CS 301
High-Performance Computing

Lab 8 - Binary Search and Fast Fourier Transform

Problem 2: Fast Fourier Transform

Jay Dobariya (202101521)
Akshar Panchani (202101522)

April 8, 2024

Contents

1 Introduction

In this lab report,Because of its efficiency over the naive Discrete Fourier Transform (DFT) computation, the Fast Fourier Transform (FFT) technique is essential in signal processing and partial differential equation solving. The Cooley-Tukey radix-2 Decimation in Time (DIT) FFT algorithm's C implementation is the main topic of this study. We will compare it to the naive DFT computation, examine its performance, and talk about how scalable it is with respect to input size.

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

TThe task involves putting the Cooley-Tukey radix-2 DIT FFT technique into C and using it to calculate the DFT of a 2^n complex array. Bit-reversal permutation and butterfly computations are the two primary steps in the implementation. To find out how the FFT implementation performs in relation to the naive DFT computation and how it scales with the size of the input array, it must be examined.

3.1.1 Algorithm description

3.2 Serial Binary Search Algorithm:

1. Bit-Reversal Permutation: Permute the input sequence $x[n]$ by a bit-reversal permutation to yield the sequence $x'[n]$. This step reorders the input sequence such that the index of each sample n is bit-reversed.

2. Butterfly Computations: Process the permuted sequence $x'[n]$ through $\log_2 N$ stages of "butterfly" computations. Each stage operates on pairs of complex numbers from the sequence output by stage $s-1$, where the pairs are separated by 2^{s-1} samples. Each pair is combined by a "butterfly" computation to yield two output samples. Twiddle factors are used in the butterfly computation to introduce necessary phase shifts for the FFT computation.

3.3 The complexity of the algorithm- serial

The time complexity of the Cooley-Tukey radix-2 DIT FFT algorithm is $O(N \log N)$, where N is the size of the input array. This complexity arises from the bit-reversal permutation and the butterfly computations. The parallel complexity of the algorithm remains $O(N \log N)$ as well.

3.4 Information about parallel implementation

Due to their independence for various pairs of samples, the butterfly computations can be parallelized in order to parallelize the FFT algorithm. To guarantee that all butterfly computations for one stage are finished before the start of the subsequent step, precise synchronization is necessary. Dealing with the non-local memory access pattern in the butterfly computations, which can result in cache inefficiency and increased memory latency, is another difficulty when parallelizing the FFT algorithm. The FFT algorithm's parallel complexity is still $O(N \log N)$ in spite of these difficulties.

3.5 The complexity of the algorithm - Parallel

The Cooley-Tukey radix-2 DIT FFT algorithm's parallel complexity similarly stays at $O(N \log N)$. Parallelization can speed up the algorithm's execution time, but it has no effect on the algorithm's underlying asymptotic complexity. The FFT algorithm can be parallelized at each step, but precise synchronization is needed to ensure accuracy. Therefore, the asymptotic complexity does not change even with possible parallel speedup.

3.6 Profiling using HPC Cluster (with gprof)

The screenshots of profiling using the HPC Cluster are given below

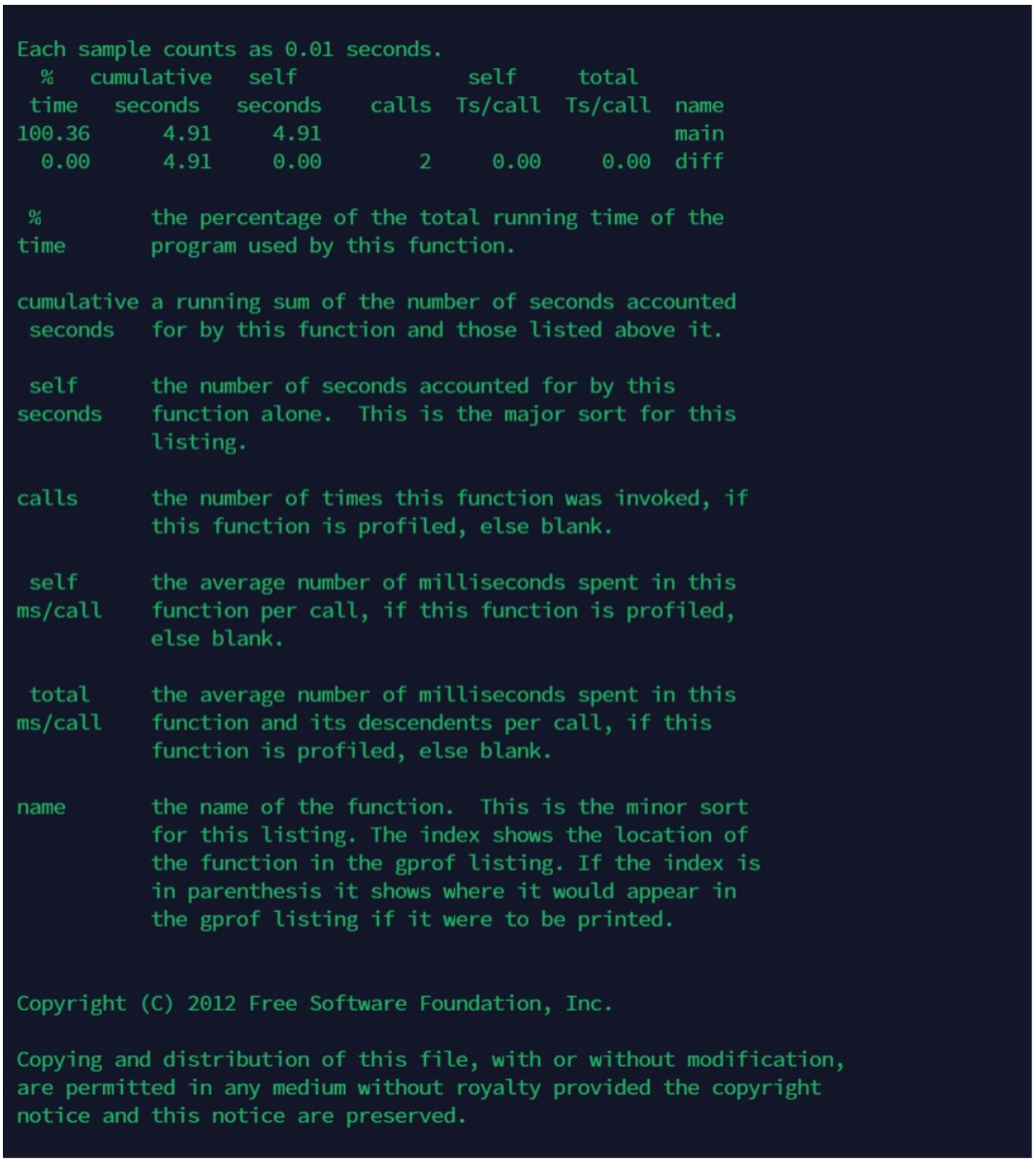

Figure 1: Screenshot of the terminal from HPC Cluster

3.7 Lab207 PC Graph and HPC Cluster Graphs

3.7.1 Graph of Problem Size vs Totaltime for LAB207 PCs

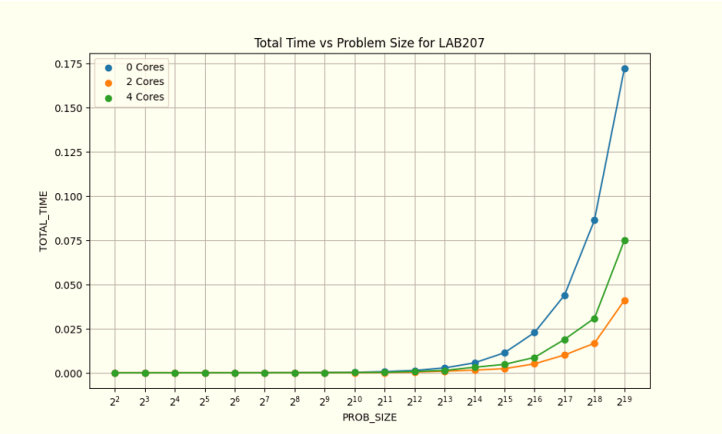

Figure 2: Total mean execution time (Total time) vs Problem size plot for **(Hardware: LAB207 PC, Problem: FFT)**.

3.7.2 Graph of Problem Size vs Algorithm time for LAB207 PCs

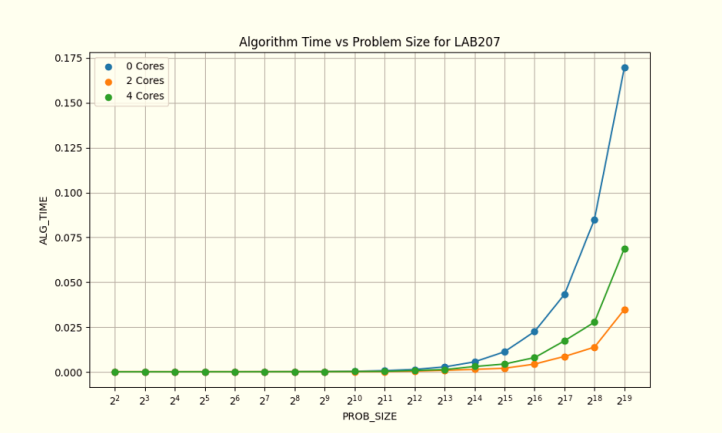

Figure 3: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: LAB207 PC, Problem: FFT)**.

3.7.3 Graph of Efficiency vs Core for LAB207 PCs

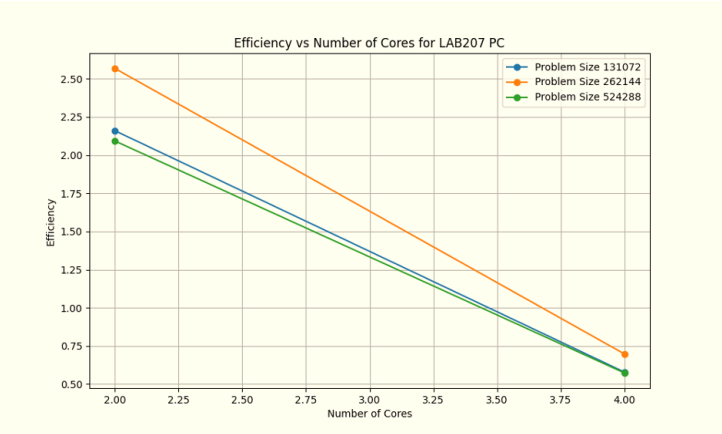

Figure 4: Graph of Efficiency vs Core plot for **(Hardware: LAB207 PC, Problem: FFT)**.

3.7.4 Graph of Efficiency vs Problem size for LAB207 PCs

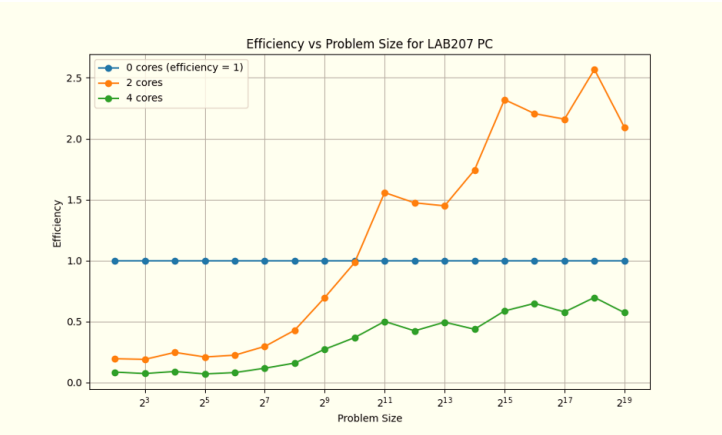

Figure 5: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: LAB207 PC, Problem: FFT)**.

3.7.5 Graph of Speedup vs Core for LAB207 PCs

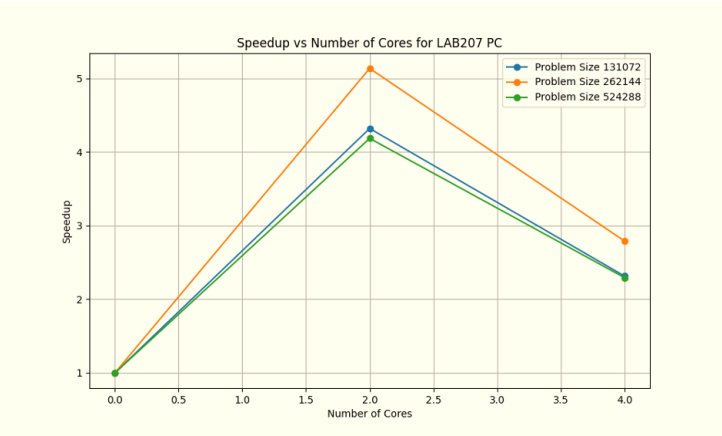

Figure 6: Graph of Speedup vs Core for **(Hardware: LAB207 PC, Problem: FFT)**.

3.7.6 Graph of Speedup vs Problem size for LAB207 PCs

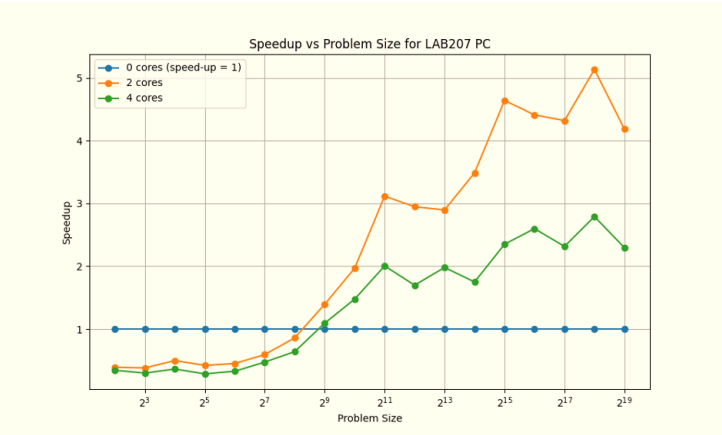

Figure 7: Graph of Speedup vs Problem size for **(Hardware: LAB207 PC, Problem: FFT)**.

3.7.7 Graph of Problem Size vs Totaltime for HPC CLUSTER

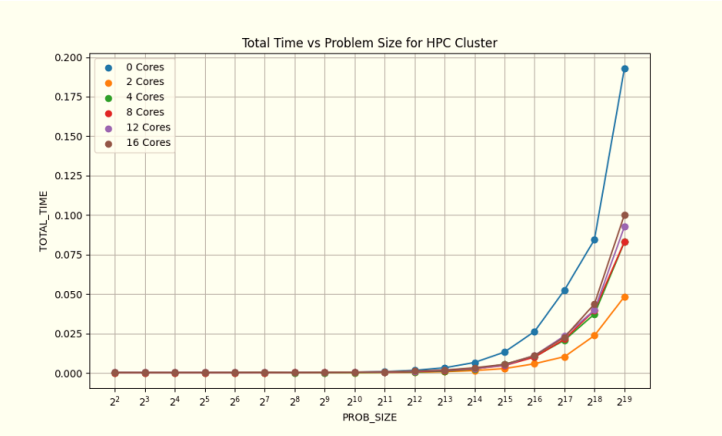

Figure 8: Total mean execution time (Total time) vs Problem size plot for **(Hardware: HPC CLUSTER, Problem: FFT)**.

3.7.8 Graph of Problem Size vs Algorithm time for HPC CLUSTER

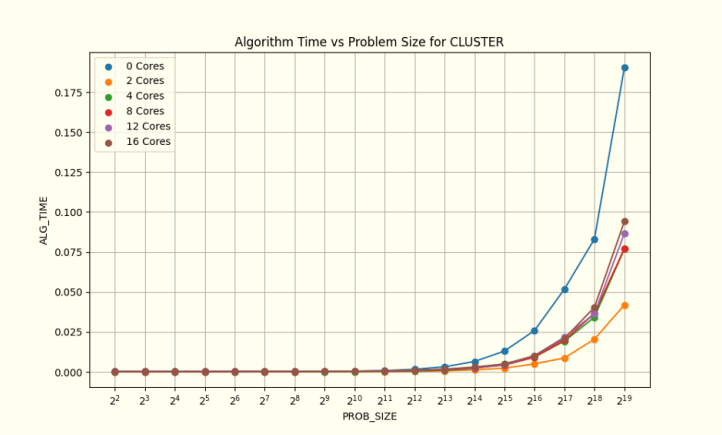

Figure 9: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: HPC CLUSTER, Problem: FFT)**.

3.7.9 Graph of Efficiency vs Core for HPC CLUSTER

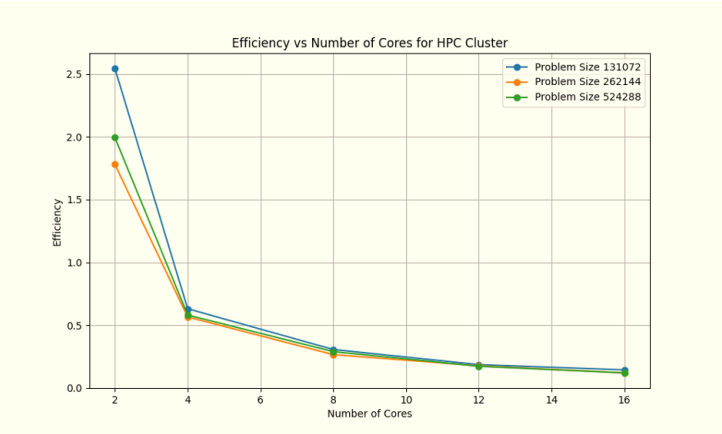

Figure 10: Graph of Efficiency vs Core plot for **(Hardware: HPC CLUSTER, Problem: FFT)**.

3.7.10 Graph of Efficiency vs Problem size for HPC CLUSTER

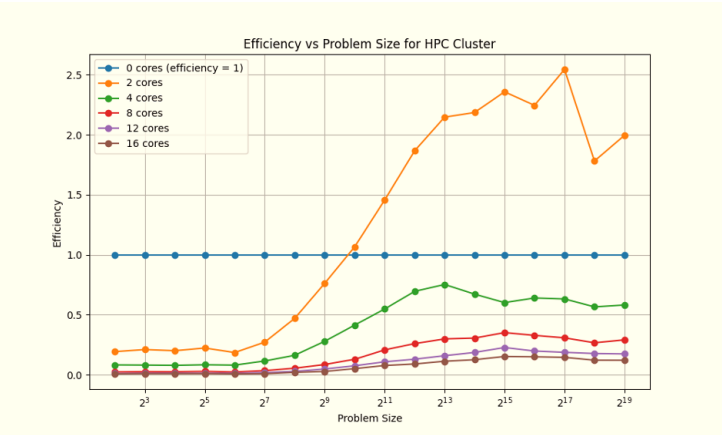

Figure 11: Total mean execution time (Algorithm time) vs Problem size plot for **(Hardware: HPC CLUSTER, Problem: FFT)**.

3.7.11 Graph of Speedup vs Core for HPC CLUSTER

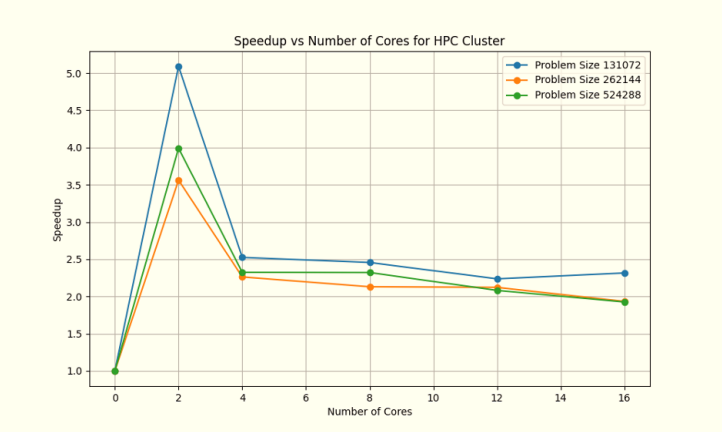

Figure 12: Graph of Speedup vs Core for **(Hardware: HPC CLUSTER, Problem: FFT)**.

3.7.12 Graph of Speedup vs Problem size for HPC CLUSTER

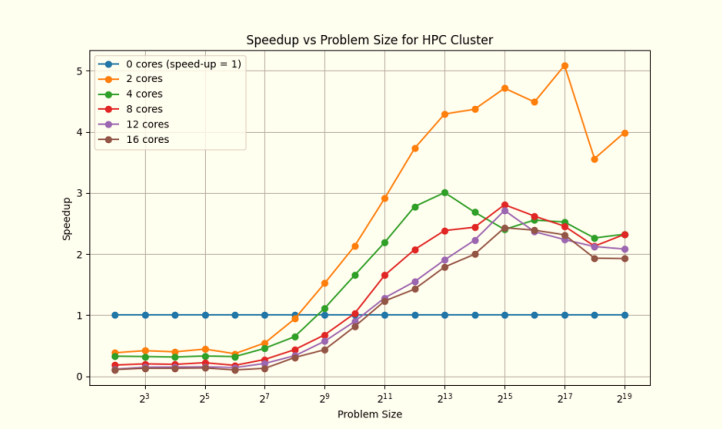

Figure 13: Graph of Speedup vs Problem size for **(Hardware: HPC CLUSTER, Problem: FFT)**.

4 Conclusions

- Use the Cooley-Tukey radix-2 DIT FFT approach to efficiently compute the DFT of a sequence. Its $O(N \log N)$ time complexity makes it significantly faster than the naive DFT computation for large input sizes. Parallelizing the FFT technique can increase its speed even further, although significant gains in speed necessitate resolving synchronization and memory access pattern problems.