# Introduction to Parallel and Distributed Processing

## Assignment 2

Author : Kavitha Elizebeth Varughese
UBID : 50560694

---

## Overview

In this assignment, we design and implement and efficient MPI program for a fast application of sorting small integers. We will benchmark our solution to understand the scalability and performance of our parallel model over a sequential model.

## Problem Description

Given an array of $n$ small integers $X=[x_0, x_1, \dots, x_{n-1}]$ that is distributed among $p << n$ processors. Sort the array and distribute them among the processors such that each processors have none or a part of the array. We must ensure that $X_{li:lj}$ and $X_{ki:kj}$ distributed to processors with ranks l and k should be such that $X_{li:lj} < X_{ki:kj}$ if $l < k$.

For this problem, small integers are defined as 16-bit integers.

## Proposed Solution

Our goal is to sort the small integers efficiently without a large all-to-all exchange of data. As we are dealing with small integers ( there are only 65536 values), we will be performing a counting sort.

**Streamlined Operations for Parallelization**

1. In each processor, initialize an counting array of size 65536. These will be our buckets for our counting sort. In this array, we will count the number of occurrences of each value in $X$.

   We must note that the values in $X$ can be negative values as well. That's why, when we are incrementing the values in our counting array, we must add an OFFSET of 32768 to each value in $X$ and increment for this index in our counting array. *[APPENDIX 1.1]*

2. Once we have the counting array of each processor ready, we can reduce the counting array across all the processors such that we have the cumulative count in processor 0. Now that we have our buckets full of all the values in $X$. We can broadcast the counting array to every processor for redistributing the data. We will be performing this operation using MPI constructs. *[APPENDIX 1.2]*

3. Finally, to redistribute the values, we will divide 65536 to $p$ processors and ask each processor to expand their portion of the counting array and store the values in their local $X$.

   Here, the values which are redistributed need not be uniformly distributed across the processors. It is important to resize the local array $X$ to accommodate their portion of data. *[APPENDIX 1.3]*

## Time Complexity

### Sequential

We will only have 1 processor for a sequential model. The processor will have to go through all the values in X and then all the values in the counting array to reorder them . So the processor will visit every element exactly twice.

Therefore, the time complexity will be ***O(n)***.

### Parallel

All processors are only dealing with ***n/p*** values at a time. Reduce and broadcast will take ***O(log p)*** .
Thus, the overall time complexity should be ***O(n/p) + log(p)***

## Performance Analysis

### System Specification

We run the code across different nodes. The specifications of the nodes are as follows

Hosts used are in the cpn-f10-XX.core.ccr.buffalo.edu set.

| | |
|---|---|
| Number of Cores | 24 |
| CPU | Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz |
| Operating System | Ubuntu 24.04.1 LTS |
| Compiler | g++ (Gentoo 10.4.0 p5) 10.4.0 |
| L1d cache | 768 KiB |

### Input Specification

The code has been run on a single processor to record sequential time, followed by 8 nodes with 2/4/8/16 cores for 16/32/64/128 cores across nodes.

The code has been tested with a wide range of value of ***n*** to get a complete understanding of memory issues when it comes to parallel processing. The values are also selected to keep the ratio of ***n/p*** constant.

### Results

| N\P | 1 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| **10000000** | 0.0152988 | 0.0158068 | 0.00835738 | 0.00516979 | 0.00375179 |
| **160000000** | 0.225135 | 0.251104 | 0.132117 | 0.0658466 | 0.0360321 |
| **320000000** | 0.448639 | 0.499836 | 0.263452 | 0.129822 | 0.0658156 |
| **640000000** | 0.895514 | 0.998953 | 0.519678 | 0.259313 | 0.135098 |
| **1280000000** | 1.79503 | 1.99522 | 1.04248 | 0.509849 | 0.271325 |

| N\P | 1 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| 5000 | 0.000561981 | 0.00121498 | 0.00108987 | 0.00149408 | 0.00168185 |
| 80000 | 0.000681464 | 0.00128306 | 0.00124299 | 0.00169662 | 0.00175185 |
| 160000 | 0.000780402 | 0.00194903 | 0.00129435 | 0.00162687 | 0.00172206 |
| 320000 | 0.00105284 | 0.00171584 | 0.00148921 | 0.00172987 | 0.00173508 |
| 640000 | 0.0014159 | 0.00204221 | 0.00166504 | 0.00231145 | 0.00178473 |

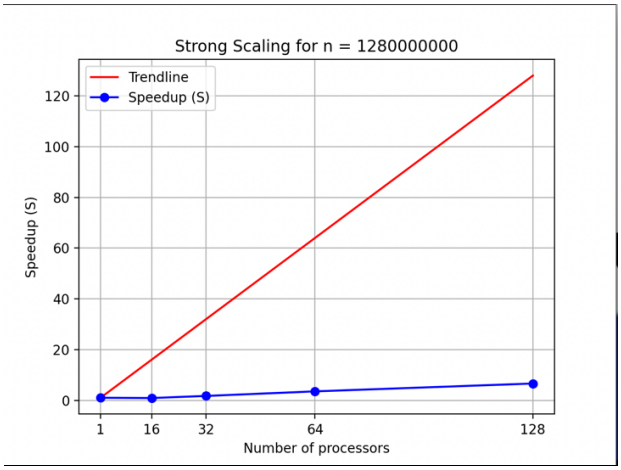
## Strong Scaling Analysis

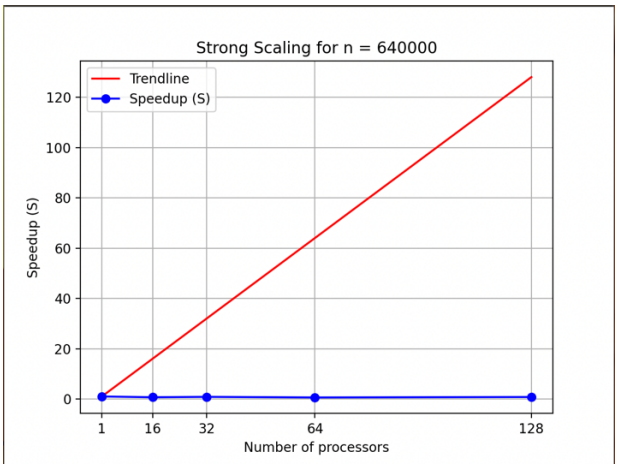
n = 12800000000

| P | Speedup | Efficiency % |
|---|---|---|
| 1 | 1.0 | 100.0 |
| 16 | 0.8996651998275880 | 5.622907498922420 |
| 32 | 1.7218843526974100 | 5.380888602179420 |
| 64 | 3.5207090726862300 | 5.501107926072230 |
| 128 | 6.615792868331340 | 5.168588178383860 |

n = 640000

| P | Speedup | Efficiency % |
|---|---|---|
| 1 | 1.0 | 100.0 |
| 16 | 0.6933175334564030 | 4.333234584102520 |
| 32 | 0.8503699610820160 | 2.6574061283813 |
| 64 | 0.6125592160764890 | 0.9571237751195140 |
| 128 | 0.7933412897188930 | 0.6197978825928850 |



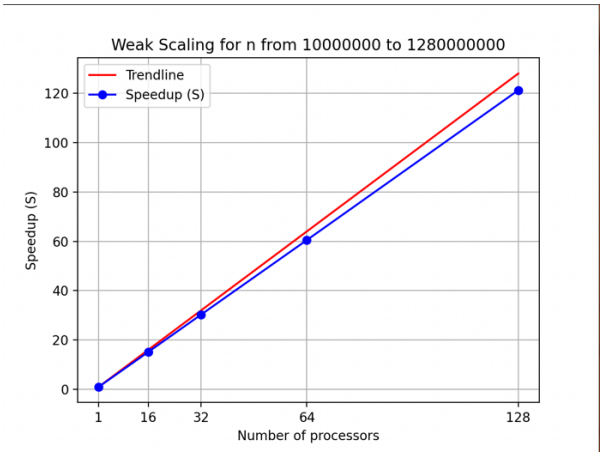

## Weak Scaling Analysis

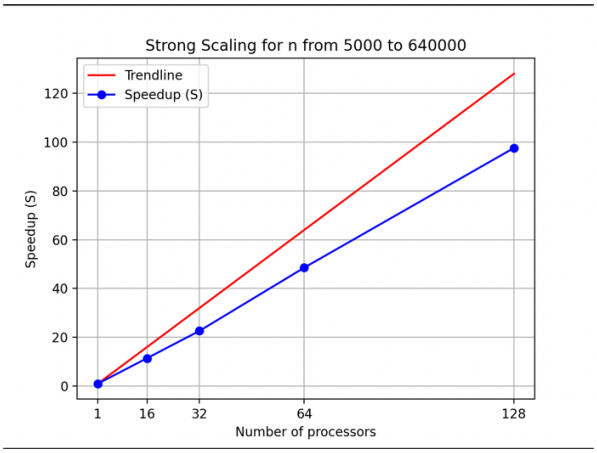
n from 10000000 to 1280000000

| P | Speedup | Efficiency % |
|---|---|---|
| 1 | 1.0 | 100.0 |
| 16 | 15.138590134938500 | 94.61618834336580 |
| 32 | 30.298613672140100 | 94.68316772543790 |
| 64 | 60.49022948030640 | 94.51598356297870 |
| 128 | 121.22126215617800 | 94.70411105951430 |

n from 5000 to 640000

| P | Speedup | Efficiency % |
|---|---|---|
| 1 | 1.0 | 100.0 |
| 16 | 11.431150310480900 | 71.44468944050570 |
| 32 | 22.615137602076400 | 70.67230500648860 |
| 64 | 48.55187400926150 | 75.86230313947110 |
| 128 | 97.58654602121860 | 76.23948907907710 |




**Observation**

The problem is not strongly scalable. This is because of the resizing of memory space in each process when redistributing the data. It is evident that there will exists at least one processor without any values in its local X. Thus, there will be processors who would need to allocate more memory to be able to accommodate the redistributed values. This extra need for memory highly effects the efficiency of our solution.

On the other hand, we can see that the problem is weakly scalable. We also see that the problem doesn't perform well for lower values of n. For higher values of n, the computation cost is probably high enough to overlook the memory costs. But, for lower values, the memory costs outshine computation costs, resulting in lesser efficiency.

With this problem, we can clearly understand the problems of memory allocations in parallel processing.

## References

1. https://www.researchgate.net/publication/331221666_Parallel_Counting_Sort_A_Modified_of_Counting_Sort_Algorithm
2. https://docs.ccr.buffalo.edu/en/latest/

## Appendix

1.1. Finding the count of every value of X locally in every processor.

```
// Each processor, count the occurrences of every number and save it in their local_count
std::vector<int> local_count(RANGE, 0);

for (const auto &x : Xi) {
    local_count[x + OFFSET]++;
}
```

1.2. Performing MPI Reduce and Broadcast to compute the global count of values in X across all processors.

```cpp
std::vector<int> global_count(RANGE, 0);

// Combine all the local counts and find the global min and max
MPI_Reduce(local_count.data(), global_count.data(), RANGE, MPI_INT, MPI_SUM, 0, comm);

// Send global count to all processors as well as the global min and max
MPI_Bcast(global_count.data(), RANGE, MPI_INT, 0, comm);
```

### 1.3. Redistribution of data within every processor.

```cpp
/*
    Divide global count vector equally to each processor between the min and max values
    Let each processor convert the counts back to the values and build Xi locally
*/

int start = rank * (RANGE / size);
int end = (rank + 1) * (RANGE / size);
int loc_n = std::accumulate(global_count.begin() + start, global_count.begin() + end, 0);

Xi.resize(loc_n);

// Convert global counts to Xi
int index = 0;
for (int i = start; i < end; i++ ) {
    while (global_count[i]-- > 0) {
        Xi[index++] = i - OFFSET;
    }
}
```