Chp 2 Exercise 1

•Find the code for Merge Sort

•The algorithm lends itself well to Parallel Computing in that it allows for separate processes to work on separate parts of the data.

```
SUBROUTINE mergeSort(array)

    If len(array) > 1 Then

       # This is the point where the array is divided into two subarrays

       halfArray = len(array) / 2



       FirstHalf = array[:halfArray]

       # The first half of the data set



       SecondHalf = array[halfArray:]

       # The second half of the data set



       # Sort the two halves

       mergeSort(FirstHalf)

       mergeSort(SecondHalf)
```

```
        k = 0



        # Begin swapping values

        While i < len(FirstHalf) and j < len(SecondHalf)

           If FirstHalf[i] < SecondHalf[j] Then

             array[k] = FirstHalf[i]

             i += 1

           Else

             array[k] = SecondHalf[j]

             j += 1

             k += 1

           EndIf

        EndWhile

     EndIf



     outputList(array)

ENDSUBROUTINE



SUBROUTINE outputList()
```

```
    FOR i in len(array)

        OUTPUT array[i]

    ENDFOR

ENDSUBROUTINE




array = [6, 5, 12, 10, 9, 1]

mergeSort(array)
```

•Assume you have 8 processors that all have access to the same shared memory.

•Given how Merge Sort works how might you assign work to each processor? – There are a couple of approaches
•Assuming it takes zero extra time to setup your approach from the standard single processor approach how much faster do you think your approach is? – Later in the book we will learn specific approaches to measuring this.

Answer:

My idea is that the part of the merge sort "divide and conquer" can be done at same time but instead of 2 parts [left and right] , we can do 4 parts using pairs of core (4 pairs as total) that each of the pair would deal with the left and right of that subpart of the array before it gets put back together and since the memory is shared we can make a specific space allocated to the size of the original array.

I would assign work to pairs (4 pairs total), where each pair would behave as merge sort. Like in it we split the array into left and right then into smaller subsets. The pair cores can each individually reorder its own part at the same time which then can be merge sorted again. My idea is that if we split it into 4 parts using the 4 pairs this should makes it at least 4 times faster since we are not relaying on data dependency, which I am referring to the idea that the subsets get reorder then gets put back together. This is assuming they shared the same memory.

For a 1024-item array, the time difference is negligible due to the parallelization overhead. The speedup becomes more noticeable with larger arrays. Therefore, I will use other sizes that I think would showcase some sort of difference between my approach and the normal merge sort algorithm.

To test my theory I will do small test at 10k , 100k, and 1 million mark of array size.

Experiment outputs:

**I will provide the python files in case you wanted to look at the code or test using the same code.

```
The 10,000 mark
Original array (first 20 elements): [1269, 3457, 9599, 6405, 4511, 110,
4395, 7852, 4042, 1418, 808, 1072, 9924, 6935, 5627, 611, 4897, 9071,
1240, 1274]


Time taken by normal merge sort: 20909.0 nanoseconds
Time taken by parallel merge sort: 99119.0 nanoseconds


Sorted array (first 20 elements): [1, 1, 3, 3, 6, 6, 7, 8, 8, 8, 9, 10,
11, 12, 12, 12, 13, 13, 16, 17]
```

```
the 100,000 mark
Original array (first 20 elements): [43243, 92222, 17105, 52416, 70964,
8218, 28799, 10124, 82364, 33295, 20980, 33368, 78218, 11495, 65685,
29229, 83767, 14790, 97598, 50393]


Time taken by normal merge sort: 257291.0 nanoseconds
Time taken by parallel merge sort: 204146.0 nanoseconds


Sorted array (first 20 elements): [2, 2, 3, 4, 5, 5, 5, 6, 9, 10, 11, 13,
13, 14, 14, 16, 16, 16, 17, 18]
```

```
The 1,000,000 mark
```

```
Original array (first 20 elements): [378856, 360646, 644216, 368032,
531559, 236548, 692654, 295215, 800781, 272874, 81460, 686812, 152922,
200947, 151152, 811108, 142325, 330679, 192263, 255513]


Time taken by normal merge sort: 3109289.0 nanoseconds
Time taken by parallel merge sort: 1601020.0 nanoseconds


Sorted array (first 20 elements): [3, 4, 8, 9, 12, 15, 15, 15, 17, 17, 18,
18, 18, 19, 22, 22, 23, 27, 28, 29]
```

Conclusion:

Based on the little experiment and the hypothesis, I can conclude that this method does improve the performance of merge sort algorithm from around the 100,000 mark and above.