Midsem OS
CED18I063
TIME: 10:00AM

Question:
Dear SUJOY DATTA,
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.

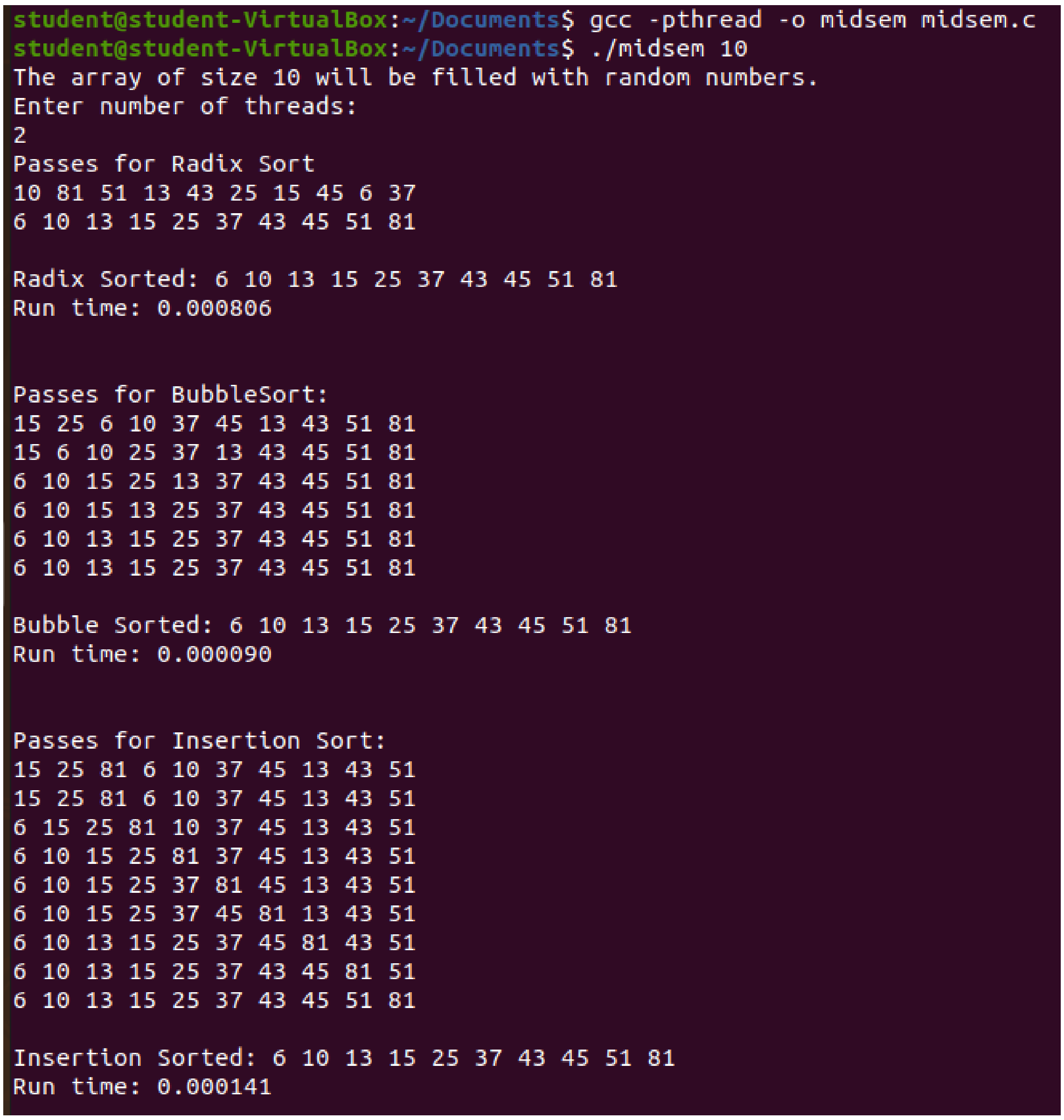
Code:
```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
int arrradix[10000];
int array[10000];
struct data
{
	int arr[10000];
	int n;
};
struct count_sort_data
{
	struct data *d;
	int exp;
};

int getMax(int arr[], int n)
{
	int mx = arr[0];
	for (int i = 1; i < n; i++)
		if (arr[i] > mx)
			mx = arr[i];
	return mx;
}
void print(int arr[], int n)
{
	for (int i = 0; i < n; i++)
		printf("%d ", arr[i]);

	printf("\n");
}

void *countSort(void *param)
{
	struct count_sort_data *csd = (struct count_sort_data *)param;

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

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

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

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

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

void *RadixSort(void *param)
{

	struct data *d = param;
	int m = getMax(d->arr, d->n);

	printf("Enter number of threads: \n");
	int noofthreads = 0;
	scanf("%d",&noofthreads);
	struct count_sort_data cd[noofthreads];
	pthread_t tid[noofthreads];
	pthread_attr_t attr;
	pthread_attr_init(&attr);
	printf("Passes for Radix Sort\n");
	int i = 0;
	for (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 BubbleSort: \n");
	for (int i = 1; i < d->n; i++)
	{
		int tcheck = 0;
		for (int j = 0; j < d->n - i; j++)
		{
			if (d->arr[j] > d->arr[j + 1])
			{
				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 (int i = 1; i < d->n; i++)
	{
		for (int j = i; j > 0; j--)
		{
			if (d->arr[j - 1] > d->arr[j])
			{
				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);
}
```

```
int main(int argc, char *argv[])
{
	clock_t t1, t2;
	srand(time(0));
	if (argc < 2)
		printf("Wrong number of parameters. Usage: ./<binary><size of array> \n");
	else
	{
		int n = atoi(argv[1]);
		printf("The array of size %d will be filled with random numbers.\n",n);
		for (int i = 0; i < n; i++)
		{
			arrradix[i] = rand()%101;
			array[i] = rand()%101;
		}

		struct data d[3];
		for (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[3];
		pthread_attr_t attr;
		pthread_attr_init(&attr);


		t1 = clock();
		pthread_create(&tid[0], &attr, RadixSort, &d[0]);
		pthread_join(tid[0], NULL);
		t2 = clock();

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

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

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

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

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

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

                printf("\nInsertion Sorted: ");
                for (int i = 0; i < n; i++)
                        printf("%d ", d[2].arr[i]);

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

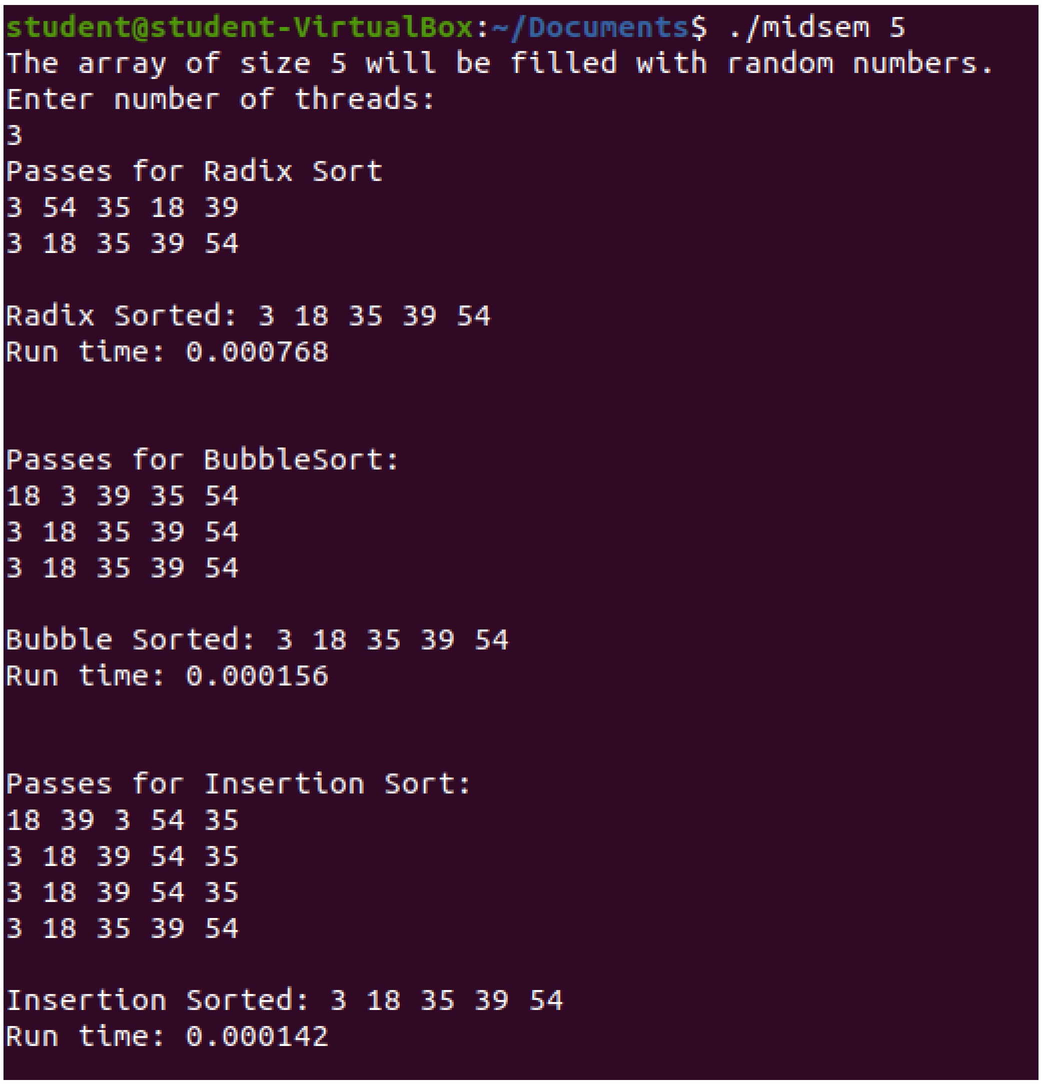
OUTPUT

1.



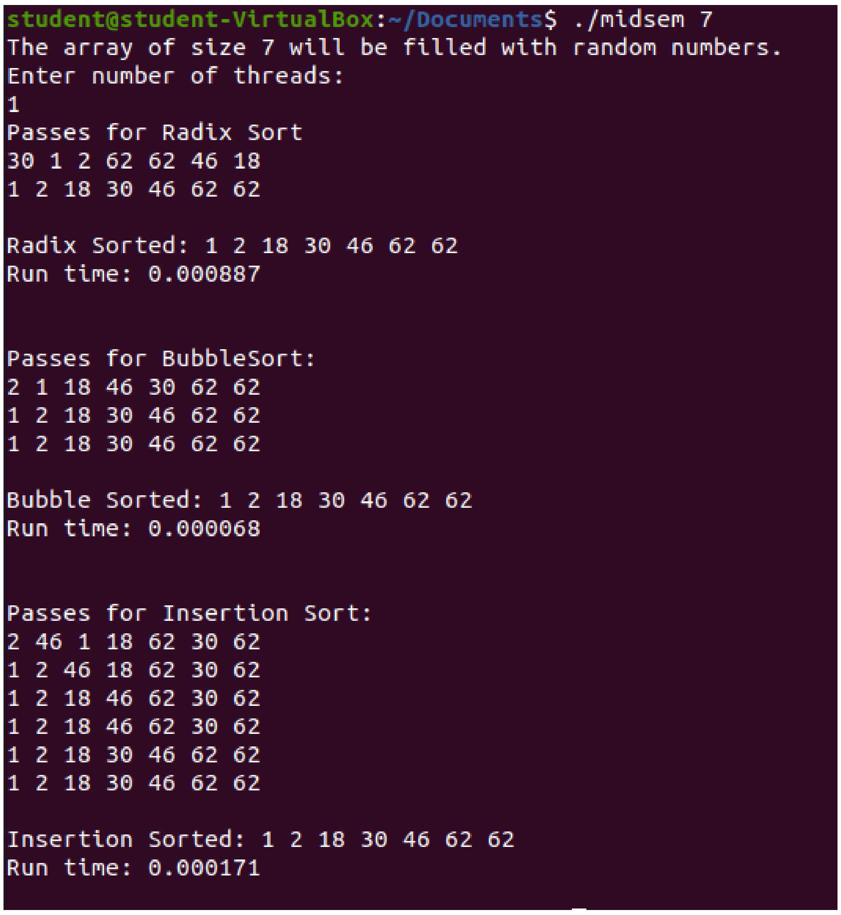
```
student@student-VirtualBox:~/Documents$ gcc -pthread -o midsem midsem.c
student@student-VirtualBox:~/Documents$ ./midsem 10
The array of size 10 will be filled with random numbers.
Enter number of threads:
2
Passes for Radix Sort
10 81 51 13 43 25 15 45 6 37
6 10 13 15 25 37 43 45 51 81

Radix Sorted: 6 10 13 15 25 37 43 45 51 81
Run time: 0.000806


Passes for BubbleSort:
15 25 6 10 37 45 13 43 51 81
15 6 10 25 37 13 43 45 51 81
6 10 15 25 13 37 43 45 51 81
6 10 15 13 25 37 43 45 51 81
6 10 13 15 25 37 43 45 51 81
6 10 13 15 25 37 43 45 51 81

Bubble Sorted: 6 10 13 15 25 37 43 45 51 81
Run time: 0.000090


Passes for Insertion Sort:
15 25 81 6 10 37 45 13 43 51
15 25 81 6 10 37 45 13 43 51
6 15 25 81 10 37 45 13 43 51
6 10 15 25 81 37 45 13 43 51
6 10 15 25 37 81 45 13 43 51
6 10 15 25 37 45 81 13 43 51
6 10 13 15 25 37 45 81 43 51
6 10 13 15 25 37 43 45 81 51
6 10 13 15 25 37 43 45 51 81

Insertion Sorted: 6 10 13 15 25 37 43 45 51 81
Run time: 0.000141
```

2.



```
student@student-VirtualBox:~/Documents$ ./midsem 5
The array of size 5 will be filled with random numbers.
Enter number of threads:
3
Passes for Radix Sort
3 54 35 18 39
3 18 35 39 54

Radix Sorted: 3 18 35 39 54
Run time: 0.000768


Passes for BubbleSort:
18 3 39 35 54
3 18 35 39 54
3 18 35 39 54

Bubble Sorted: 3 18 35 39 54
Run time: 0.000156


Passes for Insertion Sort:
18 39 3 54 35
3 18 39 54 35
3 18 39 54 35
3 18 35 39 54

Insertion Sorted: 3 18 35 39 54
Run time: 0.000142
```

3.



```
student@student-VirtualBox:~/Documents$ ./midsem 7
The array of size 7 will be filled with random numbers.
Enter number of threads:
1
Passes for Radix Sort
30 1 2 62 62 46 18
1 2 18 30 46 62 62

Radix Sorted: 1 2 18 30 46 62 62
Run time: 0.000887


Passes for BubbleSort:
2 1 18 46 30 62 62
1 2 18 30 46 62 62
1 2 18 30 46 62 62

Bubble Sorted: 1 2 18 30 46 62 62
Run time: 0.000068


Passes for Insertion Sort:
2 46 1 18 62 30 62
1 2 46 18 62 30 62
1 2 18 46 62 30 62
1 2 18 46 62 30 62
1 2 18 30 46 62 62
1 2 18 30 46 62 62

Insertion Sorted: 1 2 18 30 46 62 62
Run time: 0.000171
```

Explanation:

What is radix sort?
Distributes elements of a data set into buckets, based on the element's key, beginning with the least significant part of the key. After each pass, the elements are collected from the buckets, keeping the internal order, and then redistributed according to the next most significant part of the key.

Time complexity
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)

Algorithm
Radix sort is a stable distribution sorting algorithm, that can sort data in linear time. In mathematics, radix is the base of numeral system. For example, the decimal system have a radix of ten, as it include digits 0 − 9. 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 element 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 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 ability to sort in linear time and are efficient on few digits. There are two different versions of radix sort. As previously mentioned, least significant key (LSD) sorts a data set using the least significant key as a starting point and moving upwards. In contrast, most significant key (MSD) start with the most significant key and moves downwards

Code:
As guided by the question, I created threads for comparing multi threaded radix with bubble and insertion sort. The pthread library was used for the execution. Also, the size of the array is provided by the user which is filled by rand() with numbers between 0-100. As an additional support of providing the depth or the number of threads is also provided. In the radix sort function, threads are being created as provided by the user and the array is divided into multiple parts to pass into count sort support threaded function. For the hashing indexed method, arrayradix[] is also created which helps in the ease of sorting.

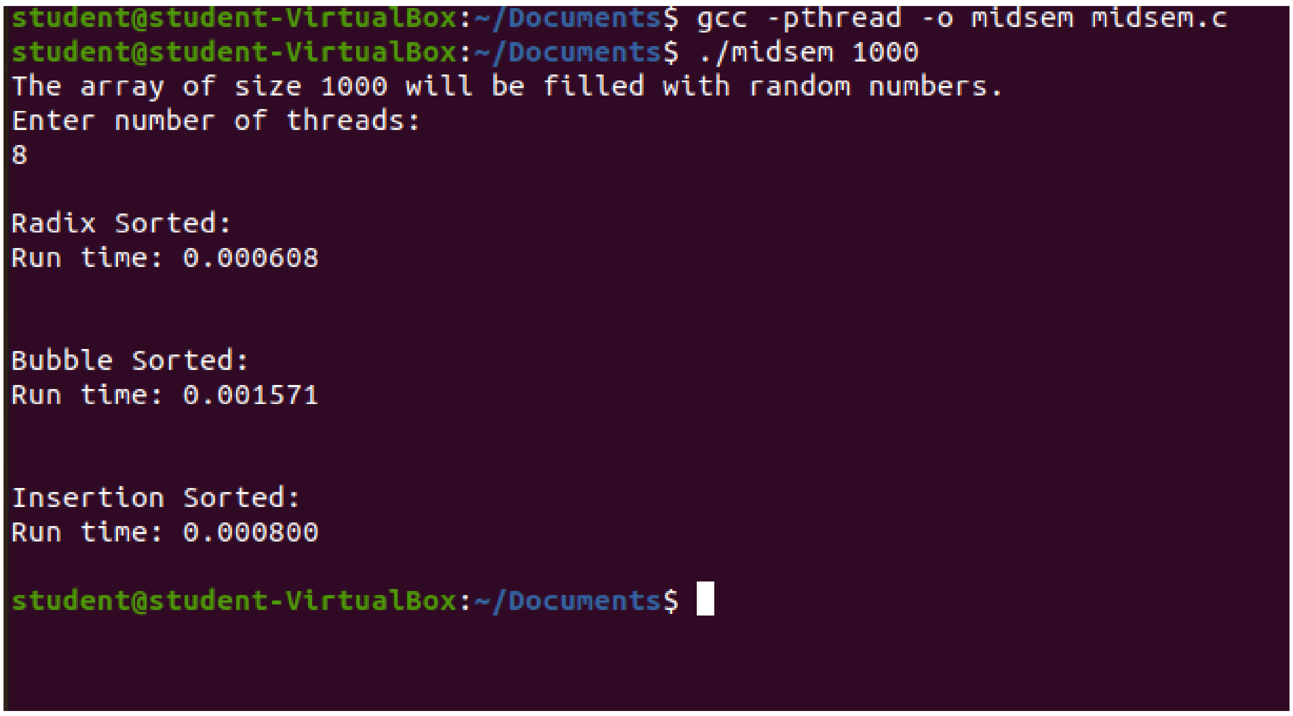
As far as the bubble and insertion sort algorithms are concerned, I created 2 separate threads for that. There is no variation in their algorithms, also provision has been made to display the passes for each algorithm. Noticeable comparison is done using clock() differences.

Larger Picture
To ensure that a parallel algorithm fulfils its intent, the processors needs to be synchronized. If not, two separate concurrent threads might access the same data and try to processes it, known as race condition. As a result, the data could end up corrupt or cause the application to either fail its purpose or provide false data. To avoid this, techniques like read- and write privileges, semaphores or mutual exclusion can be used.
As observed by the outputs the multi threaded aspect should take a lesser amount of time as compared to serial sorting algorithms, but a phenomenon called **parallel slowdown** occurs in this case.

In this, the parallelisation of an algorithm makes the execution of an application to run slower. This is most often related to the communication between processing threads. After a certain number threads have been created, in relation to the purpose of the application, threads will spend most of its time communicating with each other rather than processing data. With the creation of each processing thread, comes a cost in terms of an overhead. When the total cost of the overhead becomes a greater factor than the extra resources it provides, the parallel slowdown occurs.

A key aspect of an efficient parallel implementation is to distribute the workload evenly over the processors available. While the processor with the smaller part will finish its execution fast in relation to the second one, the overall execution time for the application will be dependant on processor with the greatest factor. Because of this, a parallel algorithm needs to ensure even balanced workload in order to maximize the performance. **For larger size arrays, that do not violate memory violation, multithreaded algorithm will perform better than the serial execution of sorting algorithms.**

Another noticeable aspect, was that when the user enters a large number of threads, t**he time taken by radix sort is the least**. In distributed memory systems, data movement between processors can cause problems. The cost of moving data between processors can possibly become a greater factor than the execution time. Due to this, the task of moving data between processors should be kept to the minimum.

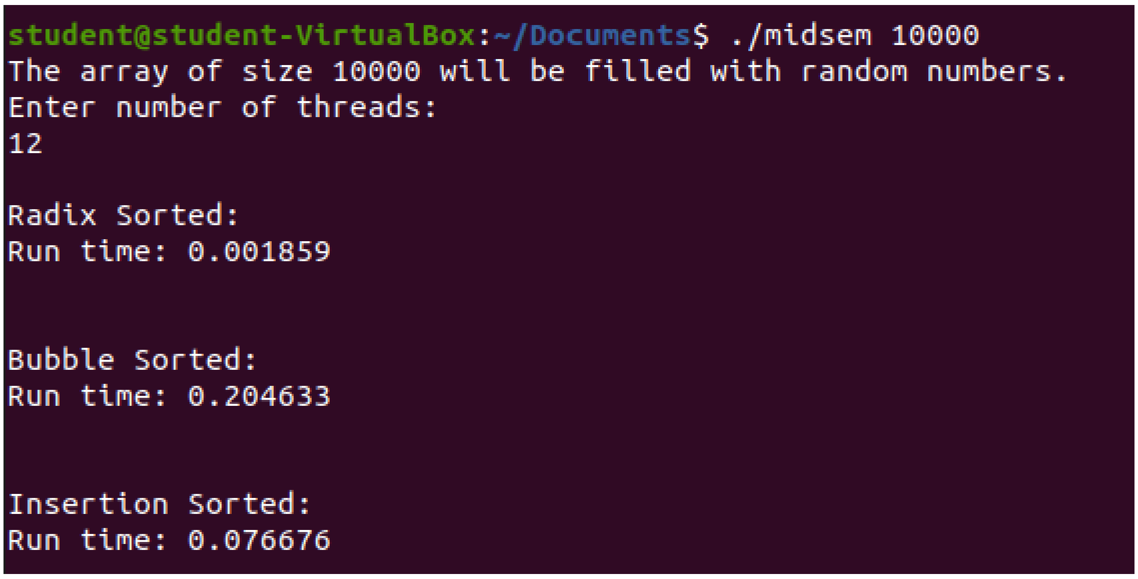


```
student@student-VirtualBox:~/Documents$ gcc -pthread -o midsem midsem.c
student@student-VirtualBox:~/Documents$ ./midsem 1000
The array of size 1000 will be filled with random numbers.
Enter number of threads:
8

Radix Sorted:
Run time: 0.000608


Bubble Sorted:
Run time: 0.001571


Insertion Sorted:
Run time: 0.000800

student@student-VirtualBox:~/Documents$
```

Comparison

1. For a large sized array(n=1000)

Time taken by Radix Sort (multithreaded)	<	Time taken by Insertion Sort	<	Time taken by bubble sort



```
student@student-VirtualBox:~/Documents$ ./midsem 10000
The array of size 10000 will be filled with random numbers.
Enter number of threads:
12

Radix Sorted:
Run time: 0.001859


Bubble Sorted:
Run time: 0.204633


Insertion Sorted:
Run time: 0.076676
```