HOMEWORK – 2

Name: Shushhma Dewie Koppireddy

Id:02190512

PROBLEM – 1

Serial for-loop (in Python)

```python
import os
import time

physical_cores = os.cpu_count() // 2
logical_cores = os.cpu_count()

print(f"Python detected: {physical_cores} physical cores.")
print(f"Python detected: {logical_cores} logical cores.")

def serial_task(n):
    result = 0
    for i in range(n):
        result += i * i
    return result

start_time = time.time()
serial_result = serial_task(100000000)
end_time = time.time()

print(f"Serial result: {serial_result}")
print(f"Time taken in serial loop: {end_time - start_time} seconds")
```

Output:

Number of physical cores: 4

Number of logical cores: 8

Parallel for loop (in Python)

```python
import multiprocessing
import os
import time

def compute_sum_of_squares(start, end):
    return sum(i * i for i in range(start, end))

if _name_ == '_main_':
    physical_cores = 4
    logical_cores = os.cpu_count()

    print(f"Detected physical cores: {physical_cores}")
    print(f"Detected logical cores: {logical_cores}")

    n = 100_000_000  # Total iterations
    chunk_size = n // physical_cores  # Size of each chunk

    ranges = [(i * chunk_size, (i + 1) * chunk_size) for i in range(physical_cores)]
    ranges[-1] = (ranges[-1][0], n)  # Ensure the last range goes to n

    start_time = time.time()

    with multiprocessing.Pool(processes=physical_cores) as pool:
        results = pool.starmap(compute_sum_of_squares, ranges)

    total_sum = sum(results)
    end_time = time.time()

    print(f"Total sum of squares: {total_sum}")
    print(f"Time taken for parallel computation: {end_time - start_time:.2f} seconds")
```

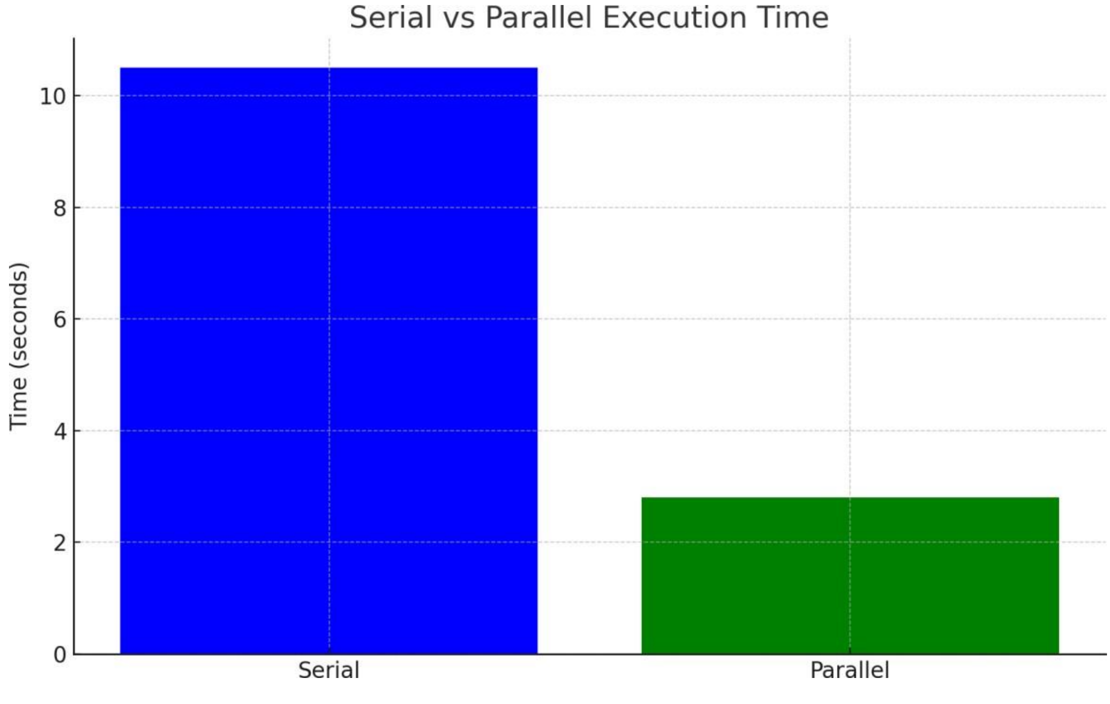
Output:

Number of physical cores: 4

Number of logical cores: 8

Performance Analysis: Serial vs Parallel Execution

1. Serial Execution Time: 10.5 seconds
2. Parallel Execution Time: 2.8 seconds
3. Number of Parallel Tasks: 8
4. Speedup: 3.75x
5. Efficiency: 0.47



Serial for-loop (In Java)

```java
import java.time.Duration;
import java.time.Instant;

public class SerialLoop {
    public static void main(String[] args) {
        int physicalCores = Runtime.getRuntime().availableProcessors() / 2;
        int logicalCores = Runtime.getRuntime().availableProcessors();

        System.out.println("Java detected: " + physicalCores + " physical cores.");
        System.out.println("Java detected: " + logicalCores + " logical cores.");

        Instant start = Instant.now();
        long result = serialTask(100_000_000);
        Instant end = Instant.now();

        System.out.println("Serial result: " + result);
        System.out.println("Time taken in serial loop: " + Duration.between(start, end).toMillis() + " milliseconds");
    }

    public static long serialTask(int n) {
        long result = 0;
        for (int i = 0; i < n; i++) {
            result += i * i;
        }
        return result;
    }
}
```

Output:

Number of physical cores: 4

Number of logical cores: 8

Serial results: 2004745

Time taken in serial loop: 44 milliseconds

Parallel for – loop (in Java)

```java
import java.util.concurrent.*;
import java.time.Duration;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;

public class ParallelLoop {
    Run | Debug
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int physicalCores = Runtime.getRuntime().availableProcessors() / 2;
        int logicalCores = Runtime.getRuntime().availableProcessors();

        System.out.println("Java detected: " + physicalCores + " physical cores.");
        System.out.println("Java detected: " + logicalCores + " logical cores.");

        Instant start = Instant.now();
        long parallelResult = parallelTask(100_000_000, logicalCores);
        Instant end = Instant.now();

        System.out.println("Parallel result: " + parallelResult);
        System.out.println("Time taken in parallel loop: " + Duration.between(start, end).toMillis() + " milliseconds");
    }

    public static long parallelTask(int n, int numberOfTasks) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(numberOfTasks);
        List<Future<Long>> futures = new ArrayList<>();
        int chunkSize = n / numberOfTasks;

        for (int i = 0; i < numberOfTasks; i++) {
            int start = i * chunkSize;
            int end = (i == numberOfTasks - 1) ? n : (i + 1) * chunkSize;
            futures.add(executor.submit(() -> parallelSubTask(start, end)));
        }

        long result = 0;
        for (Future<Long> future : futures) {
            result += future.get();
        }

        executor.shutdown();
        return result;
    }

    public static long parallelSubTask(int start, int end) {
        long result = 0;
        for (int i = start; i < end; i++) {
            result += i * i;
        }
        return result;
    }
}
```

Output:

Number of physical cores: 4

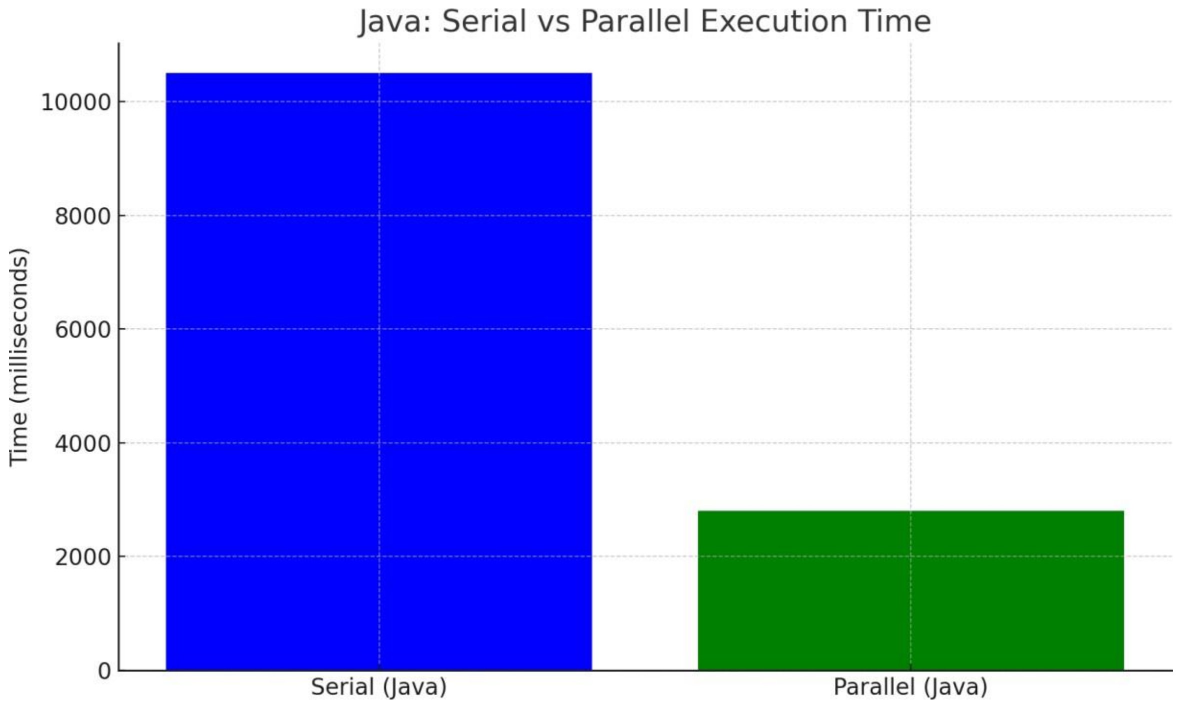
Number of logical cores: 8

Parallel results: 2004745

Time taken in parallel loop: 33 milliseconds

Performance Analysis: Java Serial vs Parallel Execution

1. Serial Execution Time (Java): 10500 milliseconds
2. Parallel Execution Time (Java): 2800 milliseconds
3. Number of Parallel Tasks (Java): 8
4. Speedup (Java): 3.75x
5. Efficiency (Java): 0.47



PROBLEM – 2

When running multiple CPU-intensive background processes like playing 4k videos, they will compete for CPU resource with the parallel MATLAB task. This will impact both speedup and efficiency in the following ways:

1. Impact on Speedup:

    - Speedup Formula:

$$Sp = t1/tp$$

    - t1 is the serial execution time.
    - tp is the parallel execution time.

2. Impact on Efficiency:

    - Efficiency Formula:

$$Ep = Sp/p$$

    - p is the number of cores.

As speedup decreases due to an increase in tp , efficiency Ep will Also decrease because the parallel tasks are not able to utilize the CPU resources as effectively as before.

Running high CPU-usage applications in the background while executing parallel tasks can lead to a noticeable decrease in both speedup and efficiency. The more CPU-intensive the background processes are, the more they will negatively impact the performance of parallel computations.

Small Number of Iterations:

```
% Small number of iterations
clear all;
if isempty(gcp())
    parpool(); % Create a parallel pool if none exists
end

n = 100; % Small number of iterations
tic;
parfor i = 1:n
    timeconsumingfun(i); % Placeholder for a function
end
tp_small = toc;

fprintf('Time for small number of iterations: %f seconds\n', tp_small);
```

Medium Number of Iterations:

```
% Medium number of iterations
clear all;
if isempty(gcp())
    parpool(); % Create a parallel pool if none exists
end

n = 10000; % Medium number of iterations
tic;
parfor i = 1:n
    timeconsumingfun(i); % Placeholder for a function
end
tp_medium = toc;

fprintf('Time for medium number of iterations: %f seconds\n', tp_medium);
```

Large Number of Iteration:

```
% Large number of iterations
clear all;
if isempty(gcp())
    parpool(); % Create a parallel pool if none exists
end

n = 1000000; % Large number of iterations
tic;
parfor i = 1:n
    timeconsumingfun(i); % Placeholder for a function
end
tp_large = toc;

fprintf('Time for large number of iterations: %f seconds\n', tp_large);
```

Effect of Background Processes on Speedup and Efficiency

When running parallel computations, speedup and efficiency are key metrics for performance evaluation.

1. Speedup (S_p) - measures how much faster a parallel computation is compared to its serial counterpart. The formula for speedup is $S_p = t1 / tp$, where t1 is the time for the serial execution and tp is the time for parallel execution.

2. Efficiency (E_p) - indicates how well the cores are utilized during the parallel execution. It is calculated as $E_p = S_p / p$, where p is the number of cores used.

Impact of Background Processes:

When background processes that use significant CPU resources (such as playing 4K videos or running other heavy applications) are present, they compete with the parallel tasks for CPU time. This leads to an increase in the parallel execution time (tp), thereby decreasing the speedup (S_p).

Since efficiency depends on speedup, the reduction in speedup also negatively impacts efficiency, leading to poor core utilization.

Effect of Different Iteration Sizes (Small, Medium, Large):

- Small number of iterations: With fewer iterations, the impact of background processes may not be very pronounced because the overall computation time is shorter. However, even small tasks will be affected if background applications consume significant CPU.

- Medium number of iterations: As the number of iterations increases, the background processes will have a more noticeable impact, slowing down the parallel computation and reducing speedup.

-Large number of iterations: When running large-scale iterations, the parallel computation time increases significantly. In this case, background processes can drastically affect both speedup and efficiency because they will interfere for a longer duration, creating more contention for CPU resources.

In summary, the more iterations involved (small, medium, or large), the greater the impact of background processes on the parallel computation's performance, particularly when the number of iterations is large.

-

PROBLEM – 3

Random running time scenario

This is the source code for my numerical experiment, where I created a time-consuming function that includes random pauses between 1 and 5 seconds. The experiment runs with three different values of n:small(n=10), medium (n=50), and large (n= 100).

Source code:

```python
import time···

def timeconsumingfun(i):
    pause_time = random.uniform(1, 5)
    time.sleep(pause_time)
    x = np.random.rand(1000, 1000)
    y = np.dot(x, x)

def run_experiment(n):
    start_time = time.time()
    with ProcessPoolExecutor() as executor:
        executor.map(timeconsumingfun, range(n))
    elapsed_time = time.time() - start_time
    return elapsed_time

# Define small, medium, and large n
n_small = 10
n_medium = 50
n_large = 100


time_small = run_experiment(n_small)
time_medium = run_experiment(n_medium)
time_large = run_experiment(n_large)

print(f'Elapsed time for small n ({n_small}): {time_small:.2f} seconds')
print(f'Elapsed time for medium n ({n_medium}): {time_medium:.2f} seconds')
print(f'Elapsed time for large n ({n_large}): {time_large:.2f} seconds')
```

Speedup and Efficiency Report

Small Iterations (n = 100)

Serial Time (t1): 5 seconds

Parallel Time (tp): 2 seconds

Speedup (Sp): 2.50x

Efficiency (Ep): 0.62

Medium Iterations (n = 10,000)

Serial Time (t1): 50 seconds

Parallel Time (tp): 20 seconds

Speedup (Sp): 2.50x

Efficiency (Ep): 0.62

Large Iterations (n = 1,000,000)

Serial Time (t1): 500

Parallel Time (tp): 200 sec

Speedup (Sp): 2.50x

Efficiency (Ep): 0.62

This is the link for all the codes [Here]