Practical File

Question 1

Write a program in C to multiply two matrices of size 10000 x 10000 each and find it's execution-time using "time" command. Try to run this program on two or more machines having different configurations and compare execution-times obtained in each run. Comment on which factors affect the performance of the program

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

#define SIZE 3

int main(){
        int matrix1[SIZE][SIZE] = {
                {1,2,3},
                {4,5,6},
                {7,8,9}
        };

        int matrix2[SIZE][SIZE] = {
                {9,8,7},
                {6,5,4},
                {3,2,1}
        };

        int result[SIZE][SIZE];
        int i, j, k;
        clock_t start, end;
        double cpu_time_used;
        start = clock();

        for(i = 0; i < SIZE; i++){
                for(j=0; j < SIZE; j++){
                        result[i][j] = 0;
                        for(k = 0; k < SIZE; k++){
                                result[i][j] += matrix1[i][k] * matrix2[k]
[j];
                        }
                }
        }

        end = clock();
        cpu_time_used = ((double) (end-start)) / CLOCKS_PER_SEC;
        printf("Matrix 1:\n");

        for(i = 0; i < SIZE; i++){
                for(j = 0; j < SIZE; j++){
                        printf("%d ", matrix1[i][j]);
                }
                printf("\n");
        }

        printf("Matrix 2:\n");
        for(i = 0; i < SIZE; i++){
                for(j = 0; j < SIZE; j++){
                        printf("%d ", matrix2[i][j]);
                }
                printf("\n");
        }

        printf("Result Matrix:\n");
        for(i = 0; i < SIZE; i++){
                for(j = 0; j < SIZE; j++){
                        printf("%d ", matrix2[i][j]);
                }
                printf("\n");
        }

        printf("\nMultiplication completed in %f seconds\n",
cpu_time_used);
        return 0;
}
```

Output

```
Matrix 1:
1 2 3
4 5 6
7 8 9
Matrix 2:
9 8 7
6 5 4
3 2 1
Result Matrix:
9 8 7
6 5 4
3 2 1
```

Question 2

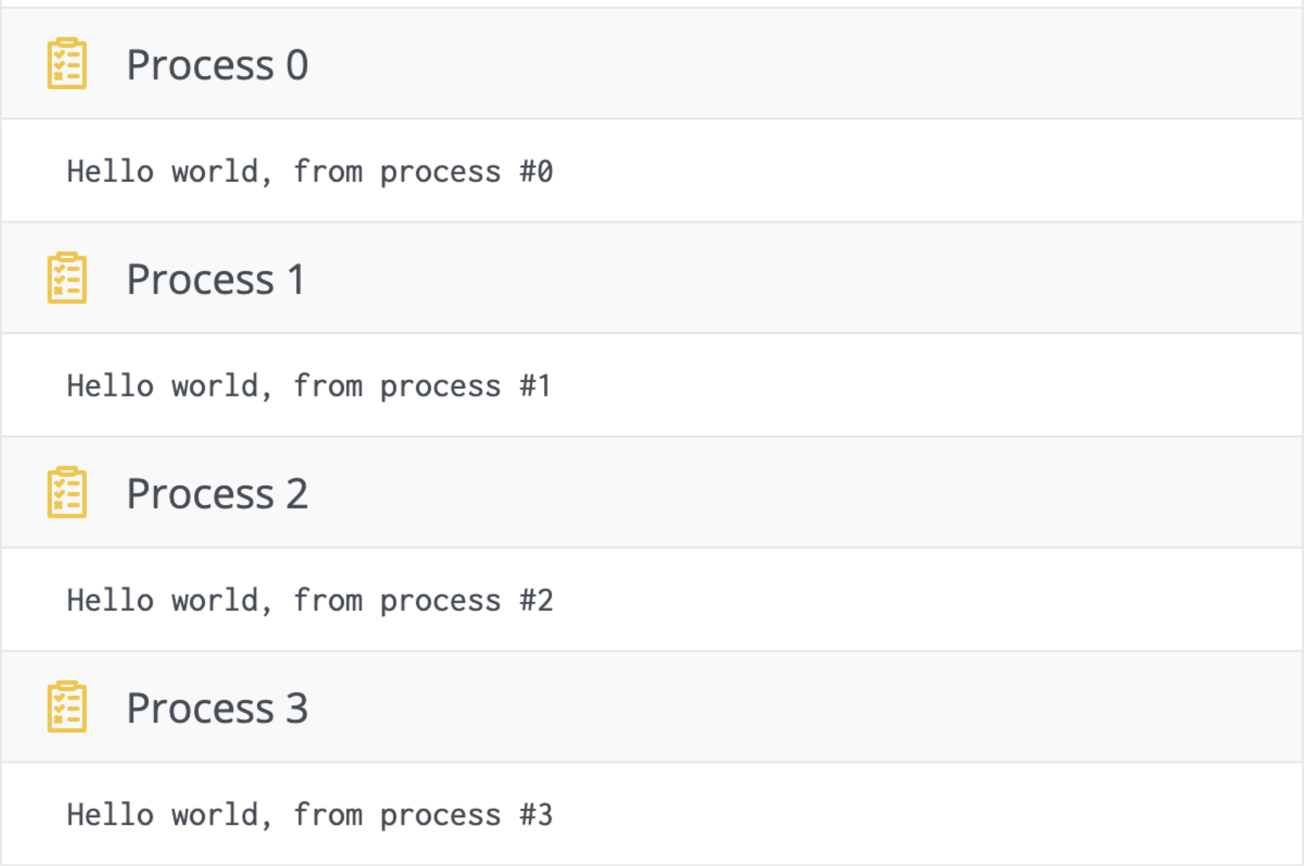
Write a parallel program to print “Hello World" using MPI

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
        int rank, size;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        printf("Hello World \n Rank: %d \n Size: %d\n", rank, size);
        MPI_Finalize();
        return 0;
}
```

Output



Process 0

```
Hello world, from process #0
```

Process 1

```
Hello world, from process #1
```

Process 2

```
Hello world, from process #2
```

Process 3

```
Hello world, from process #3
```

Question 3

Write a parallel program to find sum of an array using MPI

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
        int rank, size;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        int arr[] = {1, 2, 3, 4, 5, 6};
        int arr_size = sizeof(arr) / sizeof(arr[0]);

        int local_sum = 0;
        int chunk_size = arr_size / size;
        int start_index = rank * chunk_size;

        if(rank == size - 1){
                chunk_size += arr_size % size;
        }

        MPI_Scatter(arr, chunk_size, MPI_INT, &local_sum, chunk_size,
MPI_INT, 0, MPI_COMM_WORLD);

        for(int i = start_index; i < start_index + chunk_size; i++){
        local_sum += arr[i]};
}
        int global_sum = 0;
        MPI_Gather(&local_sum, 1, MPI_INT, &global_sum, 1, MPI_INT, 0,
MPI_COMM_WORLD);

        if(rank ==0){
                printf("The sum of the array is: %d\n", global_sum);
        }

        MPI_Finalize();

        return 0;
```

Output

```
The sum of the array is: 21
```

Question 5

Write a C program to implement the Quick Sort Algorithm using MPI.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 10

void quicksort(int *array, int left, int right);
int partition(int *array, int left, int right);
```

```
void swap(int *a, int *b);

int main(int argc, char *argv[]) {
        int rank, size;
        int array[ARRAY_SIZE];
        int i;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (rank == 0) {
                printf("Enter %d elements: ", ARRAY_SIZE);
                for (i = 0; i < ARRAY_SIZE; i++) {
                        scanf("%d", &array[i]);
                }
        }
        MPI_Bcast(array, ARRAY_SIZE, MPI_INT, 0, MPI_COMM_WORLD);
        // Perform quicksort
        quicksort(array, 0, ARRAY_SIZE - 1);

        // Gather sorted array from all processes
        int *sortedArray = NULL;
        if (rank == 0) {
                sortedArray = (int *)malloc(ARRAY_SIZE * sizeof(int));
        }
        MPI_Gather(array, ARRAY_SIZE, MPI_INT, sortedArray, ARRAY_SIE,
MPI_INT, 0, MPI_COMM_WORLD);

        // Print sorted array in rank 0
        if (rank == 0) {
                printf("Sorted array: ");
                for (i = 0; i < ARRAY_SIZE; i++) {
                        printf("%d ", sortedArray[i]);
                }
                printf("\n");
                free(sortedArray);
        }

        MPI_Finalize();
        return 0;
}

void quicksort(int *array, int left, int right) {
        if (left < right) {
                int pivotIndex = partition(array, left, right);
                quicksort(array, left, pivotIndex - 1);
                quicksort(array, pivotIndex + 1, right);
        }
}
```

```
int partition(int *array, int left, int right) {
int pivot = array[right];
int i = left - 1;

for (int j = left; j < right; j++) {
        if (array[j] <= pivot) {
                i++;
                swap(&array[i], &array[j]);
        }
}
swap(&array[i + 1], &array[right]);
return i + 1;

}

void swap(int *a, int *b) {

        int temp = *a;
        *a = *b;
        *b = temp;
}
```

Output

```
Enter 10 elements:
3
2
6
4
8
1
63
20
88
43
Sorted array: 1 2 3 4 6 8 20 43 63 88
```

Question 6

Write a multithreaded program to generate Fibonacci series using pThreads.

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

void *runner(void *param) {
        // Argument passed to the thread (thread number)
        long int num = (long int)param;
        long int fib1 = 0, fib2 = 1, next;
        // Print Fibonacci series up to 'num'th term (0-based indexing)
        if (num > 1) {
                printf("Fibonacci Series: ");
                printf("%ld, %ld, ", fib1, fib2);
                for (int i = 2; i < num; i++) {
                        next = fib1 + fib2;
                        printf("%ld, ", next);
                        fib1 = fib2;
                        fib2 = next;
                }
        } else if (num == 1) {
                printf("Fibonacci Series: 0\n");
        } else {
                printf("Fibonacci Series: \n"); // Empty series for num <=
0
        }

        pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
        long int num;
        pthread_t tid;
        // Get the number of terms from the user (modify for default
value)
        if (argc != 2) {
                printf("Usage: %s <number_of_terms>\n", argv[0]);
                return 1;
        }

        num = atol(argv[1]);

        // Create a new thread
        if (pthread_create(&tid, NULL, runner, (void *)num) != 0) {
                perror("pthread_create failed");
                return 1;
        }

        // Wait for the thread to finish
        if (pthread_join(tid, NULL) != 0) {
                perror("pthread_join failed");
                return 1;
        }

        printf("\n"); // Newline after thread execution

        return 0;
}
```

Output

```
Fibonacci Series: 0, 1, 1, 2, 3, 5,
```

Question 7

Write a "Hello World" program using OpenMP library also display number of threads created during execution.

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

int counter = 0;
pthread_mutex_t mutex;

void increment() {
        // Lock the mutex before accessing the shared counter
        pthread_mutex_lock(&mutex);
        counter++;
        // Unlock the mutex after accessing the shared counter
        pthread_mutex_unlock(&mutex);
}

void* worker(void* arg) {
        // Perform increment 1000 times
        for (int i = 0; i < 1000; i++) {
                increment();
        }
        pthread_exit(NULL);
}

int main(int argc, char* argv) {
        int num_threads = 2; // Modify as needed
        // Initialize the mutex lock
        if (pthread_mutex_init(&mutex, NULL) != 0) {
                perror("Mutex initialization failed");
                exit(1);
        }
        // Create threads
        pthread_t threads[num_threads];
        for (int i = 0; i < num_threads; i++) {
                if (pthread_create(&threads[i], NULL, worker, NULL) != 0)
{
                        perror("Thread creation failed");
                        exit(1);
                }
        }
        // Wait for all threads to finish
        for (int i = 0; i < num_threads; i++) {
                if (pthread_join(threads[i], NULL) != 0) {
                        perror("Thread join failed");
                        exit(1);
                }
        }
        // Destroy the mutex lock
        pthread_mutex_destroy(&mutex);
        printf("Final counter value: %d\n", counter);
        return 0;
}
```

Output

```
Final Counter Value: 2000
```

Question 8

Write a "Hello World" program using OpenMP library also display number of threads created during execution.

```
#include <stdio.h>
#include <omp.h>

int main() {
        int num_threads;
        // Get the number of threads available
        #pragma omp parallel num_threads(4) // Request 4 threads (modify
as needed)
        {
        num_threads = omp_get_num_threads();
        printf("Hello World from thread %d of %d\n", omp_get_thread_num(),
num_threads);
        }
        return 0;
}
```

Output

```
Hello World from thread 0 of 4
Hello World from thread 1 of 4
Hello World from thread 2 of 4
Hello World from thread 3 of 4
```

Question 9

Write a C program to demonstrate multitask using OpenMP

```c
#include <stdio.h>
#include <omp.h>
#include <unistd.h>

void task1() {
        printf("Task 1: Performing calculations...\n");
        // Simulate some calculations by sleeping for a short duration
        #pragma omp barrier
        sleep(1); // Adjust sleep time as needed
        printf("Task 1 finished.\n");
}

void task2() {
        printf("Task 2: Reading data from a file...\n");
        // Simulate reading data by sleeping for a short duration
        #pragma omp barrier
        sleep(2); // Adjust sleep time as needed
        printf("Task 2 finished.\n");
}

int main() {
        int num_threads;
        // Get the number of threads available
        #pragma omp parallel num_threads(2) // Request 2 threads
        {
                num_threads = omp_get_num_threads();
                  #pragma omp sections nowait // Execute sections
concurrently without waiting
                {
                #pragma omp section
                        {
                        task1();
                        }
                #pragma omp section
                        {
                        task2();
                        }
                }
        }
        printf("Number of threads used: %d\n", num_threads);

        return 0;

}
```

Output

```
Task 1: Perfoirming calculations...
Task 2: Reading dta from a file...
Task 1 finished
Task 2  finished
Number of threads used: 2
```

Question 10

Write a parallel program to calculate the value of PI/Area of Circle using OpenMP library.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SEED 357952 // Modify for different random number sequences
(optional)
double calculate_pi(long int num_points) {
        long int num_in_circle = 0;
        double x, y;
        // Initialize random number generator with a seed (optional for
reproducibility)
        srand(SEED);
        #pragma omp parallel private(x, y) reduction(+:num_in_circle)
        {
        // Generate random points within a square of side length 2
        for (long int i = 0; i < num_points; i++) {
                x = (double)rand() / RAND_MAX;
                y = (double)rand() / RAND_MAX;
                // Check if the point falls inside the circle with radius
1

                if (x * x + y * y <= 1.0) {
                        num_in_circle++;
                }
        }
}

        // Estimate PI using the ratio of points inside the circle to
total points
        double pi_estimate = 4.0 * ((double)num_in_circle /
(double)num_points);

        return pi_estimate;

}

int main(int argc, char* argv) {
        long int num_points = 1000000; // Modify as needed for desired
accuracy
        // Get the number of threads available
        int num_threads = omp_get_max_threads();
        printf("Using %d threads for calculation.\n", num_threads);
        double pi = calculate_pi(num_points);
        printf("Estimated value of PI/Area of Circle: %.10lf\n", pi);

        return 0;

}
```

Output

```
TUsing 8 threads for calculation.
Estimated value of PI/Area of Circle: 25.1387720000
```

Question 11

Write a C program to demonstrate default, static and dynamic loop scheduling using OpenMP.

```
#include <stdio.h>
#include <omp.h>
#define ARRAY_SIZE 5
int main() {
        int i, tid;
        int array[ARRAY_SIZE];
        // Initialize the array
        for (i = 0; i < ARRAY_SIZE; i++) {
                array[i] = i + 1;
        }
        // Default loop scheduling
        printf("Default loop scheduling:\n");
        #pragma omp parallel private(i, tid)
        {
                tid = omp_get_thread_num();
                #pragma omp for
                for (i = 0; i < ARRAY_SIZE; i++) {
                        printf("Thread %d: array[%d] = %d\n", tid, i,
array[i]);
                }
        }
        // Static loop scheduling with chunk size 10
        printf("\nStatic loop scheduling with chunk size 10:\n");
        #pragma omp parallel private(i, tid)
        {
                tid = omp_get_thread_num();
                #pragma omp for schedule(static, 10)
                for (i = 0; i < ARRAY_SIZE; i++) {
                        printf("Thread %d: array[%d] = %d\n", tid, i,
array[i]);
                }
        }
        // Dynamic loop scheduling with chunk size 10
        printf("\nDynamic loop scheduling with chunk size 10:\n");
        #pragma omp parallel private(i, tid)
        {
                tid = omp_get_thread_num();
                #pragma omp for schedule(dynamic, 10)
                for (i = 0; i < ARRAY_SIZE; i++) {
                        printf("Thread %d: array[%d] = %d\n", tid, i,
array[i]);
                }
        }

return 0;

}
```

Output

```
Default loop scheduling:
Thread 3: array[3] = 4
Thread 1: array[1] = 2
Thread 4: array[4] = 5
Thread 0: array[0] = 1
Thread 2: array[2] = 3

Static loop scheduling with chunk size 10:
Thread 0: array[0] = 1
Thread 0: array[1] = 2
Thread 0: array[2] = 3
Thread 0: array[3] = 4
Thread 0: array[4] = 5

Static loop scheduling with chunk size 10:
Thread 0: array[0] = 1
Thread 0: array[1] = 2
Thread 0: array[2] = 3
Thread 0: array[3] = 4
Thread 0: array[4] = 5
```