LP1 Assignment HPC 3

Date: 24th September, 2020

Title: Parallel Sorting Algorithms

Problem Definition:

For Bubble Sort and Merge Sort, based on existing sequential algorithms, design and implement parallel algorithm utilizing all resources available.

Learning Objectives:

- Learn Parallel Decomposition of Problem
- Learn Parallel Computing using CUDA

Learning Outcomes:

- I will be able to perform parallel computing using CUDA libraries
- I will be able to decompose a problem into sub problems, learn how to use GPUs, and solve problems using threads on GPU

S/W & H/W Packages:

1. Operating System : 64-bit Open source Linux or its derivative

2. Programming Language: C/C++

3. NVidia CUDA-enabled GPU

4. CUDA API

5. Google Colaboratory(uses Tesla K80 GPU)

Related Mathematics:

Let S be the system set:

S = {s; e; X; Y; Fme;DD;NDD; Fc; Sc}

s=start state

e=end state

X=set of inputs

X = {X1}

where

X1 = Elements of Vector

Y= Output Set (Sum or Product of elements of Vector/Matrix)

Fme is the set of main functions

Fme = {f1,f2,f3}

where

f1 = decomposition function

f2 = function to generate

f3 = function to display results

DD= Deterministic Data Vector / Matrix of Elements.

NDD=Non-deterministic data No non deterministic data

Fc =failure case: No failure case identified for this application

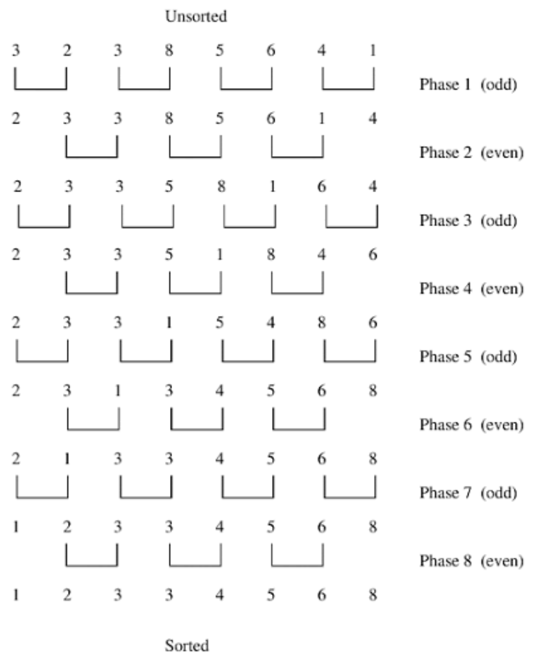
Concepts Related to Theory:

Bubble Sort: Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2) where n is the number of items. We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending. To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

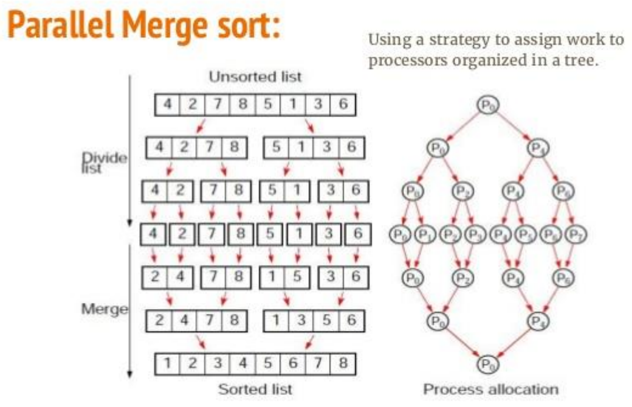
Merge Sort: Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sub lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list. The top-down merge sort approach is the methodology which uses recursion mechanism. It starts at the Top and proceeds downwards, with each recursive turn asking the same question such as “What is required to be done to sort the array?” and having the answer as, “split the array into two, make a recursive call, and merge the results.”, until one gets to the bottom of the array-tree.

Message Passing Interface: Message Passing Interface (MPI) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and application specialists (William Gropp). MPI is a specification not an implementation. The main purpose of MPI is to develop a widely used standard for writing message passing programs and achieving practical, portable, efficient and flexible standard for message passing. MPI is a mixture of functions, subroutines, and methods and also have different syntax from other languages. In parallel programming the problem will be divided into sub problems or processes and assign to different processors. The rule of MPI is to establish communication among the processes, hence MPI is used for distributed memory not for the shared memory. This means that the system works only for sending and receiving the data with their own memory.

Parallel Bubble Sort: Odd-Even Transposition Sort is based on the Bubble Sort technique. It compares two adjacent numbers and switches them, if the first number is greater than the second number to get an ascending order list. The opposite case applies for a descending order series. Odd-Even transposition sort operates in two phases − odd phase and even phase. In both the phases, processes exchange numbers with their adjacent number in the right.



Parallel Merge Sort: The idea is to take advantage of the tree structure of the algorithm to assign work to processes. If we ignore communication time, computations still only occur when merging the sub lists. But now, in the worst case, it takes 2s − 1 steps to merge all sub lists (two by two) of size s in a merging step. Since in total there are log(n) merging steps, it takes log X (n) i=1 (2 i − 1) steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of O(n)



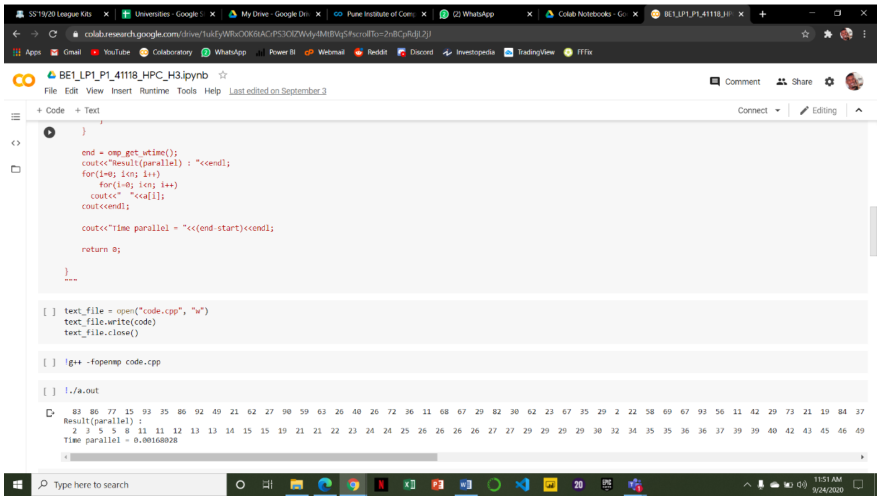
References:

1. CUDA Documentation: https://docs.nvidia.com/cuda/

Steps for Execution:

1. Open Google Colaboratory
2. Change Runtime type from CPU to GPU
3. Get nvcc plugin for Jupyter Notebook
4. Write parallel sorting algorithms.
5. Use !g++ -fopenmp filename.cpp to compile the code.
6. Use !./a.out to give output.

Output:



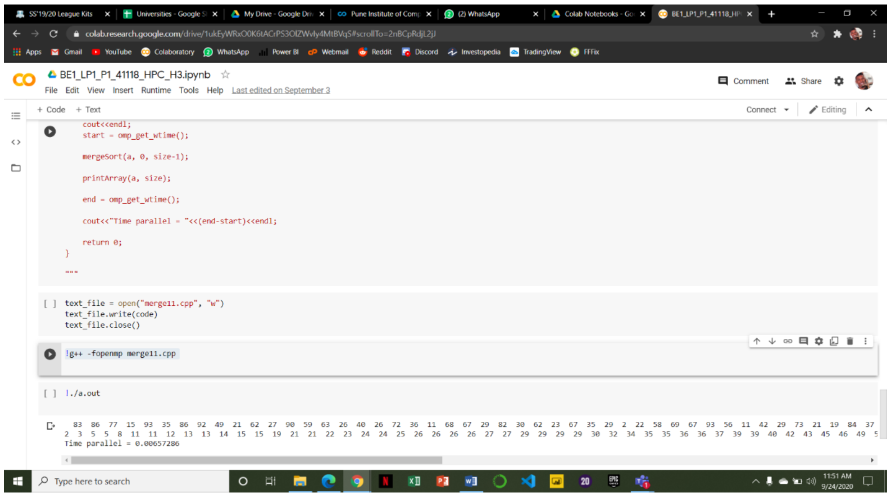
```
    }

    end = omp_get_wtime();
    cout<<"Result(parallel) : "<<endl;
    for(i=0; i<n; i++)
        for(i=0; i<n; i++)
      cout<<"  "<<a[i];
    cout<<endl;

    cout<<"Time parallel = "<<(end-start)<<endl;

    return 0;

}
"""
```

```
text_file = open("code.cpp", "w")
text_file.write(code)
text_file.close()
```

```
!g++ -fopenmp code.cpp
```

```
!./a.out
```

```
   83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36  11  68  67  29  82  30  62  23  67  35  29  2  22  58  69  67  93  56  11  42  29  73  21  19  84  37
Result(parallel) :
   2  3  5  5  8  11  11  12  13  13  14  15  15  19  21  21  22  23  24  24  25  26  26  26  26  27  27  29  29  29  29  30  32  34  35  35  36  36  37  39  39  40  42  43  45  46  49
Time parallel = 0.00168028
```



```
    cout<<endl;
    start = omp_get_wtime();

    mergeSort(a, 0, size-1);

    printArray(a, size);

    end = omp_get_wtime();

    cout<<"Time parallel = "<<(end-start)<<endl;

    return 0;
}

"""
```

```
text_file = open("merge11.cpp", "w")
text_file.write(code)
text_file.close()
```

```
!g++ -fopenmp merge11.cpp
```

```
!./a.out
```

```
   83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36  11  68  67  29  82  30  62  23  67  35  29  2  22  58  69  67  93  56  11  42  29  73  21  19  84  37
 2  3  5  5  8  11  11  12  13  13  14  15  15  19  21  21  22  23  24  24  25  26  26  26  26  27  27  29  29  29  29  30  32  34  35  35  36  36  37  39  39  40  42  43  45  46  49  5
 Time parallel = 0.00657286
```

Notebook Link:

https://colab.research.google.com/drive/1ukEyWRxO0K6tACrPS3OlZWvly4MtBVqS?usp=sharing

Conclusion:

I have successfully performed parallel sorting algorithms using openmp interface.