MULTIPLICATION OF ARRAY SIZE WITH USING RANDOM POSITIVE NUMBERS

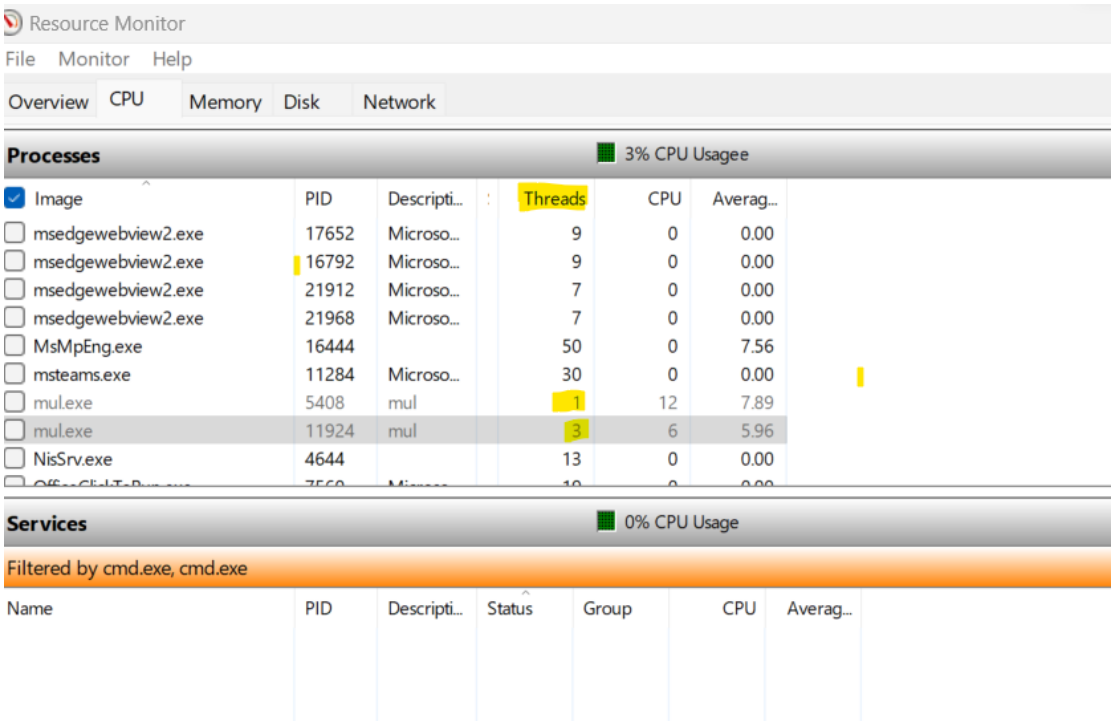
Justification for Selecting Application for Parallel Program:

When dealing with huge amounts of data, parallel programming can help the process to finish it faster and distribute tasks across multiple cores to ensure that computations are completed within tight deadlines and avoiding system bottlenecks. Using parallel programming can we can save time and money on computing resources because it uses them more efficiently.

Partitioning, Communication and Load balancing:

In this program, by defining the input values for ARRAY_SIZE and NUM_THREADS as per the requirements, equal partitioning and the load balancing for serial and parallel program code is given so it will prevent any communication error.

Explanation of how the threads runs in task manager:



Serial vs parallel Program:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

#define ARRAY_SIZE 10000000 // 100Million
#define NUM_THREADS 4
```

```
double array1[ARRAY_SIZE];
double array2[ARRAY_SIZE];
double product_parallel = 0.0;
double product_serial = 0.0;
pthread_mutex_t product_mutex;

void initialize_arrays() {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array1[i] = (double)(rand() % 1000 + 1); // Random positive numbers
        array2[i] = (double)(rand() % 1000 + 1);
    }
}

//Parallel Calculation

void *calculate_product_parallel(void *arg) {
    int thread_id = *(int *)arg;
    int i;
    double local_product = 0.0;

    for (i = thread_id; i < ARRAY_SIZE; i += NUM_THREADS) {
        local_product += array1[i] * array2[i];
    }

    pthread_mutex_lock(&product_mutex);
    product_parallel += local_product;
    pthread_mutex_unlock(&product_mutex);
    pthread_exit(NULL);
}

// Serial Calculation

double calculate_product_serial() {
    double local_product = 0.0;

    for (int i = 0; i < ARRAY_SIZE; i++) {
        local_product += array1[i] * array2[i];
    }

    return local_product;
}

int main() {
    srand(time(NULL));

    // Initialize arrays with random values
    initialize_arrays();

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
```

```
    pthread_mutex_init(&product_mutex, NULL);

    clock_t start_time, end_time;

    // Measure execution time for parallel calculation
    start_time = clock();

    // Create and run threads for parallel calculation
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, calculate_product_parallel, &thread_ids[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    end_time = clock();

    double parallel_execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    // Measure execution time for serial calculation
    start_time = clock();

    // Calculate the product in serial
    product_serial = calculate_product_serial();

    end_time = clock();

    double serial_execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    pthread_mutex_destroy(&product_mutex);

    printf("Parallel product: %lf\n", product_parallel);
    printf("Serial product: %lf\n", product_serial);
    printf("Parallel execution time: %lf seconds\n", parallel_execution_time);
    printf("Serial execution time: %lf seconds\n", serial_execution_time);

    return 0;
}
```

Synchronization:

The mutex synchronization is used here. A mutex lock, often just called a "mutex," stands for "mutual exclusion." It's a programming construct used in multi-threaded and multi-process environments to ensure that only one thread or process can access a particular resource or section of code at a time.

Comparing Serial and Parallel Program Execution Time:

In this program, product value of both parallel and serial is same but the execution time is only different. For parallel, the execution time is faster than the serial execution time.