Nithin S
221IT085

IT301 Lab Assignment 5

Q1. To understand the concept of task and taskwait. Execute the following code and write the results and observation.

Code

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

int fibonacci(int n)
{
    if(n<=1) return n;
    int x,y;
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task shared(x)
            {
                x=fibonacci(n-1);
            }
            #pragma omp task shared(y)
            {
                y=fibonacci(n-2);
            }
            #pragma omp taskwait
        }
    }
    return x+y;
}

int main()
{
    int n=15;
    omp_set_num_threads(2);
    double start=omp_get_wtime();
    int result=fibonacci(n);
    double end=omp_get_wtime();
    printf("Fibonacci of %d is %d\n",n,result);
    printf("Time Taken: %f seconds\n",end-start);
    return 0;
}
```

Output

```
● nithin@pavilion:~/Codes/Sem5/IT301/Lab/Lab_5$ gcc 1.c -fopenmp
● nithin@pavilion:~/Codes/Sem5/IT301/Lab/Lab_5$ ./a.out
  Fibonacci of 15 is 610
  Time Taken: 0.000582 seconds
○ nithin@pavilion:~/Codes/Sem5/IT301/Lab/Lab_5$ 
```

Analysis:

The code uses OpenMP tasks to parallelize the recursive calls. Each call to fibonacci(n-1) and fibonacci(n-2) is encapsulated within a separate task, allowing them to potentially execute concurrently.

The #pragma omp single directive ensures that only one thread (typically the master thread) is responsible for spawning the tasks. This avoids multiple threads creating redundant tasks for the same computation.

The #pragma omp taskwait directive in OpenMP is used to synchronize tasks within a parallel region.

In the context of the provided Fibonacci function, taskwait ensures that the main thread waits for the completion of all previously spawned tasks before proceeding.

Specifically, after creating two tasks—one for computing fibonacci(n-1) and another for fibonacci(n-2)—the taskwait directive forces the program to pause and wait until both tasks are finished.

This ensures that the values of x and y (which store the results of the two recursive calls) are correctly computed and available before summing them and returning the result.

Without taskwait, the function could proceed without waiting for the completion of the tasks, leading to undefined behavior as x and y may not be fully computed.

Thus, taskwait plays a critical role in synchronizing parallel tasks and maintaining correct program flow in parallel execution.

Q2. Execute the following OpenMP program to implement a linked list traversal in parallel using task and taskwait. Write the results and observation

Code

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

typedef struct Node{
    int data;
    struct Node*next;
}Node;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void traverseNode(Node* node) {
    if (node == NULL) return;
    printf("%d -> ", node->data);
    #pragma omp task
    {
        traverseNode(node->next);
    }
    #pragma omp taskwait
}

void traverseLinkedList(Node* head) {
    if (head == NULL) return;
    #pragma omp parallel
    {
        #pragma omp single
        {
            traverseNode(head);
        }
    }
}
```

```c
int main()
{
    Node* head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);
    head->next->next->next = createNode(40);
    head->next->next->next->next = createNode(50);
    printf("Linked list traversal: ");
    omp_set_num_threads(2);
    traverseLinkedList(head);
    printf("NULL\n");
    Node* current = head;
    Node* nextNode;
    while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
    }
    return 0;
}
```

Output

```
nithin@pavilion:~/Codes/Sem5/IT301/Lab/Lab_5$ gcc 2.c -fopenmp
nithin@pavilion:~/Codes/Sem5/IT301/Lab/Lab_5$ ./a.out
Linked list traversal: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
nithin@pavilion:~/Codes/Sem5/IT301/Lab/Lab_5$
```

Analysis

Parallel region defined by #pragma omp parallel in the traverseLinkedList function. Spawns a team of threads (in this case, 2 threads as set by omp_set_num_threads(2)).

#pragma omp single ensures that only one thread (typically the master thread) executes the traverseNode function initially. Prevents multiple threads from redundantly starting the traversal.

Within traverseNode, #pragma omp task is used to create a new task for traversing the next node.

This allows different parts of the linked list to be traversed in parallel, potentially utilizing multiple threads

#pragma omp taskwait ensures that the function waits for the spawned task (traversing the next node) to complete before proceeding.

Maintains the correct order of traversal and ensures that all tasks complete before the function returns.

Q3. 3. Write a sequential program to add elements of two arrays (C[i]=A[i]+B[i]). Convert the same program for parallel execution. Initialize arrays with random numbers. Consider the array size as 10k, 50k, and 100k. Analyze the results for maximum number of threads and various schedule () functions. Based on the observations, carry out the analysiof the total execution time and explain the results (i.e. writing your observations) by plotting the graph.

Serial Code

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

int main() {
    int sizes[]={5000,10000,50000,1000000};
    FILE *fp = fopen("execution_times.csv", "a");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "ArraySize,ExecutionTime\n");
    for(int i=0;i<4;i++){
        int n = sizes[i];
        int *A = (int*)malloc(n * sizeof(int));
        int *B = (int*)malloc(n * sizeof(int));
        int *C = (int*)malloc(n * sizeof(int));
        srand(time(0));
        for (int i = 0; i < n; i++) {
            A[i] = rand() % 100;
            B[i] = rand() % 100;
        }
        clock_t start = clock();
        for (int i = 0; i < n; i++) {
            C[i] = A[i] + B[i];
        }
        clock_t end = clock();
        double total_time = (double)(end - start) / CLOCKS_PER_SEC;
        printf("Sequential execution time of array of size %d : %f seconds\n", n,total_time);
        fprintf(fp, "%d, %f\n", n, total_time);
        free(A);
        free(B);
        free(C);
    }
    fclose(fp);
    return 0;
}
```

Static Schedule

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

int main() {

    int sizes[] = {5000, 10000, 50000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int thread_counts[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int num_thread_counts = sizeof(thread_counts) / sizeof(thread_counts[0]);
    FILE *fp = fopen("execution_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "ArraySize,ThreadCount,Schedule,ExecutionTime\n");
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int *A = (int*)malloc(n * sizeof(int));
        int *B = (int*)malloc(n * sizeof(int));
        int *C = (int*)malloc(n * sizeof(int));
        if (A == NULL || B == NULL || C == NULL) {
            printf("Memory allocation failed for size %d\n", n);
            free(A);
            free(B);
            free(C);
            continue;
        }
        srand(time(0));
        for (int j = 0; j < n; j++) {
            A[j] = rand() % 100;
            B[j] = rand() % 100;
        }
        double start_time = omp_get_wtime();
        for (int j = 0; j < n; j++) {
            C[j] = A[j] + B[j];
        }
        double end_time = omp_get_wtime();
        double serial_time = end_time - start_time;
        fprintf(fp, "%d,1,serial,%.6f\n", n, serial_time);
        printf("Serial execution time for array size %d: %f seconds\n", n, serial_time);

        for (int t = 0; t < num_thread_counts; t++) {
            int num_threads = thread_counts[t];
            if (num_threads == 1) {
                continue;
            }
            omp_set_num_threads(num_threads);
```

```c
        double end_time = omp_get_wtime();
        double serial_time = end_time - start_time;
        fprintf(fp, "%d,1,serial,%.6f\n", n, serial_time);
        printf("Serial execution time for array size %d: %f seconds\n", n, serial_time);

        for (int t = 0; t < num_thread_counts; t++) {
            int num_threads = thread_counts[t];
            if (num_threads == 1) {
                continue;
            }
            omp_set_num_threads(num_threads);
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(static)
            for (int j = 0; j < n; j++) {
                C[j] = A[j] + B[j];
            }
            end_time = omp_get_wtime();
            double parallel_time = end_time - start_time;
            fprintf(fp, "%d,%d,static,%.6f\n", n, num_threads, parallel_time);
            printf("Parallel execution time for array size %d with %d threads (static schedule): %f
        }
        free(A);
        free(B);
        free(C);
    }
    fclose(fp);
    printf("Execution times have been recorded in 'execution_times.csv'.\n");

    return 0;
}
```

Static with chunk size

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main() {
    int sizes[] = {5000, 10000, 50000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int thread_counts[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int num_thread_counts = sizeof(thread_counts) / sizeof(thread_counts[0]);
    FILE *fp = fopen("execution_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "ArraySize,ThreadCount,Schedule,ExecutionTime\n");
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int *A = (int*)malloc(n * sizeof(int));
        int *B = (int*)malloc(n * sizeof(int));
        int *C = (int*)malloc(n * sizeof(int));
        if (A == NULL || B == NULL || C == NULL) {
            printf("Memory allocation failed for size %d\n", n);
            free(A);
            free(B);
            free(C);
            continue;
        }
        srand(time(0));
        for (int j = 0; j < n; j++) {
            A[j] = rand() % 100;
            B[j] = rand() % 100;
        }
        double start_time = omp_get_wtime();
        for (int j = 0; j < n; j++) {
            C[j] = A[j] + B[j];
        }
        double end_time = omp_get_wtime();
        double serial_time = end_time - start_time;
        fprintf(fp, "%d,1,serial,%.6f\n", n, serial_time);
        printf("Serial execution time for array size %d: %f seconds\n", n, serial_time);
        for (int t = 0; t < num_thread_counts; t++) {
            int num_threads = thread_counts[t];
            if (num_threads == 1) {
                continue;
            }
            omp_set_num_threads(num_threads);
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(static,1000)
            for (int j = 0; j < n; j++) {
```

```c
            for (int j = 0; j < n; j++) {
                C[j] = A[j] + B[j];
            }
            end_time = omp_get_wtime();
            double parallel_time = end_time - start_time;
            fprintf(fp, "%d,%d,staticChunkSize,%.6f\n", n, num_threads, parallel_time);
            printf("Parallel execution time for array size %d with %d threads (static , 1000 schedul
        }
        free(A);
        free(B);
        free(C);
    }
    fclose(fp);
    printf("Execution times have been recorded in 'execution_times.csv'.\n");
    return 0;
}
```

Dynamic with Chunk Size

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main() {
    int sizes[] = {5000, 10000, 50000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int thread_counts[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int num_thread_counts = sizeof(thread_counts) / sizeof(thread_counts[0]);
    FILE *fp = fopen("execution_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "ArraySize,ThreadCount,Schedule,ExecutionTime\n");
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i void *malloc(size_t __size)
        int *A = (int*)
        int *B = (int*) Allocate SIZE bytes of memory.
        int *C = (int*)malloc(n * sizeof(int));
        if (A == NULL || B == NULL || C == NULL) {
            printf("Memory allocation failed for size %d\n", n);
            free(A);
            free(B);
            free(C);
            continue;
        }
        srand(time(0));
        for (int j = 0; j < n; j++) {
            A[j] = rand() % 100;
            B[j] = rand() % 100;
        }
        double start_time = omp_get_wtime();
        for (int j = 0; j < n; j++) {
            C[j] = A[j] + B[j];
        }
        double end_time = omp_get_wtime();
        double serial_time = end_time - start_time;
        fprintf(fp, "%d,1,serial,%.6f\n", n, serial_time);
        printf("Serial execution time for array size %d: %f seconds\n", n, serial_time);
        for (int t = 0; t < num_thread_counts; t++) {
            int num_threads = thread_counts[t];
            if (num_threads == 1) {
                continue;
            }
            omp_set_num_threads(num_threads);
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(dynamic,1000)
            for (int j = 0; j < n; j++) {
```

```c
                continue;
            }
            omp_set_num_threads(num_threads);
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(dynamic,1000)
            for (int j = 0; j < n; j++) {
                C[j] = A[j] + B[j];
            }
            end_time = omp_get_wtime();
            double parallel_time = end_time - start_time;
            fprintf(fp, "%d,%d,dynamicChunkSize,%.6f\n", n, num_threads, parallel_time);
            printf("Parallel execution time for array size %d with %d threads (static schedule): %f
        }
        free(A);
        free(B);
        free(C);
    }
    fclose(fp);
    printf("Execution times have been recorded in 'execution_times.csv'.\n");
    return 0;
}
```

Guided Schedule

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main() {
    int sizes[] = {5000, 10000, 50000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int thread_counts[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int num_thread_counts = sizeof(thread_counts) / sizeof(thread_counts[0]);
    FILE *fp = fopen("execution_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "ArraySize,ThreadCount,Schedule,ExecutionTime\n");
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int *A = (int*)malloc(n * sizeof(int));
        int *B = (int*)malloc(n * sizeof(int));
        int *C = (int*)malloc(n * sizeof(int));
        if (A == NULL || B == NULL || C == NULL) {
            printf("Memory allocation failed for size %d\n", n);
            free(A);
            free(B);
            free(C);
            continue;
        }
        srand(time(0));
        for (int j = 0; j < n; j++) {
            A[j] = rand() % 100;
            B[j] = rand() % 100;
        }
        double start_time = omp_get_wtime();
        for (int j = 0; j < n; j++) {
            C[j] = A[j] + B[j];
        }
        double end_time = omp_get_wtime();
        double serial_time = end_time - start_time;
        fprintf(fp, "%d,1,serial,%.6f\n", n, serial_time);
        printf("Serial execution time for array size %d: %f seconds\n", n, serial_time);
        for (int t = 0; t < num_thread_counts; t++) {
            int num_threads = thread_counts[t];
            if (num_threads == 1) {
                continue;
            }
            omp_set_num_threads(num_threads);
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(guided)
            for (int j = 0; j < n; j++) {
```

```c
        omp_set_num_threads(num_threads);
        start_time = omp_get_wtime();
        #pragma omp parallel for schedule(guided)
        for (int j = 0; j < n; j++) {
            C[j] = A[j] + B[j];
        }
        end_time = omp_get_wtime();
        double parallel_time = end_time - star
        fprintf(fp, "%d,%d,guided,%.6f\n", n, num_threads, parallel_time);
        printf("Parallel execution time for array size %d with %d threads (guided schedule):
    }
    free(A);
    free(B);
    free(C);
}
fclose(fp);
printf("Execution times have been recorded in 'execution_times.csv'.\n");
return 0;
```

Runtime Schedule

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main() {
    int sizes[] = {5000, 10000, 50000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int thread_counts[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int num_thread_counts = sizeof(thread_counts) / sizeof(thread_counts[0]);
    FILE *fp = fopen("execution_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "ArraySize,ThreadCount,Schedule,ExecutionTime\n");
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int *A = (int*)malloc(n * sizeof(int));
        int *B = (int*)malloc(n * sizeof(int));
        int *C = (int*)malloc(n * sizeof(int));
        if (A == NULL || B == NULL || C == NULL) {
            printf("Memory allocation failed for size %d\n", n);
            free(A);
            free(B);
            free(C);
            continue;
        }
        srand(time(0));
        for (int j = 0; j < n; j++) {
            A[j] = rand() % 100;
            B[j] = rand() % 100;
        }
        double start_time = omp_get_wtime();
        for (int j = 0; j < n; j++) {
            C[j] = A[j] + B[j];
        }
        double end_time = omp_get_wtime();
        double serial_time = end_time - start_time;
        fprintf(fp, "%d,1,serial,%.6f\n", n, serial_time);
        printf("Serial execution time for array size %d: %f seconds\n", n, serial_time);
        for (int t = 0; t < num_thread_counts; t++) {
            int num_threads = thread_counts[t];
            if (num_threads == 1) {
                continue;
            }
            omp_set_num_threads(num_threads);
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(runtime)
            for (int j = 0; j < n; j++) {
```

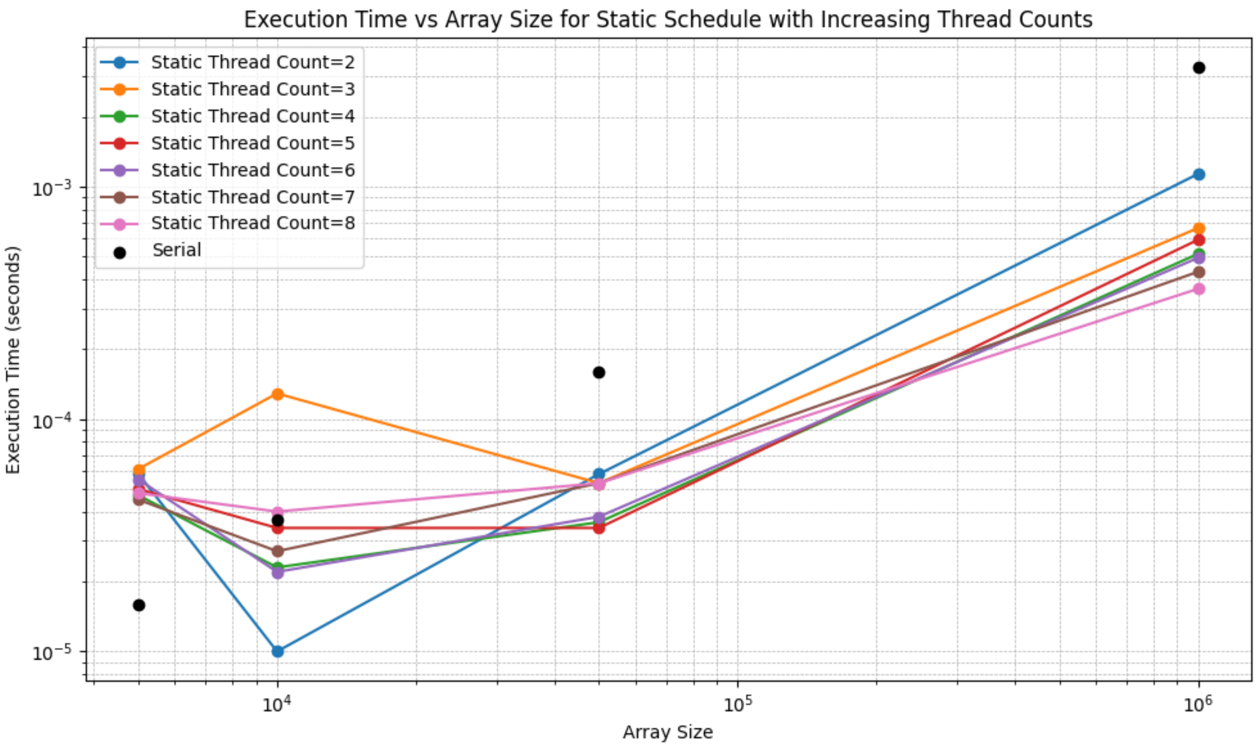
```
            start_time = omp_get_wtime();
            #pragma omp parallel for schedule(runtime)
            for (int j = 0; j < n; j++) {
                C[j] = A[j] + B[j];
            }
            end_time = omp_get_wtime();
            double parallel_time = end_time - start_time;
            fprintf(fp, "%d,%d,runtime,%.6f\n", n, num_threads, parallel_time);
            printf("Parallel execution time for array size %d with %d threads (runtime schedule): %f
        }
        free(A);
        free(B);
        free(C);
    }
    fclose(fp);
    printf("Execution times have been recorded in 'execution_times.csv'.\n");
    return 0;
}
```

Table Data (execution time in seconds taken when thread count = 8)

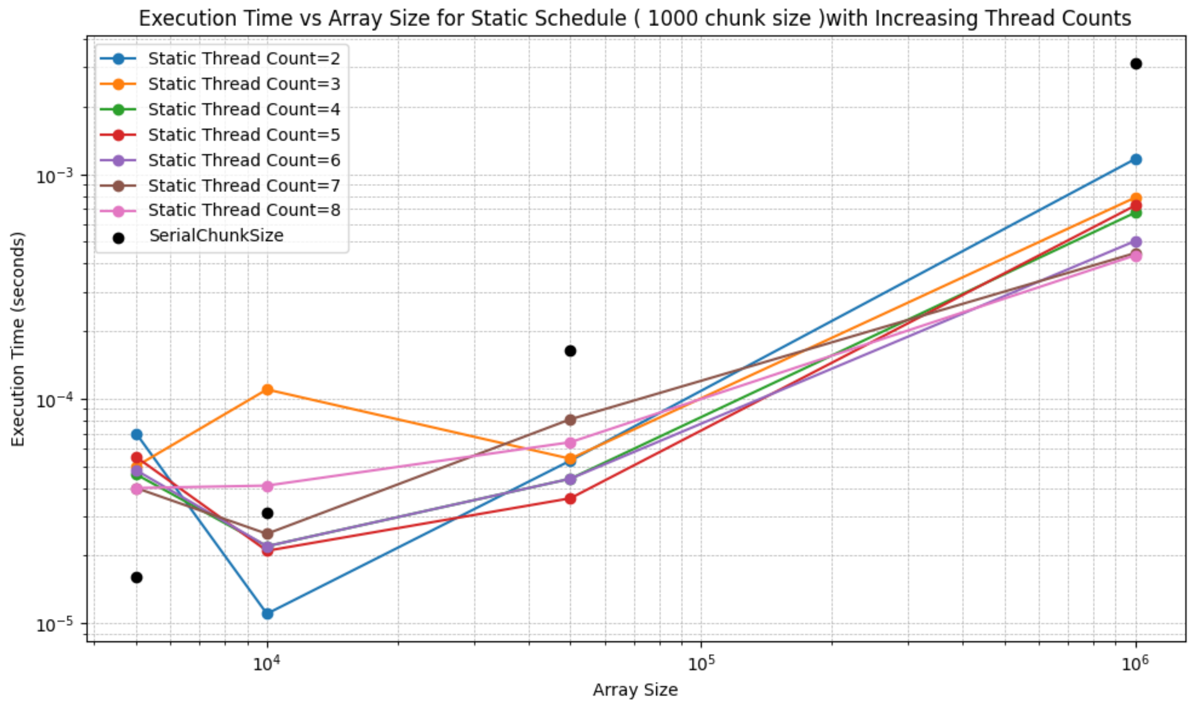
()	Execution time for number of iterations 5k	Execution time for number of iterations 10k	Execution time for number of iterations 50k	Execution time for number of iterations 100k
Sequential Execution	0.000016	0.000037	0.000159	0.000559
Static	0.000048	0.000040	0.000053	0.000364
Static, chunksize	0.000040	0.000041	0.000064	0.000436
Dyanmic,chunksize	0.000047	0.000043	0.000052	0.000332
Guided	0.000046	0.000041	0.000055	0.000366
Runtime	0.000045	0.000049	0.000057	0.000386

Graphical Analysis

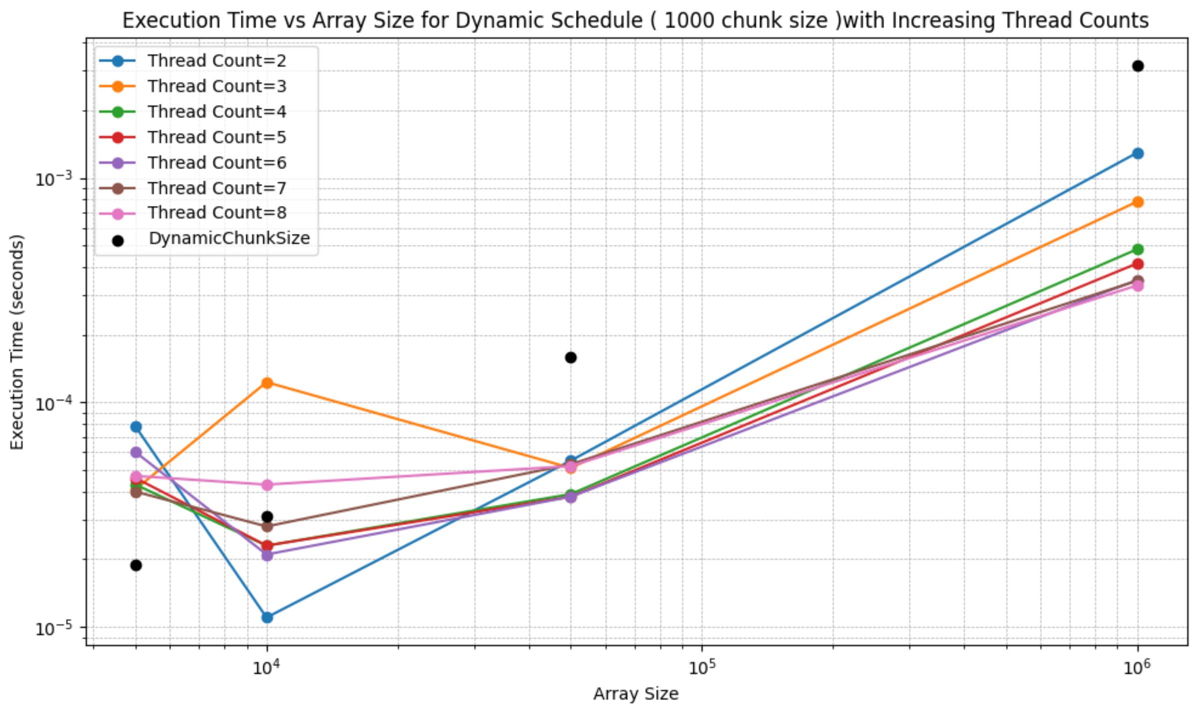
Static Schedule vs Serial as Array Size increases for different number of threads



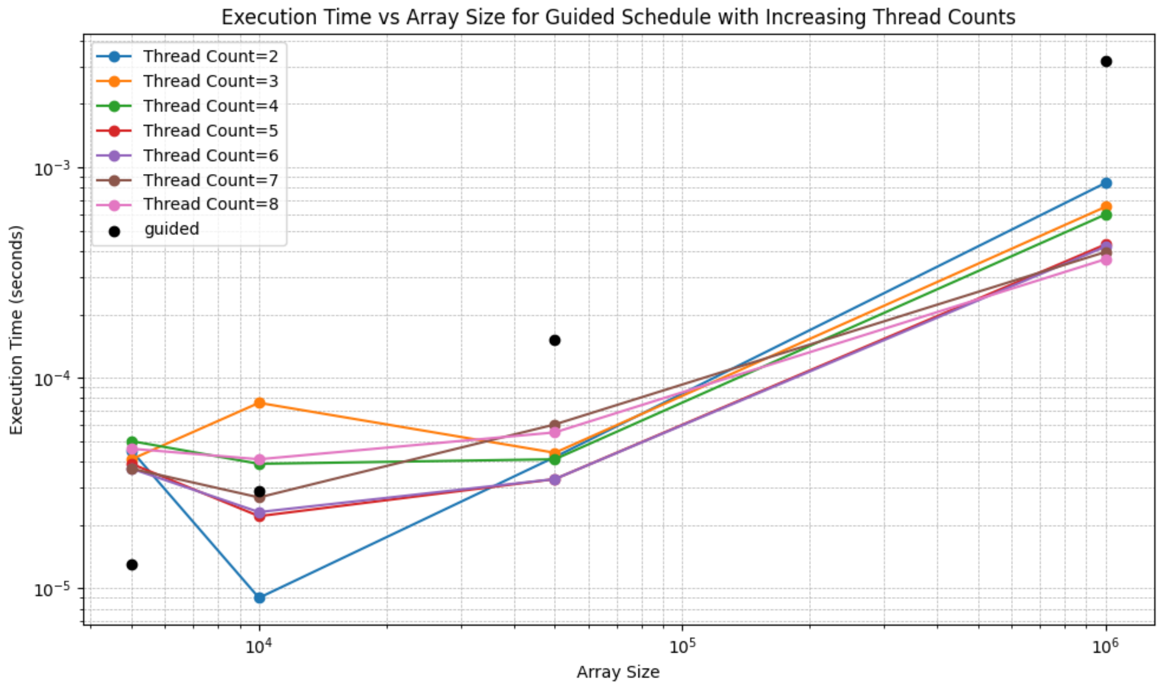
Static with chunk size 1000 Schedule vs Serial as Array Size increases for different number of threads



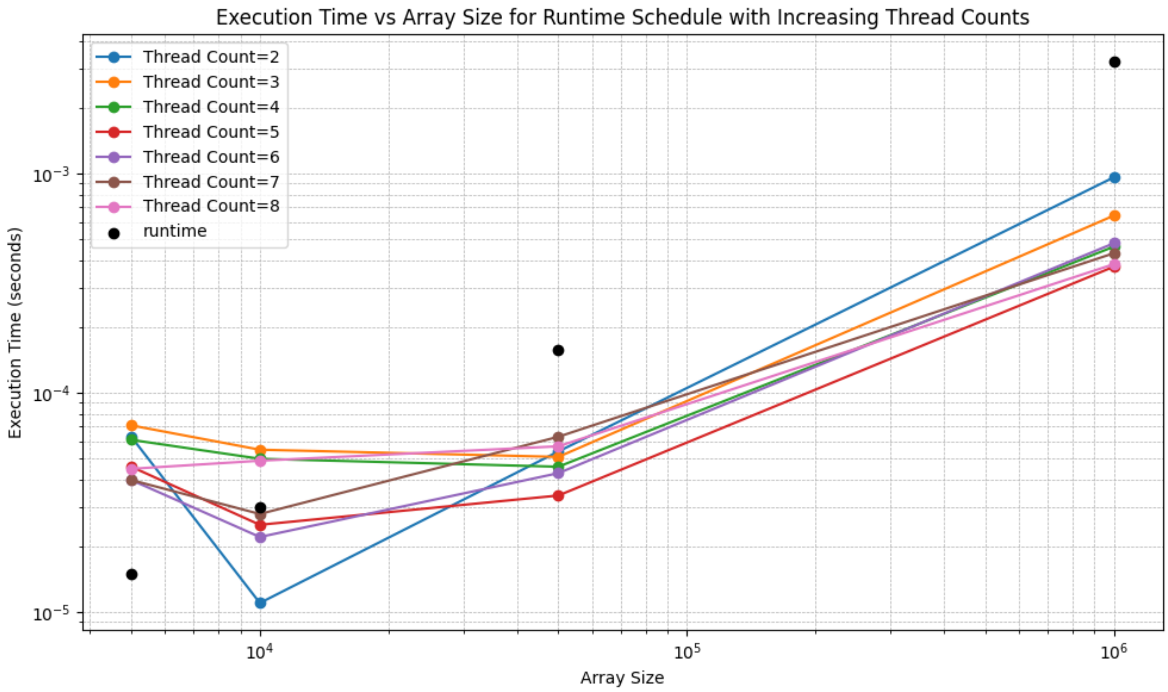
Dynamic with chunk size 1000 Schedule vs Serial as Array Size increases for different number of threads



Guided Schedule vs Serial as Array Size increases for different number of threads



Runtime Schedule vs Serial as Array Size increases for different number of threads (OMP_SCHEDULE was set to auto by default)



Execution Time vs Array Size comparing different schedules