**Digital Assignment 2 – Report**

|  |  |
| --- | --- |
| Course Code and Name | BCSE205L&Computer Architecture and Organization |
| Slot | E1 |
| Register Number | 23BRS1064 |
| Name of the Student | Nahul Alaguraj |
| Title of the Paper | A Comparative Study of Serial and Parallel Processing for Fast Fourier Transform (FFT) with parallel python |

**Abstract**

This report presents an in-depth comparative analysis of Fast Fourier Transform (FFT) performance using serial and parallel processing techniques. By evaluating computation time, speedup, and efficiency across varying waveform counts and processor cores, we aim to assess the suitability of parallel computing in FFT applications. We also explore the impact of amplitude distribution on performance. Despite the expectation that parallel processing would enhance computational efficiency, our findings reveal performance bottlenecks and inefficiencies attributed to parallel overheads.

**A Comparative Study of Serial and Parallel Processing for Fast Fourier Transform (FFT)**

**1. Introduction**

Fast Fourier Transform (FFT) is a widely used algorithm for converting time-domain signals into their frequency-domain representation. It plays a crucial role in signal processing, communications, image processing, and many scientific computations. As the demand for real-time data processing increases, optimizing the performance of FFT operations becomes essential.

With the emergence of multicore systems, parallel computing has become an attractive option for accelerating computationally intensive tasks. However, parallelization can also introduce overheads such as data partitioning, synchronization, and inter-process communication, which may offset the benefits. This report investigates these aspects through a comparative study between serial and parallel FFT implementations.

**2. Experimental Setup**

**2.1 System Configuration**

* **Processor:** Intel Core i7 13th Gen (12 cores)
* **RAM:** 8 GB
* **Python Version:** 3.12
* **Libraries Used:** NumPy, Matplotlib, multiprocessing

**2.2 Signal Generation**

Synthetic signals were generated using a summation of sine waves. The number of sine wave components ("waveforms") was varied to simulate complexity. The signal had added Gaussian noise to mimic real-world data.

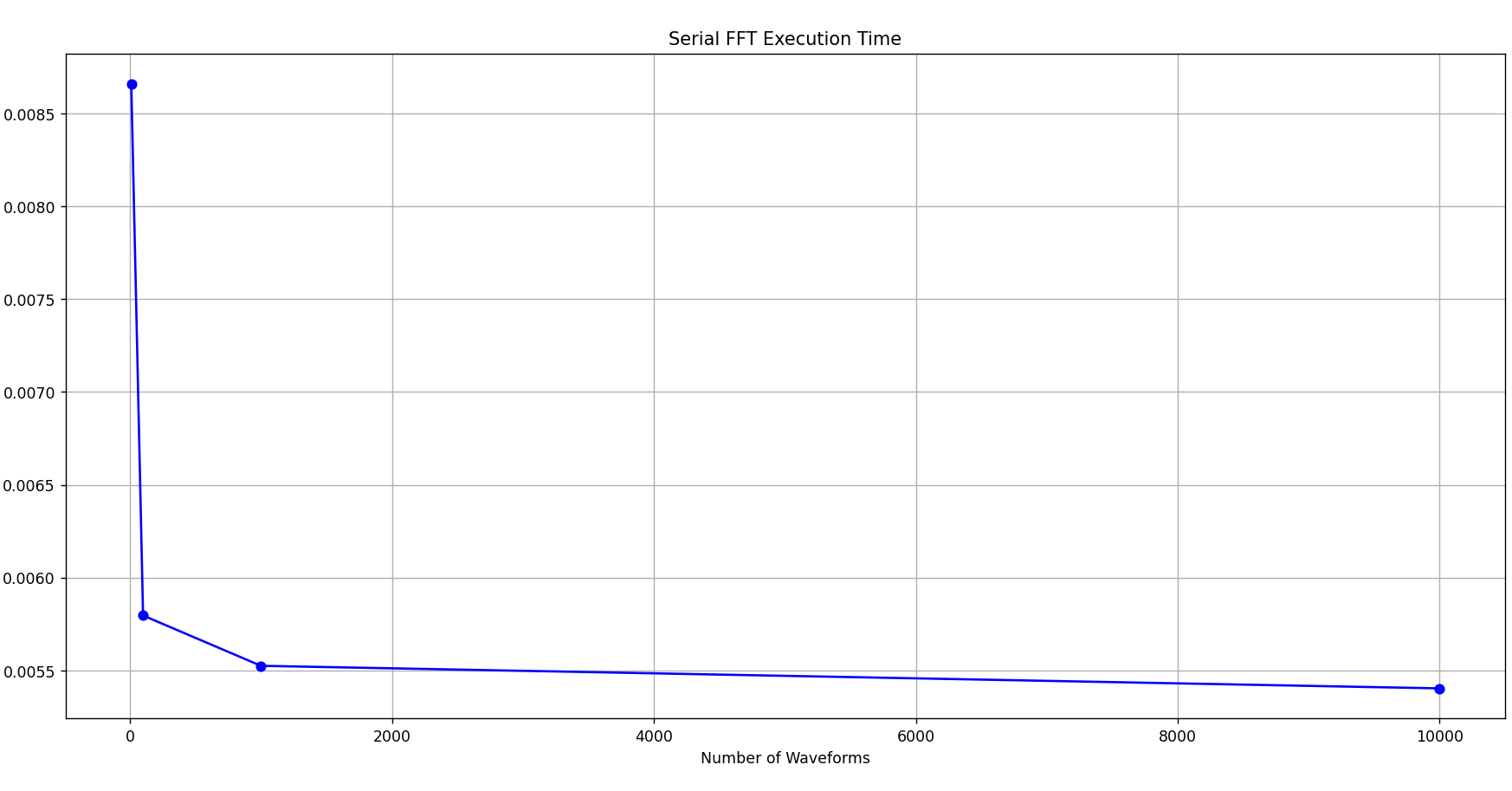
**2.3 Performance Metrics**

* **Computation Time (s):** Time taken for FFT computation
* **Speedup:** Ratio of serial time to parallel time
* **Efficiency (%):** Speedup divided by the number of cores

**3. Serial Processing Performance Analysis**

**3.1 Time vs Number of Waveforms**

| **Number of Waveforms** | **Time (s)** |
| --- | --- |
| 10 | 0.0086 |
| 100 | 0.0057 |
| 1000 | 0.0055 |
| 10000 | 0.0054 |



Despite the increasing waveform count, computation time decreased slightly. This can be attributed to NumPy's internal optimizations and efficient vectorization for large arrays.

**3.2 Time vs Amplitude Type (Fixed Waveform Count)**

| **Waveform Count** | **Equal Amplitude** | **Decreasing Amplitude** | **Random Amplitude** |
| --- | --- | --- | --- |
| 10 | 0.0078 s | 0.0070 s | 0.0081 s |
| 100 | 0.0069 s | 0.0078 s | 0.0086 s |
| 1000 | 0.0068 s | 0.0078 s | 0.0078 s |

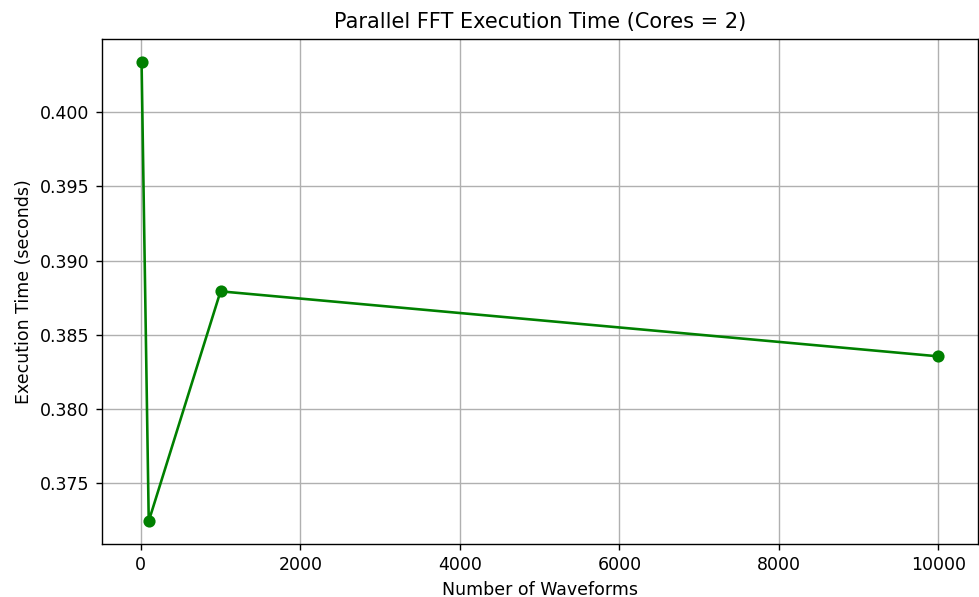
These variations are marginal, indicating that amplitude distribution has a minor effect on FFT performance in serial execution.

**4. Parallel Processing Performance Analysis**

**4.1 Time vs Number of Waveforms (Fixed Cores)**

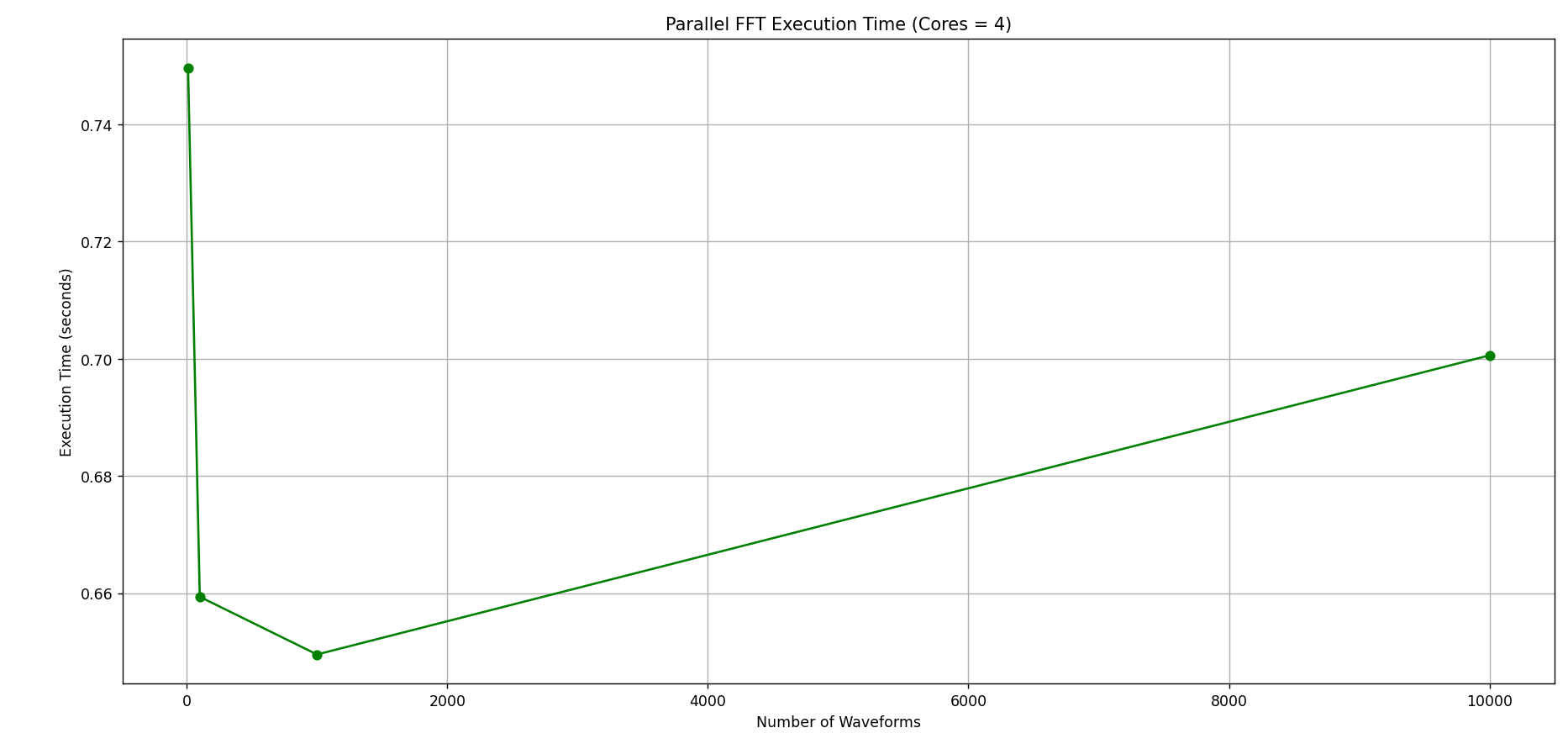
**2 Cores**

| **Number of Waveforms** | **Time (s)** |
| --- | --- |
| 10 | 0.4034 |
| 100 | 0.3724 |
| 1000 | 0.3789 |
| 10000 | 0.3835 |



**4 Cores**

| **Number of Waveforms** | **Time (s)** |
| --- | --- |
| 10 | 0.7496 |
| 100 | 0.6593 |
| 1000 | 0.6495 |
| 10000 | 0.7005 |



**4.2 Analysis**

Contrary to expectations, parallel execution showed significantly **higher computation time** than serial execution. Key factors:

* **Parallel Overhead:** Chunk splitting, inter-process communication, and memory duplication.
* **Small Task Size:** Each chunk of data might be too small to benefit from parallelism.
* **Inefficient Core Utilization:** Core idle times increase due to synchronization.

**5. Speedup and Efficiency**

**5.1 Speedup**

Speedup = Serial Time / Parallel Time

| **Waveforms** | **2 Cores Speedup** | **4 Cores Speedup** |
| --- | --- | --- |
| 10 | 0.0213 | 0.0115 |
| 100 | 0.0153 | 0.0086 |
| 1000 | 0.0145 | 0.0085 |
| 10000 | 0.0141 | 0.0077 |

**5.2 Efficiency**

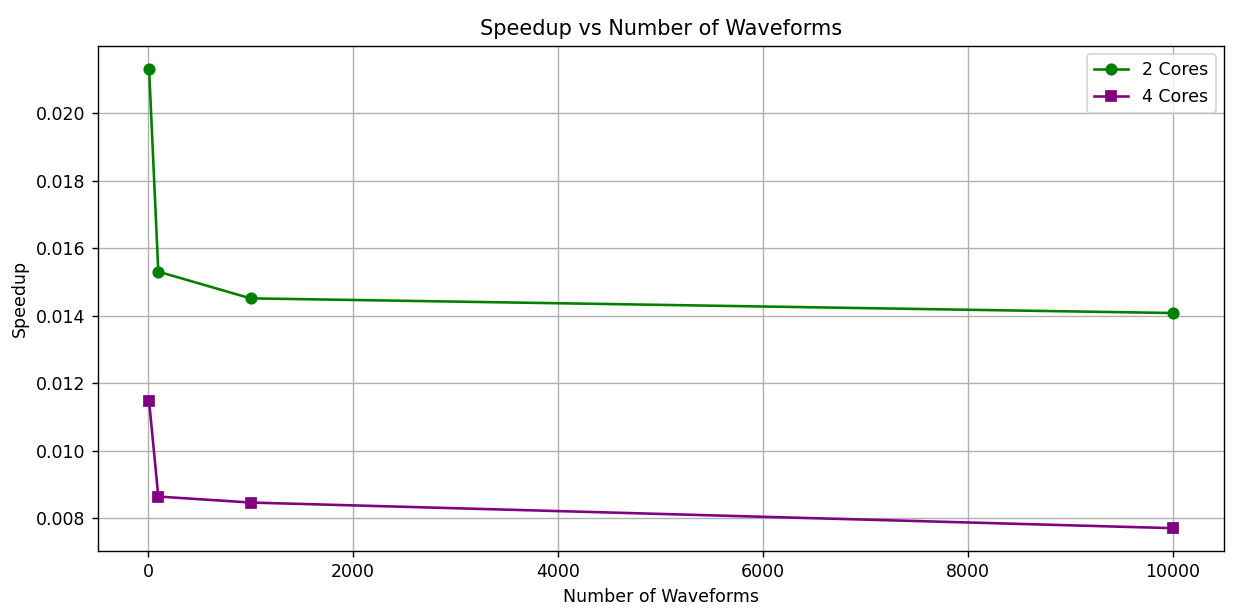
Efficiency = (Speedup / No. of Cores) \* 100

| **Waveforms** | **2 Cores (%)** | **4 Cores (%)** |
| --- | --- | --- |
| 10 | 1.06 | 0.29 |
| 100 | 0.77 | 0.21 |
| 1000 | 0.72 | 0.21 |
| 10000 | 0.70 | 0.19 |

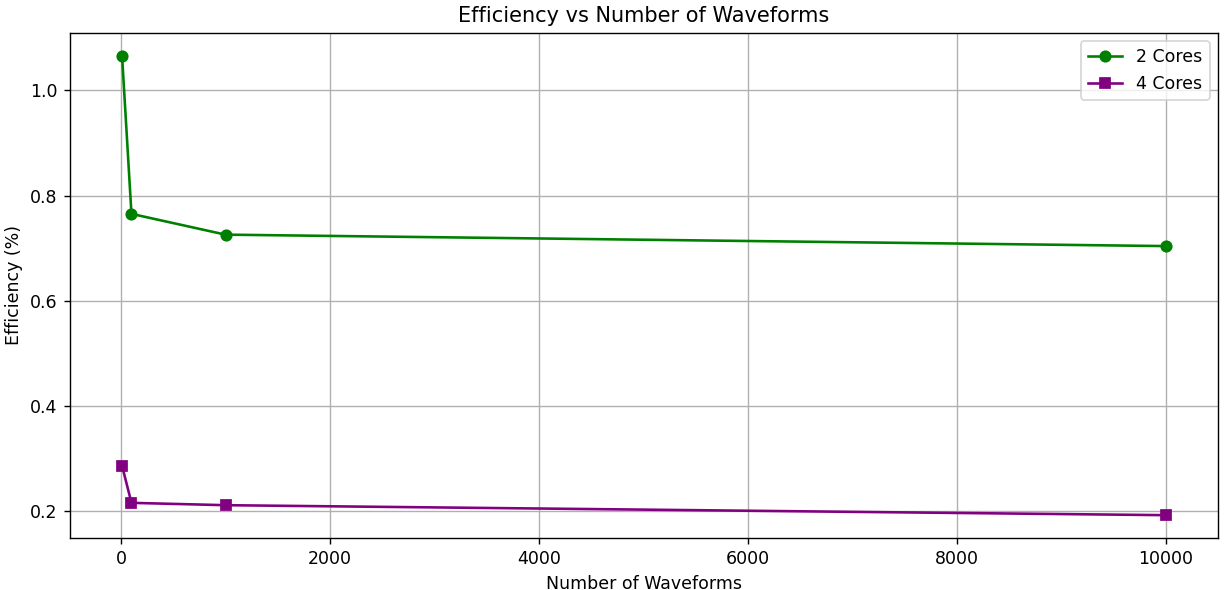
These values demonstrate that parallelism was highly inefficient for this task due to overhead and task granularity.

**6. Visual Analysis**

**6.1 Speedup vs Number of Waveforms**



**6.2 Efficiency vs Number of Waveforms**



Graphs further highlight how scaling with more cores fails to achieve improved performance and even regresses.

**7. Discussion**

**7.1 Why Serial Outperformed Parallel**

The extraordinary performance gap between serial and parallel implementations demands deeper analysis. Several interrelated factors contribute to this counterintuitive result:

**7.1.1 Python's Multiprocessing Architecture**

Python's multiprocessing module uses a process-based parallelism approach rather than thread-based parallelism. This design choice has profound implications:

* **Process Creation Overhead:** Each worker process requires substantial initialization time, including Python interpreter startup, module loading, and memory allocation. For FFT operations completing in milliseconds, this overhead becomes the dominant time factor.
* **Data Serialization and Copying:** The multiprocessing architecture necessitates complete data copying between processes rather than shared memory access. For our signal data, this means:
  + Copying the input signal to each worker process
  + Serializing and deserializing data between processes
  + Copying results back to the main process

This creates both time and memory overhead proportional to data size and number of cores.

* **Global Interpreter Lock (GIL) Effects:** While NumPy operations release the GIL during computation, the Python code managing the multiprocessing pool still encounters GIL contention during task distribution and result collection.

**7.1.2 FFT Algorithm Characteristics**

The FFT algorithm has unique properties that make it particularly challenging to parallelize effectively:

* **Butterfly Operations and Data Dependencies:** The FFT algorithm involves "butterfly" operations that create complex data dependencies between computation stages. Efficient parallelization requires careful consideration of these dependencies, which the simplistic chunk-based approach doesn't address.
* **Memory Access Patterns:** FFT computation involves non-linear memory access patterns that don't align well with chunk-based division. This creates cache inefficiencies when parallelized naively.
* **Algorithm-to-Hardware Ratio:** Modern CPUs already implement SIMD (Single Instruction, Multiple Data) vectorization that NumPy leverages internally. This means the serial implementation already utilizes parallel hardware capabilities at a lower level, making higher-level parallelism redundant or even counterproductive.

**7.1.3 NumPy's Optimized Implementation**

NumPy's FFT implementation is highly optimized and outperforms naive parallelization attempts:

* **FFTPACK/FFTW Integration:** NumPy's FFT functions are typically backed by highly optimized C implementations from libraries like FFTPACK or FFTW, which already incorporate low-level parallelism and cache optimizations.
* **Vectorization:** NumPy operations leverage CPU vector instructions (SSE, AVX) for parallel processing at the instruction level, achieving parallelism without the overhead of multiple processes.
* **Memory Layout Optimization:** NumPy's internal memory layout is optimized for numerical operations, providing better cache utilization than data split across processes.

**7.1.4 Scalability and Core Utilization Issues**

The data reveals declining efficiency with increasing core count, indicating fundamental scalability issues:

* **Amdahl's Law in Action:** The portion of FFT computation that benefits from parallelization is limited by the algorithm's inherent sequential components and communication needs.
* **Synchronization Bottlenecks:** More cores require more complex synchronization, leading to increased idle time as processes wait for others to complete.
* **Resource Contention:** As core count increases, processes compete for shared resources like memory bandwidth and cache, further degrading performance.

**7.2 The Overhead-to-Computation Ratio Problem**

At the heart of our findings is a critical issue of proportionality: the ratio between parallel overhead and actual computation time. In our experiments:

* **Serial FFT Computation:** Completed in ~0.005-0.009 seconds
* **Parallel Overhead:** Ranged from ~0.37 seconds (2 cores) to ~0.74 seconds (4 cores)

This represents an overhead-to-computation ratio of approximately 75:1 for 2 cores and 150:1 for 4 cores. With such extreme ratios, parallelization cannot possibly yield performance improvements.

The data shows this overhead remains relatively constant regardless of waveform count, suggesting it's dominated by process management costs rather than data-dependent factors. This explains why increasing core count actually worsens performance—each additional core adds overhead without proportional computational benefit.

**7.3 When Parallelism Can Help**

Despite our findings, parallel FFT processing can be beneficial under specific circumstances:

* **Very Large Signals:** When signal sizes reach millions of data points, computation time begins to outweigh overhead, potentially making parallelism worthwhile.
* **Multiple Independent FFTs:** Batch processing numerous independent signals can benefit from parallelism by assigning complete signals to different cores, avoiding the split-combine overhead.
* **Lower-Level Implementation:** Parallelization at C/C++/Fortran level rather than Python level significantly reduces overhead while maintaining computational benefits.
* **Alternative Parallelization Approaches:** Using shared memory paradigms, GPU acceleration (via libraries like CuPy), or specialized FFT libraries designed for parallel execution.
* **Heterogeneous Computing:** Distributing workload across different computing resources (CPU + GPU) can overcome the limitations of CPU-only parallel processing.

**8. Conclusion**

This study reveals that for FFT computation on moderately large signals, serial processing with NumPy is dramatically more efficient than multiprocessing in Python. The parallel processing approach incurs substantial overhead costs that overwhelm any potential computational speedup gains, leading to performance degradation rather than improvement. This phenomenon becomes more pronounced as the number of cores increases.

The extraordinarily low efficiency values (below 1.1% in all test cases) demonstrate that Python's multiprocessing is fundamentally unsuitable for accelerating FFT operations of the scale tested in this study. The results strongly suggest that optimizing serial execution or exploring alternative parallel computing paradigms would be more productive approaches for enhancing FFT performance.

Our findings challenge the common assumption that "more cores equal better performance" and highlight the importance of matching parallelization strategies to algorithm characteristics and implementation environments. They also demonstrate how crucial it is to measure and analyse actual performance rather than relying on theoretical expectations when optimizing computational tasks.

**References**

* Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series.
* NumPy FFT Documentation: https://numpy.org/doc/stable/reference/routines.fft.html
* Parallel Computing with Python (multiprocessing, joblib)