DESIGN AND ANALYSIS OF ALGORITHMS

Presented to Sir Waheed Abro

Abstract

The complete report for the project.

Made by

Munir Abbasi 20K-0244   Mustafa Zahid 20K-1045

Contents

Abstract 2
Introduction 2
Project Programming Design 2
  Parallel fast sorting 2
Experimental Setup 7
Results and discussions 9
  Space complexity restrictions 10
  Time complexity restrictions 10
Conclusion 12
References 12

Abstract

This document is the proper documentation of the project developed for the course of Design and Analysis of Algorithms submitted to our instructor Dr. Waheed Abro. The project includes an introduction of the project and the group members, the project design of programming, the experimental setup, results and discussions, conclusions and references used.

Introduction

This project includes a visual representation of multiple sorting algorithms. These are executed on test files with input. The visualization of the sorts is a key feature. We receive the execution time, and the space complexity. The group members are Mustafa Zahid 20K-1045 and Munir Abbasi 20K-0244.

Project Programming Design

The programming language chosen was python. This language was used since python is easy to use and its libraries can be used to make an interactive GUI and manage the backend as well. We were also able to implement the algorithm for fast parallel sort. This was coded in C++ language as multiple processors were incorporated using threads and the OpenMP library. All relevant research paper logic are matched with our code in images.

Parallel fast sorting

```
int max_threads=dimensions*cbrt(dimensions);
num_buckets=cbrt(dimensions);
```

```
buckets = (struct bucket *) calloc(dimensions-cbrt(dimensions), sizeof(struct bucket));
```

Algorithm 4—parallel sort using $n^{4/3}$ processors

1. Partition the n input numbers into $n^{2/3}$ groups each having $n^{1/3}$ elements.

```
int buck_sum=0;
for (j=cur_id; j< num_buckets*num_threads; j=j+num_threads){
    buck_sum += buckets[j].element;
}
```

2. Within each group do
   For each element, j, determine count[j] = # of i such that $c_i < c_j$) + (# of $i \le j$ such that $c_i = c_j$).

```
for (i=0; i< dimensions ;i++){
    j = A[i]/w;
    if (j > num_buckets -1)
        j = num_buckets-1;
    k = j + cur_id*num_buckets;
    b_index = buckets[k].curr_ind++;
    B[b_index] = A[i];
}
```

```
    int j,k;
int local_index;
for(int i=0;i<=local_index;i++)
binarySearch(A,j,k,i);
```

```
.nt binarySearch(int arr[], int l, int r, int x)

    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
```

E:\MUNIRABBASI\university\semester 5\algo\ALGOCOde\parallelsort.cpp - Dev-C++ 5.11

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 4.9.2 64-bit Profiling

(globals)

parallelsort.cpp

```

    buckets = (struct bucket *) calloc(dimensions-cbrt(dimensions), sizeof(struct bucket));//define the bucket structure used
    t1 = omp_get_wtime();
        int j,k;
    int local_index;
#pragma omp parallel
{
    num_threads = omp_get_num_threads();
    int real_bucket_index; // [0 : num_buckets * num_threads)
    int cur_id = omp_get_thread_num();
    workload = dimensions/num_threads;//as per paper divide total buckets numbers per processor
    int prevoius_index;
   #pragma omp for private(i,local_index)
    for (i=0; i< dimensions;i++){
        local_index = A[i]/w;
        if (local_index > num_buckets-1)
                local_index = num_buckets-1;
        real_bucket_index = local_index + cur_id*num_buckets;
        buckets[real_bucket_index].element++;//implement on the local bucket registers
    }
    int buck_sum=0;
    for (j=cur_id; j< num_buckets*num_threads; j=j+num_threads){
        buck_sum += buckets[j].element;
    }
    global_n_elem[cur_id]=buck_sum;
    #pragma omp barrier
    #pragma omp master
```

E:\MUNIRABBASI\university\semester 5\algo\ALGOCOde\parallelsort.cpp - Dev-C++ 5.11

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 4.9.2 64-bit Profiling

(globals)

parallelsort.cpp

```
    #pragma omp barrier
    #pragma omp master
    {
     for (j=1; j<num_buckets; j++){
                global_starting_position[j] = global_starting_position[j-1] + global_n_elem[j-1];
                buckets[j].start_ind = buckets[j-1].start_ind + global_n_elem[j-1];
                buckets[j].curr_ind = buckets[j-1].curr_ind + global_n_elem[j-1];//formula application for calculation of the offset

        }
    }
    #pragma omp barrier
    for (j=cur_id+num_buckets; j< num_buckets*num_threads; j=j+num_threads){
        int prevoius_index = j-num_buckets;//barrier for each bucket
        buckets[j].start_ind = buckets[prevoius_index].start_ind + buckets[prevoius_index].element;
        buckets[j].curr_ind = buckets[prevoius_index].curr_ind + buckets[prevoius_index].element;
          // binarySearch(A,j,k,i)
    }
    #pragma omp barrier
    #pragma omp for private(i, b_index)
    for (i=0; i< dimensions ;i++){
        j = A[i]/w;
        if (j > num_buckets -1)
                j = num_buckets-1;//final bucket sort
        k = j + cur_id*num_buckets;
        b_index = buckets[k].curr_ind++;
        B[b_index] = A[i];
    }
```

Then evaluate count[j] = the sum (over k) of count[j, k].

8. Do a bucket sort of all n elements.
9. END of Algorithm 4.

```c
#pragma omp for private(i)
    for(i=0; i<num_buckets; i++)
        qsort(B+global_starting_position[i], global_n_elem[i], sizeof(int), cmpfunc);
}
   total = omp_get_wtime() - t1;
    tmp = A;
    A = B;
    B = tmp;
   if (dimensions <= 40) {
        printf("A \n");
        for(i=0;i<dimensions;i++) {
            printf("%d ",A[i]);
        }
        printf("\n");
    }
    printf("Sorting %d elements took %f seconds\n", dimensions,total);
    int sorted = 1;
    for(i=0;i<dimensions-1;i++) {
```

Experimental results

Parallel sort code execution

```
E:\MUNIRABBASI\university\semester 5\algo\ALGOCOde\parallelsort.exe
Give length of array to sort
99
number of buckets is 4

         Unsorted data
41       67      34      0       69      24      78      58      62      64      5       45      81      27      61
         91      95      42      27      36      91      4       2       53      92      82      21      16      18
         95      47      26      71      38      69      12      67      99      35      94      3       11      22
         33      73      64      41      11      53      68      47      44      62      57      37      59      23
         41      29      78      16      35      90      42      88      6       40      42      64      48      46
         5       90      29      70      50      6       1       93      48      29      23      84      54      56
         40      66      76      31      8       44      39      26      23      37      38      18      82      29

        Sorted final bucket is
0        1       2       3       4       5       5       6       6       8       11      11      12      16      16
         18      18      21      22      23      23      23      24      26      26      27      27      29      29
         29      29      31      33      34      35      35      36      37      37      38      38      39      40
         40      41      41      41      42      42      42      44      44      45      46      47      47      48
         48      50      53      53      54      56      57      58      59      61      62      62      64      64
         64      66      67      67      68      69      69      70      71      73      76      78      78      81
         82      82      84      88      90      90      91      91      92      93      94      95      95      99


Sorting 99 elements took 0.000000 seconds

--------------------------------
Process exited after 1.499 seconds with return value 0
Press any key to continue . . .
```

E:\MUNIRABBASI\university\semester 5\algo\ALGOCOde\parallelsort.exe

```
Give length of array to sort
43
number of buckets is 3

         Unsorted data
41      67      34      0       69      24      78      58      62      64      5       45      81      27      61
        91      95      42      27      36      91      4       2       53      92      82      21      16      18
        95      47      26      71      38      69      12      67      99      35      94      3       11      22

        Sorted final bucket is
0        2       3       4       5       11      12      16      18      21      22      24      26      27      27
         34      35      36      38      41      42      45      47      53      58      61      62      64      67
         67      69      69      71      78      81      82      91      91      92      94      95      95      99


Sorting 43 elements took 0.000000 seconds

--------------------------------
Process exited after 0.5775 seconds with return value 0
Press any key to continue . . .
```

```
Give length of array to sort
399
number of buckets is 7

         Unsorted data
41      67      34      0       69      24      78      58      62      64      5       45      81      27      61
        91      95      42      27      36      91      4       2       53      92      82      21      16      18
        95      47      26      71      38      69      12      67      99      35      94      3       11      22
        33      73      64      41      11      53      68      47      44      62      57      37      59      23
        41      29      78      16      35      90      42      88      6       40      42      64      48      46
        5       90      29      70      50      6       1       93      48      29      23      84      54      56
        40      66      76      31      8       44      39      26      23      37      38      18      82      29
        41      33      15      39      58      4       30      77      6       73      86      21      45      24
        72      70      29      77      73      97      12      86      90      61      36      55      67      55
        74      31      52      50      50      41      24      66      30      7       91      7       37      57
        87      53      83      45      9       9       58      21      88      22      46      6       30      13
        68      0       91      62      55      10      59      24      37      48      83      95      41      2
        50      91      36      74      20      96      21      48      99      68      84      81      34      53
        99      18      38      0       88      27      67      28      93      48      83      7       21      10
        17      13      14      9       16      35      51      0       49      19      56      98      3       24
        8       44      9       89      2       95      85      93      43      23      87      14      3       48
        0       58      18      80      96      98      81      89      98      9       57      72      22      38
        92      38      79      90      57      58      91      15      88      56      11      2       34      72
        55      28      46      62      86      75      33      69      42      44      16      81      98      22
        51      21      99      57      76      92      89      75      12      0       10      3       69      61
        88      1       89      55      23      2       85      82      85      88      26      17      57      32
        32      69      54      21      89      76      29      68      92      25      55      34      49      41
        12      45      60      18      53      39      23      79      96      87      29      49      37      66
        49      93      95      97      16      86      5       88      82      55      34      14      1       16
        71      86      63      13      55      85      53      12      8       32      45      13      56      21
        58      46      82      81      44      96      22      29      61      35      50      73      66      44
        59      92      39      53      24      54      10      45      49      86      13      74      22      68
        18      87      5       58      91      2       25      77      14      14      24      34      74      72
        59      33      70      87      97      18
        Sorted final bucket is
```

```
E:\MUNIRABBASI\university\semester 5\algo\ALGOCOde\parallelsort.exe
        17      18      18      18      18      18      18      18      19      20      21      21      21      21
        21      21      21      21      22      22      22      22      22      22      23      23      23      23
        23      23      24      24      24      24      24      24      24      25      25      26      26      26
        27      27      27      28      28      29      29      29      29      29      29      29      29      30
        30      30      31      31      32      32      32      33      33      33      33      34      34      34
        34      34      34      35      35      35      35      36      36      36      37      37      37      37
        37      38      38      38      38      38      39      39      39      39      40      40      41      41
        41      41      41      41      41      42      42      42      42      43      44      44      44      44
        44      44      45      45      45      45      45      45      46      46      46      46      47      47
        48      48      48      48      48      48      49      49      49      49      49      50      50      50
        50      50      51      51      52      53      53      53      53      53      53      53      54      54
        54      55      55      55      55      55      55      55      55      56      56      56      56      57
        57      57      57      57      57      58      58      58      58      58      58      58      59      59
        59      59      60      61      61      61      61      62      62      62      62      63      64      64
        64      66      66      66      66      67      67      67      67      68      68      68      68      68
        69      69      69      69      69      70      70      70      71      71      72      72      72      72
        73      73      73      73      74      74      74      74      75      75      76      76      76      77
        77      77      78      78      79      79      80      81      81      81      81      81      82      82
        82      82      82      83      83      83      84      84      85      85      85      85      86      86
        86      86      86      86      87      87      87      87      87      88      88      88      88      88
        88      88      89      89      89      89      89      90      90      90      90      91      91      91
        91      91      91      91      92      92      92      92      92      93      93      93      93      94
        95      95      95      95      95      96      96      96      96      97      97      97      98      98
        98      98      99      99      99      99

Sorting 399 elements took 0.000000 seconds

--------------------------------
Process exited after 0.1446 seconds with return value 0
Press any key to continue . . .
```

```c
total = omp_get_wtime() - t1;
 tmp = A;
 A = B;
 B = tmp;
if (dimensions <= 400) {
 //printf("\t");
     printf("\tSorted final bucket is  \n");
     for(i=0;i<dimensions;i++) {
         printf("%d\t ",A[i]);
     }
     printf("\n\n");
 }
 printf("Sorting %d elements took %f seconds\n", dimensions,total);
```

Note how sorting elements takes such small time that the system clock can only show 0.0000000 since the time taken is so much less of the parallel sort. Now normal bucket sort takes how much time lets see.

Normal bucket Sort for 399 elements.

```
E:\MUNIRABBASI\university\semester 5\algo\Untitled1.exe
Bucket[3]: 21 21 21 21 21 21 21 21 22 22 22 22 22 22 23 23 23 23 23 23 24 24 24 24 24 24 24 25 25 26 26 26 27 27 27
Bucket[4]: 28 28 29 29 29 29 29 29 29 29 30 30 30 31 31 32 32 32 33 33 33 33 34 34 34 34 34 34
Bucket[5]: 35 35 35 35 36 36 36 37 37 37 37 37 38 38 38 38 38 39 39 39 39 40 40 41 41 41 41 41 41 41
Bucket[6]: 42 42 42 42 43 44 44 44 44 44 44 45 45 45 45 45 45 46 46 46 46 47 47 48 48 48 48 48 48
Bucket[7]: 49 49 49 49 49 50 50 50 50 50 51 51 52 53 53 53 53 53 53 53 54 54 54 55 55 55 55 55 55 55 55
Bucket[8]: 56 56 56 56 57 57 57 57 57 57 58 58 58 58 58 58 58 59 59 59 59 60 61 61 61 61 62 62 62 62
Bucket[9]: 63 64 64 64 66 66 66 66 67 67 67 67 68 68 68 68 68 69 69 69 69 69
Bucket[10]: 70 70 70 71 71 72 72 72 72 73 73 73 73 74 74 74 74 75 75 76 76 76
Bucket[11]: 77 77 77 77 78 78 79 79 80 81 81 81 81 81 82 82 82 82 82 83 83 83
Bucket[12]: 84 84 85 85 85 85 86 86 86 86 86 86 87 87 87 87 87 88 88 88 88 88 88 88 89 89 89 89 89 90 90 90 90
Bucket[13]: 91 91 91 91 91 91 91 92 92 92 92 92 93 93 93 93 94 95 95 95 95 95 96 96 96 96 97 97 97
Bucket[14]: 98 98 98 98 99 99 99 99


0.812311 is the total seconds needed for sorting 399 elements

Sorted array: 0 0 0 0 0 0 1 1 1 2 2 2 2 2 2 3 3 3 3 4 4 5 5 5 5 6 6 6 6 7 7 7 8 8 8 9 9 9 9 9 10 10 10 10 11 11 11 12 12
 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15 16 16 16 16 16 16 17 17 18 18 18 18 18 18 18 19 20 21 21 21 21 21 21 21 21
 22 22 22 22 22 22 23 23 23 23 23 23 24 24 24 24 24 24 24 25 25 26 26 26 27 27 27 28 28 29 29 29 29 29 29 29 29 30 30 30
 31 31 32 32 32 33 33 33 33 34 34 34 34 34 34 35 35 35 35 36 36 36 37 37 37 37 37 38 38 38 38 38 39 39 39 39 40 40 41 41
 41 41 41 41 41 42 42 42 42 43 44 44 44 44 44 44 45 45 45 45 45 45 46 46 46 46 47 47 48 48 48 48 48 48 49 49 49 49 49 50
 50 50 50 50 51 51 52 53 53 53 53 53 53 53 54 54 54 55 55 55 55 55 55 55 55 56 56 56 56 57 57 57 57 57 57 58 58 58 58 58
 58 58 59 59 59 59 60 61 61 61 61 62 62 62 62 63 64 64 64 66 66 66 66 67 67 67 67 68 68 68 68 68 69 69 69 69 69 70 70 70
 71 71 72 72 72 72 73 73 73 73 74 74 74 74 75 75 76 76 76 77 77 77 77 78 78 79 79 80 81 81 81 81 81 82 82 82 82 82 83 83
 83 84 84 85 85 85 85 86 86 86 86 86 86 87 87 87 87 87 88 88 88 88 88 88 88 89 89 89 89 89 90 90 90 90 91 91 91 91 91 91
 91 92 92 92 92 92 93 93 93 93 94 95 95 95 95 95 96 96 96 96 97 97 97 98 98 98 98 99 99 99 99

--------------------------------
Process exited after 1.179 seconds with return value 0
Press any key to continue . . .
```

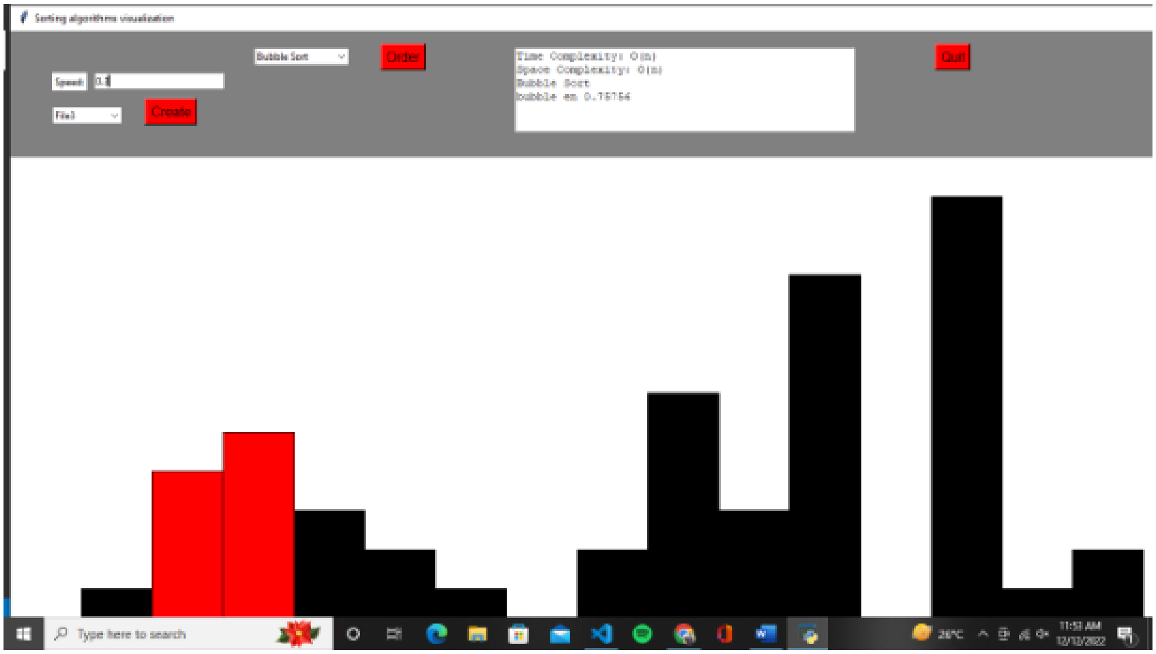
```
int t1 = omp_get_wtime();
 BucketSort(array);
  int total = omp_get_wtime() - t1;

 printf("\n\n%f is the total seconds needed for sorting 399 elements\n\n",total);
```

We cannot even deduce mathematically how fast the parallel bucket sort is as compared to a normal bucket sort as the clock time of parallel one is 0.000000, for sake of execution let's assume it is 0.1446 as mentioned in program end. For normal bucket it is less, but extremely large when compared to parallel bucket one as time is 1.179 seconds and 0.812311 for bucket.

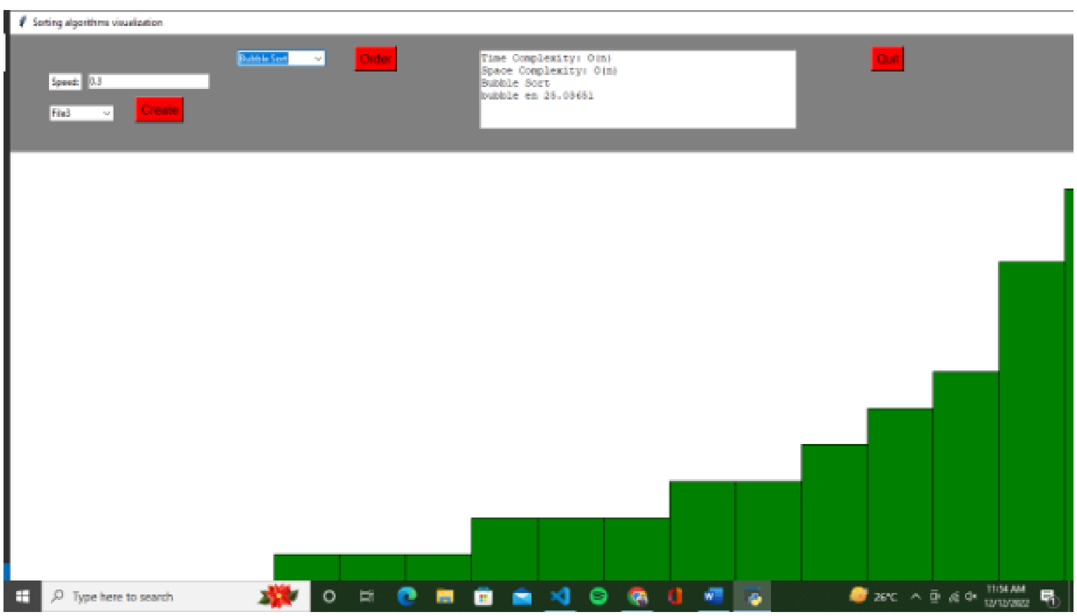
Experimental Setup

The setup for the programming was done by using Visual Studio. The code was written here. These are some screenshots for all the 10 implemented sorts.

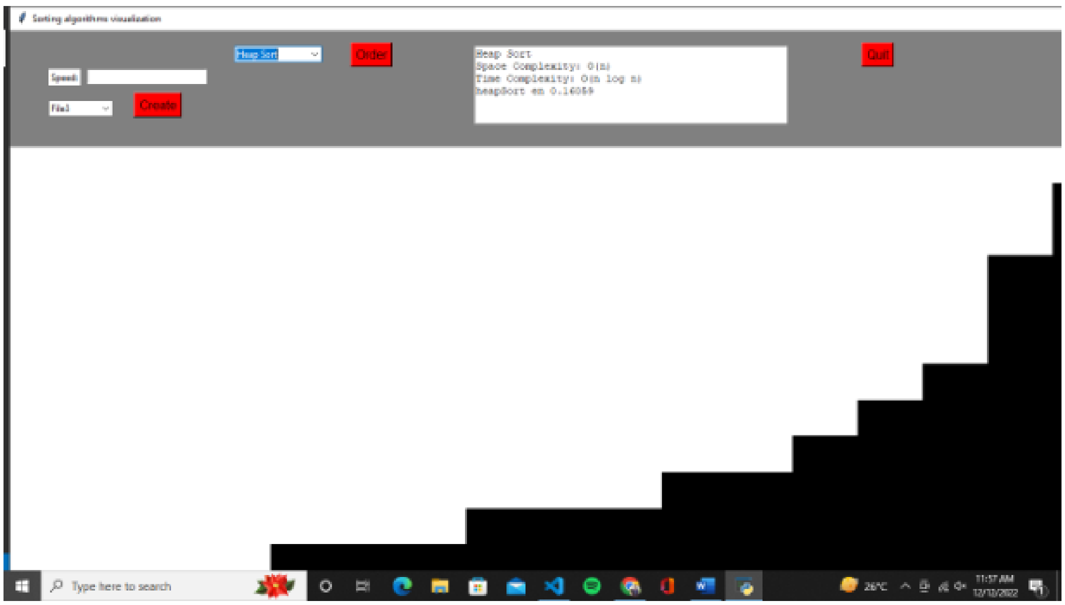
Bubble Sort in progress



Bubble Sort completed



Heap sort



Results and discussions

Let us compile some results for each sort.

Algorithm	Time Complexity			Space Complexity	Stable?
	Best	**Average**	**Worst**		
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)	Yes
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)	Yes
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)	No

Algorithm	Time Complexity			Space Complexity	Stable?
	Best	**Average**	**Worst**		
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)	No
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)	Yes
Bucket Sort	Ω(n + k)	θ(n + k)	O(n^2)	O(n)	Depends
Radix Sort	Ω(n.k)	θ(n.k)	O(n.k)	O(n + k)	Yes
Count Sort	Ω(n + k)	θ(n + k)	O(n + k)	O(k)	Yes

Space complexity restrictions

When we have a priority of using less space, then it is best to use bubble sort, insertion sort and heap sort. However in the least space for an unsorted array, it is best to use heap sort and on average and worst cases it will perform better than the other two. Heap is best if the stability of the sort is not a problem.

Time complexity restrictions

When we have a time restriction, it is best to use merge sort. It gives a fixed complexity of running time. It is stable as well. If stability is not an issue, we can use heap sort.

File 1=> Big Range, unique data

File 2=>Small range, repeated numbers

File 3=>Already sorted data

File 4=>Big range, repeated numbers

File 5=>Nearly sorted

Algorithm	Time Taken (seconds)				
	File 1	**File 2**	**File 3**	**File 4**	**File 5**
Bubble Sort	0.32318	0.11126	***0.00908***	1.42703	***0.01392***
Insertion Sort	0.32601	0.12561	0.01283	1.37864.	0.01659
Heap Sort	***0.09544***	***0.04476***	0.07625	0.25297	0.03766
Quick Sort	0.57128	0.23902	0.92755	2.17132	0.23413
Merge Sort	0.19647	0.09125	0.13542	0.47491	0.08237
Bucket Sort	0.13652	0.05219	0.07081	***0.14463***	0.04503
Radix Sort	0.62434	0.31078	0.62312	0.6209	0.30793
Count Sort	0.13651	0.13381	0.11689	0.28774	0.28388

1. For the modified quicksort or 7.4.5, we make the following deductions for the value of K.

- T(N) = running time of quicksort + running time of insertion sort (for the remaining N/K arrays of size K(or less than K))
- N (log N - log K and n/k for in bracket due to division) + N/K (K^2 as n square) = > N log(N/K) + NK.
- In theory, K has to be chosen such that : NK + N log (N/K) = N log N for same complexity==>
- K = log N (K can't be more than log N otherwise the total running time will be more than N log N which deems the change pointless).
- In practice, K should be the largest list length on which insertion sort is faster than quicksort which can be observed as well through experimental analysis.

Finding => Quicksort algorithm is efficient if the size of the input is very large. But, insertion sort is more efficient than quick sort in case of small arrays as the number of comparisons and swaps are less compared to quicksort. So we combine the two algorithms to sort efficiently using both approaches.

2. For the 8.2.4 or the modified counting sort/range. We make use of counting sort. We know that in counting sort we maintain the sum of the occurrences of all the elements in the array.
   So array[b]-array[a-1] will return the answer in O(1) time.

3. The parallel bucket sort is almost 800% faster (with the assumed values which are still very big 0.8 normal vs 0.1 parallel) than the normal bucket sort. We can be sure that this factor depends upon the number of threads used which can be the buckets as per the paper.
4. Heap sort was best for sorting files that had big range and unique data along with those with a small range and repeated numbers.
5. On already sorted data bubbles sort was fastest, had a close call with insertion sort.
6. Bucket sort was best on big range with repeated numbers.
7. Bubble sort was fastest on nearly sorted data with an again close call with insertion sort.

Conclusion

We can conclude that different sorts are feasible in different scenarios as we see from the time complexities. Another important finding is that we can design modified algorithms as per our needs to get the accurate results while keeping in mind our constraints and like time and space.

References

1. www.youtube.com
2. Geeksforgeeks.com
3. Wikipedia.com