Project – 3 Report

Describe in detailed of your system and the problem it is trying to solve.

- **Problem Statement**:
  The Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) are mathematical algorithms used to analyze and manipulate signals in the frequency domain. They are commonly employed in various fields, including signal processing, image processing, audio compression, and communication systems.
  The DFT is a transformation that converts a finite sequence of time-domain samples into its equivalent representation in the frequency domain. It expresses a signal as a sum of complex sinusoidal components, each associated with a specific frequency.
  The DFT is defined as:
  $X[k] = \sum(x[n] * e^{(-j2\pi kn/N)})$
  In this equation, X[k] represents the frequency domain representation of the signal, x[n] denotes the time-domain input sequence, N is the length of the sequence, k is the frequency index (ranging from 0 to N-1), and j is the imaginary unit (√(-1)).
  Conversely, the IDFT is used to reconstruct the original time-domain signal from its frequency domain representation. It transforms the complex sinusoidal components back into the time domain. The IDFT equation is as follows:
  $x[n] = (1/N) * \sum(X[k] * e^{(j2\pi kn/N)})$
  Here, x[n] represents the reconstructed time-domain signal, X[k] denotes the frequency domain representation, N is the length of the sequence, k ranges from 0 to N-1, and j is the imaginary unit.
  The DFT and IDFT are inverse operations of each other, meaning applying the DFT to a time-domain signal and then applying the IDFT to the resulting frequency-domain representation yields the original signal. However, it's important to note that some information can be lost during the transformation due to the discrete nature of the DFT.
  One common problem associated with the DFT and IDFT is the computational complexity. The straightforward implementation of the DFT requires O(N^2) operations, where N is the length of the input sequence. This complexity can be quite high for large sequences, leading to longer processing times.

How is this an embarrassing parallel algorithm ?
The DFT and IDFT involve the computation of individual frequency domain or time domain components separately. Each frequency or time sample can be calculated independently, without relying on the results of other samples. This independence allows for parallel processing, where different processing units can compute different components simultaneously.

Steps to run:

The usage of editor.go is as follows:

```
Usage: editor mode [seqLen] [mode] [algo] [number of threads]
seqLen  = Number of data points used for DFT - IDFT calculation.
mode     = (s) Serial and (p) Parallel
algo     = (balance) Call thread balancing exec and (steal) Call thread stealing executor
number of threads = Runs the parallel version of the program with the specified number of threads.
```

Inside proj3 run editor.go. Examples are as follows:

```
>  go run editor.go 1000 s                        // serial version (1000  num discrete points)
>  go run editor.go 10000 p balance 32        // parallel version (32 threads 10000 discrete data points)
> go run editor.go 100000 p steal 32          // parallel version (32 threads 10000 discrete data points)
```

The script inside proj3 ` benchmark-proj3.sh` runs the serial and parallel versions (balance and steal) for N = 1000, 10000, 100000 and plot the curvers measure for correctness.

Correctness graphs are named as : dft_plot_(N_val)_(algo).png

For speedup graphs:

> plot_speedup.py plots the speedup for all N = 1000, 10000, 100000 from data in file *time.txt*

To run this : python3 plot_speedup.py
File are stored as follows: speedup_plot_(N_val).png

A description of how you implemented your parallel solutions.

For parallelising DFT and IDFT,

- When Executor service is calls `NewWorkBalancingExecutor `,
  The executor service on init spawns threads (num of thread spawned = capcaity). These threads are waiting to consume tasks from the global queue into their own local queues.
- Every task submitted the task into the global queue. Where the threads are waiting for the tasks to be availabe in the queue to PopTop from the global queue and PushBottom into their local queues.
- Based on the flag
  - **If balanced algorithm**: Then once the queue finished its own tasks in its local queue. It runs the loadbalancing , In this algorithm a random size is chosen from [0..currQueueSize], and if the rand_size == currQueueSize. The this queue will load balance. It will choose a random victim and then if abs(victimSize - currQueueSize)

>= thresholdBalance. Then the currQueue will balance between itself and the victim by diff/2.

- **If stealing algorithm**: Then if the queue is Empty, a random victim is chosen from all the local queue (which ranges from [1...capacity]). Then the tasks are taken from the victim queue. And the number of tasks taken = threshold.

Here both DFT and IDFT run independently on their own executor services.

Input Data: The program asks for the number of points (N). Based on that discrete sinusoidal points are created. This data acts as input data to DFT and the result from DFT acts as input to IDFT.

All the implementation is inside dft folder parallel.go is the parallel implementation, and serial.go is the serial implementation editor.go is the interface inside proj3/ to run the algorithms.

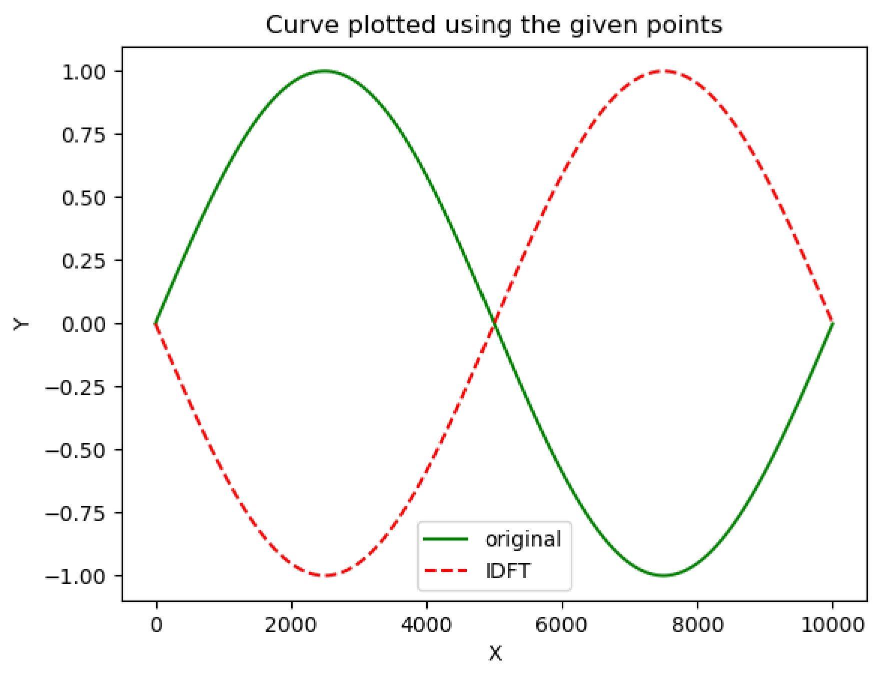
Measure of Correctness:

Expectation:

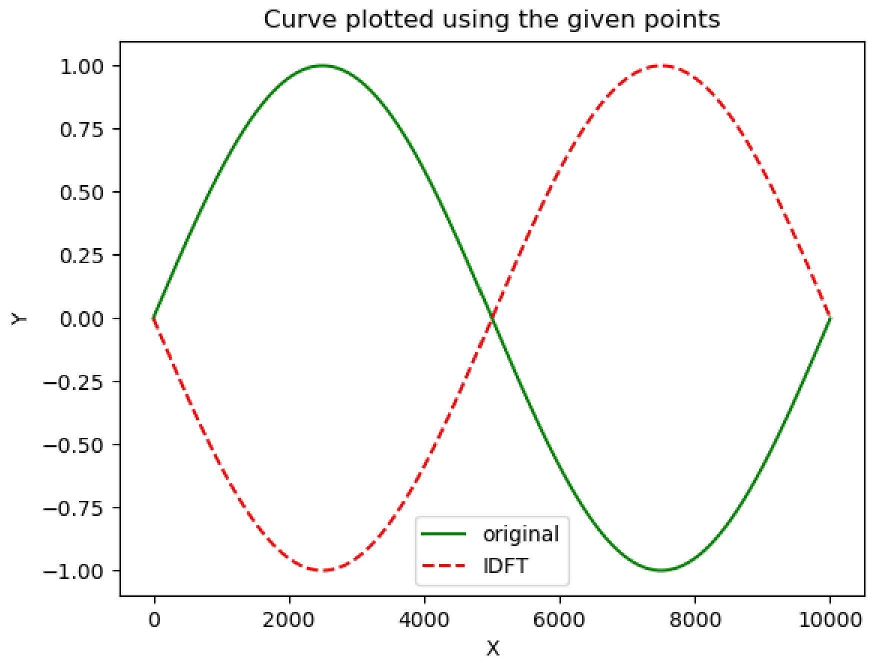
The resultant from IDFT should be inverse of the original data (an extra negative during conversion). The data can be compared from the graphs created by plot.py. The data points are stored in dataDFT.txt and dataIDFT.txt . The script plot.py runs and makes plot from the existing data points.

Reality:

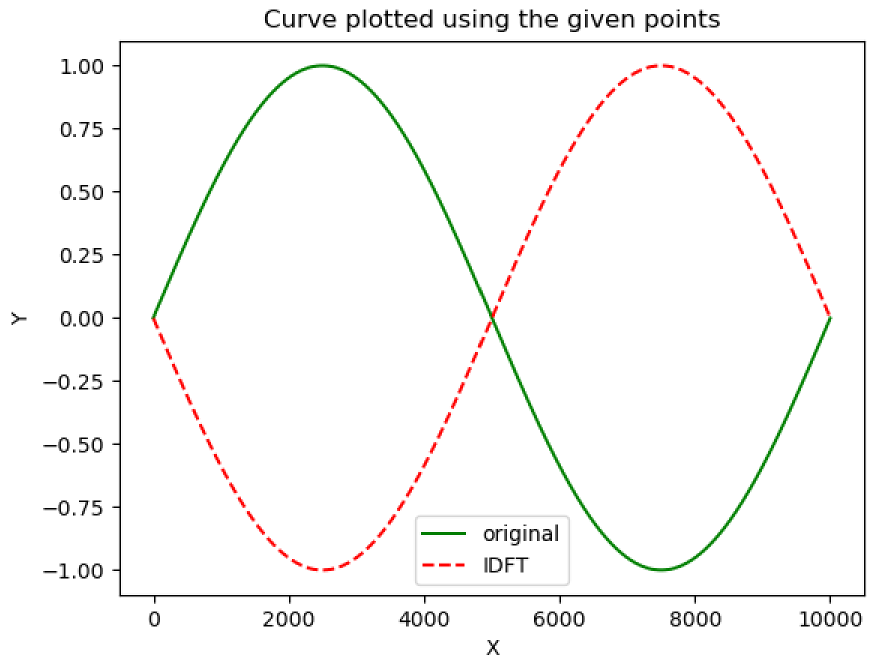
Serial Implementation : N = 10000



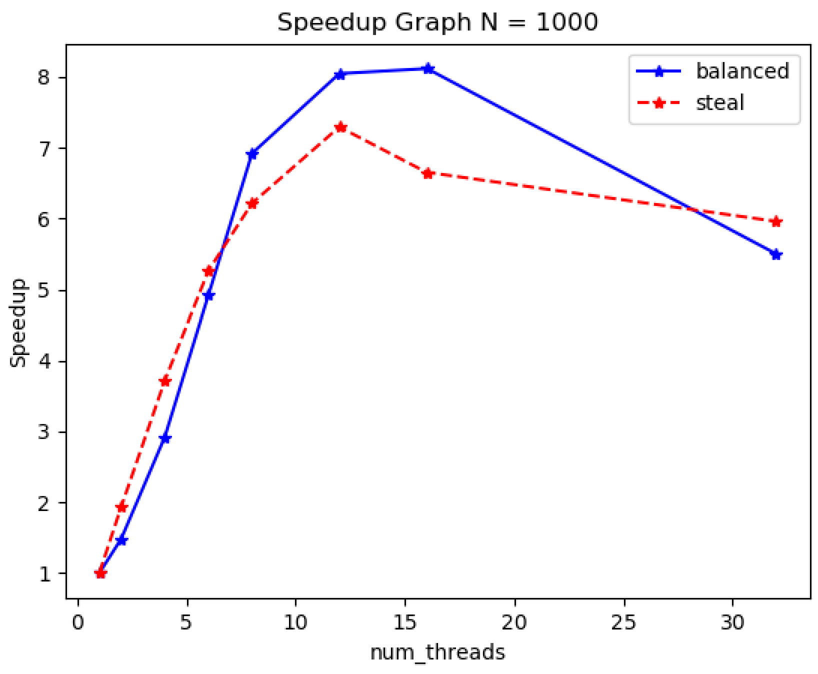
Parallel Implementation Using Stealing Algorithm N = 10000

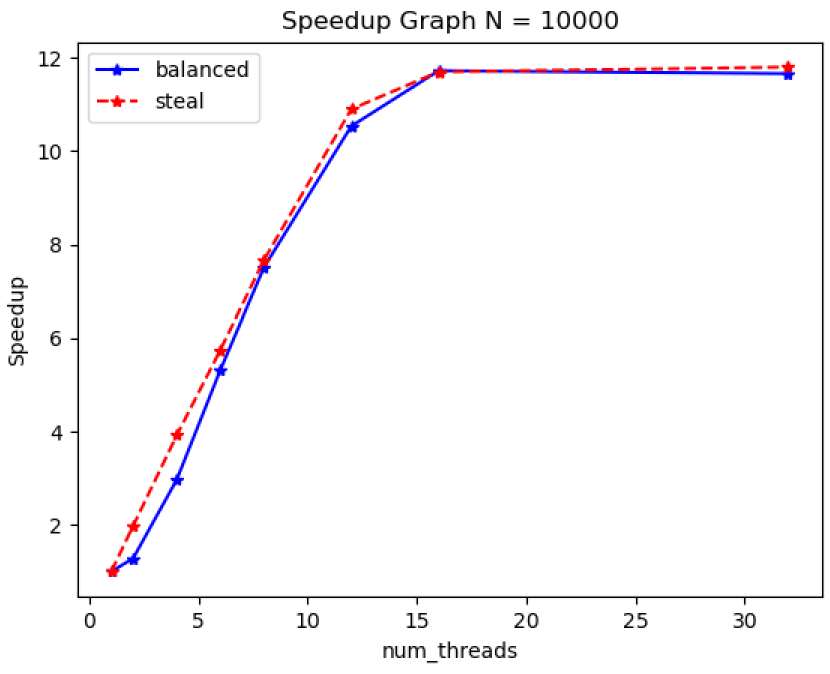


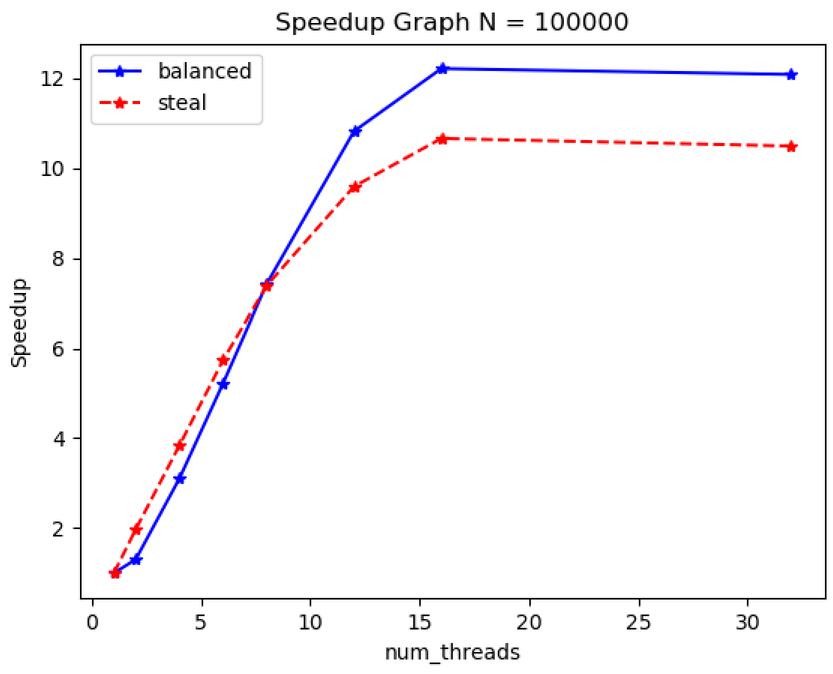
Parallel Implementation Using Balancing Algorithm N = 10000



Speedup Graphs


Speedup Graph N = 1000
balanced
steal
Speedup
num_threads
1
2
3
4
5
6
7
8
0
5
10
15
20
25
30


Speedup Graph N = 10000
balanced
steal
Speedup
num_threads
2
4
6
8
10
12
0
5
10
15
20
25
30



Describe the challenges you faced while implementing the system. What aspects of the system might make it difficult to parallelize? In other words, what to you hope to learn by doing this assignment?

- Implementing Balancing and Stealing algorithms
  The implementation of the above algorithms after understanding the role of runnable and callable was difficult but very interesting.
- After implementing the basic structure of balancing and stealing algorithms I played around with the threshold and thresoldBalance parameters to get the maximum speedup. I achieved maximum speedup when setting

  ThresholdBalance=(N/threadCount) * 10

  Threshold=threadCount

- Deciding on the input data and how to demonstrate correctness
  The input data must be discrete points after trying different functions and discrete data points I decided to go for sinusoidal implementation.

Specifications of the testing machine you ran your experiments on (i.e. Core Architecture (Intel/AMD), Number of cores, operating system, memory amount, etc.)

Linux peanut cluster

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit

Byte Order:	Little Endian
Address sizes:	46 bits physical, 48 bits virtual
CPU(s):	64
On-line CPU(s) list:	0-63
Thread(s) per core:	2
Core(s) per socket:	6
Model name:	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz
Stepping:	7
CPU MHz:	800.013

What are the hotspots (i.e., places where you can parallelize the algorithm) and bottlenecks (i.e., places where there is sequential code that cannot be parallelized) in your sequential program? Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?

Hotspots : Each frequency or time sample can be calculated independently, without relying on the results of other samples. This independence allows for parallel processing, where different processing units can compute different components simultaneously.

Bottlenecks: Here I am waiting for dft to finish generating the data and only then idft starts computation.

What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchronization overhead? As you try and answer these questions, we strongly prefer that you provide data and measurements to support your conclusions.

Limited speedup due to

- For this implementation I used lock queues rather than lock free queues. The queue implementation I wrote is dependent on coarse grained technique, this is one of the limitations t my parallelism.
- Since every task is pushed into the global queue and then the threads pull into their own local queue. This overhead is one of the reasons that limit the speedup. In other case if I randomly choose a thread and push the task into its local queue this overhead could totally be eliminated.
  When I implemented this technique the speedup achieved was:
  For (N = 100000)
  - balancing max speedup = 13.32
  - Stealing max speedup = 14.12

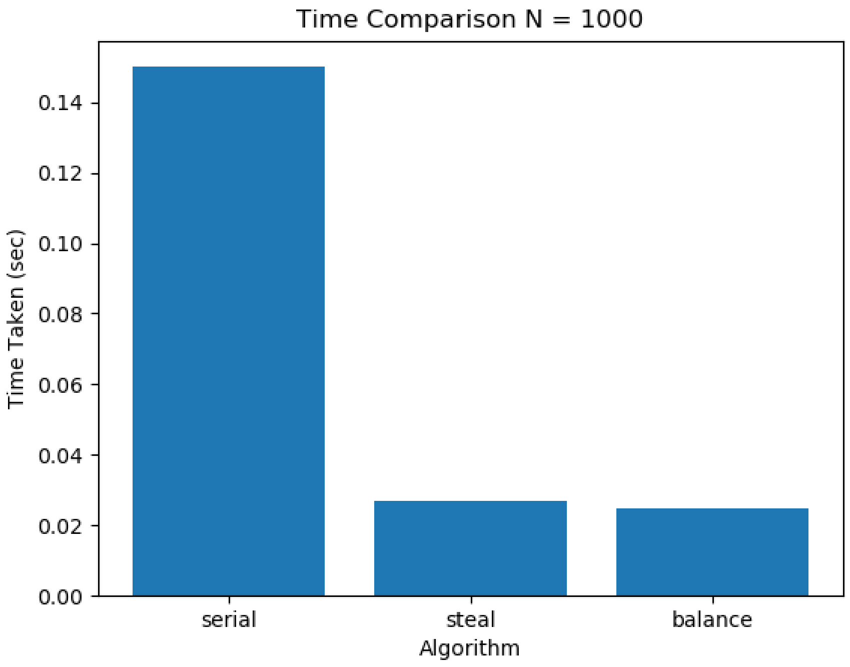
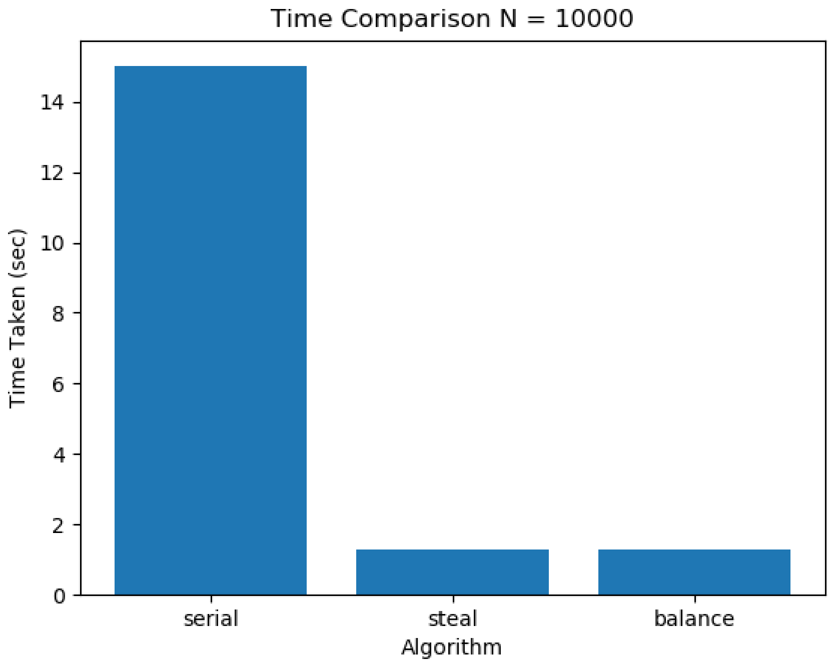
Compare and contrast the two parallel implementations. Are there differences in their speedups?

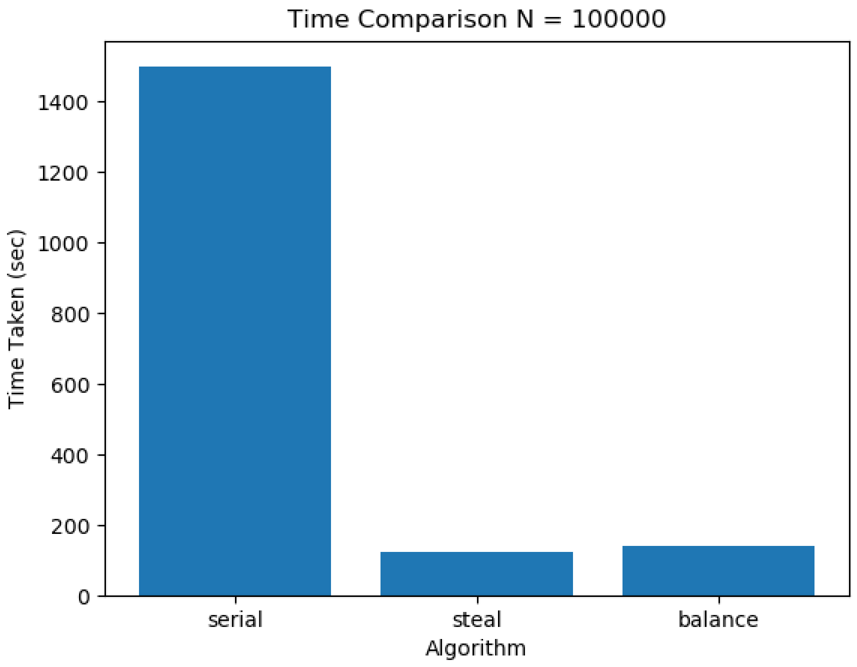
Compairison:

- **Balancing algorithm**: Then once the queue finished its own tasks in its local queue. It runs the loadbalancing , In this algorithm a random size is chosen from [0..currQueueSize], and if the rand_size == currQueueSize. The this queue will load balance. It will choose a random victim and then if abs(victimSize - currQueueSize) >= thresholdBalance. Then the currQueue will balance between itself and the victim by diff/2.
- **Stealing algorithm**: Then if the queue is Empty, a random victim is chosen from all the local queue (which ranges from [1...capacity]). Then the tasks are taken from the victim queue. And the number of tasks taken = threshold.

Both stealing and balancing achieved similar speedups for lower number of threads, but as the number of threads is increased the stealing performs a little faster.
This could be due to the following reason:

Load Imbalance: The DFT computation involves different frequency components, and the workload can vary depending on the characteristics of the input signal. The balancing algorithm aims to distribute the workload evenly among processing units. However, if there is a significant imbalance in the workload, some processing units may finish their tasks much earlier than others, leading to idle time. In contrast, the stealing algorithm allows processing units to dynamically steal tasks from others, which can help mitigate load imbalance and better utilize available resources.


Time Comparison N = 1000
Time Taken (sec)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
serial
steal
balance
Algorithm

Time Comparison N = 10000
Time Taken (sec)
0
2
4
6
8
10
12
14
serial
steal
balance
Algorithm



Time Comparison N = 100000
Time Taken (sec)
0
200
400
600
800
1000
1200
1400
serial
steal
balance
Algorithm