Group 6
ID: 2016B1A70703P Name: Mayank Jasoria
ID: 2016B4A70487P Name: Puneet Anand

Bubble Sort

Execution Requirements

The program expects a single command line argument specifying the name of the input file. The structure of the file should be such that the first line specifies the input size, and every other element must be an integer, in a new line (new line may be replaceable with a space, but that has not been tested). A few sample input files have been added as part of the submission.

Sample compilation, assuming mpich as the platform:
Either run 'make', or `mpicc bubblesort_parallel.c`
mpiexec -n <num_processors> ./a.out <intput_file>

Here, num_processors should not exceed the size of the input file.

Design

Initially, the root processes reads the input, and pads the array with infinity at the end if needed, such that the number of elements in the array becomes perfectly divisible by the number of processes. Then comes the partitioning phase, where each process receives its partition of the array for computation.

Within each process, the arrays are sorted using any sequential sorting algorithm, which in our case, is quicksort. Once all arrays belonging to all processes are sorted, odd-even transposition phases are started. In this, iterations equal to the number of processes are run, and in each iteration, either an odd-to-even compare-split operation, or an even-to-odd compare-split operation takes place among all processes which can participate. Here, compare-split is a technique in which two processes exchange their arrays, and a sorting is performed such that both processes contain sorted arrays where all elements of the process having lower rank are smaller than the elements of the other process. An odd-to-even transposition means that processes having odd ranks perform a compare-split with the immediately higher even ranked process, and an even-odd transposition is the reverse operation.

Finally, the results are collected in a sorted order into a single root process and printed to a file called "output.txt" as the output of the execution.

Analysis of the algorithm:

Assuming,
Number of processes = p
Number of elements to be sorted = n
[assuming that $p < n$, and p divides n (we can always add extra elements to make this happen)]

Step 0: Spawning of Processes - $O(log\ p)$
Step 1 : Partition of the Input into the p processes (using Scatter) - $O(n/p\ log\ p)$

Step 2: Internal sorting for each of the n/p elements chunk - $O(n/p \log n/p)$
Step 3: Alternating Odd/Even Transposition phases:

1 Compare Split operation consists of

$O(n/p)$ comparisons

$O(n/p)$ pairwise communication

Total of p Compare Split operations ($p/2$ odd and $p/2$ even)

So total complexity: $O(n)$ (for comparisons) + $O(n)$ (for communications)

$T_p = O(n/p \log n/p) + O(n) + O(n) + O(n/p \log p) + O(\log p)$

Theoretical Speedup = $Tseq \div Tp = (n \log n) \div ((n/p \log n/p) + (n) + (n) + (n/p \log p))$

Efficiency = $Speedup / No.\ of\ Processors = (n \log n) \div (p * ((n/p \log n/p) + (n) + (n) + (n/p \log p)))$

Actual Speedup estimation:

The following table has been derived for running the algorithm from 1 to 10 processes on a dataset of 10,000 elements, which has the presence of duplicate entries, on a system having four logical processors.

Number of Processes	**Execution Time (seconds)**	**Speedup (T_s/T_p)**
1	0.00234	1
2	0.00169	1.384615
3	0.001357	1.724392
4	0.001296	1.805556
5	0.160008	0.014624
6	0.253854	0.009218
7	0.313078	0.007474
8	0.29602	0.007905
9	0.612558	0.00382
10	0.57216	0.00409

Table: Execution time and Computed Speedup for different number of processes

From the table above and the two graphs below, it is evident that the highest speedup is observed when the number of processes is 4, beyond which there is a significant increase in the execution time and therefore the speedup becomes lesser than 1. This is because there is a speedup till the number of processes is not more than the number of available processors, according to the theoretical speedup. However, once the number of processes becomes more than the available processors, the speedup decreases drastically due to scheduling overheads which are primarily a result of processes having to wait for their turn.

The graphs below are plots of the execution time and observed speedup against the number of processes (execution time plotted till 5 processes for clarity). These graphs clearly depict the increase in speedup for the first 4 processes, followed by the sharp decline when number of processes is 5 or more.

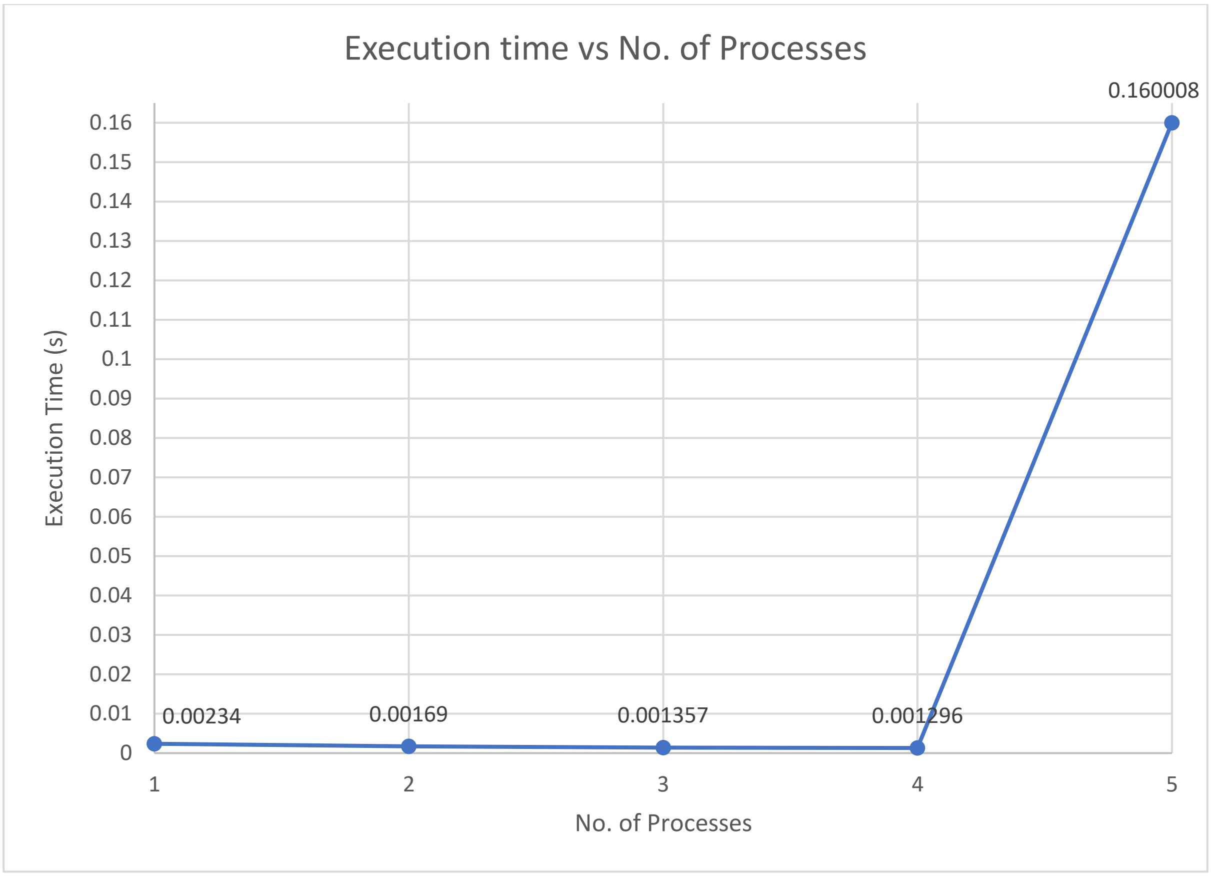

Fig 1: Graph of execution time vs no. of processes

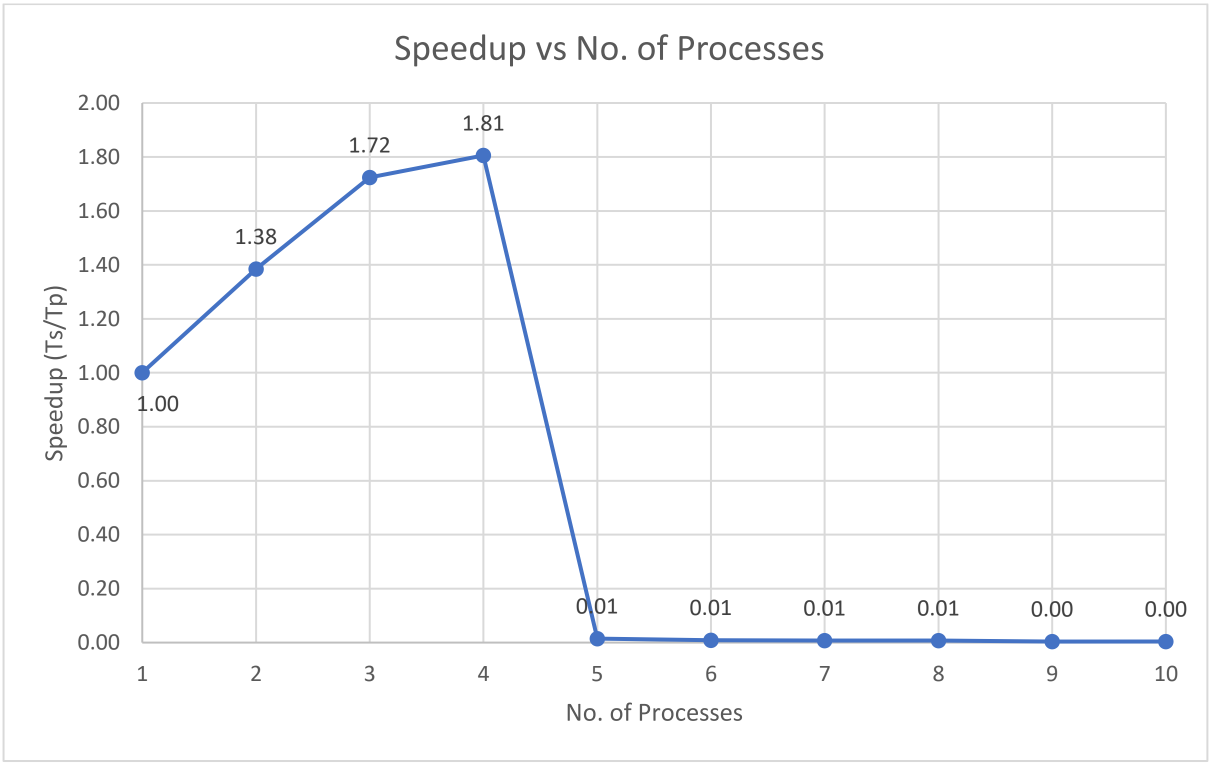

Fig 2: Graph of observed speedup vs no. of processes