Assignment-1 Report
OS2 – CS3523

Name: *Taha Adeel Mohammed*
Roll Number: *CS20BTECH11052*

Code Design

```c
typedef struct pair{
    long int X;
    long int Y;
} Pair;
```

- We first define a struct pair to hold the co-ordinates of the destinations.

```c
int num_of_threads;
Pair pt_0;
int num_of_pts;
Pair *pts;

Pair *closest_pts;
```

- We declare global variables to hold our data, namely the number of threads, the source point, the number of destination points, and the array containing the destination points.
- We also have another array (closest points) which will hold the closest point found by each thread.

We also define some utility functions as follows

```c
/**
 * @brief Parses a string of the form (x,y), where x and y are numbers and stores it in pt.
 *
 * @param pair
 * @param pt
 */
void read_pair(char* pair, Pair *pt){
    int l_location, comma_location, r_location;
    for(int i=0; pair[i] != '\0'; ++i){
        if(pair[i] == '(')
            l_location = i;
        if(pair[i] == ',')
            comma_location = i;
        if(pair[i] == ')')
            r_location = i;
    }

    char temp_x[10], temp_y[10];
    strncpy(temp_x, &pair[l_location + 1], comma_location - l_location - 1);
    strncpy(temp_y, &pair[comma_location + 1], r_location - comma_location - 1);
    temp_x[comma_location - l_location - 1] = '\0';
    temp_y[r_location - comma_location - 1] = '\0';

    pt->X = atoi(temp_x);
    pt->Y = atoi(temp_y);
}
```

- The function **read_pair()** parses a string of the form (x,y), where x and y are numbers and stores it in a Pair pt.
- To do so, we first find the locations of the brackets and the commas.
- Then the substring between them is copied to a temporary string, from which it is converted to an int and stored in the pair.

```c
/**
 * @brief Returns the distance between two points
 *
 * @param p1
 * @param p2
 * @return double
 */
double get_distance(Pair* p1, Pair* p2){
    double dist_sqr = (p1->X - p2->X)*(p1->X - p2->X) + (p1->Y - p2->Y)*(p1->Y - p2->Y);
    return sqrt(dist_sqr);
}
```

- The function **get_distance()** finds the distance between two pairs using the Euclidean Distance formula.

We then have the function which find the closest point in the array as follows.

```c
/**
 * @brief Returns the closest pair from an array/subarray pointed to by "arr" and of size "size".
 *
 * @param arr
 * @param size
 * @return Pair
 */
Pair get_closest(Pair* arr, int size){
    Pair* closest = &arr[0];
    double min_dist = get_distance(&pt_0, closest);

    for(int i = 0; i < size; ++i){
        double dist = get_distance(&pt_0, &arr[i]);
        // usleep(10000);
        if(dist < min_dist){
            min_dist = dist;
            closest = &arr[i];
        }
    }
    return *closest;
}
```

- The function **get_closest()** returns the closest pair from an array/subarray pointed to by "arr" and of size "size".
- This is done by having a variable closest and min_dist which keep track of the current closest point (initialized to the first point in the array).
- The we loop through the array updating “closest” and “min_dist” if we find a closer element (The distances are calculated using the get_distance() function).
- Then the closest pair is returned by the function.

```c
/**
 * @brief A wrapper function for the get_closest() function. This function is called by the threads.
 *
 * @param thread_num An integer which keeps track of which thread has called the function.
 * @return void*
 */
void* get_closest_wrapper(void* thread_num){
    int thread_id = *(int*)thread_num;
    free(thread_num);
    int size = num_of_pts/num_of_threads;

    if(thread_id < (num_of_threads - 1))
        closest_pts[thread_id] = get_closest(pts + thread_id*size, size);
    else
        closest_pts[thread_id] = get_closest(pts + thread_id*size, num_of_pts - thread_id*size);
}
```

- The function **get_closest_wrapper()** is a wrapper function for the get_closest() function.
- It is called by the threads when they are created, hence its return type and parameter type are void*.

- The parameter it takes is the id (0 to num_of_threads-1) of the thread that calls it. This lets it know which subarray the function should work on.
- We typecast the void* parameter to int and store the value in it to get the thread_id, then free the memory allocated to thread_num(memory allocated in the main() function.).
- The size of the subarrays the threads operate on is then found.
- Then the get_closest() function is called for the appropriate subarray.
- To take care of cases where the number of points is not a perfect multiple of the number of threads, we pass all remaining points to the last thread.
- The return value of get_closest() is stored in the global array "closest_pts".

- In the main function,

```c
int main(){
    // Read input
    char* filename = "input.txt";
    FILE* input_file = fopen(filename, "r");
    if(input_file == NULL){
        printf("Error opening file.\n");
        return EXIT_FAILURE;
    }

    fscanf(input_file, "%d", &num_of_threads);
    closest_pts = (Pair*) malloc(num_of_threads * sizeof(Pair));

    char pair[25];
    fscanf(input_file, "%s", pair);
    read_pair(pair, &pt_0);

    fscanf(input_file, "%d", &num_of_pts);
    pts = (Pair*)malloc(num_of_pts*sizeof(Pair));

    for(int i=0; i<num_of_pts; ++i){
        fscanf(input_file, "%s", pair);
        read_pair(pair, &pts[i]);
    }

    fclose(input_file);
```

- ♦ We first have to read the input.
- ♦ The file with the input in it is opened, and we begin scanning it.
- ♦ The number of threads is read from the first line of the file and appropriate memory is allocated to closest_pts.
- ♦ Then the src point is read, followed by the number of points and then the destination points.
- ♦ After reading the input completely, the file is closed.

```c
    // Start measuring time
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Start the threads
    pthread_t *threads = (pthread_t*) malloc(num_of_threads * sizeof(pthread_t));
    for(int i = 0; i < num_of_threads; ++i){
        int* j = (int*)malloc(sizeof(int));
        *j = i;
        pthread_create(&threads[*j], NULL, &get_closest_wrapper, (void*)(j));
    }

    // Wait for the threads to finish execution
    for(int i = 0; i < num_of_threads; ++i)
        pthread_join(threads[i], NULL);

    // Find the closest point among the closest points found by the different threads
    Pair closest = get_closest(closest_pts, num_of_threads);

    // Stop and measure the time.
    clock_gettime(CLOCK_REALTIME, &end);
    double elapsed_microseconds = (end.tv_sec - start.tv_sec)*1e6 + (end.tv_nsec - start.tv_nsec)*1e-3;
```

- We then start the timer, followed by spawning the threads.
- The threads are spawned in a for loop with the function get_closest_wrapper() and the parameters 0,1,2,…
- The value of i is copied to a new address to make it thread-safe, as multiple threads would access the address of i if it is passed directly to the threads.
- Hence the threads start executing parallelly and find the closest point in their subarrays.
- Meanwhile, the main function after spawning all the child process, waits for them to finish executing by calling pthread_join() in a for loop.
- Once all the threads finish execution, the array closest_pts would contain the closest points found by each thread.
- Then the main function finds the closest point among all these points.
- Finally, the timer is stopped and the elapsed time is calculated.

```
    printf("%.2lf microseconds\n", elapsed_microseconds);
    printf("(%ld, %ld)\n", closest.X, closest.Y);


    return EXIT_SUCCESS;
}
```

- Finally, the time taken for the parallel execution and the closest point are outputted to standard output.
- Then main() terminates successfully.

Result Analysis

- An important point to be noted is that thread creation and destruction also takes a significant amount of time. This can be called thread overhead.
- For a “small” test size of 1000ish input points, the thread overhead is significantly greater than the time required to traverse through the points and compare them for the closest point.
- Hence the normal execution times graphs are different from what one might expect.
- Therefore, I have considered two different cases for the result analysis. One: the code is timed normally.
- And in the other, before each comparision, the code is made to sleep for a small time(~1 ms (using usleep(1000)). This causes the thread overhead to become insignificant and we get the results we expect.

The test inputs for all the following graphs are included in the zip file.

1) Execution Time vs Number of points

❖ Normal Execution

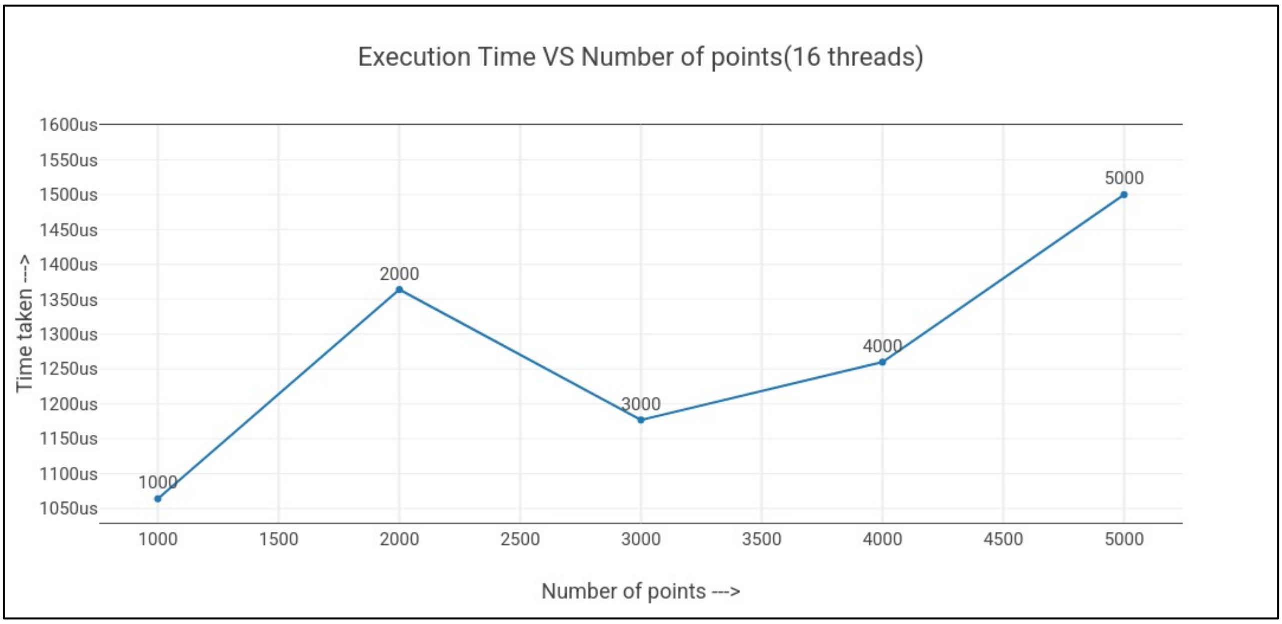
```
taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 1000
1064.23 microseconds
(51204, 82693)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 2000
1364.20 microseconds
(60295, 72713)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 3000
1177.51 microseconds
(82255, 42874)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 4000
1260.34 microseconds
(94124, 52718)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 5000
1500.88 microseconds
(30423, 7141)
```



- **Execution with sleep called**

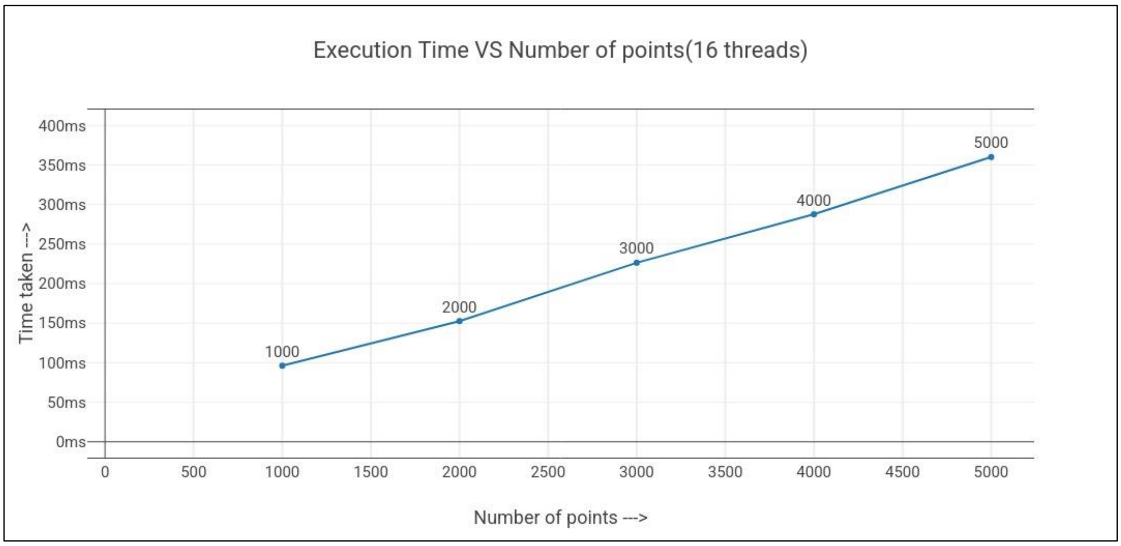
```
taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 1000
96204.47 microseconds
(51204, 82693)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 2000
152674.99 microseconds
(60295, 72713)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 3000
226480.55 microseconds
(82255, 42874)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 4000
287764.79 microseconds
(94124, 52718)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 5000
359935.97 microseconds
(30423, 7141)
```



2) Execution Time VS Number of threads

- **Normal execution**

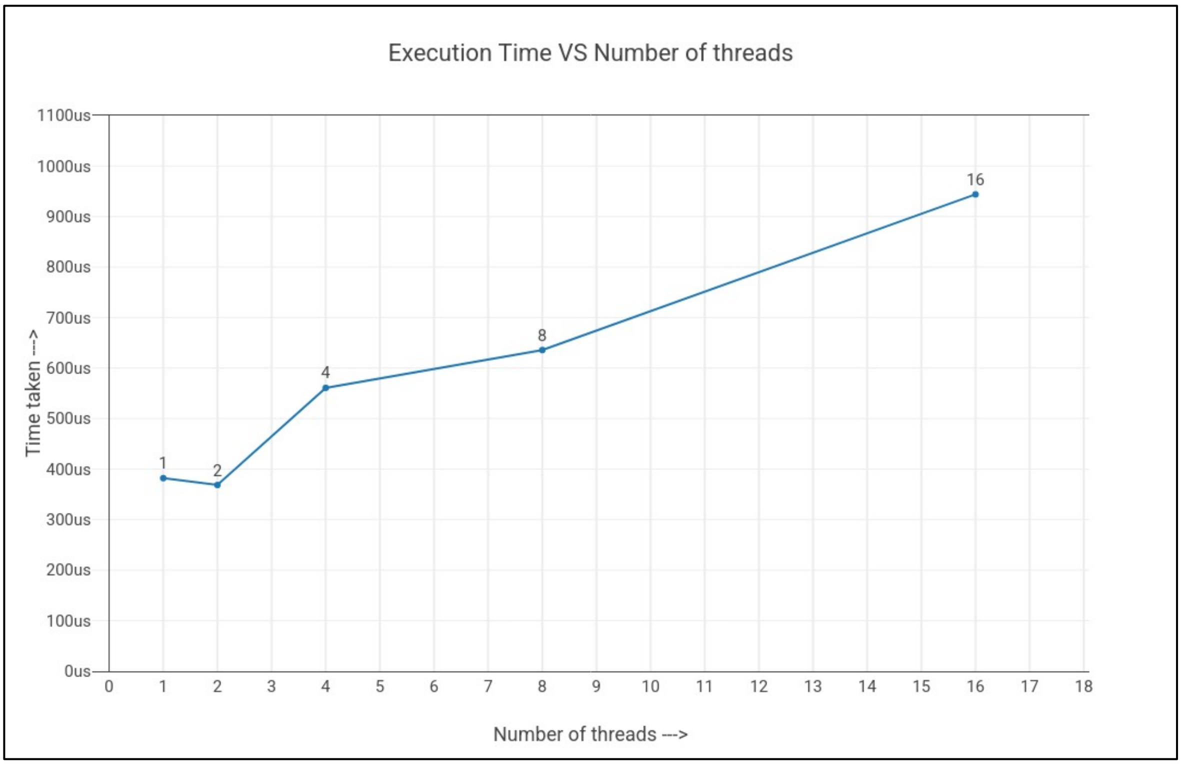
```
taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 1     Number of points: 5000
382.09 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 2     Number of points: 5000
368.55 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 4     Number of points: 5000
560.75 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 8     Number of points: 5000
635.68 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 5000
943.82 microseconds
(30423, 7141)
```



❖ Execution with sleep

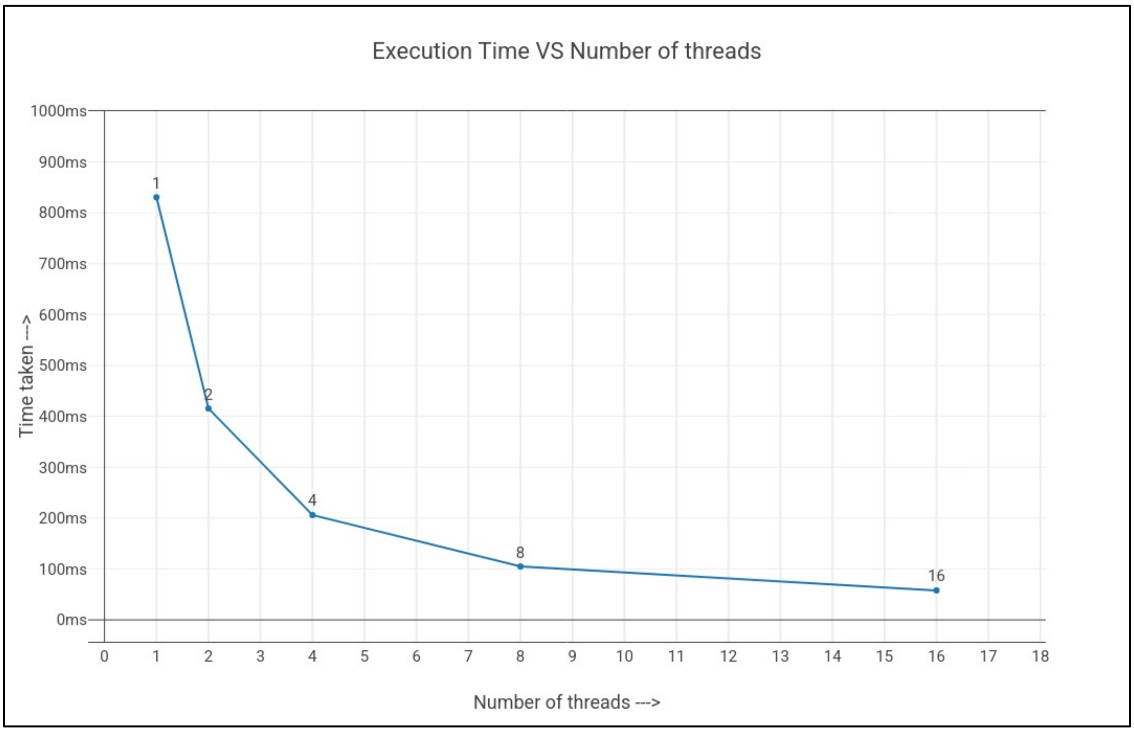
```
taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 1     Number of points: 5000
830713.58 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 2     Number of points: 5000
415509.85 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 4     Number of points: 5000
205991.76 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 8     Number of points: 5000
105460.63 microseconds
(30423, 7141)

taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 16    Number of points: 5000
58012.52 microseconds
(30423, 7141)
```



- Sequential execution of the code(without creating any threads) gives the following time.

```
taha_adeel@IdeaPad: ~/Desktop/OS 2/Assignment 1
$ ./main
Number of threads: 0     Number of points: 5000
87.16 microseconds
(30423, 7141)
```

We can see that this is much better than the parallel execution as the gain due to the task being done parallelly is insignificant compared to the overload.