الجامعة العربية الأمريكية
ARAB AMERICAN UNIVERSITY

Arab American University

Faculty of Engineering

Computer Systems Engineering

Parallel & Distributed Systems

Project 1

Name: Feras jarrar ID: 202111567

Supervised by: Dr. Hussein Younis

Introduction

The algorithm I have chosen is Merge Sort which is an algorithm that sorts arrays using divide and conquer the reason I have chosen this algorithm because it is easy to parallelize since the array can be divided between different pthreads.

Sequential implementation

This function will recursively call itself dividing the array by half each time and each half will be divided till there is only 1 element which is considered to be sorted then it will be sent to the merge function to sort these elements in the array.

```
24  void merge_sort(int arr[], int l, int r) {
25      if (l < r) {
26          int m = (l + r) / 2;
27          merge_sort(arr, l, m);
28          merge_sort(arr, m + 1, r);
29          merge(arr, l, m, r);
30      }
31  }
```

This function only receives sorted parts of the array when called from it arguments and sort the parts by comparing each element and putting the smaller element first in the array until all the elements are sorted then when one of the 2 arrays is all sorted it will put the rest of the elements of the other array after the sorted elements since all the rest of the elements are sorted as well.

```
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    i = 0; j = 0; k = l;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

Time Methodology

```
int main() {
    float time[5];
    for(int i=0; i<5; i++){
    int arr[SIZE];

    for(int i=0; i < SIZE; i++) {
        arr[i] = rand() % 10001;
    }

    auto start = std::chrono::high_resolution_clock::now();
    merge_sort(arr, 0, SIZE - 1);
    auto end = std::chrono::high_resolution_clock::now();
    check_sort(arr);
    printf("\nTime taken: %f seconds\n", std::chrono::duration<double>(end - start).count());
    time[i] = std::chrono::duration<double>(end - start).count();
}
    float avg = (time[0] + time[1] + time[2] + time[3] + time[4]) / 5.0;
    printf("\nAverage time taken: %f seconds\n", avg);
    return 0;
}
```

This is the main section of my code and is the start and end of the main thread the first two line of codes would create a for loop which will sort the function 5 times and get the time for each one to calculate the average time to calculate the time I used the chrono library putting the start before calling the merge sort function and the end after calling the merge sort to accurately calculate the time and putting the total time (end-start) into an array each time after the loop the average time be printed with each cycle time and it success.

Parallelization strategy

I parallelized the Algorithm by dividing the array between the threads number so each thread will have different array locations and each sort it half after that the main thread will wait until all threads are finished to merge each thread part.

The struct will have the location of the array (shared array) and each thread will have different l and r representing each thread array size.

```
struct ThreadData {
    int *arr;
    int l;
    int r;
};
```

This function will ensure that each thread will call the merge_sort and merge functions (previously mentioned) to sort it part putting the result into the array.

```
void* thread_merge_sort(void* args) {
    ThreadData *data = (ThreadData*)args;
    merge_sort(data->arr, data->l, data->r);
    return NULL;
}
```

Experiments

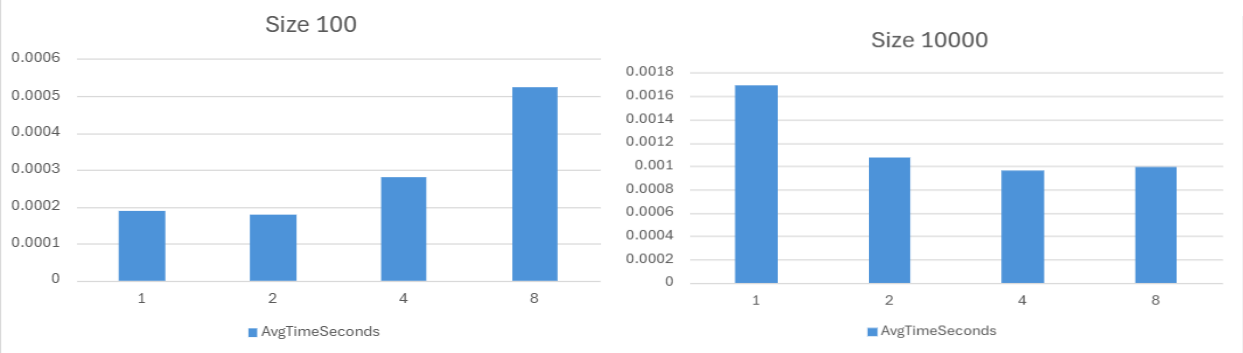
Hardware specs: 4 cores, 8 logical cores, windows 11 Home, core i5 9300h.

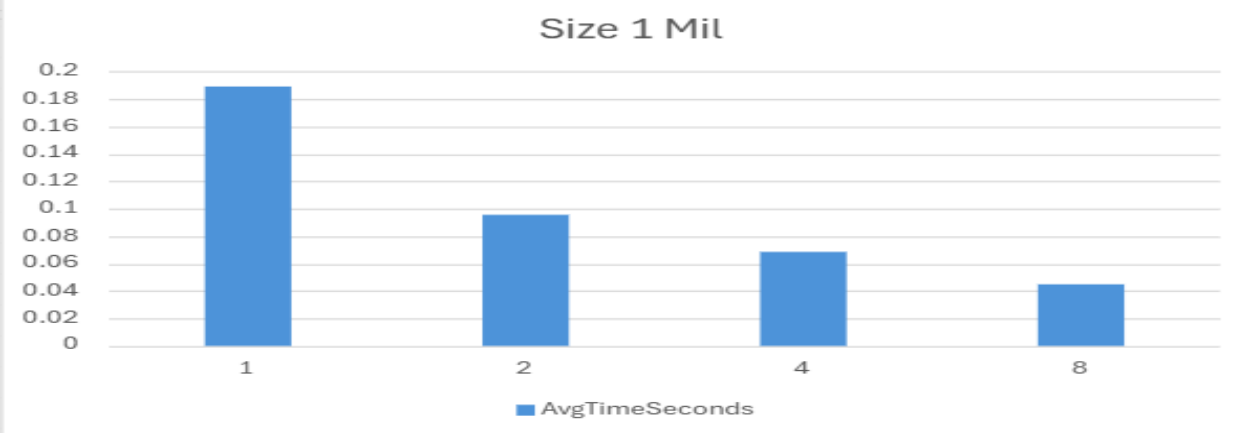
Input sizes and thread counts

- Array sizes: 100, 10000, 1000000 (1 mil)
- Thread counts: 1, 2, 4 ,8

Results

Graphs:





Size	Sequential	1 Thread	2Threads	4Threads	8Threads
100	0.000008	0.000144	0.000186	0.000247	0.000443
10000	0.001295	0.001401	0.000859	0.000748	0.000854
1 Mil	0.159338	0.161630	0.081769	0.060175	0.041831

Speed up Table:

Thread num	100	10000	1 Mil
1 Thread	0.05556X	0.924X	0.9858X
2 Threads	0.043X	1.507X	1.949X
4 Threads	0.0324X	1.731X	2.648X
8 Threads	0.018X	1.516X	3.81X

Discussion

As can be observed from the tables the speed up was lower at 1 thread because it is basically the same as the sequential one however it used extra functions and has a larger sequential part which resulted in lower speed up.

For the 8 threads the speed up was pretty good at large arrays because the size of the parallel size of the code was way larger than the sequential part of the code however it can be noted that the load for each core was different due to the number of threads and array size, and the speed was slightly different each time due to core use sometimes a core would be used for the os which would result in slower time.

Amdahl Law

Using Amdahl Law, we can calculate the Parallel fraction by using the speed up as the result and (1 – F parallel) in place of sequential part to get the values:

Size: 10000, F parallel= 0.389 Size: 1 Mil, F parallel= 0.8428

Which would result in the following predictions:

Size	Pred: 1Th	Pred: 2Th	Pred: 4Th	Pred: 8^{Th}
10000	1X	1.24X	1.41X	1.51X
1 Mil	1X	1.72X	2.718X	3.809X

It can be noted that the Prediction wasn't accurate due to unexpected factors like cores uptime the different in both algorithms (sequential parts) and such.

Conclusion

Transforming an algorithm from sequential to parallel can be challenging due to the different methods available the Synchronization and finding and observing critical sections of the code however in most cases parallelizing an algorithm won't be worth the effort in most case if the input size was small and might even return diminishing results also parallelizing an algorithm may result in special cases which also needs to be observed.

(at some point in my code my algorithm worked only for certain inputs)

Note: used ChatGPT to fix the algorithm thread parts merge since it had problems for different sizes depending on the number of threads.