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=40 millions**.

Math expression.

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

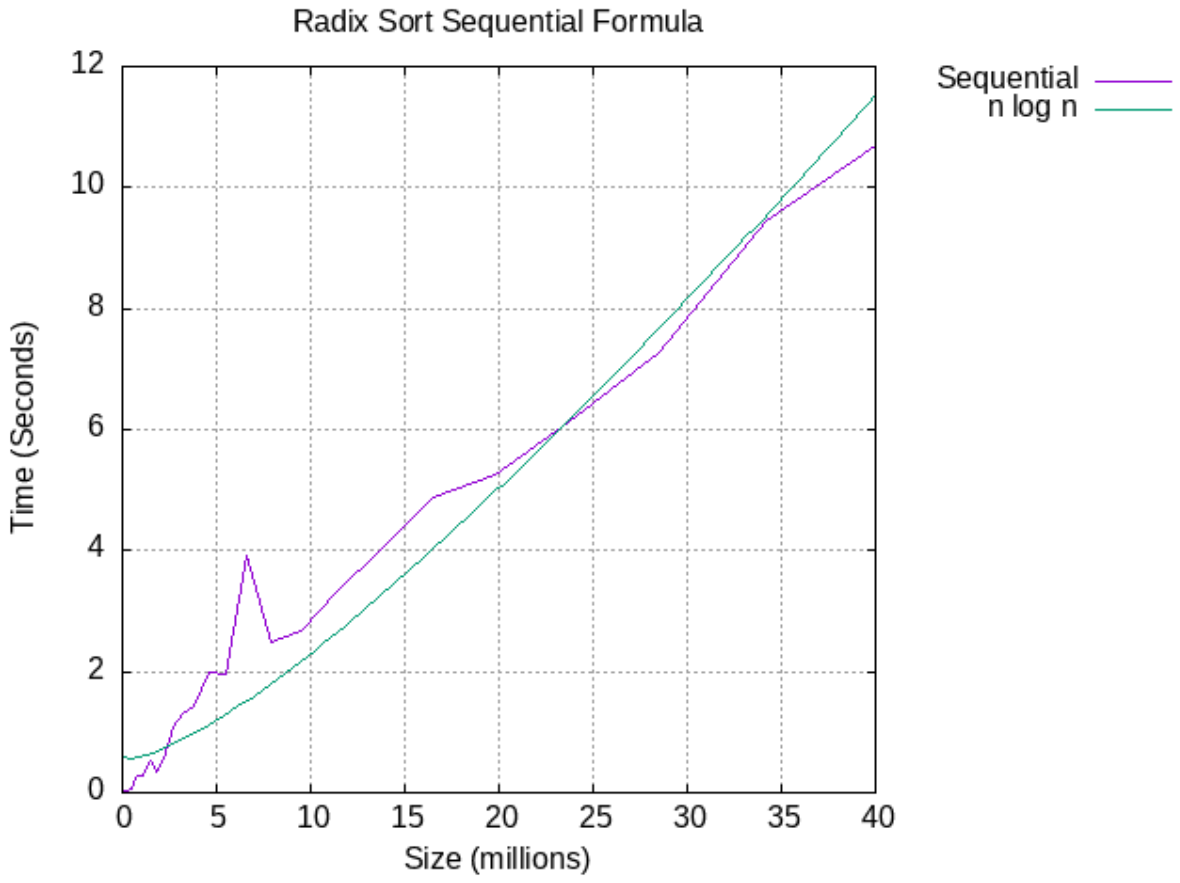
We know from the theory that the theoretical math expresion is O(n log n)
Therefore the formula for the plotting would be the following:

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                  = 0.0741033     +/- 0.003123     (4.214%)
b                  = 0.595424      +/- 0.131        (22.01%)
```

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

Before starting to program the radix sort parallel algorithm I thought about the possible approaches. I tried to apply parallelism in many ways but there did not seem to be improvements, instead it could even take longer. This could be done by the implicit cost because of the mechanism inside openmp library and maybe the creation and destruction of the threads.
My final choice was to apply the parallelism at the update of the array which gave me some improvements compared to the sequential.

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

Because the algorithm being almost sequential by design is harder for making parallelism improve in other fields. Here is an explanation because I chose this field rather than the rest.

This first loop is too small to really take advantage of it. The number of keys are small numbers and the cost of parallelizing does not seem to be worth it.

```
for(j=0; j<numKeys; j++){
    countKeys[j]=0;
}
```

This one would need an atomic operation or if not a semaphore to protect the variable.

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

This one is almost sequential by itself, making it challenging to get any improvements.

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

And for the sorting one we have the same problem.

```
for(j=number_of_elements-1; j>=0; j--){
    aux2 = rawArray[j];
    aux1 = (aux2 >> (i*number_of_bits)) & bits;
    index = countKeys[aux1];
    countKeys[aux1]--;
    outputArray[index-1] = rawArray[j];
}
```

Global variables are in danger of being changed therefore they must be protected. Making it hard to parallelize.

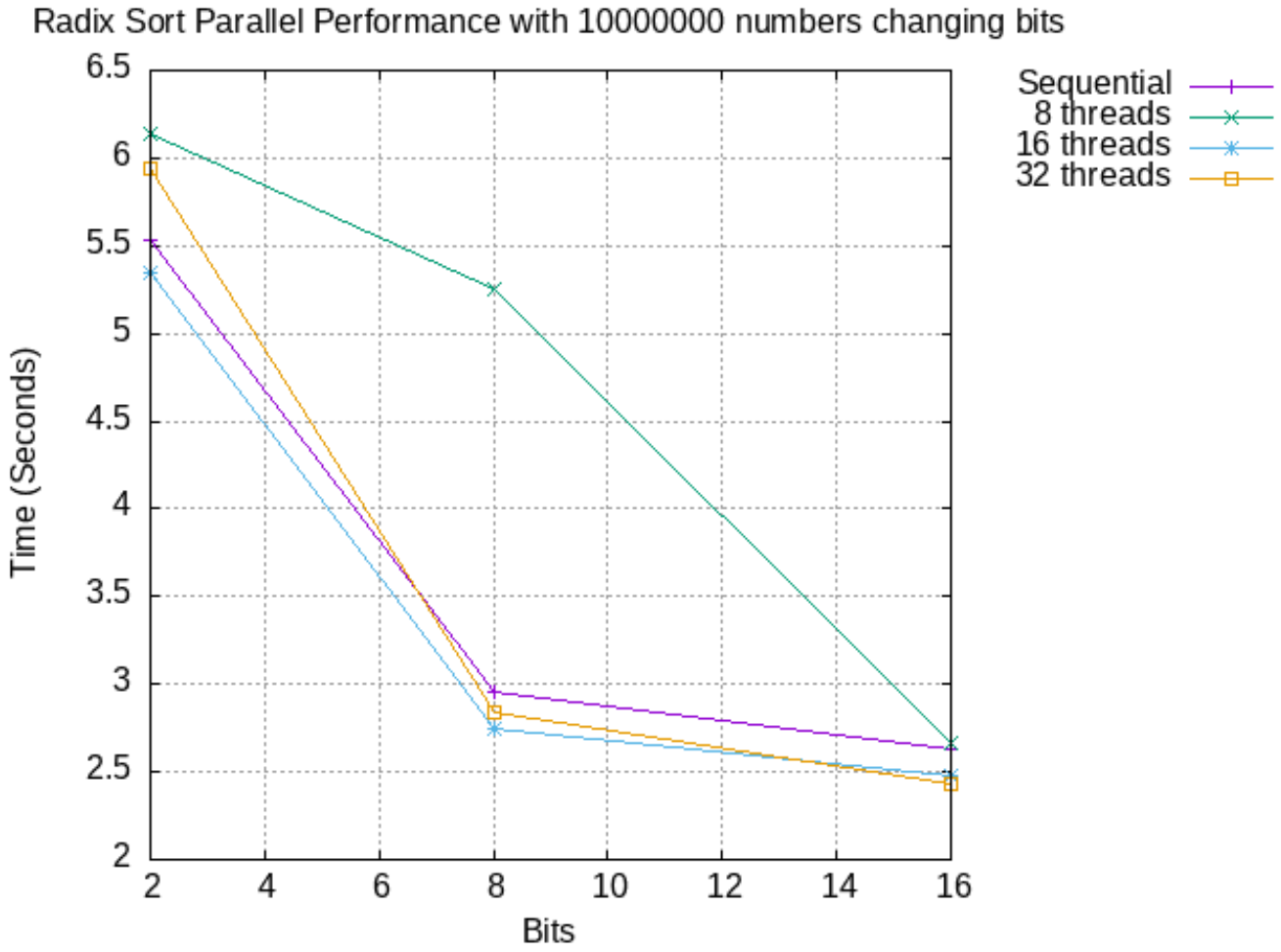
To be able to rerun the program changing the number of threads without recompiling I decided to make the number of threads be a parameter of the program.

```
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

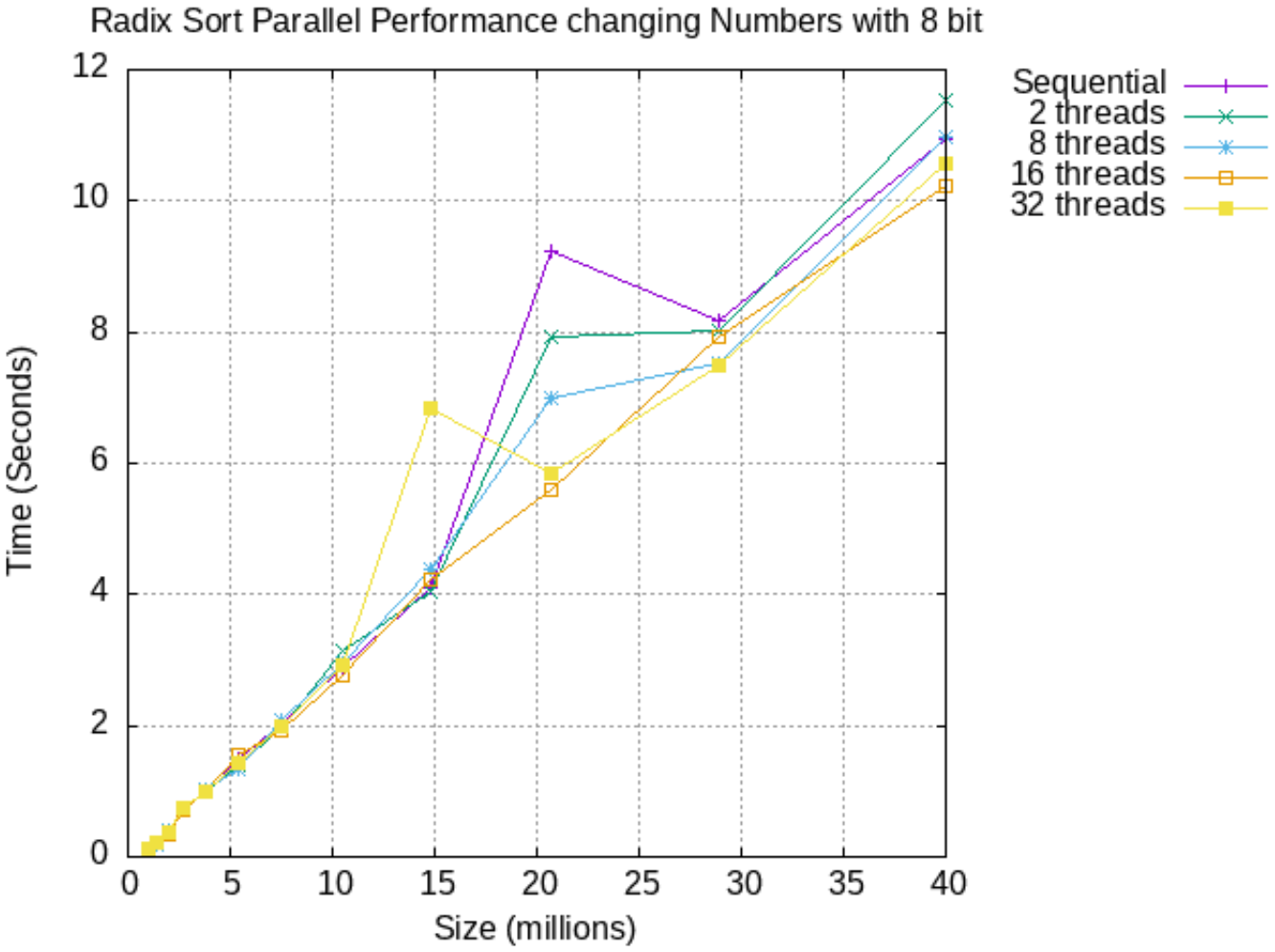
As there are now 3 possible things that can be changed, let's see how it can affect.

First changing the bits for a fixed input of numbers.



We can see that not using enough threads as 8 is not effective due to the fact that it can take for time to create the mechanism than actually using it. At 16 and 32 threads we get a speedup compare to the sequential.

And for a deeper study lets see how can change making the amount of numbers grow until the value we got at problem 2 and see the different performances for different values for the number of threads.



Here we can see that the time is pretty similar for small sizes and that using 2 or 8 threads seems to have less improvements than using 16 or 32 threads. Therefore we can see there is a speed up in the use of parallelization when the input increases.

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 16 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:

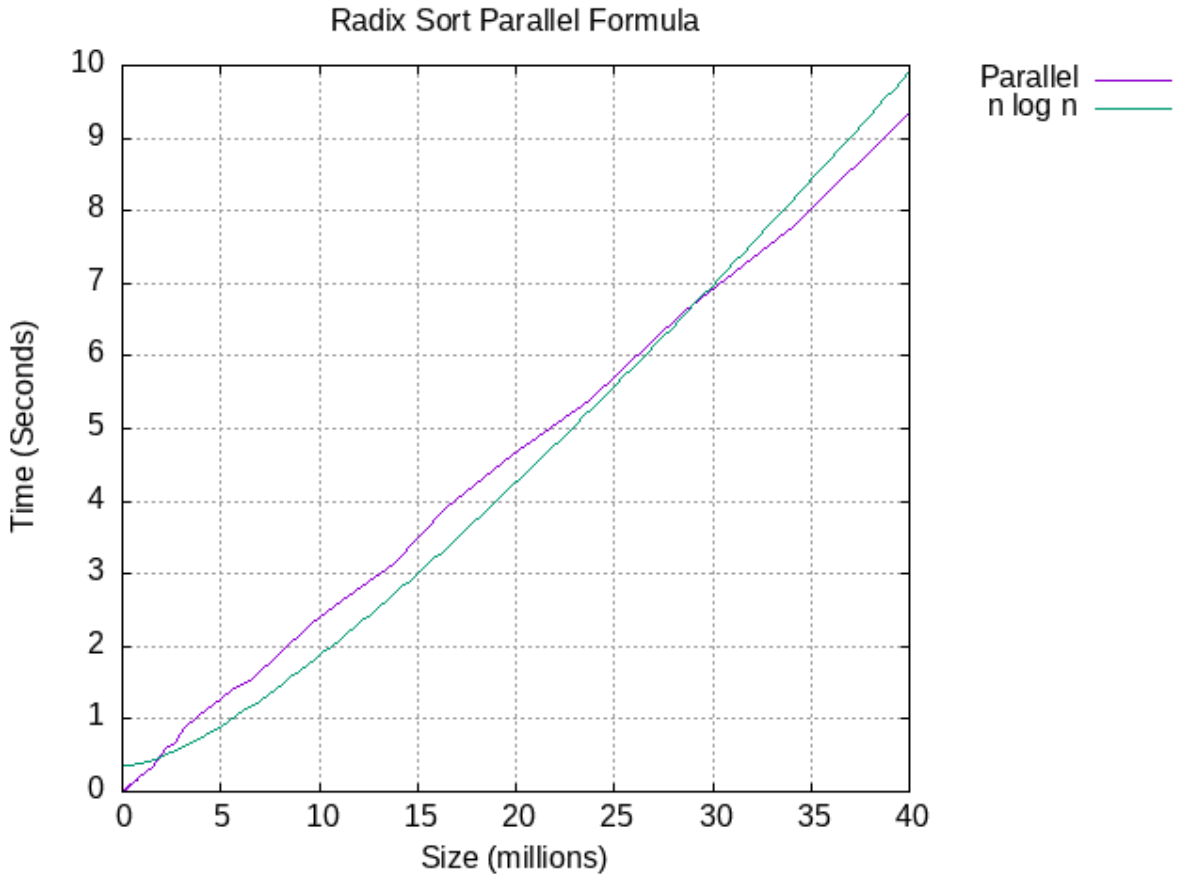
```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 0.0646016     +/- 0.001572     (2.433%)
b                  = 0.382748      +/- 0.06597      (17.23%)
```

Previous ones for sequential:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 0.0741033     +/- 0.003123     (4.214%)
b                  = 0.595424      +/- 0.131        (22.01%)
```

The SpeedUp would be TSequential/TParallel = 0.0741033 / 0.0646016 = 1,147
15% of speedUp from the parallel compare to the sequential

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.

MaxN -> TotalMemoryAvaible/16bytes

CODE FILES:

(Only the C files, Scripts are in the Zip)

radix_sequential.c

```c
#include "mt19937-64.c"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <time.h>
/* #define MAX_NUMBER 8 */ //Only for developing
#define MAXNBITS sizeof(unsigned long long) * 8


int check_number_of_elements(int number);
unsigned long long max(unsigned long long list[], int lenght);
int getNumberOfIterationsFromMax(unsigned long long maxNum, int numBits);
int countBits(unsigned long long number);
void getStringFromNumber(unsigned long long number, char * numberInBits);


int main(int argc, char *argv[]) {
    if (argc != 3) {
        // At least 3 arguments
        printf("Usage: %s number_of_elements number_of_bits\n", argv[0]);
        return -1;
    }

    char *endptr;
    unsigned long long number_of_elements = strtoull(argv[1], &endptr, 10);

    // Check if the coversion was done successfully
    if (endptr == argv[1]) {
        printf("La conversión falló. Asegúrate de que ingresaste un número válido.\n");
        return 1;
    }

    int number_of_bits = atoi(argv[2]);
    if (number_of_bits<1){
        printf("\tERROR, the minimum number of bits is 1\n");
        return -1;
    }
    if (number_of_bits>=64){
        printf("\tERROR, the maximum number of bits is 63\n");
        return -1;
    }

    printf("Radix Sequential Sorting with %llu elements and %d bits\n", number_of_elements,
number_of_bits);

    unsigned long long* rawArray = malloc(number_of_elements * sizeof(unsigned long long));
    unsigned long long* outputArray = malloc(number_of_elements * sizeof(unsigned long long));
    unsigned long long* countKeys;
```

```
if (!rawArray || !outputArray) {
    fprintf(stderr, "Memory allocation failed\n");
    free(rawArray); // It's safe to call free on NULL
    free(outputArray);
    return -1;
}

unsigned long long randomNum=0, maxNum=0, aux1=0, aux2=0, numKeys=1;
unsigned int bits;
int numIterations, i, j, randomValue;
double time_elapsed;
clock_t start, end;

// Random seed
srand((unsigned) time(NULL));
randomValue = rand();
init_genrand64(randomValue);

for (i=0; i<number_of_elements-1; i++){
    randomNum = genrand64_int64();
    rawArray[i] = randomNum;
}
randomNum = genrand64_int64();
rawArray[number_of_elements-1] = randomNum;

// Get the max number
maxNum = max(rawArray, number_of_elements);
numIterations = getNumberOfIterationsFromMax(maxNum, number_of_bits);

for(i=1; i<=number_of_bits; i++){
    numKeys = numKeys*2;
}

countKeys = malloc(numKeys * sizeof(unsigned long long));
if(!countKeys){
    fprintf(stderr, "Memory allocation failed\n");
    free(rawArray);
    free(outputArray);
    free(countKeys);
    return -1;
}

bits = (1U << number_of_bits) - 1;

// Start timing
start = clock();
for(i=0; i<numIterations; i++){

    // Faster than a loop to fill
    memset(countKeys, 0, numKeys * sizeof(unsigned long long));

    // Fill the key array
    for(j=0; j<number_of_elements; j++){
        aux2 = rawArray[j];
        aux1 = (aux2 >> (i*number_of_bits)) & bits; //aux1 has the bits we are studing in this iteration
        countKeys[aux1]++;
    }
```

```
        // Do cumulative sum
        for(j=1; j<numKeys; j++){
            countKeys[j]= countKeys[j-1] + countKeys[j];
        }

        // Fill the output array
        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 studing in this
            countKeys[aux1]--;
            outputArray[countKeys[aux1]] = rawArray[j];

        }

        // Faster than a loop to copy
        memcpy(rawArray, outputArray, number_of_elements * sizeof(unsigned long long));

    }
    // End timing
    end = clock();

    // Check if the array is sorted well
    for(j=0; j< number_of_elements-1; j++){
        if(rawArray[j] > rawArray[j+1]){
            printf("ERROR, the array is not well sorted\n");
            free(rawArray);
            free(outputArray);
            free(countKeys);
            return -1;
        }
    }

    printf("Success!, Array Sorted succesfully :)\n");
     // Calculate the time elapsed
    time_elapsed = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Time elapsed: %f seconds\n", time_elapsed);

    free(rawArray);
    free(outputArray);
    free(countKeys);
    return 0;
}


/*
    This function gets the Maximum value of the list
 */
unsigned long long max(unsigned long long *list, int lenght){
    unsigned long long maximum;
    int i;

    if (list == NULL || lenght<=0){
        return 0;
    }

    maximum = list[0];
```

```
    for (i=0; i<lenght; i++){
        if (maximum<list[i]) maximum=list[i];
    }
    return maximum;
}


/*
    This function gets the number of Iterations we will need to sort the elements.
    It would be done by dividing the max number by the number of bits were given
    when running the program
*/
int getNumberOfIterationsFromMax(unsigned long long maxNum, int numBits){

    int numberOfBitsMaximum = countBits(maxNum);
    int numberOfIterations = ((numberOfBitsMaximum-1) / numBits) +1;

    return numberOfIterations;
}


/*
    This function gives the number of bits are actually used in a number
    For example if the number if 9 in decimal, in binary would be represented
    as 000...1001, Therefore this function would return 4 as the maximum 1 in binary
    is at position 4
*/
int countBits(unsigned long long number)
{
    int currentBit, i;
    char numberInBits[MAXNBITS+1]="\0";

    for (i=0; i<MAXNBITS; i++){
        currentBit = number & 1;
        number >>= 1; //Shift the number 1 bit
        numberInBits[i]= '0'+currentBit;
    }

    i = MAXNBITS-1;
    for(; i>=0; i--){
        if(numberInBits[i] == '1'){
            break;
        }
    }

    return i+1;
}


void getStringFromNumber(unsigned long long number, char * numberInBits){
    int currentBit, i;

    for (i=0; i<MAXNBITS; i++){
        currentBit = number & 1;
        number >>= 1;
        numberInBits[i]= '0'+currentBit;
    }
}
```

radix_parallel.c

```c
#include "mt19937-64.c"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <time.h>
#include <omp.h>
/* #define MAX_NUMBER 8 */ //Only for developing
#define MAXNBITS sizeof(unsigned long long) * 8


int check_number_of_elements(int number);
unsigned long long max(unsigned long long list[], int lenght);
int getNumberOfIterationsFromMax(unsigned long long maxNum, int numBits);
int countBits(unsigned long long number);
void getStringFromNumber(unsigned long long number, char * numberInBits);


int main(int argc, char *argv[]) {
    if (argc != 4) {
        // At least 4 arguments
        printf("Usage: %s number_of_elements number_of_bits number_of_threads\n", argv[0]);
        return -1;
    }

    char *endptr;
    unsigned long long number_of_elements = strtoull(argv[1], &endptr, 10);

    // Check if the coversion was done successfully
    if (endptr == argv[1]) {
        printf("La conversión falló. Asegúrate de que ingresaste un número válido.\n");
        return 1;
    }

    int number_of_bits = atoi(argv[2]);
    if (number_of_bits<1){
        printf("\tERROR, the minimum number of bits is 1\n");
        return -1;
    }
    if (number_of_bits>=64){
        printf("\tERROR, the maximum number of bits is 63\n");
        return -1;
    }

    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);
```

```
    printf("Radix Parallel Sorting with %llu elements, %d bits, %d threads\n", number_of_elements,
number_of_bits, number_of_threads);

    unsigned long long* rawArray = malloc(number_of_elements * sizeof(unsigned long long));
    unsigned long long* outputArray = malloc(number_of_elements * sizeof(unsigned long long));
    unsigned long long* countKeys;

    if (!rawArray || !outputArray) {
        fprintf(stderr, "Memory allocation failed\n");
        free(rawArray); // It's safe to call free on NULL
        free(outputArray);
        return -1;
    }

    unsigned long long randomNum=0, maxNum=0, aux1=0, aux2=0, numKeys=1, index=0;
    unsigned int bits;
    int numIterations, i, j, randomValue;
    double time_elapsed;
    double start, end;

    // Random seed
    srand((unsigned) time(NULL));
    randomValue = rand();
    init_genrand64(randomValue);

    for (i=0; i<number_of_elements-1; i++){
        randomNum = genrand64_int64();
        rawArray[i] = randomNum;
    }
    randomNum = genrand64_int64();
    rawArray[number_of_elements-1] = randomNum;

    // Get the max number
    maxNum = max(rawArray, number_of_elements);
    numIterations = getNumberOfIterationsFromMax(maxNum, number_of_bits);

    for(i=1; i<=number_of_bits; i++){
        numKeys = numKeys*2;
    }

    countKeys = malloc(numKeys * sizeof(unsigned long long));
    if(!countKeys){
        fprintf(stderr, "Memory allocation failed\n");
        free(rawArray);
        free(outputArray);
        free(countKeys);
        return -1;
    }

    bits = (1U << number_of_bits) - 1;

    // Start timing
    start = omp_get_wtime();
    for(i=0; i<numIterations; i++){

        // Lets see
        for(j=0; j<numKeys; j++){
```

```
        countKeys[j]=0;
    }

    // Fill the key array
    // PARALELISE 1 - count phase
    for(j=0; j<number_of_elements; j++){
        unsigned long long aux2 = rawArray[j];
        unsigned int aux1 = (aux2 >> (i*number_of_bits)) & bits;
        countKeys[aux1]++;
    }

    // Do cumulative sum
    // PARALELISE 2 -Possible paralelise 2 (more challenging)
    for(j=1; j<numKeys; j++){
        countKeys[j]= countKeys[j-1] + countKeys[j];
    }

    // Fill the output array
    // PARALELISE 3 - Order the elements
    //#pragma omp parallel for
    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 studing in this
        index = countKeys[aux1];
        countKeys[aux1]--;
        outputArray[index-1] = rawArray[j];
    }

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


// End timing
end = omp_get_wtime();

// Check if the array is sorted well
for(j=0; j< number_of_elements-1; j++){
    if(rawArray[j] > rawArray[j+1]){
        printf("ERROR, the array is not well sorted\n");
        free(rawArray);
        free(outputArray);
        free(countKeys);
        return -1;
    }
}

printf("Success!, Array Sorted succesfully :)\n");
 // Calculate the time elapsed
time_elapsed = end - start;

printf("Time elapsed: %f seconds\n", time_elapsed);

free(rawArray);
free(outputArray);
free(countKeys);
```

```
    return 0;
}


/*
    This function gets the Maximum value of the list
*/
unsigned long long max(unsigned long long *list, int lenght){
    unsigned long long maximum;
    int i;

    if (list == NULL || lenght<=0){
        return 0;
    }

    maximum = list[0];

    for (i=0; i<lenght; i++){
        if (maximum<list[i]) maximum=list[i];
    }
    return maximum;
}


/*
    This function gets the number of Iterations we will need to sort the elements.
    It would be done by dividing the max number by the number of bits were given
    when running the program
*/
int getNumberOfIterationsFromMax(unsigned long long maxNum, int numBits){

    int numberOfBitsMaximum = countBits(maxNum);
    int numberOfIterations = ((numberOfBitsMaximum-1) / numBits) +1;

    return numberOfIterations;
}


/*
    This function gives the number of bits are actually used in a number
    For example if the number if 9 in decimal, in binary would be represented
    as 000...1001, Therefore this function would return 4 as the maximum 1 in binary
    is at position 4
*/
int countBits(unsigned long long number)
{
    int currentBit, i;
    char numberInBits[MAXNBITS+1]="\0";

    for (i=0; i<MAXNBITS; i++){
        currentBit = number & 1;
        number >>= 1; //Shift the number 1 bit
        numberInBits[i]= '0'+currentBit;
    }

    i = MAXNBITS-1;
    for(; i>=0; i--){
        if(numberInBits[i] == '1'){
```

```
            break;
        }
    }

    return i+1;
}


void getStringFromNumber(unsigned long long number, char * numberInBits){
    int currentBit, i;

    for (i=0; i<MAXNBITS; i++){
        currentBit = number & 1;
        number >>= 1;
        numberInBits[i]= '0'+currentBit;
    }
}
```