Assignment
Sorting Small Integers with MPI

Akash Sharma
UBIT-as475
as475@buffalo.edu
Department of Computer Science and Engineering
University at Buffalo
Buffalo, NY 14214

1 November 2021

1 Introduction

This assignment is to implement and analyze efficient MPI program for sorting small integers.

2 Code used in sorting integer MPI

```
include <iostream>
include <omp.h>
include <vector>
include <math.h>
include <algorithm>
using namespace std;

    void isort(std::vector<short int> Xi, MPI_Commcomm)
    int rank = 0, size = 0;
int m = Xi.size();
MPI_Comm_rank(comm, rank);
MPI_Comm_size(comm, size);

    std::sort(std::begin(Xi), std::end(Xi));
int p = size;
int x = -(p - 1);
int z = 0;
vector<int> buf(2*(p - 1)+1, 0);
```

for (int j = x; j <= p - 1; j++)

//cout « j«’’;
for (int i = 0; i < m; i++)

if (Xi[i] == j)

buf[z]++;

z++;

vector<int>sum$_c ount(2 * (p - 1) + 1, 0)$;

int r$_s ize = 2 * (p - 1) + 1$;

MPI_Reduce(buf[0], sum_count[0], r_size, MPI_INT, MPI_SUM, 0 , comm);

//Broadcasting Frequecy of elements
MPI_Bcast(sum$_c ount[0], 2 * (p - 1) + 1, MPI_INT, 0, comm)$;

int l = 0;
int y = 0;
int j,k;
vector<short int> sort;
for (int i = 0; i < size - 1; i++)

if (rank == i)

int c1 = sum$_c ount[i]$;
$intt = sum_c ount[i]$;
$for(j = 0; j < t; j + +)$

if (c1 == 0)
break;
//cout « i - (p - 1);
sort.push$_b ack(i - (p - 1))$;
$c1 - -$;

Xi = sort;

if (rank = size - 1)

for (int i = size - 1; i < 2 * size - 1; i++)
// int c1 = sum$_c ount[i]$;
$intt = sum_c ount[i]$;
$for(j = 0; j < t; j + +)$

if (c1 == 0)
break;
//cout « i - (p - 1);
sort.push$_b ack(i - (p - 1))$;
$//cout << "s : " << sort[i] << "\n"; c1 - -$;
Xi = sort;

3 Code run for different threads and row(n)and column(m) value

3.1 Time in code run -1

p\n	10k	20k	40k	80k	100k	500k	1000k
2	0.000203397	0.00031966	0.000405	0.001001	0.001473	0.006426	0.011162
4	0.0226284	0.0226041	0.019993	0.021077	0.022426	0.021674	0.022951
8	0.0395235	0.0395235	0.045208	0.045208	0.037074	0.036578	0.038263
16	0.0691523	0.0693193	0.067567	0.070414	0.062172	0.061146	0.06127
2	0.000160299	0.00028243	0.000479	0.000896	0.001322	0.005793	0.010672
4	0.0205235	0.0208517	0.020352	0.021597	0.020635	0.02265	0.024582
8	0.0205235	0.0208517	0.020352	0.021597	0.037161	0.033584	0.0373
16	0.0616378	0.0594366	0.063499	0.059305	0.063199	0.06459	0.069623
2	0.000182779	0.00027327	0.000534	0.00101	0.001311	0.005805	0.009386
4	0.0209921	0.0200199	0.019963	0.020225	0.02099	0.021314	0.024851
8	0.0209921	0.0200199	0.019963	0.020225	0.036732	0.036356	0.035253
16	0.0616982	0.0628688	0.065572	0.059472	0.064677	0.064712	0.06569

Figure 1: Code run 3 time to get the values

3.2 Average Time of all 3 code run

p\n	10k	20k	40k	80k	100k	500k	1000k
2	0.000182	0.000292	0.000473	0.000969	0.001368	0.00600784	0.01040639
4	0.021381	0.021159	0.020102	0.020966	0.021351	0.0218795	0.02412773
8	0.027013	0.026798	0.028507	0.02901	0.036989	0.035505967	0.0369386
16	0.064163	0.063875	0.065546	0.063063	0.063349	0.063482367	0.0655274

Figure 2: Average Time

4 SpeedUp for increasing Processors

p\n	10k	20k	40k	80k	100k	500k	1000k
2	1	1	1	1	1	1	1
4	0.00852	0.01379	0.023511	0.046205	0.064096	0.274587628	0.43130395
8	0.006743	0.010888	0.016579	0.033393	0.036998	0.16920649	0.2817212
16	0.002839	0.004568	0.007211	0.015361	0.021602	0.094637934	0.1588097

Figure 3: Speedup

4.1 Efficiency for increasing Processors

p\n	10k	20k	40k	80k	100k	500k	1000k
2	0.5	0.5	0.5	0.5	0.5	0.5	0.5
4	0.001065	0.001724	0.002939	0.005776	0.008012	0.034323453	0.05391299
8	0.000843	0.001361	0.002072	0.004174	0.004625	0.021150811	0.03521515
16	0.000355	0.000571	0.000901	0.00192	0.0027	0.011829742	0.01985121

Figure 4: Efficiency

5 Graph

5.1 Efficiency

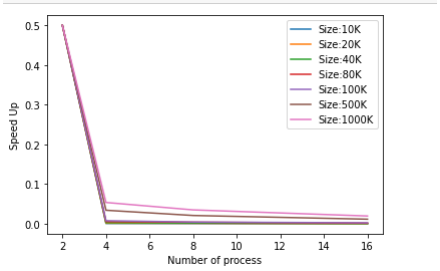

Figure 5: Efficiency

5.2 Speed Up

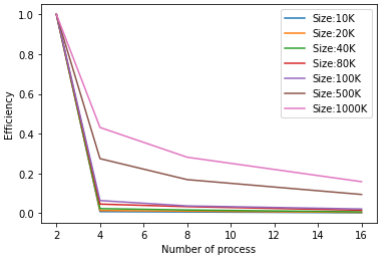

Figure 6: Speed Up

6 Result

By observing table and graph, Algorithm is strong scaling property with overheads due to communication cost, as increases in threads there is no significant decrease in time in compare to increase number of threads.