CS 301
High-Performance Computing

Lab 8 - Binary Search and Fast Fourier Transform

Problem 1: Parallel Binary Search

Jay Dobariya (202101521)
Akshar Panchani (202101522)

April 8, 2024

Contents

1 Introduction

In this lab report,a key algorithm of computer science, binary search is renowned for its effective time complexity. It is very helpful for finding sorted arrays because it repeatedly divides the search space in half. Nevertheless, parallelization offers a chance to improve performance while working with unsorted arrays. In this research, we describe the use of OpenMP for the parallel implementation and performance analysis of the binary search algorithm.

2 Hardware Details

2.1 Hardware Details for LAB207 PCs

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 6
- On-line CPU(s) list: 0-5
- Thread(s) per core: 1
- Core(s) per socket: 6
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 155
- Model name: Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz
- Stepping: 10
- CPU MHz: 799.992
- CPU max MHz: 4100.0000
- CPU min MHz: 800.0000
- BogoMIPS: 6000.00
- Virtualization: VT-x
- L1d cache: 192KB
- L1i cache: 192KB

- L2 cache: 1.5MB
- L3 cache: 9MB
- NUMA node0 CPU(s): 0-5

2.2 Hardware Details for HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 24
- On-line CPU(s) list: 0-23
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- Stepping: 2
- CPU MHz: 2642.4378
- BogoMIPS: 4804.69
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K
- NUMA node0 CPU(s): 0-5,12-17
- NUMA node1 CPU(s): 6-11,18-23

3 Problem 2

3.1 Brief description of the problem

The task at hand is to put into practice a parallel binary search algorithm that can find a target value quickly by sorting through an unsorted array of numbers. This approach is important because it uses parallel processing capabilities to significantly increase search speed for huge arrays.

3.1.1 Algorithm description

3.2 Serial Binary Search Algorithm:

1. Start with the entire array as the search space.
2. Set the lower bound of the search space to the first index of the array and the upper bound to the last index.
3. While the lower bound is less than or equal to the upper bound:
   (a) Calculate the middle index of the current search space.
   (b) Compare the target value with the element at the middle index.
   - If they are equal, return the middle index (target found).
   - If the target value is less than the middle element, update the upper bound to be one less than the middle index.
   - If the target value is greater than the middle element, update the lower bound to be one more than the middle index.
4. If the target value is not found after the loop, return -1 (target not found).

3.3 Parallel Binary Search Algorithm:

1. Divide the array into equal-sized chunks.
2. Assign each chunk to a different thread for searching.
3. Each thread operates independently on its assigned chunk using the serial binary search algorithm.
4. If any thread finds the target value, return its index.
5. If none of the threads find the target value, return -1 (target not found).

3.4 The complexity of the algorithm- serial

The time complexity of the serial binary search algorithm is O(log n), where n is the size of the array. This logarithmic complexity arises from the fact that with each iteration, the search space is halved.

3.5 Information about parallel implementation

OpenMP directives are used in the binary search algorithm's parallel implementation to divide the search workload among several threads. Concurrent searching is made possible by the fact that each thread works on a different section of the array, potentially cutting down on search time.

3.6 The complexity of the algorithm - Parallel

The complexity of the parallel binary search algorithm remains O(log n) in terms of time complexity. However, the parallel version may exhibit improved performance due to concurrent execution of search tasks.

3.7 Profiling using HPC Cluster (with gprof)

The screenshots of profiling using the HPC Cluster are given below

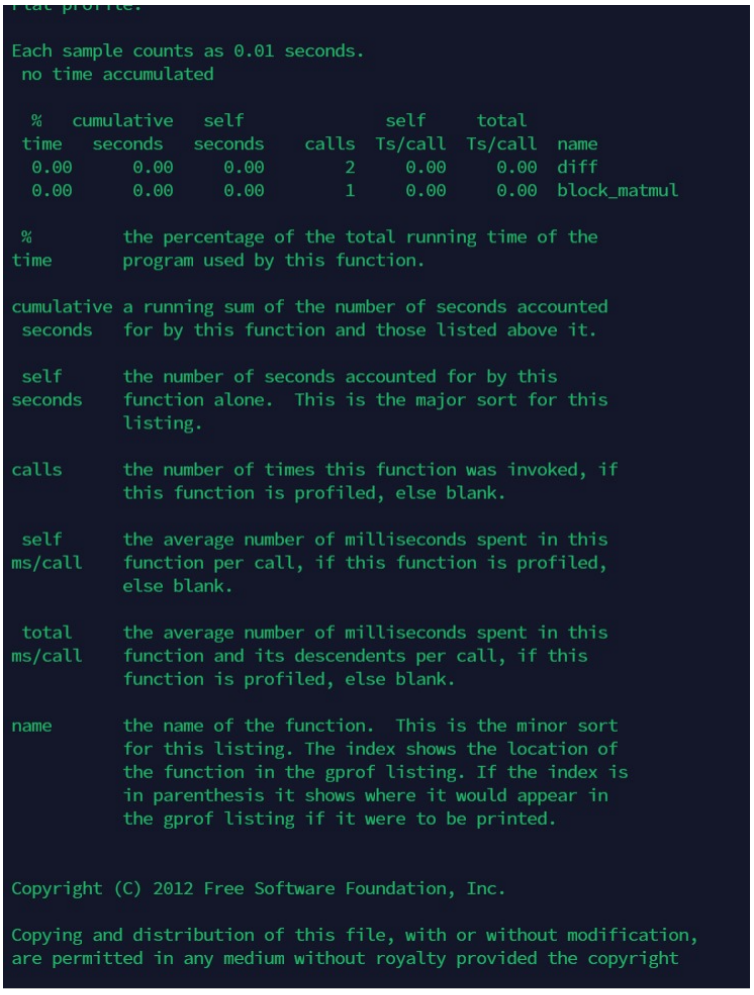

Figure 1: Screenshot of the terminal from HPC Cluster

3.8 Lab207 PC Graph and HPC Cluster Graphs

3.8.1 Graph of Problem Size vs Totaltime for LAB207 PCs

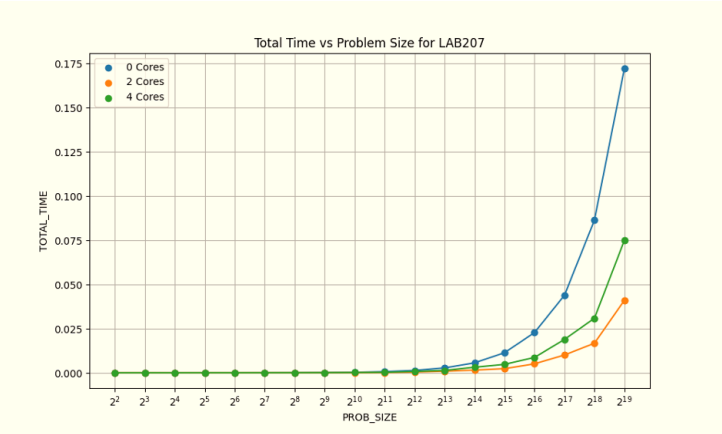

Figure 2: Total mean execution time (Total time) vs Problem size plot for **(Hardware: LAB207 PC, Problem: BINARY_SEARCH)**.

3.8.2 Graph of Problem Size vs Algorithm time for LAB207 PCs

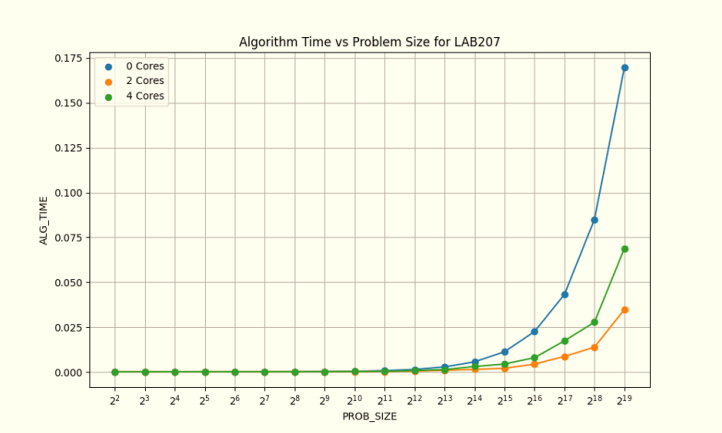

Figure 3: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: LAB207 PC, Problem: BINARY_SEARCH)**.

3.8.3 Graph of Efficiency vs Core for LAB207 PCs

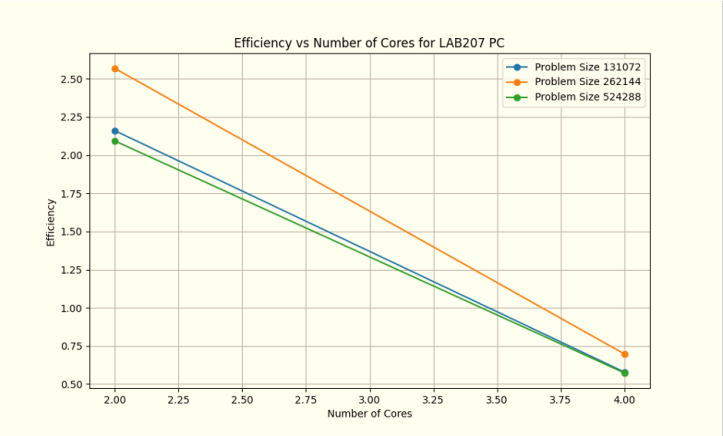

Figure 4: Graph of Efficiency vs Core plot for **(Hardware: LAB207 PC, Problem: BINARY_SEARCH)**.

3.8.4 Graph of Efficiency vs Problem size for LAB207 PCs

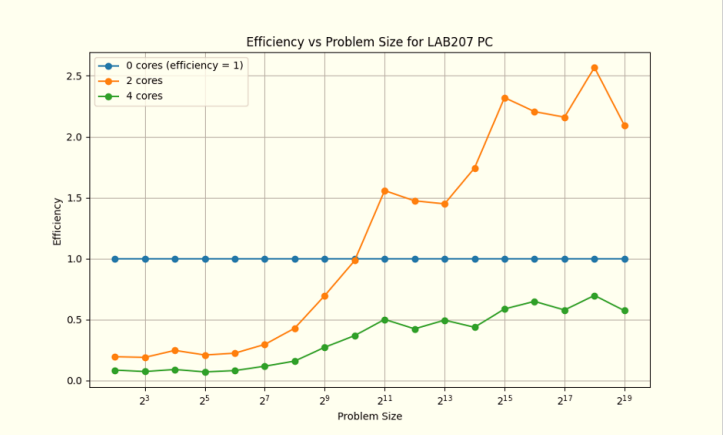

Figure 5: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: LAB207 PC, Problem: BINARY_SEARCH)**.

3.8.5 Graph of Speedup vs Core for LAB207 PCs

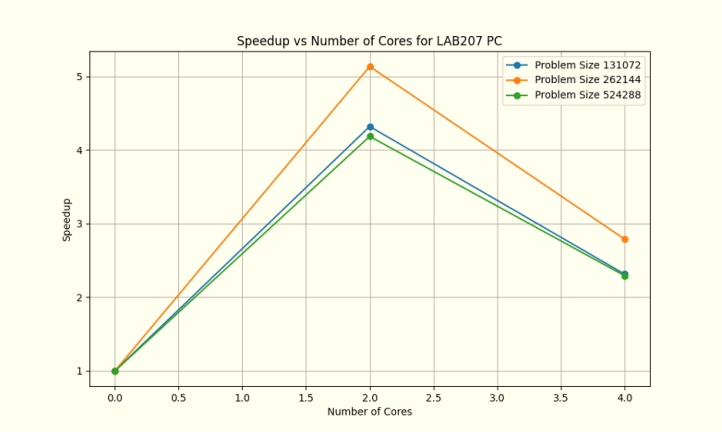

Figure 6: Graph of Speedup vs Core for **(Hardware: LAB207 PC, Problem: BINARY_SEARCH)**.

3.8.6 Graph of Speedup vs Problem size for LAB207 PCs

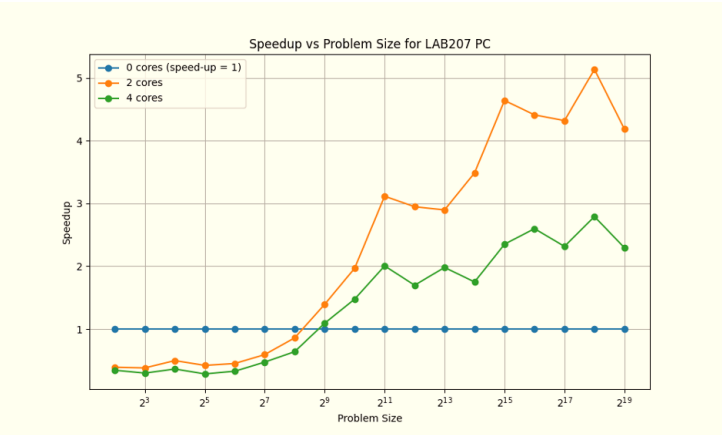

Figure 7: Graph of Speedup vs Problem size for **(Hardware: LAB207 PC, Problem: BINARY_SEARCH)**.

3.8.7 Graph of Problem Size vs Totaltime for HPC CLUSTER

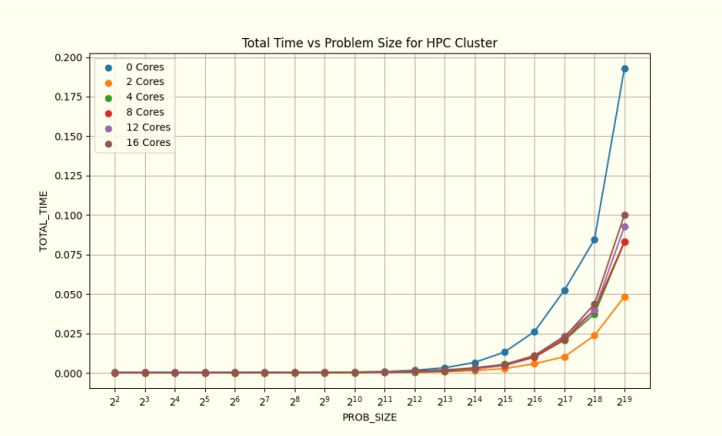

Figure 8: Total mean execution time (Total time) vs Problem size plot for **(Hardware: HPC CLUSTER, Problem: BINARY_SEARCH)**.

3.8.8 Graph of Problem Size vs Algorithm time for HPC CLUSTER

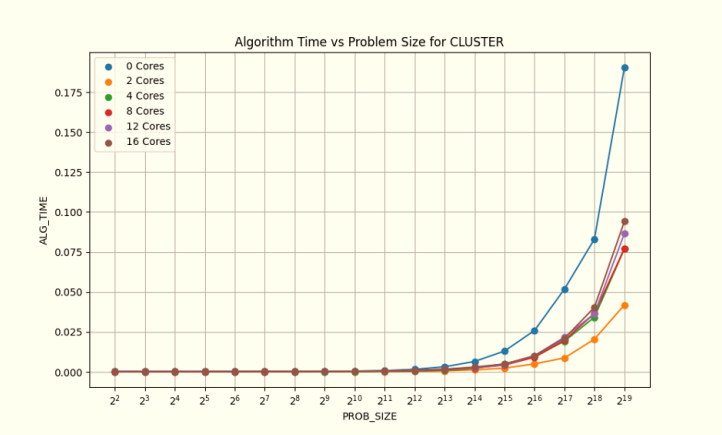

Figure 9: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: HPC CLUSTER, Problem: BINARY_SEARCH)**.

3.8.9 Graph of Efficiency vs Core for HPC CLUSTER

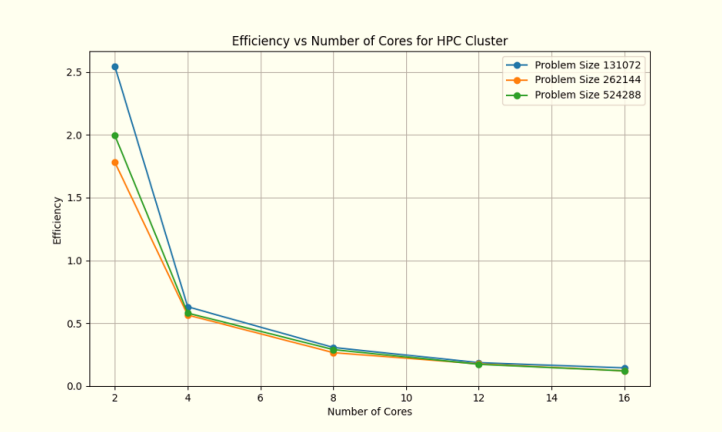

Figure 10: Graph of Efficiency vs Core plot for **(Hardware: HPC CLUSTER, Problem: BINARY_SEARCH)**.

3.8.10 Graph of Efficiency vs Problem size for HPC CLUSTER

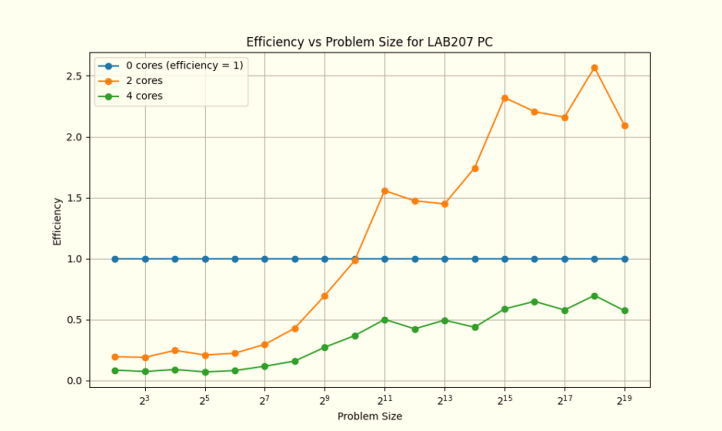

Figure 11: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: HPC CLUSTER, Problem: BINARY_SEARCH)**.

3.8.11 Graph of Speedup vs Core for HPC CLUSTER

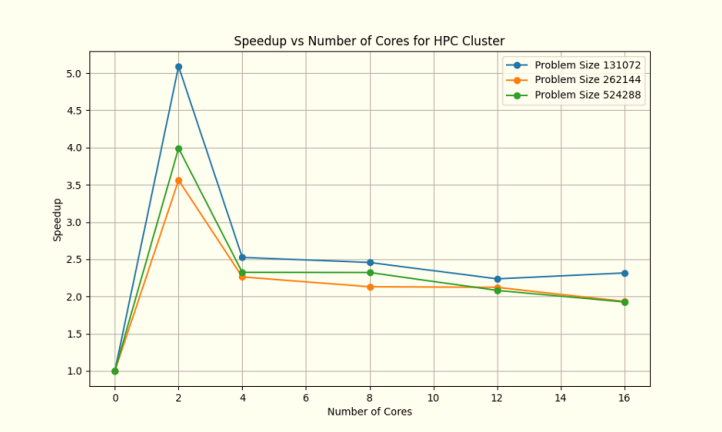

Figure 12: Graph of Speedup vs Core for **(Hardware: HPC CLUSTER, Problem: BINARY_SEARCH)**.

3.8.12 Graph of Speedup vs Problem size for HPC CLUSTER

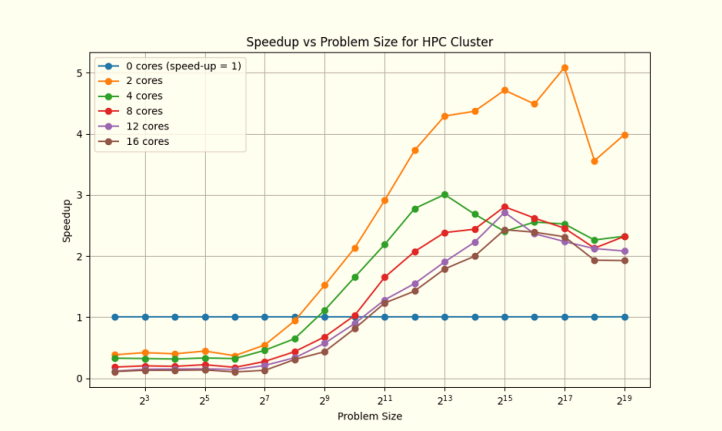

Figure 13: Graph of Speedup vs Problem size for **(Hardware: HPC CLUSTER, Problem: BINARY_SEARCH)**.

4 Conclusions

- To sum up, the parallel binary search algorithm offers a viable method for enhancing search efficiency on big unsorted arrays. Compared to the serial approach, it can find the target value faster by effectively searching through the array in parallel processing. However, the number of threads, array size, and underlying hardware architecture are only a few examples of the variables that could affect how successful the parallel implementation is.