THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

Parallel and Distributed Computing PROJECT I

ID: 202010285____

Name: Yahya Azzam____

Section: ____

1- Introduction

Algorithm choice: Binary Search

is a searching algorithm used in a sorted array by **r**epeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Thread Independence: there's no need for synchronization during the search process itself because threads operate on separate sections of the array, except when updating shared results (like the found index).

Static Division of Data (balanced workload): The array can be split into equal-sized chunks, one per thread. Each thread performs a standard binary search within its chunk, independently of others.

Asynchronous Termination: If one thread finds the target, it can notify others to terminate early reducing unnecessary computation by cancelling other threads.

2-Sequential Implementation

❖ Code explanation with snippets:

```
int binarySearch(int arr[], int left, int right, int x) {
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (arr[mid] == x)
         return mid;
      else if (arr[mid] < x)
         left = mid + 1;
      else
         right = mid - 1;
   }
   return -1;
}
```

left and right : represent the current bounds of the array segment being searched.

mid : is the index at the center of the current segment.

compares the target with arr[mid].

If equal, the target is found and the loop breaks.

If the target is smaller then:

search continues in the left half (right = mid - 1).

If the target is larger then:

search continues in the right half (left = mid + 1).

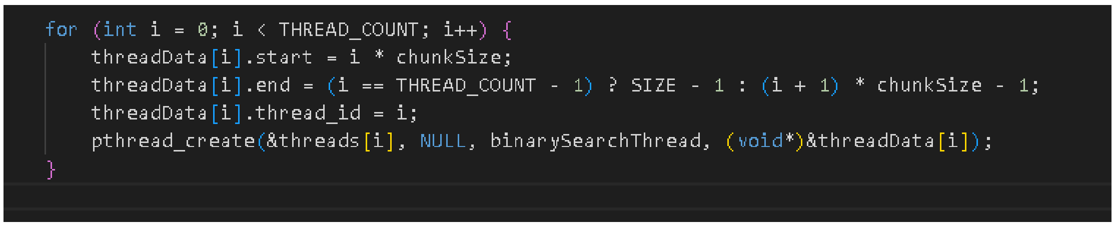
repeating until either the element is found or the search range becomes invalid (left > right).

- ❖ Time measurement methodology:
  Using <chrono> a part of C++ standard library that provides high precision. Repeating the search 5 times.
  The time is captured in seconds using <double> & in Microseconds using <microseconds>

```
auto startTime = high_resolution_clock::now();
                :
                :
                :
double en=get_time();
auto endTime = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(endTime - startTime);
```

3-Parallelization Strategy

- ❖ work dividing among threads:
  the input array is divided into equal-sized chunks based on the number of threads .Each thread independently performs a binary search within its assigned chunk.



```
for (int i = 0; i < THREAD_COUNT; i++) {
    threadData[i].start = i * chunkSize;
    threadData[i].end = (i == THREAD_COUNT - 1) ? SIZE - 1 : (i + 1) * chunkSize - 1;
    threadData[i].thread_id = i;
    pthread_create(&threads[i], NULL, binarySearchThread, (void*)&threadData[i]);
}
```

When one thread finds the target value:
It stores the index that contain the result.

& cancels all other threads to save CPU time using pthread_cancel().
a mutex lock is used when updating the shared variable resultIndex to prevent race conditions.

❖ Pthread functions/structs used:

Using struct Each thread has:

Its assigned start and end indices.
Its thread_id .

Functions used:

- pthread_create_t()
- pthread_join()
- pthread_cancel()
- pthread_mutex_init()
- pthread_mutex_lock() / unlock()
- pthread_setcancelstate() / setcanceltype()
- pthread_testcancel()

4-Experiments

Hardware:

OS: Vbox linux .

CPU: 2 core cpu

RAM: 4 GB

Compiler: g++ on linux

Input sizes and thread counts tested:

Length of the arrays tested is

Arr[1000]: 1K

Arr[100000] :100K

Arr[500000]: 500K

Arr[1000000] : 1million

Arr[10000000] : 10 million

Thread count:1 ,2, 4, 8.

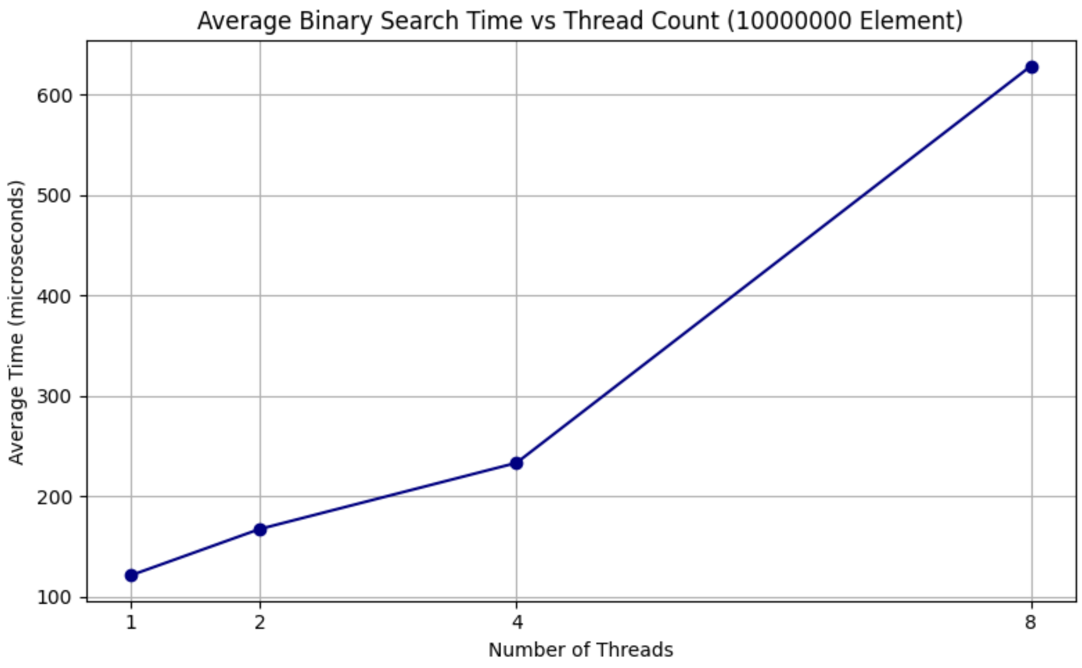
5-Results

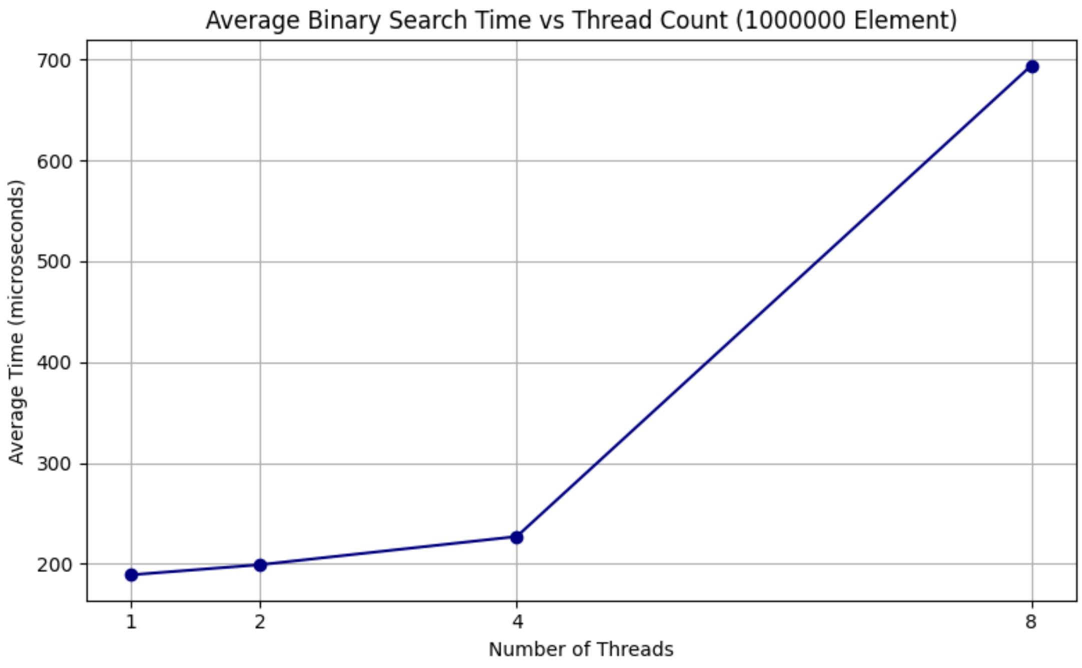
Sequential execution time

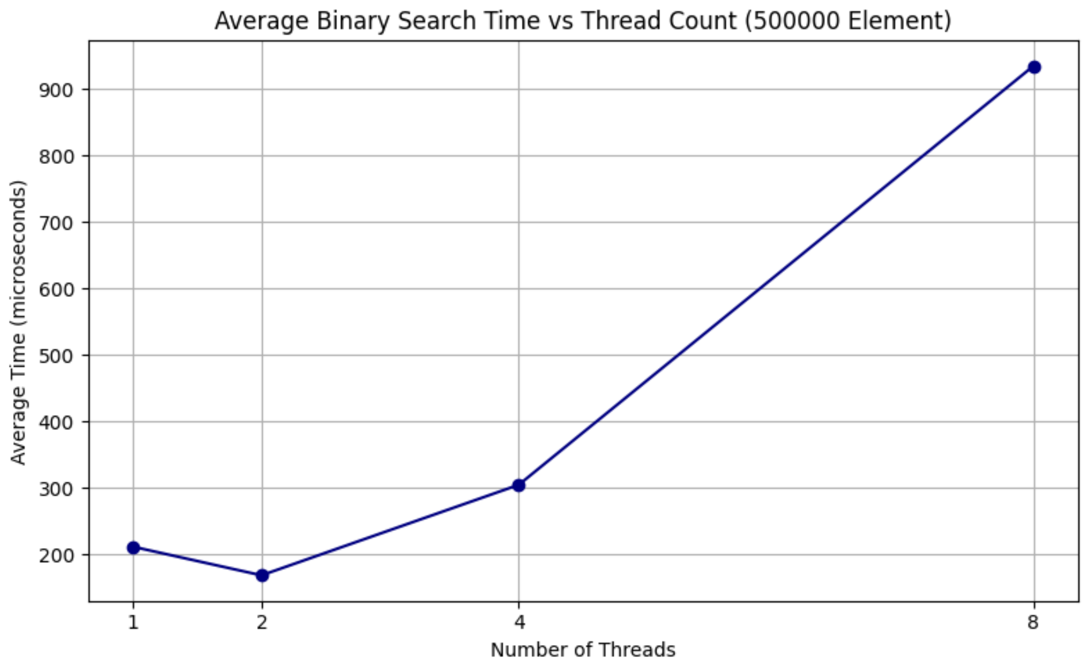
Length of array	time
10 M	3 µS
1 M	2 µS
500 K	2 µS
100 K	1 µS
1 K	Less than 1 µS

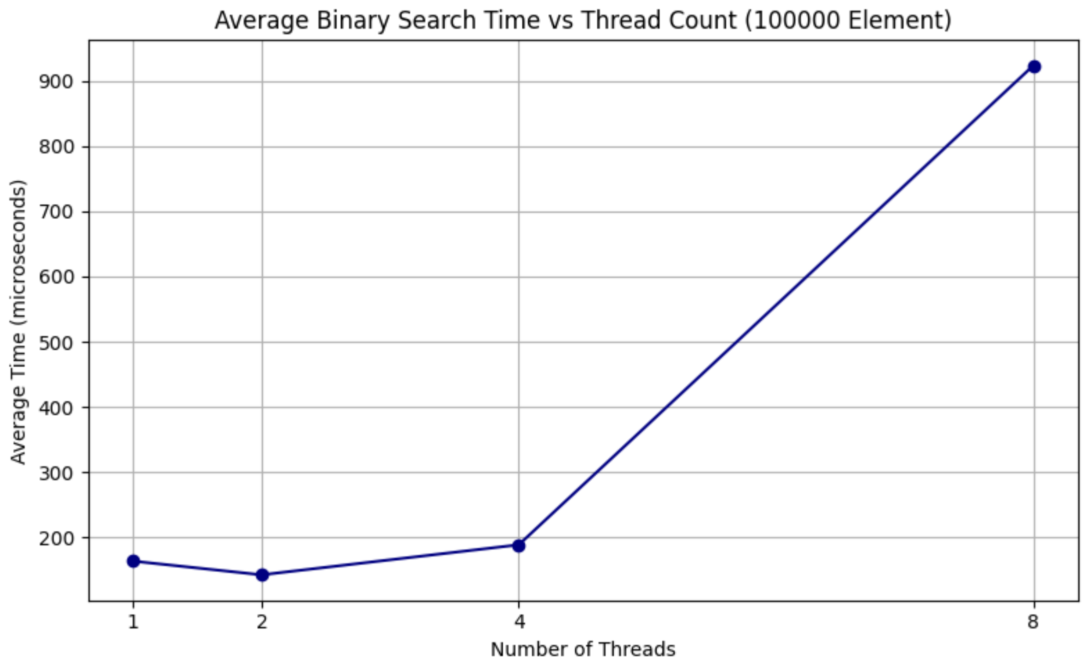
Execution time using multi threaded

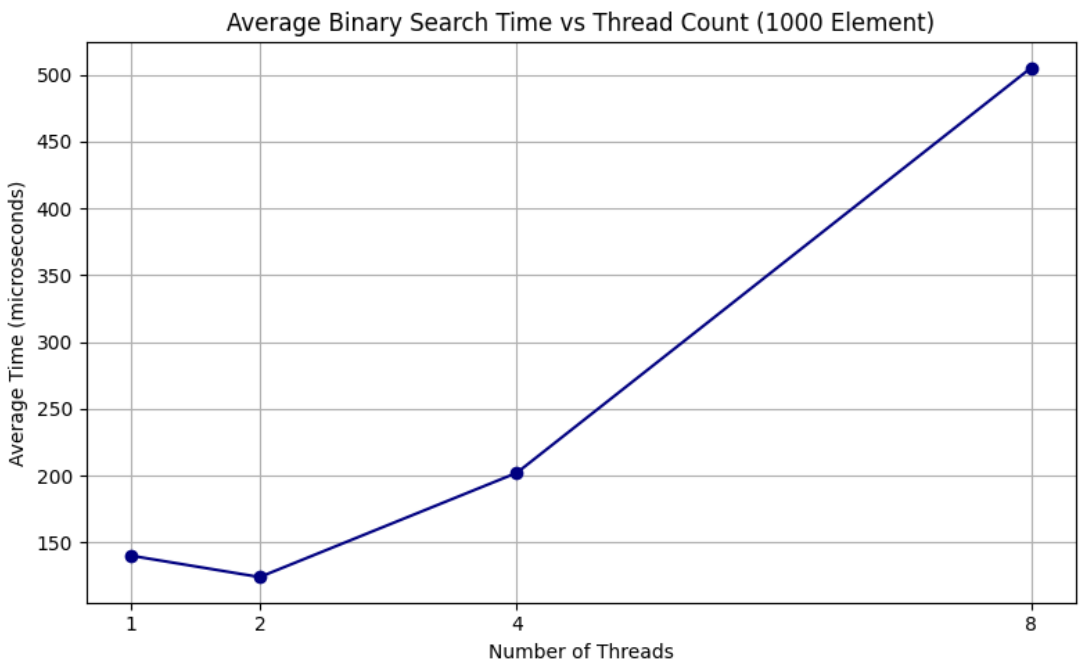
Th count	Time 10M	Time 1M	Time 500K	Time 100K	Time 1K
1	121	189	212	164	140
2	167	199	169	143	124
4	233	227	305	189	202
8	628	694	934	923	505


Average Binary Search Time vs Thread Count (10000000 Element)
Average Time (microseconds)
100
200
300
400
500
600
1
2
4
8
Number of Threads


Average Binary Search Time vs Thread Count (1000000 Element)
Average Time (microseconds)
200
300
400
500
600
700
1
2
4
8
Number of Threads

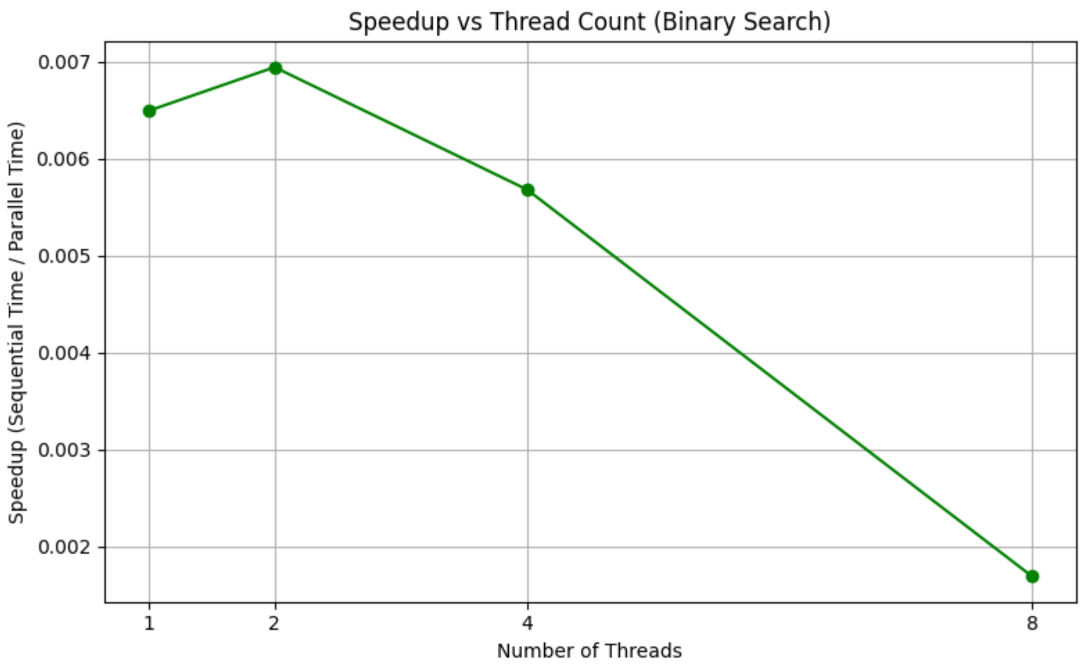

Average Binary Search Time vs Thread Count (500000 Element)
Average Time (microseconds)
200
300
400
500
600
700
800
900
1
2
4
8
Number of Threads


Average Binary Search Time vs Thread Count (100000 Element)
Average Time (microseconds)
200
300
400
500
600
700
800
900
1
2
4
8
Number of Threads



Speedup

Number of threads	speedup
2	0.00694444
4	0.00868182
8	0.00169205



6-Discussion

Graphs show that its not just sublinear it’s suboptimal and actually declining with increasing thread count, which indicates performance degradation rather than improvement .

Binary Search is Already Fast Binary search has $O(\log n)$ complexity due to very small workload, In binary search, only one thread finds the element, while others do redundant or wasted work. Threads may finish unevenly, and some threads may do no useful work at all.

especially for short tasks like binary search, Creating, managing, and destroying multiple threads adds significant overhead, The cost of threading often outweighs any performance gain

More threads increase pressure on the cache and memory bus. cache misses will degrade performance.

Actual speedup is less than Amdahl’s Law predictions due to the practical reasons listed above.

This confirms that Amdahl’s Law provides an upper bound not a guaranteed outcome.

7- Conclusion

In conclusion, sometimes may be a little improvement when using 2 threads but overall Binary search is not a good candidate for parallelization in practice because it’s already efficient and the added threading overhead reduces performance instead of improving it.

Tools used in the project:

- GCC
- VScode
- Gihub
- Python
- Excel
- chatGPT
- VBox