Practical Work 14
Signal Processing Using Parallel Processing Technologies

1. The Objectives of this Laboratory Work

• Understand the principles of parallel processing in digital signal processing (DSP)
• Learn how parallel computation improves speed and efficiency in signal analysis
• Implement parallel processing techniques for large-scale signal datasets
• Compare sequential and parallel performance using Python tools such as NumPy and multiprocessing

2. Theoretical Background

Signal processing often requires intensive mathematical computations such as convolution, Fourier transforms, filtering, and feature extraction. As data sizes grow, especially in multimedia, biomedical, and real-time systems, these operations become computationally expensive when performed sequentially.
Parallel processing technologies address this challenge by dividing tasks into smaller units that can be executed simultaneously across multiple processors or cores.

Parallel Processing Overview

Parallel processing is the simultaneous use of multiple compute resources to solve a computational problem faster. Instead of executing instructions one after another (sequentially), the task is divided into independent sub-tasks that can be processed concurrently. These sub-tasks can run on:

- Multiple CPU cores (shared-memory systems)
- Multiple processors (distributed-memory systems)
- GPU cores (graphics processing units)

This approach allows large-scale signal and image operations to complete faster and with improved performance scalability.

Relevance in Signal Processing

Many signal processing operations can be parallelized because they involve repetitive calculations on independent data samples. Examples include:

- **Convolution and correlation**
- **Fourier Transform (FFT)**
- **Filtering and denoising**
- **Feature extraction and compression**
- **Matrix operations** in digital image or sound analysis

Parallel execution enables **real-time** and **high-resolution** data processing in systems such as radar, medical imaging, and speech recognition.

Parallel Computing Models

1. **Data Parallelism:** The same operation is applied simultaneously to different data elements (e.g., applying a filter to multiple pixels or samples at once).
2. **Task Parallelism:** Different tasks are executed simultaneously (e.g., filtering one signal while computing FFT on another).
3. **Pipeline Parallelism:** Operations are divided into stages, and each stage is processed concurrently on different data streams.

Python Tools for Parallel Signal Processing

Python provides several ways to perform parallel computation:

- **NumPy + Vectorization:** Efficient low-level C operations for parallelized math.
- **Multiprocessing:** Runs independent processes on multiple CPU cores.
- **Threading:** Useful for I/O-bound signal tasks.
- **Joblib / Dask:** High-level parallel frameworks for large datasets.
- **GPU Libraries (CuPy, PyTorch, TensorFlow):** Perform parallel computations using GPU cores for massive acceleration.

Advantages of Parallel Processing

Advantages	Description
Faster Execution	Reduces overall processing time
Scalability	Can handle large datasets efficiently
Real-Time Performance	Enables real-time analysis of signals
Efficient Resource Utilization	Makes full use of multicore and GPU resources

Challenges

Challenge	Description
Synchronization	Managing data dependencies between threads
Overhead	Time cost for data splitting and merging
Load Balancing	Equal distribution of tasks among cores
Hardware Limitations	Performance depends on CPU/GPU availability

Applications

Parallel processing in signal processing is widely used in:

- **Medical Imaging (MRI, CT):** Real-time reconstruction and denoising
- **Speech and Audio Processing:** Fast feature extraction and recognition
- **Radar and Sonar Systems:** Real-time target detection
- **Telecommunications:** Fast encoding/decoding of high-bandwidth data
- **AI and Machine Learning:** Training deep neural networks on signal data

3. Practical Section: Implementing Parallel Processing for Signal Operations

Task 1: Sequential vs Parallel Filtering

Objective: Compare the performance of sequential and parallel execution in signal filtering.

Instructions:

1. Generate a large one-dimensional signal (e.g., 10^6 samples).
2. Apply a moving average filter sequentially.
3. Split the signal into chunks and process each in parallel using multiprocessing.

4. Compare execution times.

Code Example:

```
import numpy as np
import multiprocessing as mp
import time
import matplotlib.pyplot as plt

# Create synthetic signal
signal = np.random.randn(10**6)
kernel = np.ones(5) / 5

# Plot original signal
plt.figure(figsize=(10,4))
plt.plot(signal)
plt.title("Original Signal")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.tight_layout()
plt.show()

# Sequential filtering
def moving_average(sig, kernel):
    return np.convolve(sig, kernel, mode='same')

start_seq = time.time()
filtered_seq = moving_average(signal, kernel)
end_seq = time.time()

# Parallel filtering
def parallel_filter(chunk):
    return np.convolve(chunk, kernel, mode='same')

num_processes = mp.cpu_count()
chunks = np.array_split(signal, num_processes)

start_par = time.time()
with mp.Pool(num_processes) as pool:
    results = pool.map(parallel_filter, chunks)
filtered_par = np.concatenate(results)
end_par = time.time()

print(f"Sequential Time: {end_seq - start_seq:.3f}s")
print(f"Parallel Time: {end_par - start_par:.3f}s")
```

Task 2: Parallel FFT Computation

Objective: Perform Fourier Transform on multiple signals in parallel.

Instructions:

1. Generate multiple random signals.
2. Compute FFT sequentially and then in parallel.
3. Compare computation times.

Code Example:

```
import numpy as np
import multiprocessing as mp
import time
import matplotlib.pyplot as plt
from numpy.fft import fft

signals = [np.random.randn(2**16) for _ in range(8)]

def compute_fft(sig):
    return np.abs(fft(sig))

# Sequential
start_seq = time.time()
results_seq = [compute_fft(s) for s in signals]
end_seq = time.time()

# Parallel
start_par = time.time()
with mp.Pool(num_processes) as pool:
    results_par = pool.map(compute_fft, signals)
end_par = time.time()

print(f"Sequential FFT Time: {end_seq - start_seq:.3f}s")
print(f"Parallel FFT Time: {end_par - start_par:.3f}s")
```

Task 3: GPU-based Parallel Signal Operation

Objective: Accelerate signal filtering using GPU computation (if available).

Instructions:

1. Install and import CuPy (!pip install cupy-cuda11x)
2. Rewrite the filtering operation using GPU arrays.
3. Compare CPU vs GPU execution times.

Code Example:

```
import numpy as np
import time

# Create synthetic signal
signal = np.random.randn(10**6)
kernel = np.ones(5) / 5

# CPU sequential filtering
start_cpu = time.time()
filtered_cpu = np.convolve(signal, kernel, mode='same')
end_cpu = time.time()
print(f"CPU Execution Time: {end_cpu - start_cpu:.4f} s")

# ------------------------------------
# GPU PART (CuPy)
# ------------------------------------

import cupy as cp
from cupyx.scipy.signal import convolve
```

```
# Check if GPU detected
print("GPU count:", cp.cuda.runtime.getDeviceCount())

start_gpu = time.time()

# Move data to GPU
signal_gpu = cp.array(signal)
kernel_gpu = cp.array(kernel)

# GPU filtering
filtered_gpu = convolve(signal_gpu, kernel_gpu, mode='same')

# GPU computation must be synchronized
cp.cuda.Stream.null.synchronize()

end_gpu = time.time()

print(f"GPU Execution Time: {end_gpu - start_gpu:.4f} s")

# Convert GPU result back to CPU (optional)
filtered_gpu_cpu = cp.asnumpy(filtered_gpu)
```

Assignments for Students

Assignment Task 1 — Parallel Filtering

Implement a low-pass filter for a large 1D signal using both sequential and parallel processing. Measure speed improvement.

Assignment Task 2 — Parallel FFT Processing

Perform FFT on multiple signals in both sequential and parallel modes. Compare performance metrics and plot execution time differences.

Assignment Task 3 — Parallel Edge Detection in Images

Apply parallel convolution operations for edge detection (Sobel or Prewitt) on large images using multiprocessing.

Assignment Task 4 — GPU-Accelerated Signal Denoising

Use CuPy or PyTorch to perform real-time denoising on a noisy signal using a GPU. Compare CPU vs GPU performance.

Control Questions

1. What is parallel processing and why is it important in signal processing?
2. Define data parallelism and task parallelism with examples.
3. How does multiprocessing improve signal filtering performance?

4. What are the main challenges in implementing parallel signal algorithms?
5. Compare CPU-based parallelism with GPU-based parallelism.
6. What types of signal operations are most suitable for parallelization?
7. Why does parallel processing not always guarantee speed improvement?
8. How can load balancing affect parallel system performance?
9. What is the difference between synchronous and asynchronous parallel execution?
10. Explain the benefit of using FFT with parallel processing.
11. How can Python libraries like NumPy and CuPy accelerate signal computations?
12. Describe one real-world application that relies on parallel signal processing.
13. What is the effect of increasing the number of processes beyond available cores?
14. How can parallel computing help in real-time digital signal processing systems?