**PROJECT 2**

**Sample Sort**

**MPI- Only implementation**

It contains below functions –

ReceiveInput : This function receives input from flat file. Structure of flat file is mentioned in README.TXT

Distribute : Distributes the input to all processors equally.(n/p)

Sort : Insertion sort implementation.

localSplitter : Finds p or p-1 splitters based on Boolean variable passed.

allToAllCalculation : It prepares scounts and sdispls for allToall communication.

countElements : It counts the number of elements in each bucket(scounts) using BINARY SEARCH ALGORITHM.

Merge : It merges the sorted sublists received from all processors at each bucket(processor).

prepareGatherv : It prepares finalCount and finalDispls for final appending of sorted lists at each node. It gives output to root node (0).

Input is processed and distributed across all processors equally.

Each processor sorts local received input using insertion sort.

P equal spaced splitters are identified and sent to root node.

Root node receives p\*p samples and sort them using insertion sort. (**MPI\_Allgather**)

It identifies p-1 splitters and broadcasts it to all nodes.

Now, every node receives p-1 splitters which are nothing but bucket limits.

Each node finds out number of element is posses in every bucket – scounts using **BINARY SEARCH ALGORITHM.**

Each node find out displacement of bucket’s first element is posses – sdispls

Each node broadcasts scounts to all nodes using allToAll communication (**MPI\_Alltoall**)

After receiving scounts from all other nodes, each i bucket(processor) fins out number of element is suppose to be receive from j processor.- rcounts

Same way it prepares rdispls which gives position of data it should receive from j processor – rdispls.

After preparing scounts, sdispls, rcounts, rdispls every node sends ith sublists to ith processor using **MPI\_Alltoallv**.

Doing this now each bucket receives all elements in that bucket.

Every bucket merges all sorted sublissts it received from other processors using **MERGE ALGORITHM**.

Now all nodes prepare finlCount and finalDispls for sending sorted list in bucket to root node.

Then root node receives all sorted lists from other nodes and append them using MPI\_GatherV.

**MPI OpenMP implementation**

The MPI side implementation of MPI OpenMP is mostly similar to MPI only implementation.

Multithreading strategy-

Multithreading is used for local sort algorithm, binary search and merge algorithms, as given in the requirements.

Multithreading in insertion sort –

The inner loop of insertion sort is multithreaded using **‘#pragma omp for schedule(dynamic)’** directive. Schedule dynamic is used to avoid temporal imbalance in static scheduling, It will split the iteration space equally into all available threads as they become idle.

Doing so, the inner loop of insertion sort is multithreaded and when iteration space is 1 for each thread it will improve complexity from O(n­2) to O(n).

Multithreading strategy for binary search –

countElements function.

Binary search complexity is log(n) in general. This binary search variant counts the number of elements smaller than target value passed. Iteration space is divided using **‘#pragma omp for schedule(dynamic)’.**

The upper limit (hi) is divided to each thread and every thread updates shared variable count pointer.

Now the complexity each improved from log(n) to log(n/t) where t is number of threads available.

Multithreading strategy for Merge algorithm –

This is a variant implementation of merge step used merge sort algorithm. Directive **‘#pragma omp for schedule(dynamic) ordered’** is used.

**Experimental results –**

Result shows the improvement in performance while increasing the number of processors (p) and increasing number of threads(t) per process. Especially for large value of n, values of p and t makes a visible difference.

MPI OpenMP implementation shows better performance than MPI significantly for large value of n and large number of processors and threads.

For small n, using large p and t will increase the distribution and thread creation. Initialization cost than actually sorting the numbers.

Therefore, MPI-OpenMP implementation will perform well for large values of n and when we have many processors and available threads per core.

Using virtual threads (in this case 8) shows improvement in performance but obviously it is not that much of implementing with same physical threads. When total number of threads per core are limited then extra threads will be in waiting state while few threads are executing.

The experimental results for fixed n and variable p and t for both implementations are shown below. These readings are taken while system peak time so may not be very accurate,

MPI Only results

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | |  | p | t | time | | n |  |  |  | | 32 | 4 | 4 | 0.032377 | |  |  | 8 | 0.032918 | |  | 8 | 4 | 0.064538 | |  |  | 8 | 0.055195 | |  |  |  |  | | 64 | 4 | 4 | 0.034973 | |  |  | 8 | 0.032614 | |  | 8 | 4 | 0.068225 | |  |  | 8 | 0.04621 | |  |  |  |  | | 256 | 4 | 4 | 0.034642 | |  |  | 8 | 0.032861 | |  | 8 | 4 | 0.049581 | |  |  | 8 | 0.044996 | |  |  |  |  | | 1024 | 4 | 4 | 0.039343 | |  |  | 8 | 0.042861 | |  | 8 | 4 | 0.082215 | |  |  | 8 | 0.078854 | |  |  |  |  | | 4096 | 4 | 4 | 0.068606 | |  |  | 8 | 0.069836 | |  | 8 | 4 | 0.085525 | |  |  | 8 | 0.069398 | |  |  |  |  | | 16384 | 4 | 4 | 0.471979 | |  |  | 8 | 0.365828 | |  | 8 | 4 | 0.233805 | |  |  | 8 | 0.21289 |   MPI OpenMP results –   |  |  |  |  | | --- | --- | --- | --- | |  | p | t | time | | n |  |  |  | | 32 | 4 | 4 | 1.941969 | |  |  | 8 | 0.054548 | |  | 8 | 4 | 3.60655 | |  |  | 8 | 0.088746 | |  |  |  |  | | 64 | 4 | 4 | 1.469879 | |  |  | 8 | 0.059683 | |  | 8 | 4 | 4.234838 | |  |  | 8 | 0.071572 | |  |  |  |  | | 256 | 4 | 4 | 3.617837 | |  |  | 8 | 0.03237 | |  | 8 | 4 | 4.121898 | |  |  | 8 | 0.005382 | |  |  |  |  | | 1024 | 4 | 4 | 3.656736 | |  |  | 8 | 0.034353 | |  | 8 | 4 | 3.48958 | |  |  | 8 | 0.020977 | |  |  |  |  | | 4096 | 4 | 4 | 25.7656 | |  |  | 8 | 0.33271 | |  | 8 | 4 | 0.03456 | |  |  | 8 | 0.027812 | |  |  |  |  | | 16384 | 4 | 4 | 0.271979 | |  |  | 8 | 0.048326 | |  | 8 | 4 | 0.140345 | |  |  | 8 | 0.197949 | |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Performance bottleneck –**

Performance bottleneck is observed for MPI- OpenMP implementation for small n with p=4 and t=4. Moreover it was observed for n=256 and n=4096 for p=4, t=4.

**Program execution -**

Output of program executions for n=64, p=4,t=4 is attached in submission.

Output\_64\_4\_4\_MPI\_Only.txt : Output of MPI only implementation

Output\_64\_4\_4\_MPI\_OprnMP.txt : Output of MPI OPenMP implementation

Numbers.TXT : input file.

Compilation instructions are included in README.TXT. Input file structure is also mentioned in readme.txt