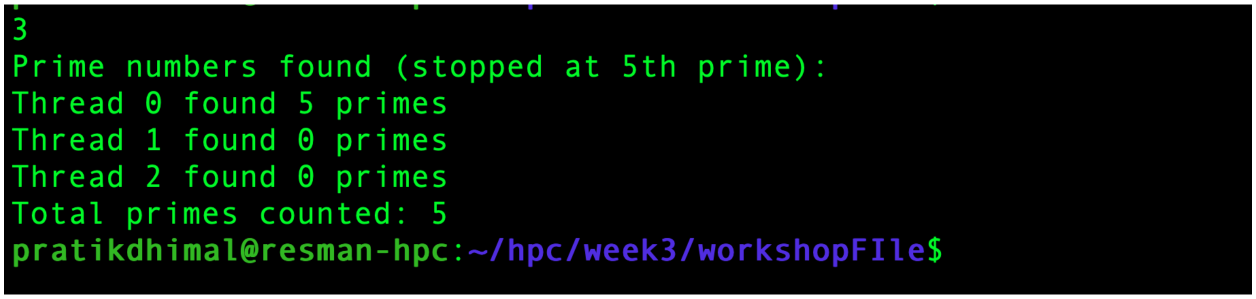
```
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ vi three.c
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ gcc three.c -o three -pthread -lm
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$ ./three
10
Thread 0 found 168 primes
Thread 1 found 135 primes
Thread 2 found 127 primes
Thread 3 found 120 primes
Thread 4 found 119 primes
Thread 5 found 114 primes
Thread 6 found 117 primes
Thread 7 found 107 primes
Thread 8 found 110 primes
Thread 9 found 112 primes
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$
```

FOUR

In the final version, threads share a global counter called total_primes, which is protected by a mutex to prevent race conditions. Each time a thread identifies a prime number, it increases both its own private counter and the global counter. When the shared total reaches five primes, the thread stops. After all threads have exited, the main program prints how many primes were found by each thread.



```c
#include <stdio.h>
#include <pthread.h>
#include <math.h>
#include <stdlib.h>

#define MAX 10000

int is_prime(int n) {
    if (n < 2) return 0;
    int i;
    for (i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

typedef struct {
    int start;
    int end;
    int prime_count;
} ThreadData;

ThreadData *thread_data;
pthread_t *threads;
int total_primes = 0;
pthread_mutex_t lock;
int stop_flag = 0;

void *find_primes(void *arg) {
    int index = (int)(long)arg;
    int i;

    for (i = thread_data[index].start; i <= thread_data[index].end; i++) {
        pthread_testcancel();

        if (stop_flag) break;

        if (is_prime(i)) {
            pthread_mutex_lock(&lock);
            if (stop_flag) {
                pthread_mutex_unlock(&lock);
                break;
            }

            total_primes++;
            thread_data[index].prime_count++;

            if (total_primes >= 5) stop_flag = 1;

            pthread_mutex_unlock(&lock);
        }
    }

    pthread_exit(NULL);
}

int main() {
    int n, i;
    scanf("%d", &n);

    threads = malloc(n * sizeof(pthread_t));
    thread_data = malloc(n * sizeof(ThreadData));
    pthread_mutex_init(&lock, NULL);

    int chunk = MAX / n;
    int start = 1;

    for (i = 0; i < n; i++) {
        thread_data[i].start = start;
        thread_data[i].end = (i == n - 1) ? MAX : start + chunk - 1;
        thread_data[i].prime_count = 0;
        start += chunk;

        pthread_create(&threads[i], NULL, find_primes, (void *)(long)i);
    }

    for (i = 0; i < n; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Prime numbers found (stopped at 5th prime):\n");
    int total = 0;

    for (i = 0; i < n; i++) {
        printf("Thread %d found %d primes\n", i, thread_data[i].prime_count);
        total += thread_data[i].prime_count;
    }

    printf("Total primes counted: %d\n", total);

    pthread_mutex_destroy(&lock);
    free(threads);
    free(thread_data);

    return 0;
}

~
```


```
3
Prime numbers found (stopped at 5th prime):
Thread 0 found 5 primes
Thread 1 found 0 primes
Thread 2 found 0 primes
Total primes counted: 5
pratikdhimal@resman-hpc:~/hpc/week3/workshopFIle$
```