# Assignment 1 - Radix Sort

## Carlos Vega de Alba

# Exercise 1 - Radix Sort Sequential

## - Explanation of Radix Sort

First of all let's explain how does Radix Sort works using my own words.
In Radix Sort we have an outer loop that would execute i times. This number of times would be defined as follows. The numbers we are sorting have 64 bits (at most), and we sort in groups of x bits (given as an input).

For example if x is 4, we would first check the 4 first least significant bits and then, do the next ones and keep doing it until we have done it 64/4=16 times.
Once we have understood how Radix Sort works, then we have to understand how the numbers are sorted inside each iteration. This is done by doing 3 sequential loops:

- First one: Goes through the numbers array counting the amount of different values we have at this iteration, these are saved in an array we will call the key array.

- Second one: Goes through this just created key array and sets where the numbers should be placed by saving the indices of where each different value should be placed in the sorted array. This is done by just adding (we will see better in the example).

- Third one: Goes again through the numbers array and now puts each element sorted by using the previous created sorted array.

Easy visualizable example: (sorting numbers of 8 bits by groups of 4 bits)
- Numbers: [**00001001**, **00011001**, **01000000**, **00111010**]

We will need to iterate as we first sort the 4 least significant bits and then the other 4 most significant bits. As we are sorting in groups of 4 bits we need the keys array to have 2 to the power of 4 as those are the possible numbers that can be generated

## FIRST ITERATION (4 least significant bits)

- Numbers: [**00001001**, **00011001**, **01000000**, **00111010**]

**First loop runs and count the times each elements appears**

(position, value)

- KeyArray: [0: 0000 -> 1
  1: 0001 -> 0
  2: 0010 -> 0
  3: 0011 -> 0
  4: 0100 -> 0
  5: 0101 -> 0
  6: 0110 -> 0
  7: 0111 -> 0
  8: 1000-> 0
  9: 1001 -> 2
  10: 1010 -> 1
  11: 1011 -> 0
  12: 1100 -> 0
  13: 1101 -> 0
  14: 1110 -> 0
  15: 1111 -> 0
  ]

**Second loop runs and sets the index of where each element should start.**

(We just add the previous value plus the current one)

- KeyArray: [0: 0000 -> 1
  1: 0001 -> 1
  2: 0010 -> 1
  3: 0011 -> 1
  4: 0100 -> 1
  5: 0101 -> 1
  6: 0110 -> 1
  7: 0111 -> 1
  8: 1000-> 1
  9: 1001 -> 3
  10: 1010 -> 4
  11: 1011 -> 4
  12: 1100 -> 4
  13: 1101 -> 4
  14: 1110 -> 4
  15: 1111 -> 4
  ]

**Third loop runs and for each number goes to the key array and looks for the position that should be placed in the sorted array and places the number there.**

(This loop goes from the end to the beginning to make the element that appears first be before in the sorted array)

(To get the actual position it necessary to decrease by 1 the value is gotten from the key array)

- Sorted array: [**01000000, 00001001**, **00011001**, **00111010**]

## SECOND ITERATION (4 most significant bits)

- Sorted array: [**01000000**, **00001001**, **00011001**, **00111010**]

**First loop runs and count the times each elements appears**

(position, value)

- KeyArray: [0: 0000 -> 1
  1: 0001 -> 1
  2: 0010 -> 0
  3: 0011 -> 1
  4: 0100 -> 1
  5: 0101 -> 0
  .
  .
  .
  15: 1111 -> 0
  ]

**Second loop runs and sets the index of where each element should start.**

- KeyArray: [0: 0000 -> 1
  1: 0001 -> 2
  2: 0010 -> 2
  3: 0011 -> 3
  4: 0100 -> 4
  5: 0101 -> 4
  .
  .
  .
  15: 1111 -> 4
  ]

**Third loop runs and for each number goes to the key array and looks for the position that should be placed in the sorted array and places the number there.**

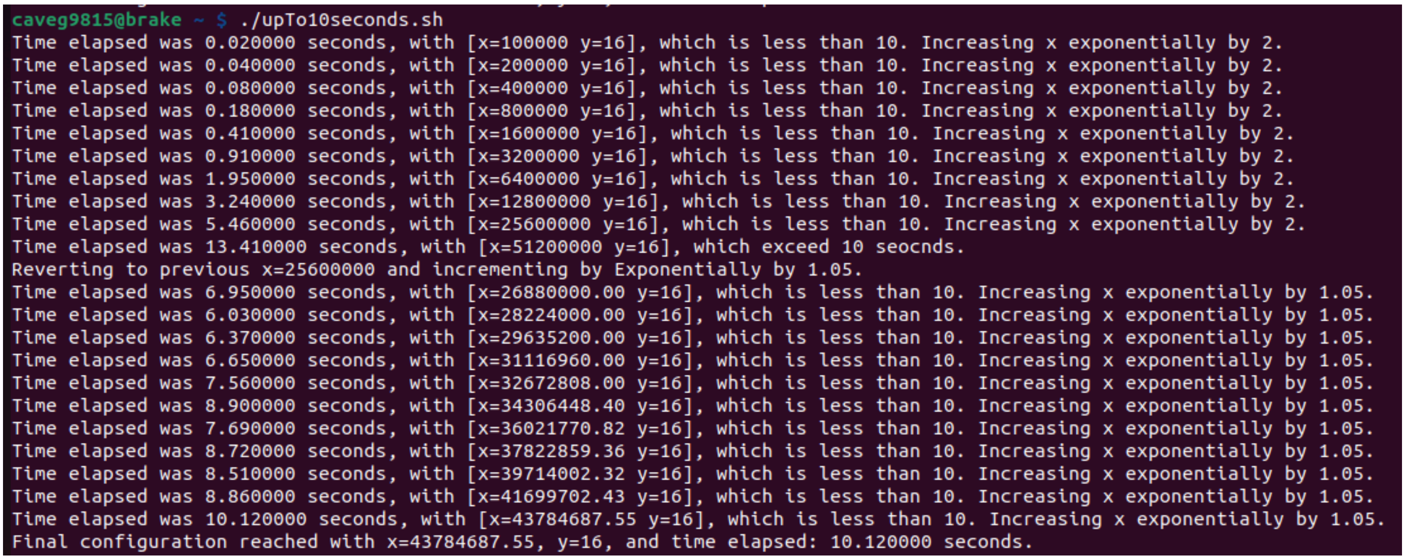
Sorted array: [**00001001**, **00011001**, **00111010**, **01000000**]

# Exercise 2 - Runtime under 10 seconds Radix Sort Sequential. Math expression

To get a precise answer of when did the algorithm reach 10 seconds I created a script named “upTo10seconds.sh” that would increase the input by a factor of 2 until it reach 10 seconds and then go back to the last value that fitted inside 10 seconds and increase the input by 1.05, this is a good idea for testing performance easily.

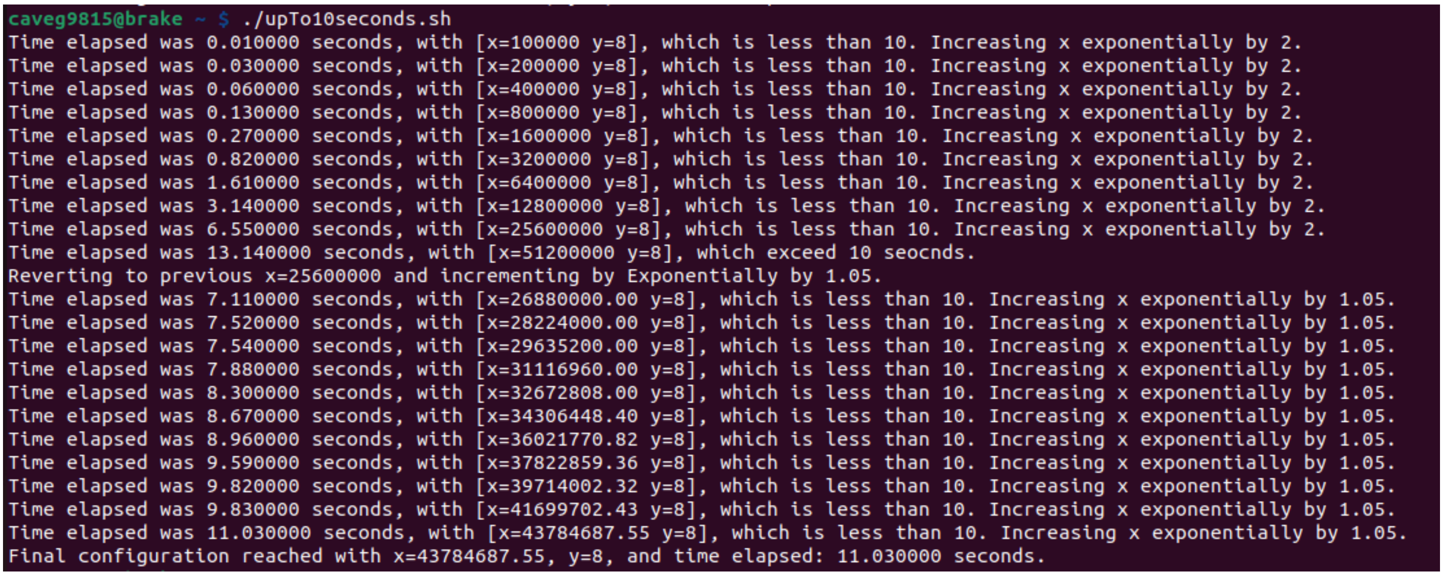
For more precision I tried with the values y=8 and y=16 that seemed to give the best results (Due to the key array does not get to small or too big and can be handled in cache efficiently). They all stopped at around almost 44 million.

y=16



```
caveg9815@brake ~ $ ./upTo10seconds.sh
Time elapsed was 0.020000 seconds, with [x=100000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.040000 seconds, with [x=200000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.080000 seconds, with [x=400000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.180000 seconds, with [x=800000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.410000 seconds, with [x=1600000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.910000 seconds, with [x=3200000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 1.950000 seconds, with [x=6400000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 3.240000 seconds, with [x=12800000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 5.460000 seconds, with [x=25600000 y=16], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 13.410000 seconds, with [x=51200000 y=16], which exceed 10 seocnds.
Reverting to previous x=25600000 and incrementing by Exponentially by 1.05.
Time elapsed was 6.950000 seconds, with [x=26880000.00 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 6.030000 seconds, with [x=28224000.00 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 6.370000 seconds, with [x=29635200.00 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 6.650000 seconds, with [x=31116960.00 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 7.560000 seconds, with [x=32672808.00 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.900000 seconds, with [x=34306448.40 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 7.690000 seconds, with [x=36021770.82 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.720000 seconds, with [x=37822859.36 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.510000 seconds, with [x=39714002.32 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.860000 seconds, with [x=41699702.43 y=16], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 10.120000 seconds, with [x=43784687.55 y=16], which is less than 10. Increasing x exponentially by 1.05.
Final configuration reached with x=43784687.55, y=16, and time elapsed: 10.120000 seconds.
```

y=8



```
caveg9815@brake ~ $ ./upTo10seconds.sh
Time elapsed was 0.010000 seconds, with [x=100000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.030000 seconds, with [x=200000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.060000 seconds, with [x=400000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.130000 seconds, with [x=800000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.270000 seconds, with [x=1600000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 0.820000 seconds, with [x=3200000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 1.610000 seconds, with [x=6400000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 3.140000 seconds, with [x=12800000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 6.550000 seconds, with [x=25600000 y=8], which is less than 10. Increasing x exponentially by 2.
Time elapsed was 13.140000 seconds, with [x=51200000 y=8], which exceed 10 seocnds.
Reverting to previous x=25600000 and incrementing by Exponentially by 1.05.
Time elapsed was 7.110000 seconds, with [x=26880000.00 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 7.520000 seconds, with [x=28224000.00 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 7.540000 seconds, with [x=29635200.00 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 7.880000 seconds, with [x=31116960.00 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.300000 seconds, with [x=32672808.00 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.670000 seconds, with [x=34306448.40 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 8.960000 seconds, with [x=36021770.82 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 9.590000 seconds, with [x=37822859.36 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 9.820000 seconds, with [x=39714002.32 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 9.830000 seconds, with [x=41699702.43 y=8], which is less than 10. Increasing x exponentially by 1.05.
Time elapsed was 11.030000 seconds, with [x=43784687.55 y=8], which is less than 10. Increasing x exponentially by 1.05.
Final configuration reached with x=43784687.55, y=8, and time elapsed: 11.030000 seconds.
```

Therefore the chosen value after these calculations would be **n=43 millions**.

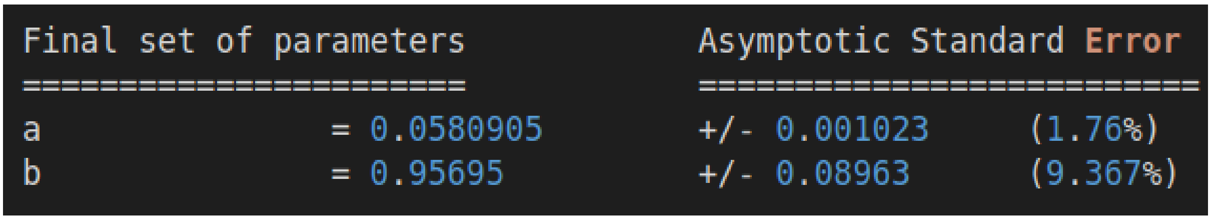
## **Math expression.**

For calculating the math expresion I used gnuplot and let it identify the values that should be taken.

We know from the theory that the theoretical math expresion is O(n log n)
Therefore the formula for the plotting would be the following:

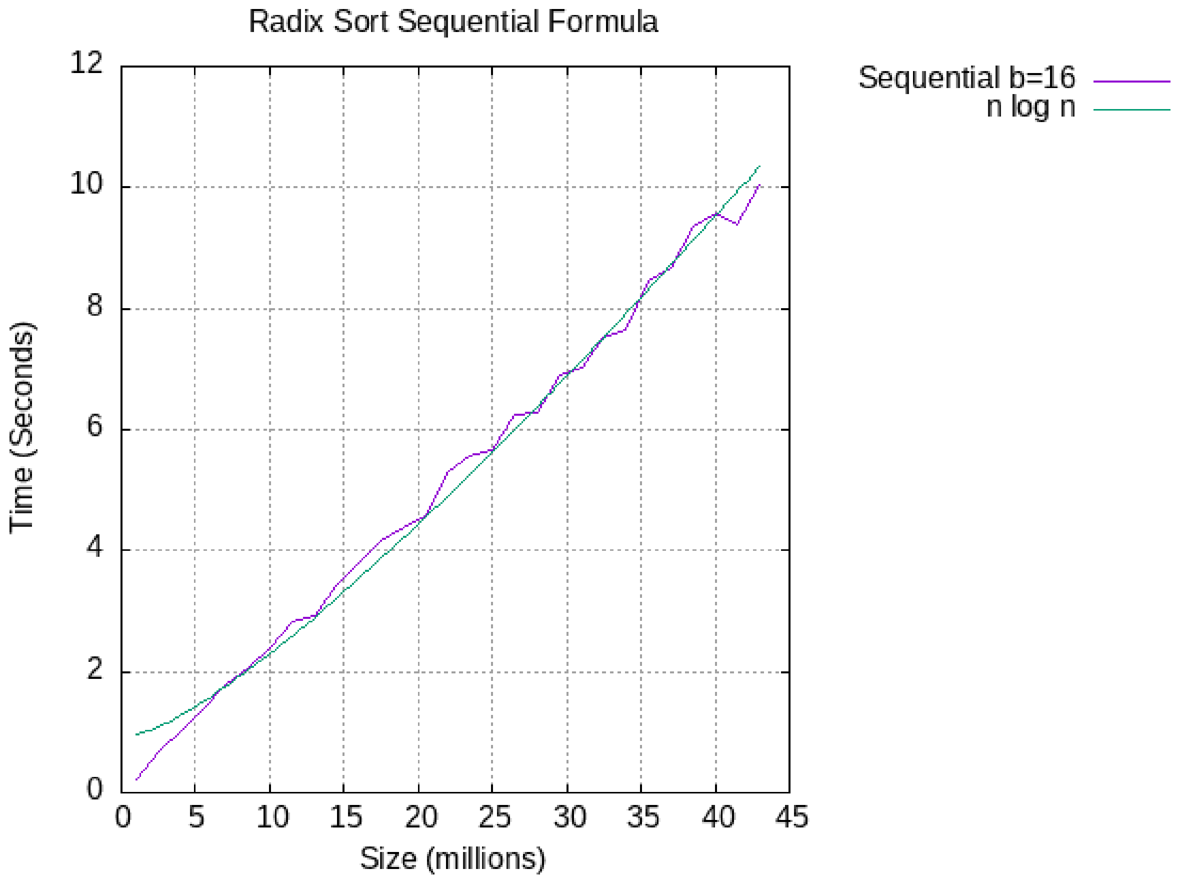
f(x) = a * x * log(x) + b  (For this calculation x is the value in millions)

We got this math values for a and b:



```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.0580905        +/- 0.001023       (1.76%)
b               = 0.95695          +/- 0.08963        (9.367%)
```

Here we have the output of it and then plot where we can see the math expression corresponds to our data:



Here we can clearly see that the formula fits the math expresion therefore we can say that the theory cost is true in practice.

# Exercise 3 - Radix Sort Parallel

The approach to make the algorithm Parallel is first to think where are the part where we are doing things that are independent from each other. For the sequential program we are going group of bits per group of bits from the least significant to most significant filling the key array in each iteration. But this is an action that could become parallel. We could first prepare this key arrays for each iteration before ordering the number, making it completely parallel.

So now instead of having only one array that would be the key_array we create as many iterations as are needed. For example if we are ordering numbers of 64 bits and the user input is that order in groups of 4 bits, then we would create 64/4=16 different arrays one for each iteration.
We parallelize this work as it is completely independent for each iteration.

```
#pragma omp parallel for
 for(i=0; i<numIterations; i++){
     for(j=0; j<number_of_elements; j++){
         unsigned long long aux2 = rawArray[j];
         unsigned int aux1 = (aux2 >> (i*number_of_bits)) & bits;
         countKeys[i][aux1]++;
     }

     for(j=1; j<numKeys; j++){
         countKeys[i][j]= countKeys[i][j-1] + countKeys[i][j];
     }

 }
```

This would make the threads fill their corresponding key array and be ready for after sequentially ordering them.

Then we would need some sequential code of ordering the numbers using this key array in each interaction.
But from copying the previous array to the next one we can take the advantage again of using another parallel region as we saw before that gave us some advantage.
This would be the code:

```
for(i=0; i<numIterations; i++){
    for(j=number_of_elements-1; j>=0; j--){
        aux2 = rawArray[j];
        aux1 = (aux2 >> (i*number_of_bits)) & bits; //aux1 has the bits we are studying in this
        index = countKeys[i][aux1];
        countKeys[i][aux1]--;
        outputArray[index-1] = rawArray[j];
    }

    #pragma omp parallel for
    for(j=0; j< number_of_elements; j++){
        rawArray[j] = outputArray[j];
    }
}
```

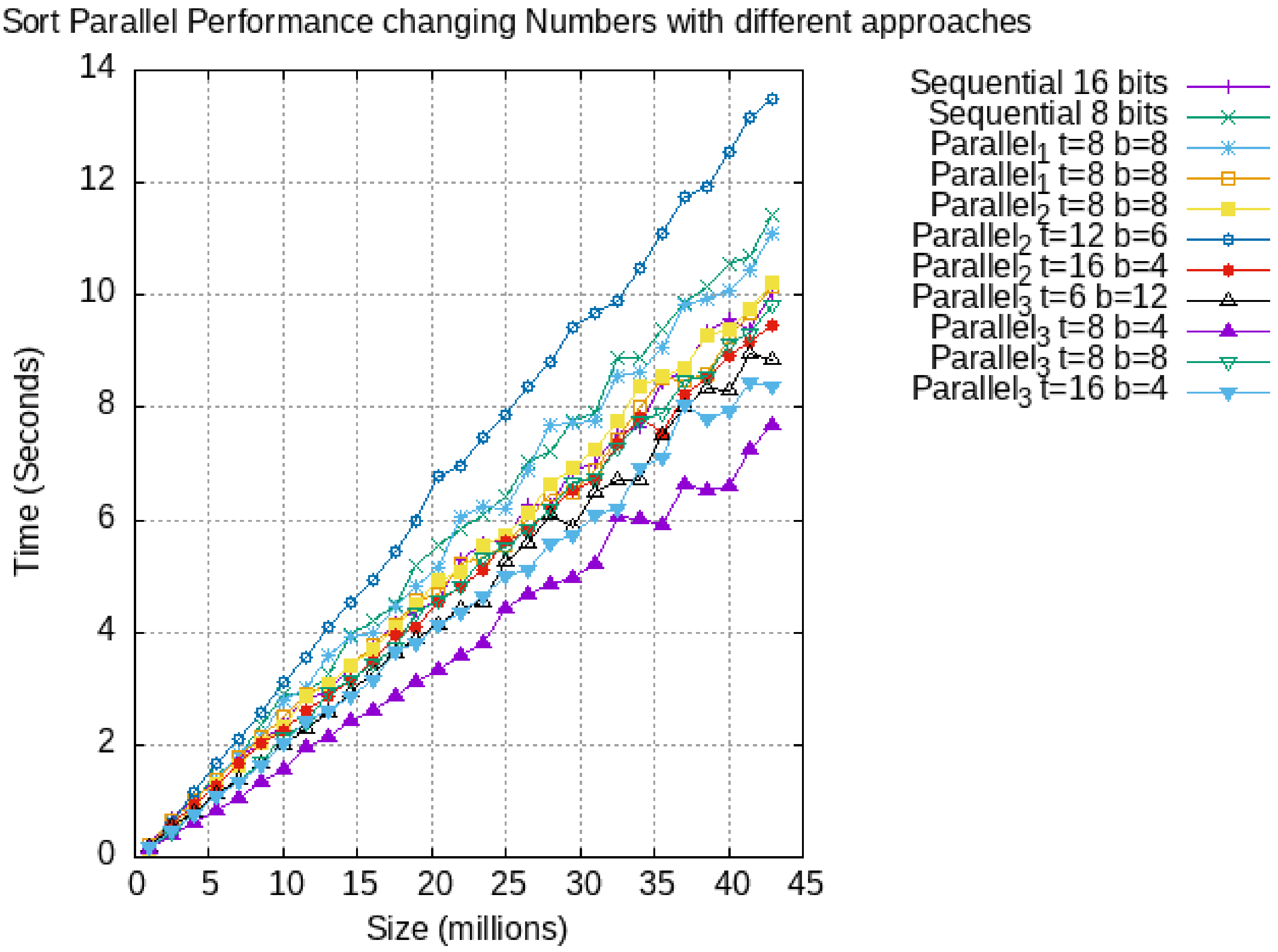
In order to choose this iteration instead of any other I plot 3 different approaches.

- Parallel1 -> Same as sequential but parallelizing only the copy of the array

- Parallel2 -> Parallelizing the creation of the key arrays as just mentioned

- Parallel3 -> Doing both, first parallelize the creation of the key array and then the copy of the array. This is the one that has been just explained above.

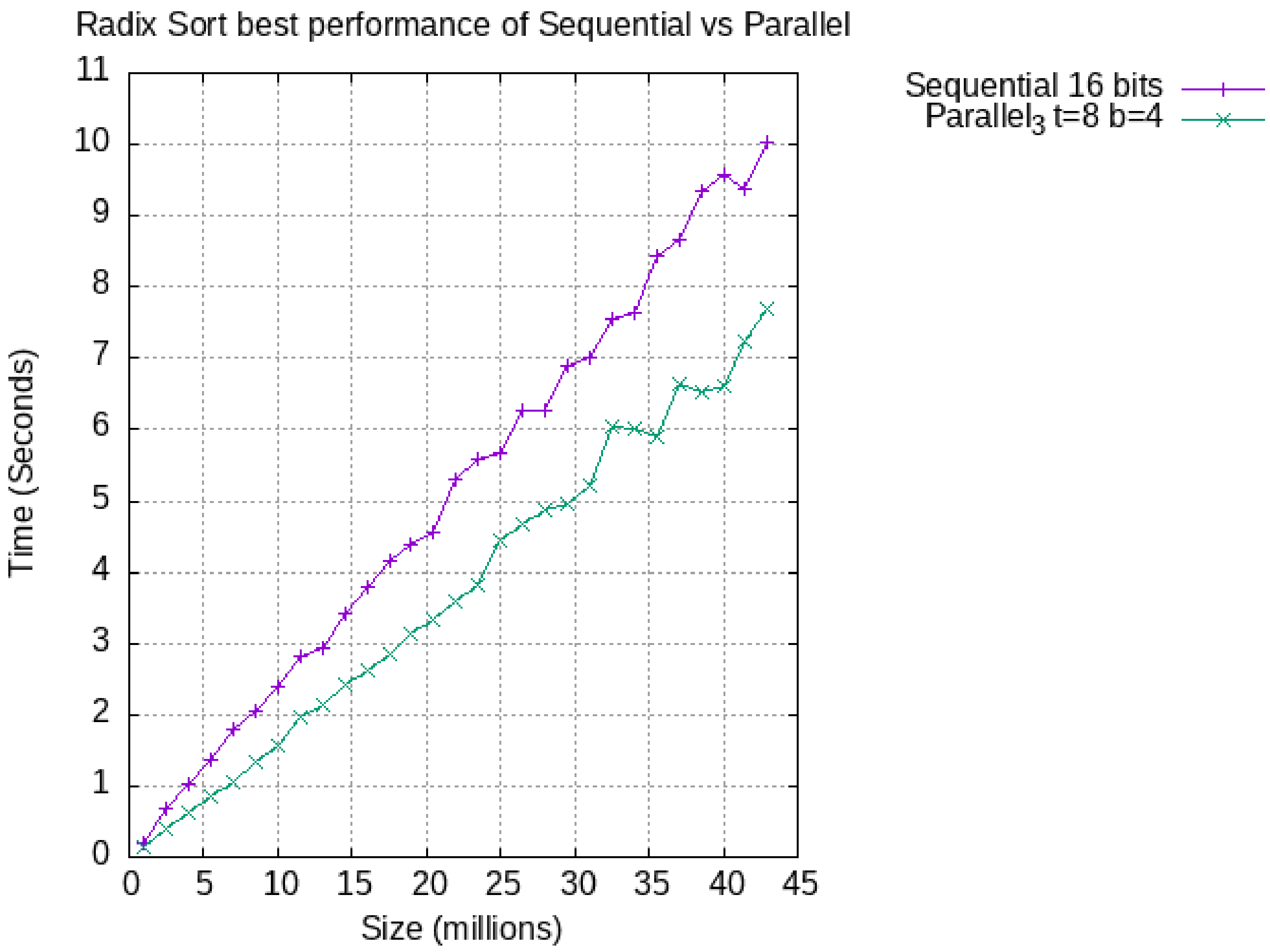
I tried with different amount of threads and different number of bits in order to get a view of what would be the best approach for each possibility. The input numbers go from 1m to 43 millions with an increment of 1,5m in each iteration. The threads and bits used are in the lines title.



As we can see the Parallel3 gives the best results, specially using 8 threads and 4 bits.

From here **Parallel3 would be the model that we would use for future analysis** as has been proven to give the best results.

Now, to be able to distinguish better we just see another plot of just the best performance of the sequential (that was using 16 bits vs the best performance of this Parallel3 version just explained, that was with 8 threads 4 bits)



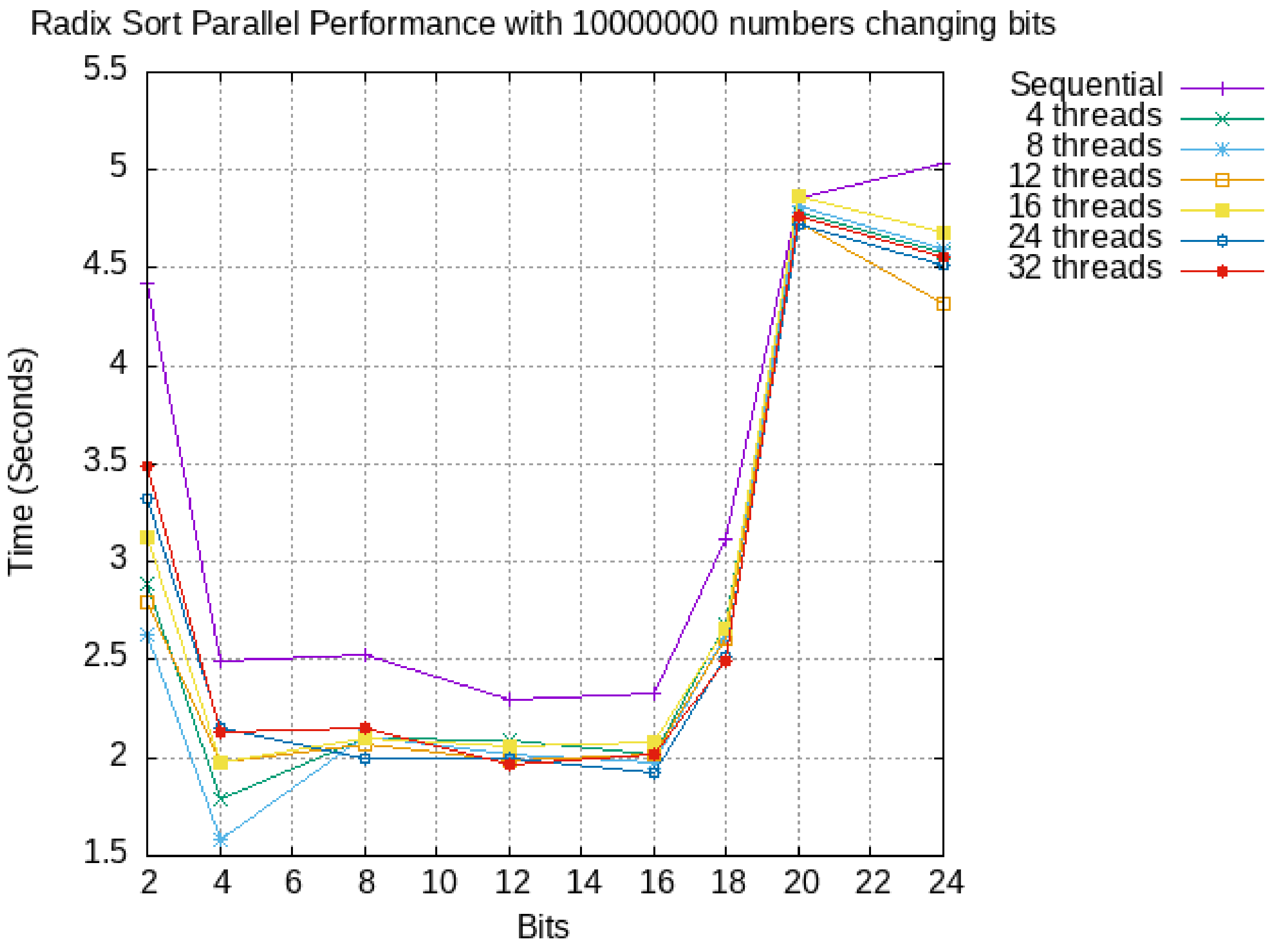
As we see, from the very beginning parallelization gave us much better results.

To be able to rerun the program changing the number of threads without recompiling I decided to make the number of threads be a parameter of the program.

```
int number_of_threads = atoi(argv[3]);
   if (number_of_threads<1){
      printf("\tERROR, the minimum number of threads is 1\n");
      return -1;
   }
   if (number_of_threads>80){
      printf("\tERROR, the maximum number of threads is 80\n");
      return -1;
   }
   omp_set_num_threads(number_of_threads);
```

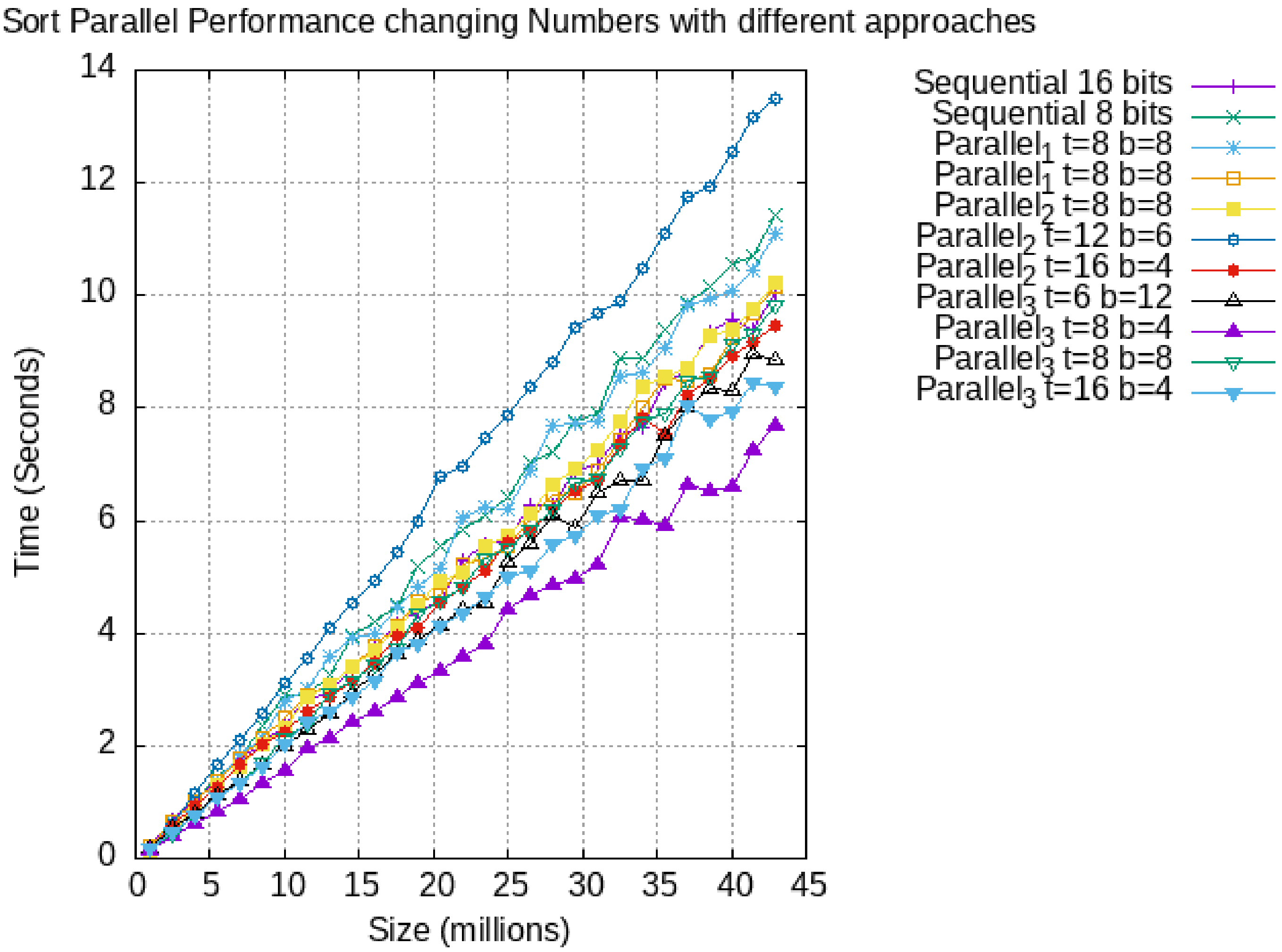
# Exercise 4 - Scaling Performance Study

Let’s study how it would affect changing the amount of bits for different number of threads.



Here we can see that the optimal number of bits is from 4 to 16. Having moreless results in the same range. Having less than 2 bits or more than 16 affects the balance of how many times we order the numbers.

If we increase the input, for any quantity of bits we can see that the algorithm increases linearly. We can take a look again at the graph we saw at problem 3



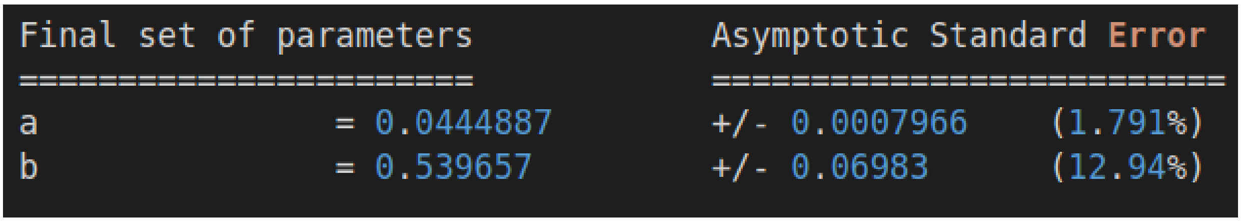
## Math expression.

For calculating the math expresion I will again use gnuplot and let it identify the values that should be taken. We will use the best approach we found that was using 16 threads and 4 bits.

We know from the theory that the theoretical math expresion is still the same, O(n log n) Therefore the formula for the plotting would remain the same as before.

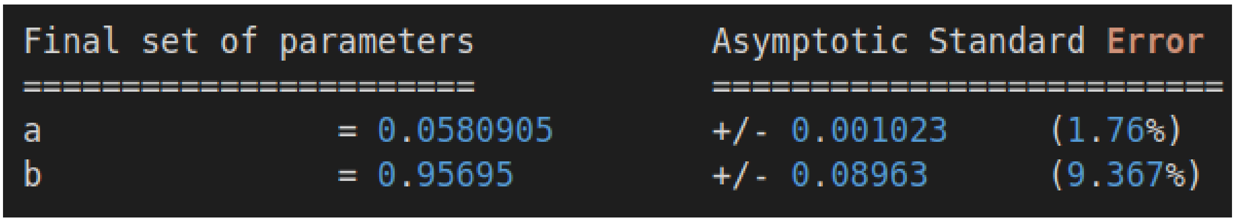
f(x) = a * x * log(x) + b  (For this calculation x is the value in millions)

We got this math values for a and b:



```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.0444887        +/- 0.0007966    (1.791%)
b               = 0.539657         +/- 0.06983      (12.94%)
```

Previous ones for sequential:



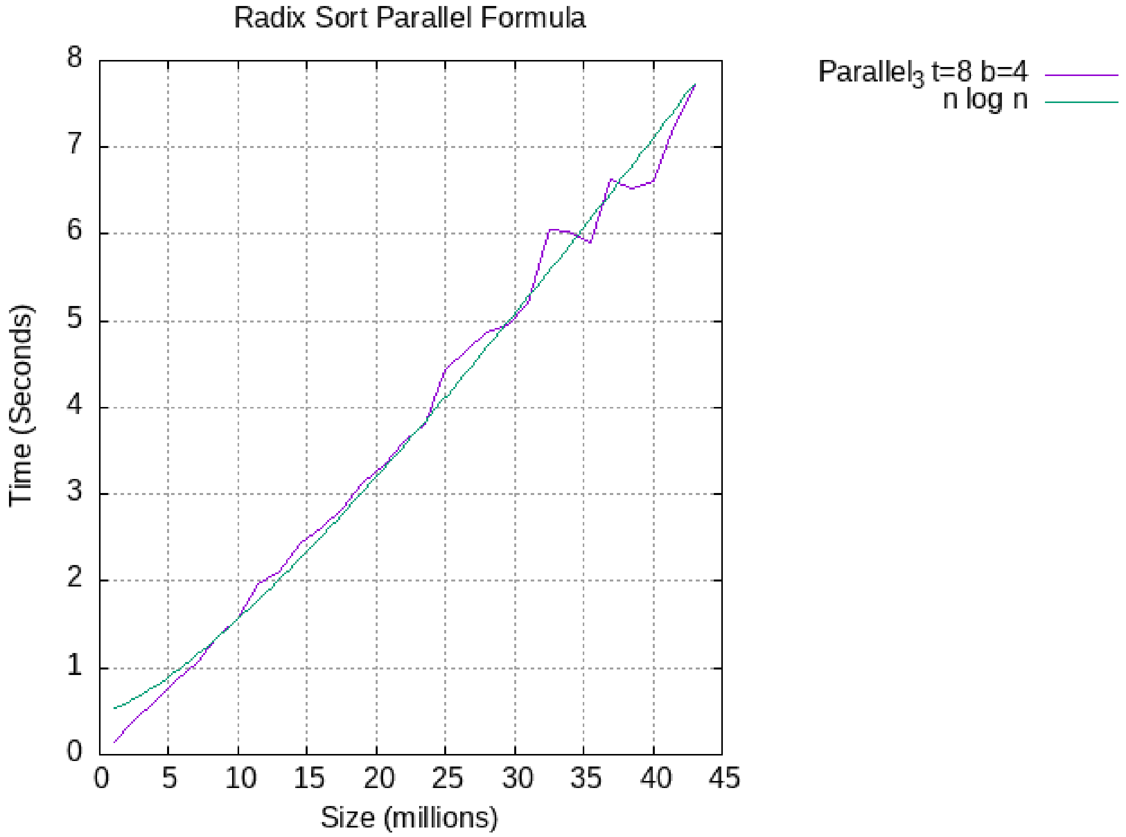
```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.0580905        +/- 0.001023     (1.76%)
b               = 0.95695          +/- 0.08963      (9.367%)
```

The SpeedUp would be TSequential/TParallel = 0,058095÷0,0444887 = 1,15 **30% of speedUp in theory** from the parallel compared to the sequential.

Let's take a practical example using the time it took for our last x that was 43m:

TSequential=10.037000 , TParallel=7.706463= 1,30. **That is 30% faster in practice.**

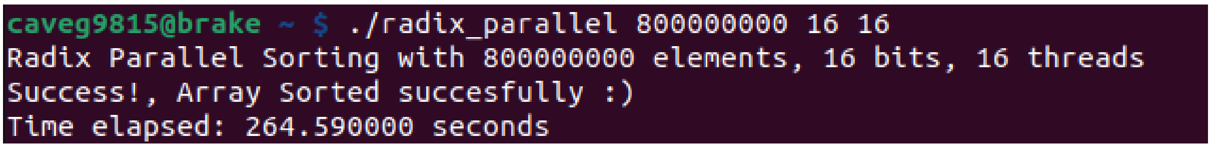
Again we output the values to a graph to visualize that the math expression corresponds to our data.



## Maximum Value of N the program can handle

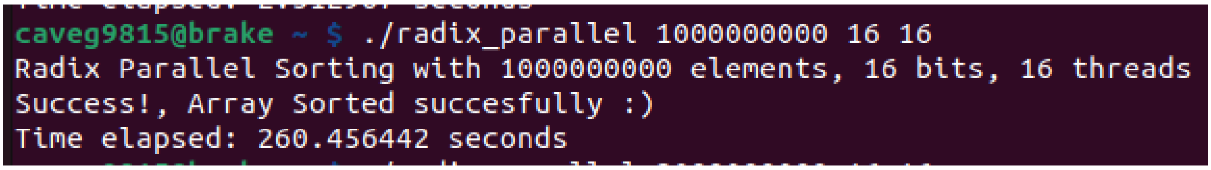
One option could be start trying until it fails:

Try for 800m



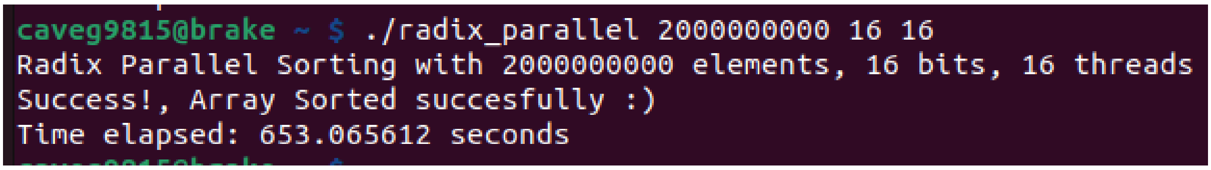
```
caveg9815@brake ~ $ ./radix_parallel 800000000 16 16
Radix Parallel Sorting with 800000000 elements, 16 bits, 16 threads
Success!, Array Sorted succesfully :)
Time elapsed: 264.590000 seconds
```

Try for 1000m



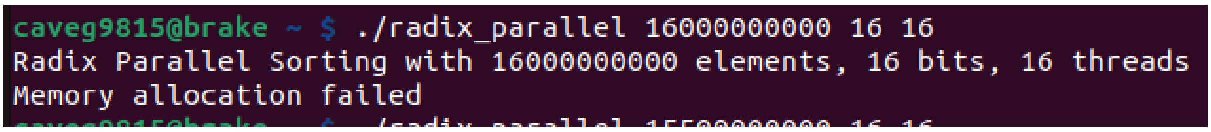
```
caveg9815@brake ~ $ ./radix_parallel 1000000000 16 16
Radix Parallel Sorting with 1000000000 elements, 16 bits, 16 threads
Success!, Array Sorted succesfully :)
Time elapsed: 260.456442 seconds
```

Try for 2000m



```
caveg9815@brake ~ $ ./radix_parallel 2000000000 16 16
Radix Parallel Sorting with 2000000000 elements, 16 bits, 16 threads
Success!, Array Sorted succesfully :)
Time elapsed: 653.065612 seconds
```

Try for 16000m



```
caveg9815@brake ~ $ ./radix_parallel 16000000000 16 16
Radix Parallel Sorting with 16000000000 elements, 16 bits, 16 threads
Memory allocation failed
```

The number of elements that can be sorted would depend on the amount of memory the computer has to be able to have the arrays in memory. For an element, ignoring the key array as it would not really affect in a normal case, we will have 2 arrays, one for the elements and another one for sorting the elements in each interaction.
A number would affect in terms of memory as 64 bits * 2 (each array) = 16bytes.
So the maximum number of elements would follow this formula. Of course for a more precise formula we will need to subtract from the TotalMemoryAvaible the memory space we would need for the key arrays and variables, but for generalization this would be a good formula:

**MaxN -> TotalMemoryAvaible/16bytes**