PAVENDHAN N
COE18B041

MIDSEM

Develop a Multithreaded Version of Radix Sort algorithm and compare it with the performance (execution time) of bubble and insertion sort algorithms. The demonstration should display the passes of the respective sorting strategies. The comparison is against the MT Radix sort with sequential versions of Bubble and Insertion Sort Algorithm. Ensure that the testing explores large sized arrays, random distribution of elements. As an addon it would be preferred to have a data creator code which initializes an array of user required size randomly given some boundary conditions. Once done with the basic comparison, you may carry out a comparison in terms of varying number of threads and its impact on the efficiency of the parallelization.

APPROACH:

As asked in the question, the method of execution is parallelized/multithreaded radix sort and sequential bubble and insertion sort, where the size is obtained from the user in the command line and acts as the upper bound to fill the array with random numbers. There are 3 threads for each sort and for radix sort, more threads are created and executed concurrently in multithreaded fashion.

LOGIC:

BUBBLE SORT:

Simple swapping of elements if they are in wrong ordered fashion

TC:

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when the array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when the array is already sorted.

Auxiliary Space: O(1)

PASSES: N-1

INSERTION SORT:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.

TC:

This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where n is the number of items.

PASSES: N-1

RADIX SORT:

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort.
Radix sort can be applied to data that can be sorted lexicographically, be they integers, words, punch cards, playing cards, or the mail.

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. **Radix sort uses counting sort as a subroutine to sort.**

I.e. Sort input array using counting sort (or any stable sort) according to the i'th digit.

EXAMPLE-

Input	1st Pass	2nd Pass	3rd Pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457

436	**657**	**355**	**657**
720	**329**	**457**	**720**
355	**839**	**657**	**839**

DETAILED LOGIC/ALGO:
Radix sort works by placing elements into buckets based on parts of the key for the element, most commonly starting with the least significant key. After each pass, the elements are put back together again into a new set. The routine is then repeated, based on the following least significant key, until all possible keys have been processed. For sorting the keys in each pass, an additional sorting algorithm is needed(count sort, in this case). Most common algorithms for this step are **bucket sort or counting sort**, both of which have the ability to sort in linear time and are efficient in a few digits. Here, counting sort is used for the additional sorting algorithm

TC:

Worst case: O(d(n + k))

Best case: O(d(n + k))

Average case: O(d(n + k))

Space complexity: Worst case: O(n + k)

PASSES: number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

CODE:

The code consists of three threads each calling respective sorting functions. The radix thread calls the radix function where no. of thread is received from the user and based on that number, the counting sort works in parallel on those threads
The other two threads work on the bubble and insertion sort where the executions are in serial manner and for all these executions, runtimes are individually calculated using time.h library and its functions.

```c
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#define ll long long

ll int arrradix[200000];
ll int array[200000];

void generate(ll int n)
{
    srand(time(0));
    for (ll int i = 0; i < n; i++)
    {
        arrradix[i] = rand() % 101;
        array[i] = rand() % 101;
    }
}

struct data
{
    ll int arr[200000];
    ll int n;
};

struct count_sort_data
{
    struct data *d;
    ll int exp;
};

ll int getMax(ll int arr[], ll int n)
{
    ll int mx = arr[0];
    for (ll int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void print(ll int arr[], ll int n)
{
    for (ll int i = 0; i < n; i++)
        printf("%lld ", arr[i]);

    printf("\n");
}
```

```c
void *countSort(void *param)
{
    struct count_sort_data *cs = (struct count_sort_data *)param;

    ll int output[(cs->d)->n]; // output array
    ll int i, count[10] = {0};

    for (i = 0; i < (cs->d)->n; i++)
        count[((cs->d)->arr[i] / cs->exp) % 10]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (i = (cs->d)->n - 1; i >= 0; i--)
    {
        output[count[((cs->d)->arr[i] / cs->exp) % 10] - 1] = (cs->d)->arr[i];
        count[((cs->d)->arr[i] / cs->exp) % 10]--;
    }

    for (i = 0; i < (cs->d)->n; i++)
        (cs->d)->arr[i] = output[i];

    print((cs->d)->arr, (cs->d)->n);
    pthread_exit(0);
}

void *RadixSort(void *param)
{
    struct data *d = param;
    ll int m = getMax(d->arr, d->n);
    ll int thread_count = 0;

    printf("\nEnter number of threads: ");
    scanf("%lld", &thread_count);

    printf("\nPasses for Radix Sort: \n");

    struct count_sort_data cd[thread_count];
    pthread_t tid[thread_count];
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    ll int i = 0;
    for (ll int exp = 1; m / exp > 0; exp *= 10)
    {
        cd[i].d = d;
        cd[i].exp = exp;
        pthread_create(&tid[i], &attr, countSort, &cd[i]);
        i++;
    }
    for (int j = 0; j < i; ++j)
    {
        pthread_join(tid[j], NULL);
        j++;
    }
    pthread_exit(0);
}

void *BubbleSort(void *param)
{
    struct data *d = param;
    printf("\nPasses for Bubble Sort: \n");
    for (ll int i = 1; i < d->n; i++)
    {
        ll int tcheck = 0;
        for (ll int j = 0; j < d->n - i; j++)
        {
            if (d->arr[j] > d->arr[j + 1])
            {
                ll int temp = d->arr[j];
                d->arr[j] = d->arr[j + 1];
                d->arr[j + 1] = temp;
                tcheck = 1;
            }
        }
        print(d->arr, d->n);
        if (tcheck == 0)
            break;
    }

    pthread_exit(0);
}

void *InsertionSort(void *param)
{
    struct data *d = param;
    printf("\nPasses for Insertion Sort: \n");
    for (ll int i = 1; i < d->n; i++)
    {
        for (ll int j = i; j > 0; j--)
        {
            if (d->arr[j - 1] > d->arr[j])
            {
                ll int temp = d->arr[j];
                d->arr[j] = d->arr[j - 1];
                d->arr[j - 1] = temp;
            }
            else
                break;
        }
        print(d->arr, d->n);
    }

    pthread_exit(0);
}

ll int main(ll int argc, char *argv[])
{
    clock_t t1, t2;
    if (argc != 2)
    {
        printf("Usage: ./a.out <SIZE>\nAborting.\n");
        exit(EXIT_FAILURE);
    }

    else
    {
        ll int n = atoi(argv[1]);
        generate(n);

        struct data d[3];
        for (ll int i = 0; i < n; i++)
        {
            d[0].arr[i] = arrradix[i];
            d[1].arr[i] = arrradix[i];
            d[2].arr[i] = arrradix[i];
        }
        d[0].n = n;
        d[1].n = n;
        d[2].n = n;

        pthread_t tid[2];
        pthread_attr_t attr;
        pthread_attr_init(&attr);


//-------------------------------------------------------------------------------
--RADIX
```

```
printf("-----------------------------------------------------------------------------------------RADIX");
        t1 = clock();
        pthread_create(&tid[0], &attr, RadixSort, &d[0]);
        pthread_join(tid[0], NULL);
        t2 = clock();

        printf("\nRadix Sort: ");
        for (ll int i = 0; i < n; i++)
            printf("%lld ", d[0].arr[i]);

        printf("\nRun time: %f\n\n", (t2 - t1) / (double)CLOCKS_PER_SEC);


//----------------------------------------------------------------------------------BUBBLE


printf("-----------------------------------------------------------------------------------------BUBBLE");
        t1 = clock();
        pthread_create(&tid[1], &attr, BubbleSort, &d[1]);
        pthread_join(tid[1], NULL);
        t2 = clock();

        printf("\nBubble Sort: ");
        for (ll int i = 0; i < n; i++)
            printf("%lld ", d[1].arr[i]);

        printf("\nRun time: %f\n\n", (t2 - t1) / (double)CLOCKS_PER_SEC);


//----------------------------------------------------------------------------------INSERTION

printf("-----------------------------------------------------------------------------------------INSERTION");
        t1 = clock();
        pthread_create(&tid[2], &attr, InsertionSort, &d[2]);
        pthread_join(tid[2], NULL);
        t2 = clock();

        printf("\nInsertion Sort: ");
        for (ll int i = 0; i < n; i++)
            printf("%lld ", d[2].arr[i]);
        printf("\nRun time: %f\n\n", (t2 - t1) / (double)CLOCKS_PER_SEC);
    }
```

RUNTIME:

DATASET SIZE:

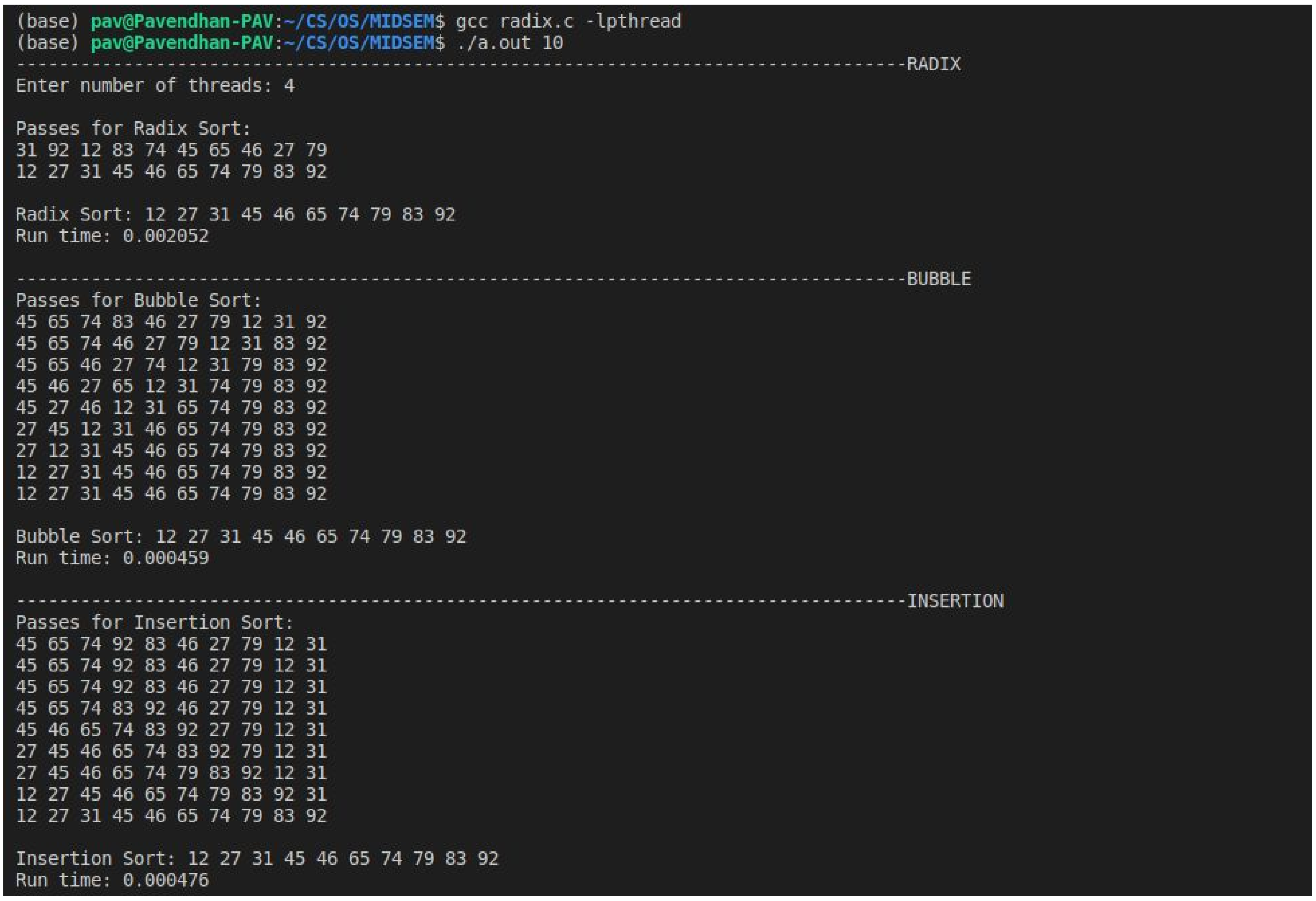
For small array size it can be seen that the multithreaded version takes more time than those of the serialised versions of sort. Well, this is not what is expected. This happens because of overhead like creating the thread and communication within those threads over processing of data. BUT in the larger picture where the dataset contains lakhs or millions of data, the parallelisation PREVAILS i.e. the time taken for sorting by the radix sort is so less compared to the serialised bubble and insertion sort

NUMBER OF THREADS:

It is observed that at SMALL DATASET, higher the no. of threads faster the performance, BUT for LARGE DATASETS, the performance is **HIGH for lower no. of threads** than higher number. This is because of the overheads caused by **intercommunication between threads rather than processing data.**
I.e. for 1 vs 2 vs 4 threads , **2 or 4** number of threads **perform best for small-medium datasets,** whereas **for Large datasets - 1 or 2** number of threads **perform best.**

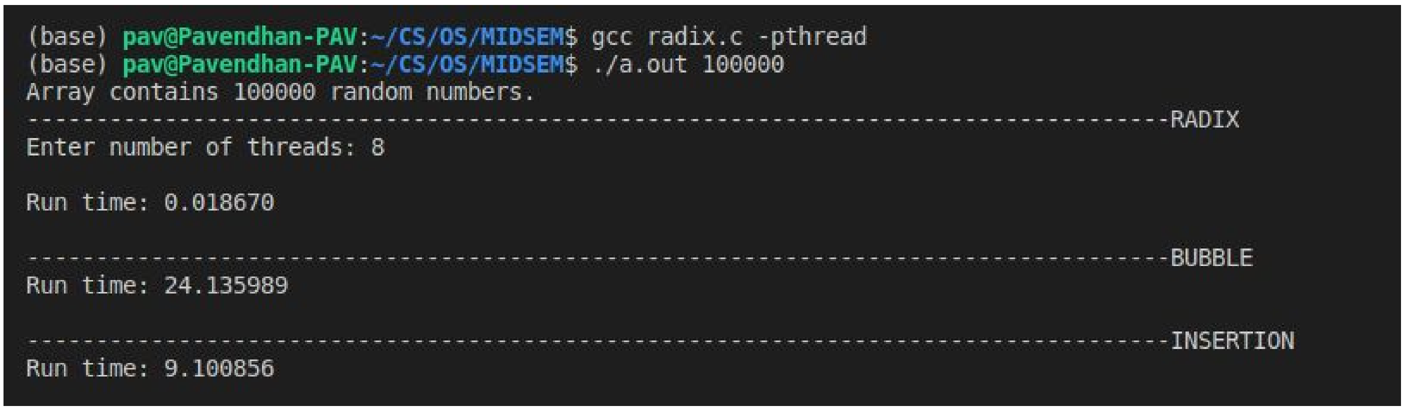
OUTPUT:

SMALL DATASET:



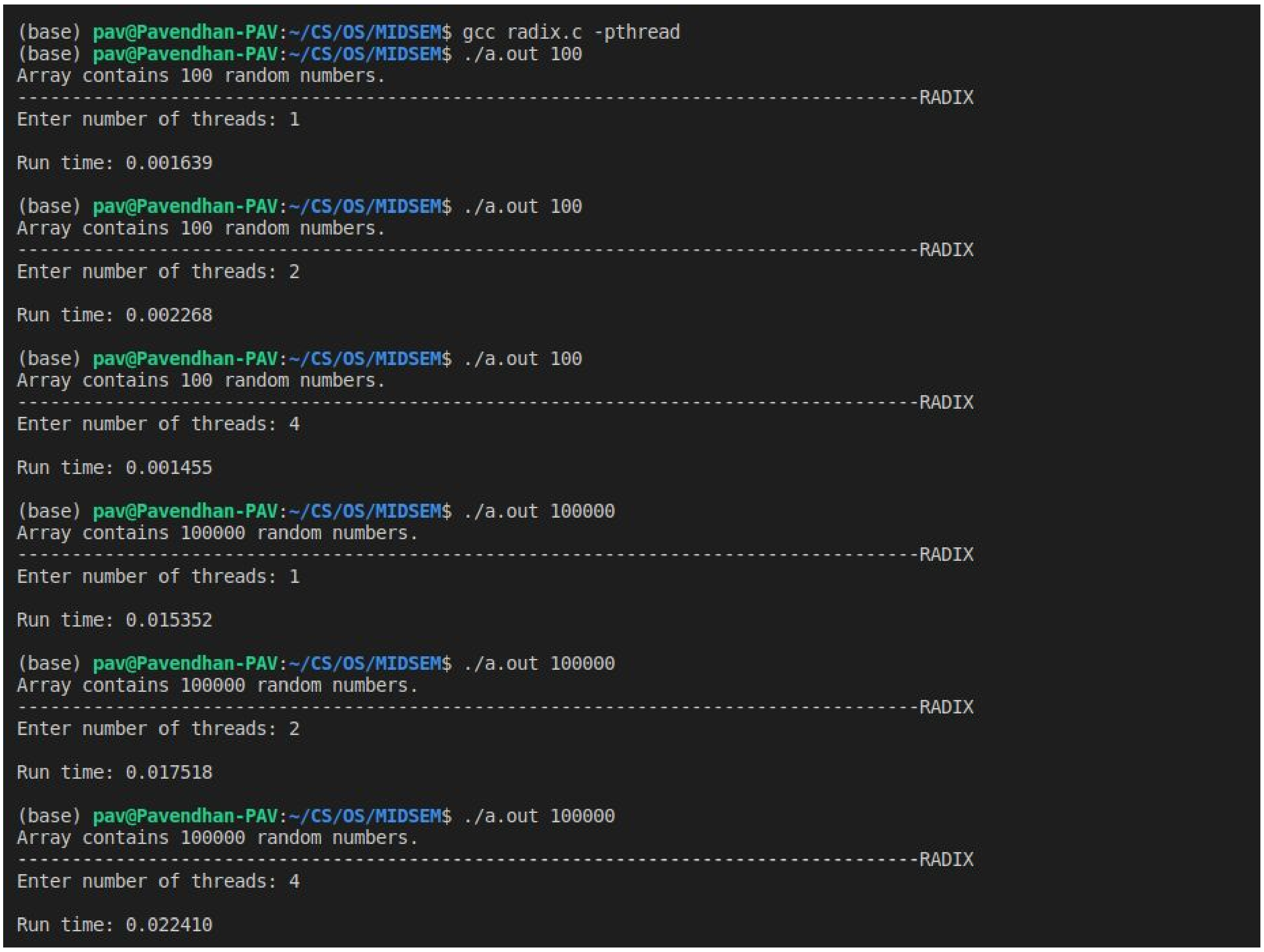
```
(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ gcc radix.c -lpthread
(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 10
-----------------------------------------------------------------------------------RADIX
Enter number of threads: 4

Passes for Radix Sort:
31 92 12 83 74 45 65 46 27 79
12 27 31 45 46 65 74 79 83 92

Radix Sort: 12 27 31 45 46 65 74 79 83 92
Run time: 0.002052

-----------------------------------------------------------------------------------BUBBLE
Passes for Bubble Sort:
45 65 74 83 46 27 79 12 31 92
45 65 74 46 27 79 12 31 83 92
45 65 46 27 74 12 31 79 83 92
45 46 27 65 12 31 74 79 83 92
45 27 46 12 31 65 74 79 83 92
27 45 12 31 46 65 74 79 83 92
27 12 31 45 46 65 74 79 83 92
12 27 31 45 46 65 74 79 83 92
12 27 31 45 46 65 74 79 83 92

Bubble Sort: 12 27 31 45 46 65 74 79 83 92
Run time: 0.000459

-----------------------------------------------------------------------------------INSERTION
Passes for Insertion Sort:
45 65 74 92 83 46 27 79 12 31
45 65 74 92 83 46 27 79 12 31
45 65 74 92 83 46 27 79 12 31
45 65 74 83 92 46 27 79 12 31
45 46 65 74 83 92 27 79 12 31
27 45 46 65 74 83 92 79 12 31
27 45 46 65 74 79 83 92 12 31
12 27 45 46 65 74 79 83 92 31
12 27 31 45 46 65 74 79 83 92

Insertion Sort: 12 27 31 45 46 65 74 79 83 92
Run time: 0.000476
```

LARGE DATASET:



```
(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ gcc radix.c -pthread
(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100000
Array contains 100000 random numbers.
-----------------------------------------------------------------------------------RADIX
Enter number of threads: 8

Run time: 0.018670

-----------------------------------------------------------------------------------BUBBLE
Run time: 24.135989

-----------------------------------------------------------------------------------INSERTION
Run time: 9.100856
```

VARIANCE IN RUNTIME FOR DIFFERENT NUMBER OF THREADS:



```
(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ gcc radix.c -pthread
(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100
Array contains 100 random numbers.
-------------------------------------------------------------------------------------RADIX
Enter number of threads: 1

Run time: 0.001639

(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100
Array contains 100 random numbers.
-------------------------------------------------------------------------------------RADIX
Enter number of threads: 2

Run time: 0.002268

(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100
Array contains 100 random numbers.
-------------------------------------------------------------------------------------RADIX
Enter number of threads: 4

Run time: 0.001455

(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100000
Array contains 100000 random numbers.
-------------------------------------------------------------------------------------RADIX
Enter number of threads: 1

Run time: 0.015352

(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100000
Array contains 100000 random numbers.
-------------------------------------------------------------------------------------RADIX
Enter number of threads: 2

Run time: 0.017518

(base) pav@Pavendhan-PAV:~/CS/OS/MIDSEM$ ./a.out 100000
Array contains 100000 random numbers.
-------------------------------------------------------------------------------------RADIX
Enter number of threads: 4

Run time: 0.022410
```