EC527 Final Project: Radix Sort

Roberto Garcia Plata

December 18, 2018

Contents

1. Description of the Algorithm

Radix Sort is a non-comparative sorting algorithm whose idea is to sort an array by comparing digit by digit from least significant to most significant.

1.1 Time Complexity Analysis

Most sorting algorithms take O(nlog(n)), such as merge sort, heap sort and others. Radix Sort is a big improvement since it will allow us to do sorting in a time of O(n+k) where elements are in the range from 1 to k. This complexity is possible because radix sort uses Counting sort as a subroutine to compute its result. One problem for counting sort arises when the elements are in range from 1 to n^2. This would give us a time complexity of O(n^2) which is worse than comparison-based algorithms. However, we can bring this down to O(n) by choosing carefully the array that will be sorted. Radix sort as explained now would take O(d*(n+b)) where d is the number of digits and b is the base representing the numbers. The value of d can be defined as O(log_b(k)) by considering the fact that our numbers have to be in the range of 1 to k. This makes our time complexity look a little more like O(log_b(k)*(n+b)). This however is starting to look just like the time complexity of comparison-based algorithms for a large value of k. This can then be improved by limiting k. Say k <= n^c where c is a constant; that would mean our new complexity is O(nlog_b(n)). This is still not good enough to beat comparison-based algorithms, so we have to get rid of the log_b(n) part of the expression. If we set b=n then what we are left with is simply O(n). What all of this means is that we can sort an array of integers with range from 1 to n^c if the numbers are represented in base n. This makes radix sort one of the fastest algorithms available for sorting arrays. The challenge of this project is to make this algorithm perform even faster by doing hardware aware coding. In order to understand how this will be done it is necessary to not only understand the time complexity of the algorithm, but also how the algorithm itself works.

1.2 Radix Sort Algorithm

As mentioned before, radix sort uses counting sort as a subroutine. So, let's start by focusing on how counting sort works. Counting sort works by counting the amount of times a particular key is repeated in the input array. In this case, counting sort will first count how many occurrences of a digit there are in our array starting with the least significant digit. This counting is done is a separate array of size 10 (because we assume that our numbers are in base 10 so we only have those many digits available.) Once this counting array is filled (sometimes known as buckets) it has to be modified to correctly depict the position of the elements. This means each element will add its current value to the value of the previous element starting from element 1 (element 0 stays the same since nothing comes before it). This new version of the counting array now shows

the position of every element in our output array. The way the output array is built is by processing again one by one the elements of our input array and using the counting array to find their position. The way it works is the following: you look at the current element in the input array and see the value of that same index in the counting array. That value is the position where that element goes into the output array, and once you place it where it goes you have to decrease the count of that element by 1. This is hard to visualize when explained only written like this, so the following image is used to show in a better was how radix sort works.

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

Figure 1. A simple example of counting array. Taken from www.geeksforgeeks.org/counting-sort/ on 12/17/18.

1.3 Radix Sort

Now that the subroutine for radix sort has been explained it is possible to see why this algorithm is an improvement over counting sort. The reason is very simple: radix sort first considers the maximum value in the array and uses that to perform and limit counting sort. Knowing this maximum value will allow us to do counting sort as many times as necessary (one time per each digit) because we will stop doing counting sort when we are operating on a digit beyond the ones

available in our maximum number. In other words, radix sort first finds the maximum value of the input array and then makes a for loop starting with variable exponent set as 1, iterating until exponent/maximum is less than 0 where the exponent increments by 10 with each iteration. This exponent variable allows us to look at the digit that we want. And incrementing it by 10 is acceptable if the base the numbers are represented in is decimal. This way you use the maximum number to know when to stop and you sort the input array from least significant digit to most significant digit using counting sort.

2. Scalar CPU Version

2.1 Functions used to implement Radix Sort

Since radix sort is a very standard algorithm there are already many implementations of it available online. All implementations use three functions to do radix sort and main. These functions are: (1) getting the maximum value in the array, (2) count sort, and (3) radix sort that uses both previous functions.

(1) Getting the maximum is a simple task that runs in O(n) complexity. There are many ways to find a maximum, and I decided to use one that was simple and efficient. The way I find the maximum is by assuming that the first element in the array is the maximum number and then going through the whole array: if there is an element that is bigger than the current maximum then I make that the new maximum and continue going evaluating the next elements. Therefore, all this function needs as an argument is the array whose maximum you want to find.

(2) Count sort is implemented as explained before. This function takes as parameters the array you want to sort and an exponent variable (exp) which will be used to determine what digit you are sorting with respect to. This function has three for loops: one for filling the counting array, one for summing the elements of that array and a last one to create the output array. If necessary this function can include a fourth for loop to copy the output array back into the input array so that you can return a pointer to the same array that you passed as an input (only now that array will be sorted.)

(3) Radix Sort is the function that brings the previous two together. Radix sort first calls the function to find the max of the array it's trying to sort. Once it has determined that value it will call counting sort from inside a for loop as many times as it is necessary. The amount of times counting sort will be called depends on the maximum value.

The last thing we need is a main function. The main function will pass the array that we want to sort to the radix sort function. There is another thing that hasn't been talked about, and that is how this input array is initialized. This can be done in main, or as I chose to do it, in a separate function. The way I create the input array is by using rand( ), this way the elements will be random, and most importantly, un-sorted. To create random elements between [1,1000] I use rand()%1000+1. The function to initialize them is then a for loop from 1 to n and I set the value of each individual element to be random using the previous expression. I also made a function to print the arrays (both initial and final) just to make sure the algorithm was correctly implemented.

2.2 Code

```
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;
}
```

Figure 2. Code for the function used to get the maximum value in the array.

```
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;

    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Figure 3. Code used to implement counting sort

```
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Figure 4. Code used to implement radix sort. Note that it uses both of the previous functions to calculate its output.

```
int main()
{

    int *array1;
    int n = ARRAY_SIZE;

    init_array(array1, n);

    printf("Old array:");
    print(array1,n);
    printf("\n");

    radixsort(array1, n);
    printf("New array:");
    print(array1, n);
    printf("\n");
    return 0;
}
```

Figure 5. Main function. This function shows how you initialize a random array and perform radix sort on it while printing it before and after to check for correctness.

2.3 Results from Scalar implementation

The results from this implementation will give us the baseline idea of how well this algorithm performs so we can look to optimize this as much as possible. The following CPEs were obtained by varying the size of the array being sorted.

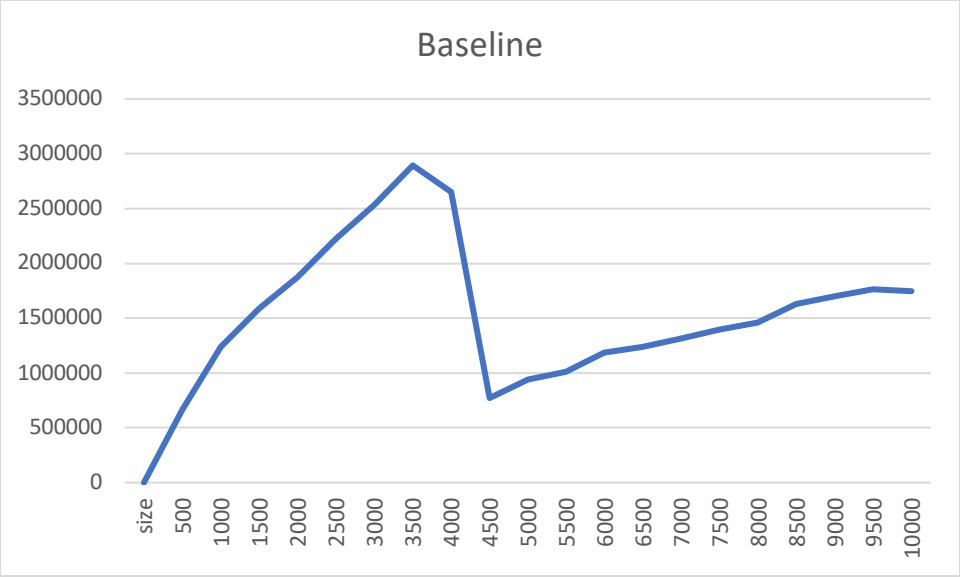

Figure 6. This image shows how the CPE changes depending on the size of the array. It is interesting to see that for the n value of 4000 this decreases considerably only to increase steadily again.

The point of doing this is, as mentioned before, to see what is the performance of the baseline and compare that to the final optimized product. There is very little that can be drawn from looking at the baseline performance, other than seeing the CPE of this implementation of radix sort. However, this baseline data will be more useful later on to make the previously mentioned comparisons.

3. Optimized Implementation

Optimizing radix sort proved to be a challenging task. Even though it looks simple since there are only a few functions, finding a good way to optimize them is not as easy as it seems. My first approach was to see what parts of the code could be optimized. The code has two main components, which are getting the maximum value and doing count sort. Count sort uses three for loops, so it is worth it to look into all of them to see what can be done to make them faster. Now the task has been narrowed down, I can look to optimize the three for loops and the getMax function.

3.1 Analysis of Possible Optimizations

(1) Optimize getting the maximum.

There are many ways to get the maximum value of an array to the point where programming languages have built in functions to do this. Since I didn't know exactly what went behind the curtains with these functions I decided to make one of my own.

The simplest and most efficient implementation that I could find was the one that was explained in the scalar version. There are few options to considerably optimize this. Vectorizing is not an option since I need to look into each individual element in the array. Threading is also not an ideal option since one thread could set the maximum variable equal to a local maximum in the scope of the thread, but there can be a bigger maximum elsewhere in the array. Unless properly synchronized, using threads to find the maximum is not the best solution. One thing that could be done is unrolling the loop so that the for loop has to be done less times. This is a simple improvement that can be easily put in place. I decided to unroll by 4 to see whether this would improve the performance of my code or not.

(2) Optimize the for loop that fills the count array.

This for loop is the one that was optimized the most. I did some research and most papers focus on the optimization of this loop. This is possible by parallelizing. However, there is a catch to this. In order to properly count how many times a digit occurs in the input array it is necessary to have a 2D array of counting buckets instead of just a 1D array as needed for the serial version. The reason for this is because several threads might be trying to increment the same bucket simultaneously and then only one of these increments would take effect. So, making a 2D array where every row is for each processor to have access to solves this problem. That way once the 2D array is filled, all you have to do is add the columns and save that result to the same final buckets array. This way that particular for loop can be parallelized and the whole program made faster.

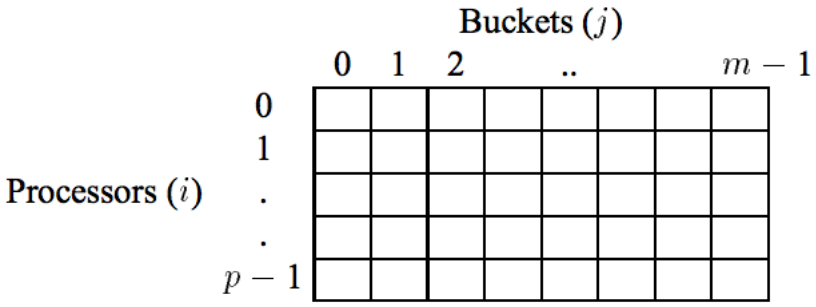

Figure 7. 2D array of counting buckets to account for different threads accessing the same bucket element simultaneously.

(3) Optimize the for loop that sums the count array.

Unfortunately, this loop cannot be parallelized. There are too many dependencies (every element depends on the sum of the previous one). The best I can attempt to do is to unroll this loop as well (as done with the getMax function).

(4) Optimize the for loop that creates the output array.

Once more time, this loop cannot be parallelized. The dependencies here are that you have to decrease by one the count of whatever index you are working on. This causes the same problem as with filling the buckets, meaning that if several threads try to access the same bucket element simultaneously only one decrease would take effect and this would give us the wrong output. Therefore, this loop cannot be vastly optimized.

3.2 Vectorizing, Parallelizing and Final Approaches

The previous comments on each individual part of the code that I attempted to optimize were the final implementations of what I did. However, I tried many things before arriving to said conclusions. The first recommendations that were given to me were to try to vectorize it. However, using vectors requires a very specific set up, which is that all the elements on the array should have the same operation done to them. One good example of this would be summing all the elements in an array. With a problem like this one, vectorizing proves to be very useful since all elements require to be summed with each other regardless of the order in which this is done. This is not the case with radix sort since the order in which things are done matter. There were some research papers that gave some ideas on how to vectorize this process. However, many of these ideas were hard for me to understand, and the pseudocode provided in most papers is not very explanatory. I decided that if I was to attempt any optimizations it should be on things that I already know am capable of doing. Therefore, I decided that vectorizing was not the route I would take (even though initially I set up my code to be vectorizable), but I would focus on simple unrolling optimizations and on parallelizing a small part of this whole process. I am hoping that these choices will yield a better performance time.

3.3 Code

This section was included here to be consistent with the way the scalar version was discussed in this report (and to adhere to the example report that was given to us). Since this is the main focus of the project, I included the code used for the optimization in a part of its own. The code can be found under “6. Optimized Code”.

3.4 Results from Optimized Radix Sort

Optimizing radix sort wasn't a regular optimization task. Most of the algorithms that are tried to be improved usually are parallelizable in a way that will improve performance considerably. If this is not the case then you can find a way to vectorize it. In my case this algorithm only had very few places where it could be optimized, and I found that most of the optimizations that could be done require deep understanding of vectorizing. This is why I decided to try as many things as I could come up with, and see which ones would actually work. Before I can present my results regardless if their performance was better, I had to make sure they were correct. This is why I will only be discussing the results that were fully successful, and I decided to leave out any other (incomplete) attempts that didn't perform as expected.

I had a few tools at my disposition. One of them was doing basic optimizations through loop unrolling. The other ones were vectorizing the code, parallelizing, and implementing Radix Sort in a GPU. Using a GPU didn't seem like an ideal option, since GPUs are better used with compute-intensive, parallel computations. The main issue with using a GPU was that this algorithm is basically not parallelizable because of all the data dependencies. The only part that could be done in parallel was one for loop inside the counting sort function. This for loop fills the counting array to keep track of how many occurrences there are of a certain digit. So, it seemed excessive (and pointless) to try and use a GPU to improve this. However, parallelizing that small part of the code, whether with OpenMP or creating the threads myself, seemed like it was worth the try.

The other options left for me to use were unrolling loops and vectorizing. Vectorizing is ideal when you have to perform the same operation on all elements, an there is little to no need of using individual elements of the data structure. This is the case since vectors won't look at the individual elements, but at chunks of the data structure on which they can operate. At least, that is my understanding on vectors. So, finding a way to vectorize this seemed too complicated for me to come up with. This is why I did some research to see how other people had optimized this code. I found that, contrary of what I thought, vectorizing can be used to optimize radix sort. The problem was that the way this is vectorized was beyond my abilities for the moment. I tried for several days to implement what those papers proposed in pseudocode, but I had no success. This is the reason why I limited my final code to only use simple optimizations (unrolling the loop) and a small parallelization.

In essence, my successful optimization attempts are focused on two approaches: unrolling for loops and parallelizing the for loop that fills the counting array. The following are my results using the two previous techniques that I mentioned.

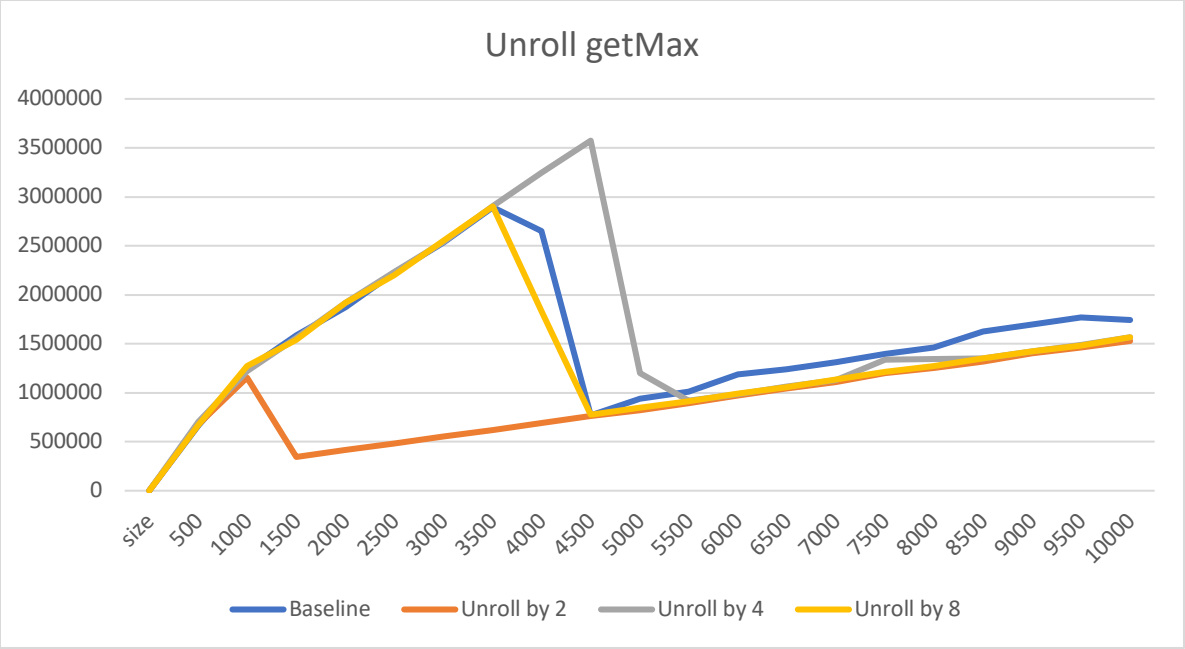

Figure 8. This figure shows my first approach at optimization, which is unrolling the for loop inside the getMax function. The purpose of this was to find what would be the ideal loop unrolling factor that I could use across all other for loops that allowed for it. Looking at this graph it might seem that all unrolling factors yield the same performance, but looking at the actual values lets us see that a factor of 2 performs a little better than the other factors.

As mentioned in the previous figure, unrolling by a factor of 2 seemed like the best option. This was implemented throughout the for loops that were available for this optimization. These for loops were (other than the one inside getMax) for loop number 2 in the count sort, which adds the values of the already filled counting array, and the loop that copies the data of the output array back to the input array. The first loop in counting sort wasn't optimized like this because this was the loop that was aimed to be parallelized.

The parallelization did not go as expected. I thought I would get a really good performance by parallelizing this loop, but in reality, performance just became much worse. The following are my results after paralleling the for loop using OpenMP. My approach was to put around the first for loop the command #pragma omp parallel for. I created a 2D array of size [Number of Threads] [10]. The number of threads I decided to use were 4 since I was doing this project on a 4-core machine. Once inside the for loop I index into a row by getting the id of the thread (omp_get_thread_num) and index 'i'. This way I built the 2D array necessary for the parallel version.

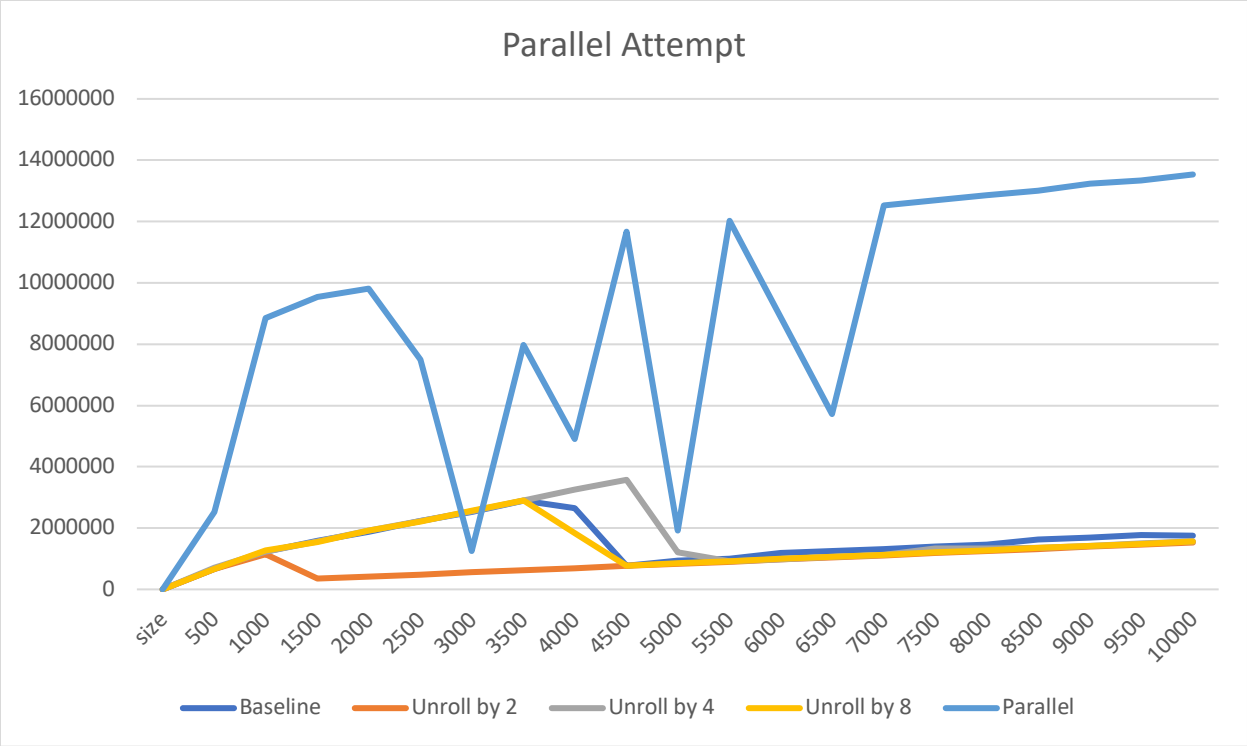

Figure 9. The function that clearly has the worse runtime is the one where the counting sort for loop was parallelized. This seems very unstable, so it's hard to tell what is really going on in there. One thing that is clear is that this algorithm is clearly better without trying to make it parallel in any way.

As the previous figure clearly shows, parallelizing the code didn't go as expected. Event when the function has a minimum it is still not better than the other implementations. The way the code was parallelized was just as recommended by Zagha and Blelloch in their paper "Radix Sort for Vector Multiprocessors". They describe the method to do it (creating a 2D array) and provide pseudocode for how they did it. However, in their case they successfully optimized the algorithm considerably. This is probably because they actually implemented vectorization in their code, something I did not achieve. Taking all the previous things into account, I decided to provide three different files with three different versions of radix sort. One file is the serial version, the other one only uses unrolling, and the last one parallelizes the for loop inside counting sort. However, the best performance comes from only unrolling the loops, a very basic (and not so useful) optimization.

4. Closing Thoughts

Optimizing code by being hardware aware is not an easy task. Doing labs where you know what the results are going to be is one thing, but actually using the skills learned in EC527 to optimize an already existing algorithm with no instruction on how to do it is another thing of its own. I found that many times it is necessary to try (and fail) many different things until something works. Following what other people have done is useful, but sometimes the pseudocode provided is very hard to implement. It is interesting to note that some ways of optimizing code might not always work as in my case with the parallel approach. It is important to understand why this could be the case and to seek for further methods of optimization. It is also important to note that most of these research papers are done using state of the art machines that were created for such tasks. However, the most useful conclusion I can take is that optimizations through hardware aware programming are still a key element in today's practices and there is a lot of room for exploration.

5. Compilation Instructions

Serial version:

```
gcc -std=c99 radix_sort_serial.c -o serial
```

Unrolled version:

```
gcc -O1 -mavx radix_sort_vector.c -lrt -lm -o vector
```

Parallel version:

```
gcc -O1 -fopenmp rs_vec_parallel.c -lrt -lm -o parallel
```

6. Optimized Code

```
// A utility function to get maximum value in v
long int getMax(vec_ptr v)
{
  long int i;
  long int get_vec_length(vec_ptr v);
  data_t *get_vec_start(vec_ptr v);
  long int length = get_vec_length(v);
  data_t *data = get_vec_start(v);
  long int max = data[0];

  for (i = 1; i < length; i++) {
    if (data[i] > max) max = data[i];
  }
  return max;
}
```

Figure 10. This figure shows the first approach at optimizing the code. I decided to use vector structures instead of arrays to hold my data. In here I get the length and a pointer to the first element of the vector that is passed in the arguments of the function.

```
void countSort(vec_ptr v, data_t exp)
{
  long int i;
  long int get_vec_length(vec_ptr v);
  data_t *get_vec_start(vec_ptr v);
  long int length = get_vec_length(v);
  data_t *data = get_vec_start(v);

  vec_ptr output = new_vec(ARRAY_SIZE); // output vector
  data_t *data_output = get_vec_start(output);

  vec_ptr count = new_vec(10);
  init_vector_zero(count, 10);
  data_t *data_count = get_vec_start(count);


  // Store count of ocurrences in vector count
  for (i = 0; i < length; i++){
    data_count[ (data[i]/exp)%10 ]++;
  }

  // Change data_count[i] so that it now contains actual
  //  position of this digit in output
  for (i = 1; i < 10; i++){
    data_count[i] += data_count[i-1];
  }

  // Build the output array
  for (i = length - 1; i >= 0; i--){
    data_output[data_count[ (data[i]/exp)%10 ] - 1] = data[i];
    data_count[ (data[i]/exp)%10 ]--;
  }

  // Copy output to v
  for (i = 0; i < length; i++){
    data[i] = data_output[i];
  }

}
```

Figure 11. This figure shows count sort using vector structures.

```
long int getMax(vec_ptr v)
{
  long int i;
  long int get_vec_length(vec_ptr v);
  data_t *get_vec_start(vec_ptr v);
  long int length = get_vec_length(v);
  data_t *data = get_vec_start(v);
  long int max = data[0];

  // Unroll by 2
  for (i = 1; i < length - 1; i+=2) {
    if (data[i] > max) max = data[i];
    else if (data[i+1] > max) max = data[i+1];
  }

  // Finish remaining elements
  for (; i < length; i++){
    if (data[i] > max) max = data[i];
  }

  return max;
}
```

Figure 12. This is the final getMax function used. As it can be seen it is similar as the one shown in Figure 10 but with the for loop unrolled by a factor of two. The extra for loop is needed to get the remaining elements of the vector.

```
void countSort(vec_ptr v, data_t exp)
{
  long int i;
  long int get_vec_length(vec_ptr v);
  data_t *get_vec_start(vec_ptr v);
  long int length = get_vec_length(v);
  data_t *data = get_vec_start(v);

  vec_ptr output = new_vec(ARRAY_SIZE); // output vector
  data_t *data_output = get_vec_start(output);

  vec_ptr count = new_vec(10);
  init_vector_zero(count, 10);
  data_t *data_count = get_vec_start(count);

  //************************* FOR LOOP 1 *********************************
  // Store count of ocurrences in vector count
  int bucket_2d[THREADS][10] = {0};
  int tid;
  omp_set_num_threads(THREADS);

#pragma omp parallel for
  for (i = 0; i < length; i++){
    tid = omp_get_thread_num();
    bucket_2d[tid][(data[i]/exp)%10]++;
  }

  for (i=0; i < 10; i++){
  data_count[i] = bucket_2d[0][i] + bucket_2d[1][i] + bucket_2d[2][i] + bucket_2d[3][i];
  }
```

Figure 13. This figure shows the first half of the new implementation of count sort. The main change happens after the first for loop. In here I declared that particular for loop as a parallel omp section. This allows for that computation to be done in different threads. Inside the loop I index by row into bucket_2d by using the thread id. I index into bucket_2d by column by using 'i' which is incremented by the for loop with every iteration. The second for loop is used to sum that array column wise and store the results back in data_count, the vector structure that will be used in the following for loops of that function.

```
//************************* FOR LOOP 2 *********************************
// Change data_count[i] so that it now contains actual
//  position of this digit in output
// Unroll by 2
for (i = 1; i < 9; i+=2){
  data_count[i] += data_count[i-1];
  data_count[i+1] += data_count[i];
 }

// Finish remaining elements
for (; i < 10; i++){
   data_count[i] += data_count[i-1];
}



//************************* FOR LOOP 3 *********************************
// Build the output array
for (i = length - 1; i >= 0; i--){
  data_output[data_count[ (data[i]/exp)%10 ] - 1] = data[i];
  data_count[ (data[i]/exp)%10 ]--;
}

//*************************EXTRA FOR LOOP*********************************
// Copy output to v with unroll by 2
for (i = 0; i < length - 1; i+=2){
  data[i] = data_output[i];
  data[i+1] = data_output[i+1];
}

for (; i < length; i++){
   data[i] = data_output[i];
}

}
```

Figure 14. This image shows the second half of the new count sort function. The second and extra for loops (extra because it just copies the results back to the input array, but it is not required for the actual sorting) are unrolled by a factor of 2 (determined to be the ideal factor previously by experimenting with getMax). As usual, unrolling requires an extra for loop to finish the last elements that haven't been worked on.

7. List of Files

The files included in the Garcia_Roberto_FinalProject.zip file are:

- EC527 Final Project, which is this report
- `radix_sort_serial.c`, which is the baseline code
- `radix_sort_vector.c`, which is the vector version
- `rs_vec_parallel.c`, which is the parallel and unrolled version