Condor Sort

By Thomas. A. Lock

thomas.lock.personal@gmail.com

Abstract

This paper will present a new novel sorting method that is based on partition sorting similar to the *Danish National Flag algorithm, and how implementations can optionally leverage the structure of entities being sorted to optimize sorting while minimizing memory usage and considering modern programming paradigms.

Sorting is simply the task of arranging a set of entities, where all entities are part of another set where they each have a hierarchal relationship with at least 1 other entity in the set (2 comes after 1, 3 comes after 2 or B comes after A and C comes after B). Most digital implementation of these hierarchies have characteristics that can be leveraged as I will show below.

Define Boundaries

The first step in implementingtu Condor is partition your problem into 3 smaller problems. So the first decision to make is how to perform this partitioning. This decision here is key for achieving the fastest possible performance out of the algorithm while keeping it stable. Typically sorting algorithm split into 2 broad types, integer and comparative. For this paper I would like to think of this methodology to be more specifically referring to structural and comparative sorting.

- For a structural approach I simple can partition our values based on our memory limits allowed. In the case of a 32bit unsigned integer I could use the first and last 8bit values as boundaries.
- The Comparative approach you can simple pick a minimum of 3 values (I will call these landmarks) from your full set which you then sort quickly most likely using **Insertion sort. These values are now your boundary values. This can be very useful feature, since for a small memory footprint you can significantly increase efficiency.

Value Swapping

Now that we have our boundary values, the next step is to separate all values into these individual groups and the approach used here differs depending on boundary defining approach. It also differs from *Danish National Flag algorithm in that counting is optional and rather than flowing in a single direction the iteration move inwards towards the middle.

- Structural can use the counting method to determine how many there are in each region and simply write/swap those values in. Most times in computing you are sorting objects or unique valued structures and simply rewriting values anywhere is not an option. So for a stable sort you must swap values from left to right into the correct region starting from the lowest position in the array for the region.
- Comparative uses the ends of the array moving towards the middle. While iterating over the values, values less than or equal to the first landmark value are moved to the beginning position

which is tracked and increments on each addition. If the value is greater than the last landmark value it is swapped to the end of array, the position is tracked and decremented.

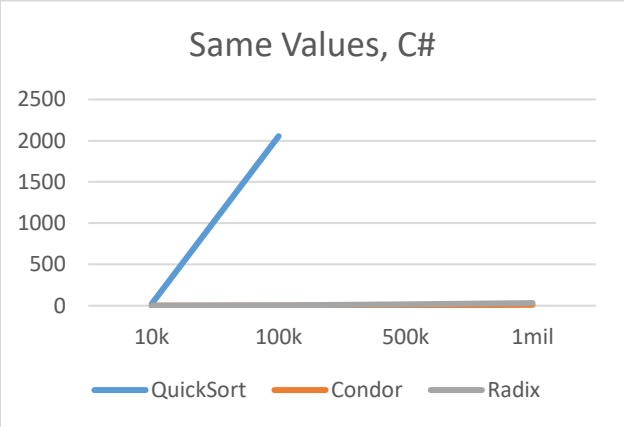
At this point if no values are greater than the last landmark or less than first and first and last landmark are equal, than all value must be equal and with a single pass you have eliminated 1 set of ordered arrangements regardless of value while also preserving their existing order.

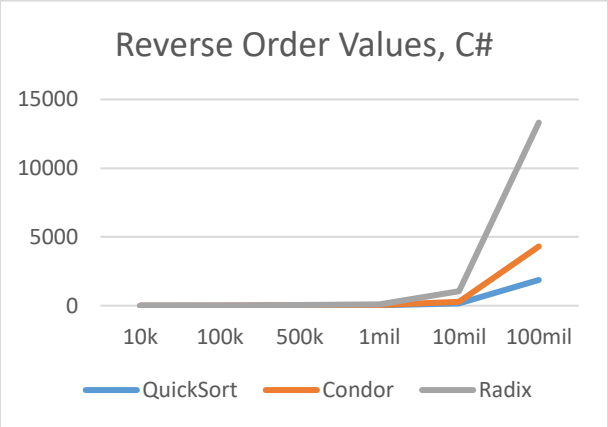
If they are not all the same values then with these new isolated regions you recursively loop over each separately. Once you reach a region that has less than 10 values it is highly efficient for you to simply sort using a method like **Insertion sort.

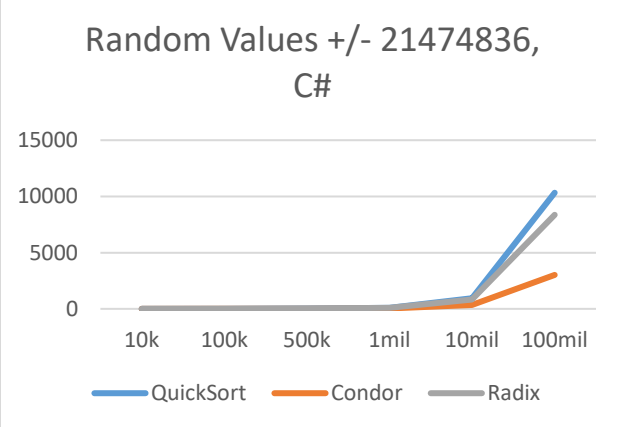
Time Complexity

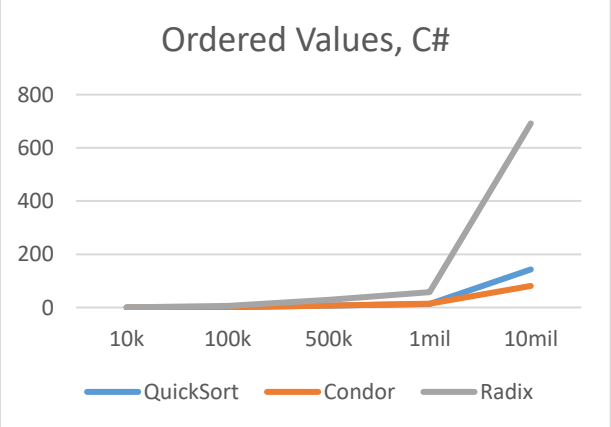
Based on these 2 key approaches, Condor has a time complexity ranging from O(n*k) where k is the amount of triple partitioning occur, to O(n*l) where l refers to how many levels for ex. 8bit counting gives 4 levels to a 32bit value. To test the time complexity of this approach, I benchmarked results against **Merge Sort, **QuickSort and **Radix implementations. These benchmarks were performed using 2 languages (C# and Objective C), 3 different platform (Windows, OSX and iOS) and 2 processor types (ARM and Intel).

In addition to different coding environments, I also tested Condor using different data set arrangements. This was especially key as some sorting algorithms perform sorting more efficiently depending on the data arrangement.









These are results using Windows 10, C# on Intel Processor with all results in milliseconds and data sets ranging from 10,000 – 1 million. The implementation utilized 3 tasks to complete the sorting.

Same Value Results (ms)				
	10k	100k	500k	1mil
QuickSort	2.9994	22.9993	123.0241	255.8526
Merge Sort	1.9996	15.5364	81.6172	169.8308
Condor	1.0007	2.9957	9.0029	18.0169
Radix	2.9993	20.9936	98.999	198.0373
Ordered Value Results (ms)				
	10k	100k	500k	1mil
QuickSort	2.002	7.999	34.5339	70.8166
Merge Sort	1.9988	15.9993	82.0054	180.9053
Condor	0.9855	5.991	28.9985	58.1065
Radix	1.576	21.06	98.0246	197.6125
Reverse Ordered Value Results (ms)				
	10k	100k	500k	1mil
QuickSort	1.9999	9.4351	40.9972	82.0167
Merge Sort	1.9995	19.5873	101.0025	201.7919
Condor	1.0006	7.9945	31.0726	61.0553
Radix	2.9992	21.0024	100.5052	197.1187
Random Values (0-256) Results (ms)				
	10k	100k	500k	1mil
QuickSort	3.6736	24.006	130.5556	269.4228
Merge Sort	2.9998	26.002	138.4812	285.7596
Condor	0.9997	2.9985	8.9973	18.9992
Radix	2.9969	21.0025	100.5646	209.0212
Random Values (+/-21474836 Results (ms)				
	10k	100k	500k	1mil
QuickSort	4.0049	28.0179	144.7297	304.4275
Merge Sort	2.9999	29.0129	160.9556	335.6161
Condor	1.9999	12.05	51.0648	89.0149
Radix	2.9994	21.0083	98.2295	198.12

These are results using Windows 10, C# on Intel Processor with all results in milliseconds and data sets ranging from 10,000 – 1 million. The implementation utilized 3 tasks to complete the sorting.

Random Values +/- 21474836, Results (ms)						
	10k	100k	500k	1mil	10mil	100mil
QuickSort	2.81	13.35	62.04	129.07	1467.13	16556.45
Condor	1.0007	6.46	21.73	42.73	665.7	5614.33
Radix	2.66	9.04	39.49	80.19	773.9	14421.63

These are results using iOS, Objective C on ARM Processor with all results in milliseconds and data sets ranging from 10,000 – 100 million. The implementation utilized 3 threads to complete the sorting.

Reverse Ordered Value Results (ms)						
	10k	100k	500k	1mil	10mil	100mil
QuickSort	0.21	1.92	8.72	16.66	164.04	1869.84
Condor	0.27	1.22	11.93	22.63	125.08	2438.55
Radix	0.65	6.4	33.24	61.55	760.15	9006.6
Random Values +/- 21474836, Results (ms)						
	10k	100k	500k	1mil	10mil	100mil
QuickSort	0.76	6.83	39.25	81.85	938.85	10328
Condor	0.41	2.56	12.71	25.22	345.33	3017.19
Radix	0.92	7.81	38.14	78.1	819.09	8362.47
Ordered Value Results (ms)						
	10k	100k	500k	1mil	10mil	100mil
QuickSort	0.14	1.53	6.69	12.94	143.04	
Condor	0.27	1.15	7.69	14.74	81.09	
Radix	0.63	5.75	29.22	57.75	691.62	
Same Value Results (ms)						
	10k	100k	500k	1mil	10mil	100mil
QuickSort	21.92	2054.69	failed	failed		
Condor	0.26	1.38	5.26	10.31		
Radix	0.42	3.48	16.59	31.56		

These are results using iOS, Objective C on Intel Processor with all results in milliseconds and data sets ranging from 10,000 – 100 million. The implementation utilized 3 threads to complete the sorting.

Space Complexity

Condor space complexity is easily visible since the method of sorting is done in place on the existing data set. This gives condor a O(1) since no additional memory is required using the comparative method. If the hybrid counting method is used, Condor has to allot additional memory for counting to determine boundaries giving it a complexity of O(k) where k can vary depending on implementation but I recommend 8bit boundary as used in these example implementations. This give k a value of 256 x 32bit x #ofThreads.

Implementation Methods

Threading

Condor can be implemented using single or multiple threads to complete the task. By being able to utilize multiple threads easily, Condor is ideal for sorting very large data sets like in large databases or in competitions such as Daytona or Indy Sorting Benchmarks. Utilizing multiple threads for very small data set sorts is not recommended as the processing cost of creating new threads is too high.

Comparative vs Hybrid

As illustrated above, Condor can be written as either just a comparative or a hybrid sort that leverages the data structure. The benefit of leveraging all available information about the data structure allowed us to easily convert an algorithm from O (n*k) too O (n*l) (where l < k) in time and turning it from O (1) to O (k) in space requirement.

Language

Although programming languages shouldn't in principle have any impact, it can actually have huge ramifications on implementation. Languages like Objective C and C# both treat arrays as classes and store information on the heap. This allows for multiple threads to work on the same object so long as they work on different areas to avoid collisions or locks. Other languages may treat them as structs storing information on the stack which multiple threads can't work on but is memory efficient.

Conclusion

Condor sort has demonstrated that depending on implementation requirements it is able to outperform **Merge Sort, **QuickSort and **Radix at almost every single set size ranging from 10k-100million as well as many data arrangements. Based off the 47 benchmarks tests I ran, Condor ranked #1 40 times and #2 7 times making it extremely balanced algorithm of time and space complexity where many other algorithm are strong in one area but weak in another. Condor's implementation can also range from a simple small code base to a very verbose high performance algorithm which enables it to have broad use cases.

Download Source Code

Condor supports System Types, NSNumber, and Object with using a specific properties that returns an Int32, Int64, Float or Double.

Objective C version: https://github.com/thomasrunner/Condor

C# Nuget: https://www.nuget.org/packages/Condor/

References

* "Dutch National Flag problem and algorithm". Faculty of Information Technology (Clayton), Monash University, Australia. 1998.

**Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.

Swift Code Example

Key Features of this implementation

- Time Complexity is O(n*k)
- Space Complexity is O(n), due to the nature Swift implementation of array
- Generic Implementation
- This is working code

```
class CondorCompareSorting<T: Comparable>{

  var items:[T] = []

  func condorSort(_ input: [T]) -> [T]

  {

    items = input

    condor(start: 0, end: items.count - 1)

    return items

  }


  func condor(start: Int, end:Int) {

    var i = start

    var j = end

    if abs(j - i) < 1 { return }

    if j - i < 10 {
```

```
    inlineInsertionSort(low: i, high: j )
    return
}

//Select landmarks using any method you like and select as many sample as you like
var landmarks:[T] = []
landmarks.append(items[start + (end - start)/4])
landmarks.append(items[start + (end - start)/2])
landmarks.append(items[start + (end - start) * (3/4)])
landmarks = insertionSort(landmarks, by: >)

var q = start
var k = end
while q <= k {
    //Moving from the start of array inward
    if self.items[q] <= landmarks[0] {
        let val = items[i]
        items[i] = items[q]
        items[q] = val
        i += 1
        q += 1
    } else if items[q] >= landmarks[2] {
        let val = items[j]
        items[j] = items[q]
        items[q] = val
        j -= 1
        k = k > j ? j : k
    } else {
        q += 1
    }
```

```
        //Moving from end of array inwards
        if self.items[k] <= landmarks[0] {
            let val = items[i]
            items[i] = items[k]
            items[k] = val
            i += 1
            q = q < i ? i : q
        } else if items[k] >= landmarks[2] {
            let val = items[j]
            items[j] = items[k]
            items[k] = val
            j -= 1
            k -= 1
        } else {
            k -= 1
        }
    }

    //Repeat until done.
    if i <= j {
        condor(start: start, end: i - 1)
        condor(start: i, end: j)
        condor(start: j + 1, end: end)
    }
    return
}

func inlineInsertionSort(low:Int, high:Int) {
    if low < high {
        for i in (low + 1)...high {
            let key = items[i]
```

```
        var j = i - 1
        while j >= 0 {
            if key < items[j] {
                items[j + 1] = items[j]
                j -= 1
            } else {
                break
            }
        }
        items[j + 1] = key;
    }
  }
}

func insertionSort<T: Comparable>(_ input: [T], by comparison: (T, T) -> Bool) -> [T]
{
  var items = input

  for index in 1..<items.count
  {
    let value = items[index]
    var position = index

    while position > 0 && comparison(items[position - 1], value) {
      items[position] = items[position - 1]
      position -= 1
    }

    items[position] = value
  }
  return items
```

```
  }
}
```

C# Code Example

To illustrate the flexibility of this algorithm, I am sharing a verbose multi tasked C# version example. This version is more ideal for larger data sets as the Task functionality adds a lot of additional computing overhead that has nothing to do with sorting.

Key Features of this implementation

- Time Complexity is O(n)
- Space Complexity is O(k)
- Limiting to 3 Threads to avoid thread explosion
- Stable Sorting with counting boundary starting positions
- This is working code

```
//32BIT UNSIGNED
private static UInt32[] numbers32;

public static void Sort(UInt32[] array)
{
    numbers32 = array;
    size = array.Length;

    MultiThreadPartitioning(0, size - 1, 0);
}

private static void MultiThreadPartitioning(int low, int high, int shift)
{
    int[] added = new int[2];

    UInt32 maskleft = (UInt32)(0x000000FF | (255 << (shift * 8)));
    UInt32 maskright = (UInt32)(0x00FFFFFF & (0x00FFFFFF >> (shift * 8)));
    int i = low;

    int[] subcount = new int[256];
    int subtotal = 0;

    int[] subcount2 = new int[256];
    int subtotal2 = 0;

    while (i < high)
    {
        if (i > high - added[1])
        {
            break;
        }

        if (numbers32[i] < 256)
        {
            subcount[numbers32[i]]++;
            subtotal++;

            if (i > low + added[0])
            {
                int j = low + added[0];
                Exchange32(i, j);
                added[0]++;
            }
```

```
                    else
                    {
                        added[0]++;
                        i = low + added[0];
                    }
                }
                else if (numbers32[i] > 16777215)
                {
                    subcount2[numbers32[i] >> 24]++;
                    subtotal2++;

                    int j = high - added[1];
                    Exchange32(i, j);
                    added[1]++;
                }
                else
                {
                    i++;
                }
            }

            Task[] tasks = new Task[3];
            tasks[0] = Task.Run(() =>
            {
                if (added[0] > 1) Sort32(0, added[0], shift, subcount, subtotal);
                subcount = null;
            });

            tasks[1] = Task.Run(() =>
            {
                if (((high + 1) - ((high - added[1]) + 1)) > 1) Sort32((high - added[1]) + 1, high + 1, 3,
subcount2, subtotal2);
            });

            tasks[2] = Task.Run(() =>
            {
                if ((high - added[1] + 1) - added[0] > 1) Sort32(added[0], high - added[1] + 1, 2, null,
0);
            });

            Task.WaitAll(tasks);

        }

        private static void Exchange32(int i, int j)
        {
            UInt32 temp = numbers32[i];
            numbers32[i] = numbers32[j];
            numbers32[j] = temp;
        }

        private static void Sort32(int low, int high, int shift, int[] subcount, int subtotal)
        {
            if (high - low > 10)
            {
                int[] count = new int[256];
                if (subcount != null)
                {
                    count = subcount;
                }

                int shiftbits = shift * 8;
                UInt32 mask = (UInt32)(0x000000FF << shiftbits);
                int totalcounted = subtotal;

                int[] sub2count = null;
                int sub2total = 0;

                //BUILD COUNTING FOR BOUNDARY
```

```
if (subcount == null)
{
    if (shift == 0)
    {
        for (int i = low; i < high; i++)
        {
            UInt32 countvalue = (UInt32)(numbers32[i] & mask);
            countvalue = countvalue >> shiftbits;
            count[countvalue]++;

            totalcounted++;
        }
    }
    else
    {
        sub2count = new int[256];
        int shiftbits2 = (shift - 1) * 8;
        UInt32 submask = (UInt32)(0xFFFFFFFF >> ((4 - shift) * 8));
        for (int i = low; i < high; i++)
        {
            UInt32 countvalue = (UInt32)(numbers32[i] & mask);
            countvalue = countvalue >> shiftbits;
            count[countvalue]++;

            if (countvalue == 0)
            {
                UInt32 subcountvalue = (UInt32)(numbers32[i] & submask);
                subcountvalue = (UInt32)(subcountvalue >> shiftbits2);
                sub2count[subcountvalue]++;
                sub2total++;
            }
            totalcounted++;
        }
    }
}

if (shift > 0)
{
    int[] boundaries = new int[256];
    int[] added = new int[256];

    //BUILD BOUNDARIES
    boundaries[0] = low;
    for (int i = 1; i < 256; i++)
    {
        boundaries[i] = boundaries[i - 1] + count[i - 1];
    }

    //int j = low;
    while (low < high)
    {
        UInt32 value = numbers32[low];
        UInt32 value32 = (UInt32)(((value & mask) >> shiftbits));

        if (low >= boundaries[value32] && low <= boundaries[value32] + count[value32])
        {
            //STABILITY LEAVE VALUE IN PLACE IF IT BELONGS IN THIS BOUNDARY
            if (low < boundaries[value32] + added[value32])
            {
                low = boundaries[value32] + added[value32];
            }
            else
            {
                low++;
                added[value32]++;
            }
        }
        else
        {
```

```
                    //PLACE VALUE IN CORRECT BOUNDARY
                    int k = boundaries[value32] + added[value32];
                    numbers32[low] = numbers32[k];
                    numbers32[k] = value;
                    added[value32]++;
                }
            }
            if (count[0] > 1)
            {
                //SORT EACH BOUNDARY
                Sort32(boundaries[0], boundaries[0] + count[0], shift - 1, sub2count, sub2total);
            }

            for (long i = 1; i < 256; i++)
            {
                if (count[i] > 1)
                {
                    //SORT EACH BOUNDARY
                    Sort32(boundaries[i], boundaries[i] + count[i], shift - 1, null, 0);
                }
            }
        }
        else
        {
            if (totalcounted > 1)
            {
                //USE COUNT SORT ON LSD BOUNDARIES
                mask = 0xFFFFFF00;
                int arrayPosition = low;

                for (int j = 0; j < 256; j++)
                {
                    int k = 0;

                    while (k < count[j])
                    {
                        UInt32 value = (UInt32)(numbers32[arrayPosition] & mask);
                        numbers32[arrayPosition] = (UInt32)(value + j);
                        arrayPosition++;
                        k++;
                    }
                }
            }
        }
    }
    else
    {
        InsertionSort.Sort(numbers32, low, high);
    }
 }

//UInt32 VERSION
private static UInt32[] numbers32;
public static void Sort(UInt32[] array)
{
    numbers32 = array;
    size = numbers32.Length;

    InsertionSort32();
}

private static void InsertionSort32()
{
    for (int i = 1; i < size; i++)
    {
        UInt32 key = numbers32[i];
        int j = i - 1;
        while (j >= 0)
        {
```

```
                if (key < numbers32[j])
                {
                    numbers32[j + 1] = numbers32[j];
                    j--;
                }
                else
                {
                    break;
                }
            }
            numbers32[j + 1] = key;
        }
    }
```

Co/mpare A/nd /or Counting = Co/nd/or the name just stuck.

This paper was written 3 years after original algorithm was created. Kind of late. April 26 2021

Revision: Mar 3 2021

Added Swift Comparable Generic version