REPORT

Abhay Vivek Kulkarni (ak6277)

Project 2: Bitonic Sort

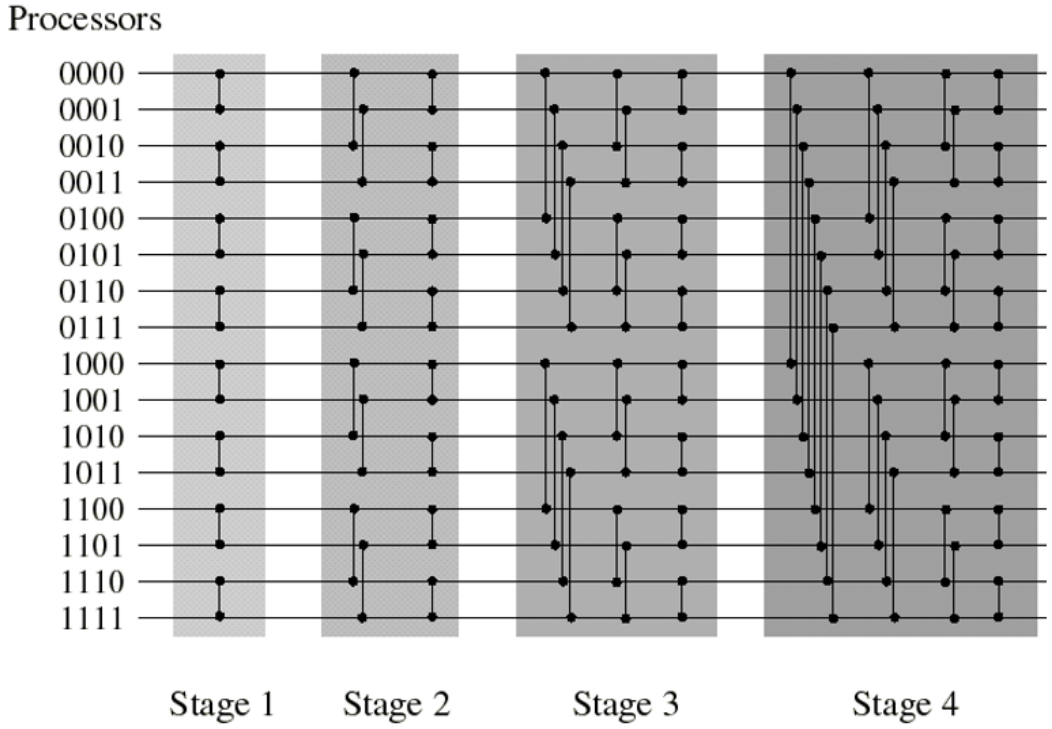
Program Workflow:

---

1) Get user input for size of raw data.
   --size-N : Generates an array of size N with random numbers
   --print-X : (optional) Prints a part of the sorted array of size X for easy readability.
   If not specified; does not print the result

Global parameters: (hardcoded)
   MASTER = Specifies the base thread
   SEED = Seed for the random value generation
   BOUND = range of random values (0 to BOUND)

2) MPI.Init()
   All processors get (N/P) elements
   Wait for all processors to get their raw data
   Stages = $\log_2 P$



3) For each stage:
Perform comparisons:
if (stage is even) and (processor's index = 0) OR
if (stage is odd) and (processor's index = 1)
Perform a min comparison
else
Perform a max comparison

```
for (i = 0; i < stages; i++) {
    for (j = i; j >= 0; j--) {
        if (((process_rank >> (i + 1)) % 2 == 0 && (process_rank >> j) % 2 == 0) ||
                ((process_rank >> (i + 1)) % 2 != 0 && (process_rank >> j) % 2 != 0)) {
            minCompare(j);
        } else {
            maxCompare(j);
        }
    }
}
```

This transforms the Raw data to Bitonic by stage 3 (in this case)
The last stage converts Bitonic sequence to sorted sequence.

4) Min-Max Comparisons:
process rank P1 compares its value with (P1 XOR (left bit shift))
swaps values based on the type of comparison

<eg> For 16 processors (Assigned 0000 to 1111)

Stage 1: 0000 - 0001	0100 - 0101
Stage 2: 0000 - 0010	0100 - 0110
Stage 3: 0000 - 0100	0100 - 0000
Stage 4: 0000 - 1000	0100 - 1100

Observations and Issues:

---

1) No significant changes can be seen in execution time for array sizes < 4M
as spawning threads in Java has an overhead which counteracts the performance gain achieved by multi-processing.

2) For <1024 sizes, regular in-built sort is extremely efficient.

3) Significant changes can be observed at array size of ~100 M.
Sequential is 7.5 times slower than parallel 8-core processing.

4) With input size >268M (16384^2)
The final master array can't accommodate that many values.
Issues with program heap/ int overflows
Assigning (long) values would add to the extra memory overhead.

Result graph: