CSC4005 Project 3

ID:120090712

How to execute the program:

Depress the zip file, the depressed folder is "120090712". Upload into the cluster. Open 120090712/src/sbatch.sh and change the following settings:

```
CURRENT_DIR="/path/to/120090712/src"
```

Then, in the cluster, use following commands to complie and subbmit the tasks:

```
cd /path/to/120090712
mkdir build && cd build
cmake ..
make -j4
cd ..
sbatch ./src/sbatch.sh
```

Then, waite for the results, the results will be in the file "Project3_results.txt" under the "path/to/120090712" directory.

How did I design and implement each parallel sorting algorithm:

Task 1 - QuckSort using MPI

```
Here, in my code, I first divide the array into several fragments and each process
will deal with one fragment.

Then each process needs to do quciksort for thier small arrays. After every
process finish their task, I get several sorted arrays and the next step for the
code to do is to merge them together.

To merge the sorted arrays together, I implemented a "k_merge" algorithm using a
min-heap. (more detials will be talked about later)

The "result" vector is what I want.
```

Task 2 - BucketSort using MPI

Here, in my code, I first create some vectors(according to the bucket number) to represent buckets. Then assign the buckets to each process(with order).

Then, every process will go through the vector and implement the bucketsort independently. After they finish their job, they will combine the buckets. As the size of their combined vector is dynamic and MASTER process doesn't know the size, the processes need to send the sizes to the MSATER process first and then send the final small vector to the MASTER process.

The MASTER process will combine the small vector from the slave process and I get what I want.

Task 3 - OddEvenSort using MPI

Here, in my code, I first divide the array into several fragments and each process will deal with one fragment.

Then each process needs to do OddEven-sort for thier small arrays. What's more, I design a way to implement the communication between the nearby process.(more detials will be talked about later)

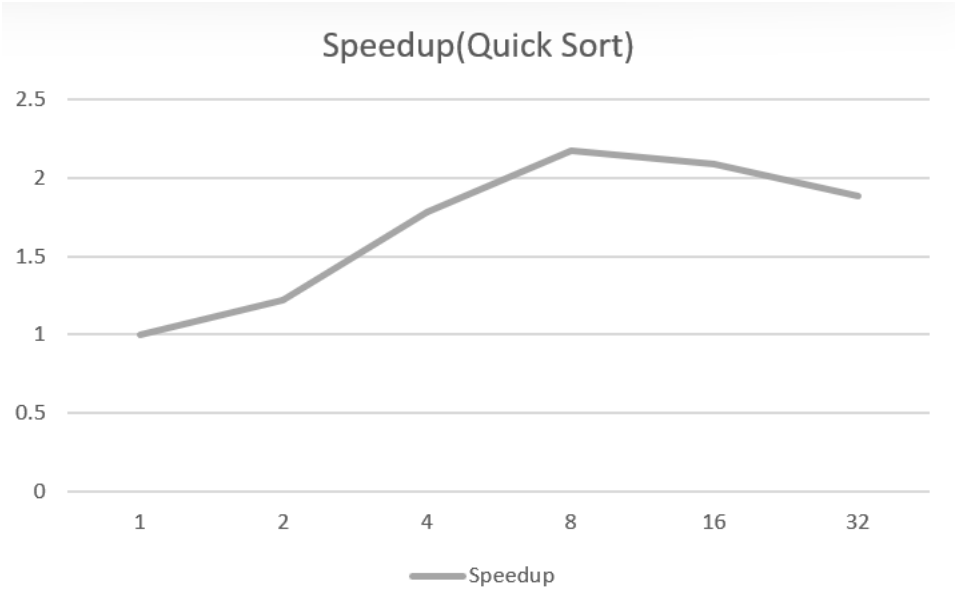
Every time when the processes finish one phase, the slave processes will send the infomation(whether thiis process finish its sorting) to the MASTER process and the MASTER process will determine whether all the sortings are finished and tell all the slave processes.

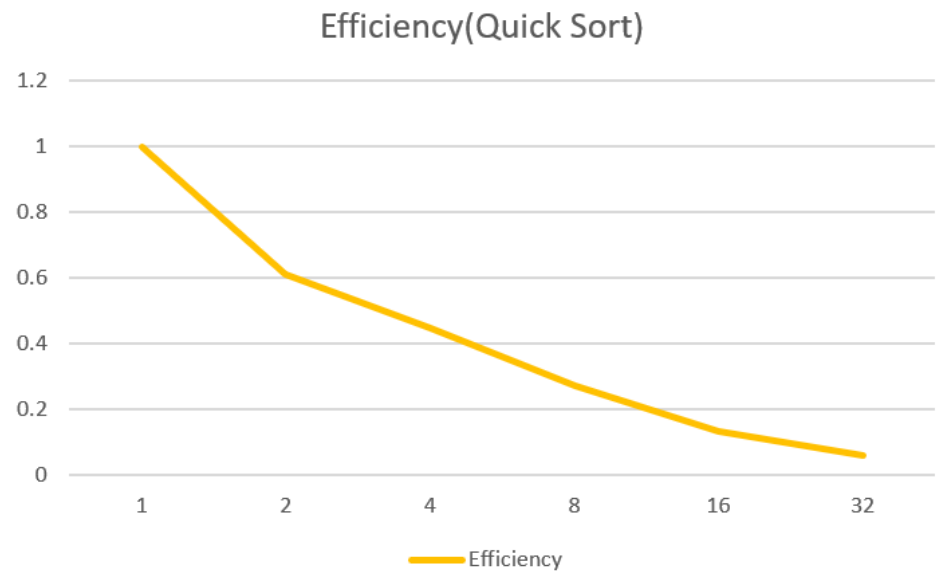
After the sorting finish, we get what we want.

Experiment results and numerical analysis

Task 1 - QuckSort using MPI

Workers	Time	Speedup	Efficiency
1	13984	1	1
2	11421	1.224411172	0.612205586
4	7851	1.781174373	0.445293593
8	6441	2.171091445	0.271386431
16	6682	2.092786591	0.130799162
32	7425	1.883367003	0.058855219





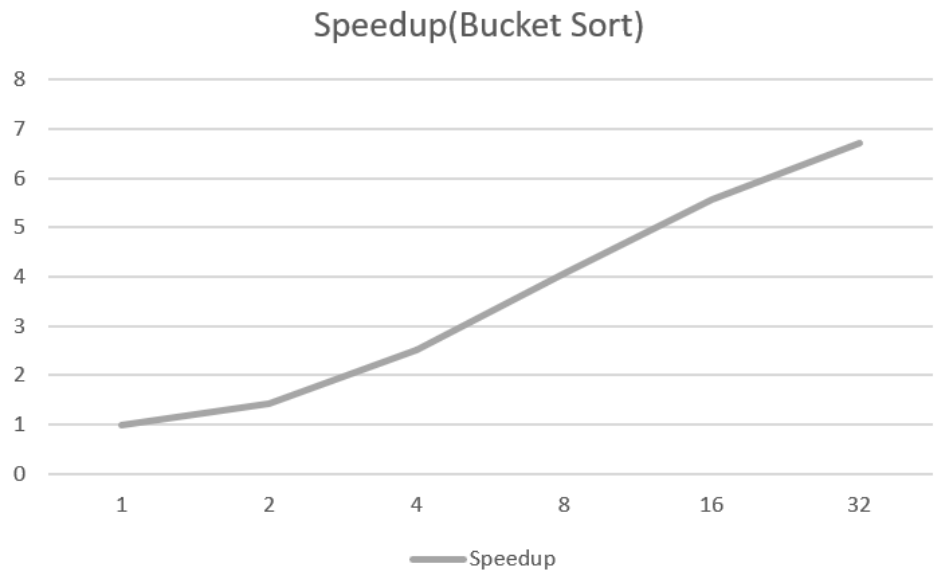
Change in Performance with Different Number of Workers

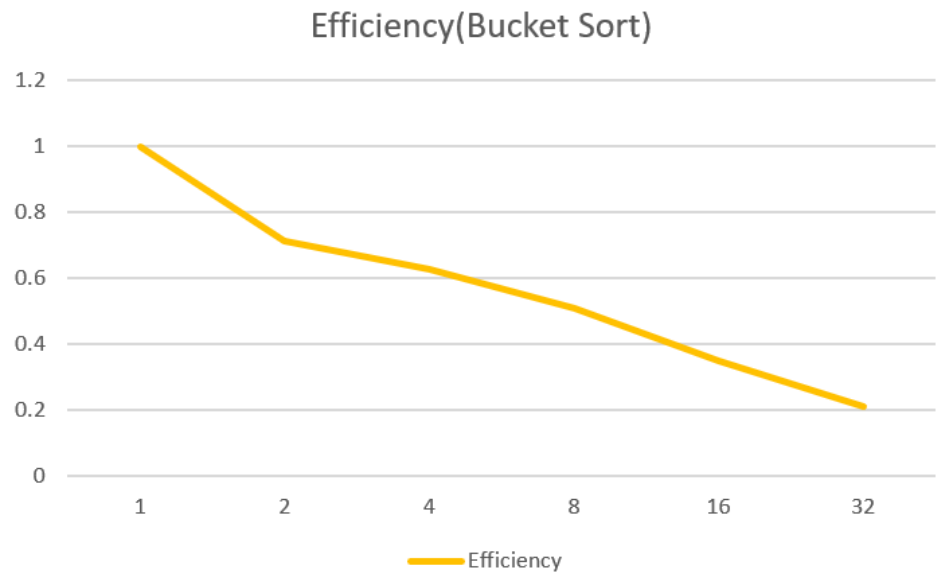
```
For task 1, the speedup increases and then decreases while the number of workers
increases. From my points of view, the reason is because the "k_merge sort" phase.
On the one hand, the smaller fragments means shorter quick sort time. the large
number of fragments means the longer "k_merge" time. The efficiency keep
decreasing because of the communication costs.
```

Task 2 - BucketSort using MPI

Workers	Time	Speedup	Efficiency
1	10563	1	1
2	7425	1.422626263	0.711313131
4	4206	2.511412268	0.627853067

Workers	Time	Speedup	Efficiency
8	2600	4.062692308	0.507836538
16	1902	5.55362776	0.347101735
32	1576	6.702411168	0.209450349



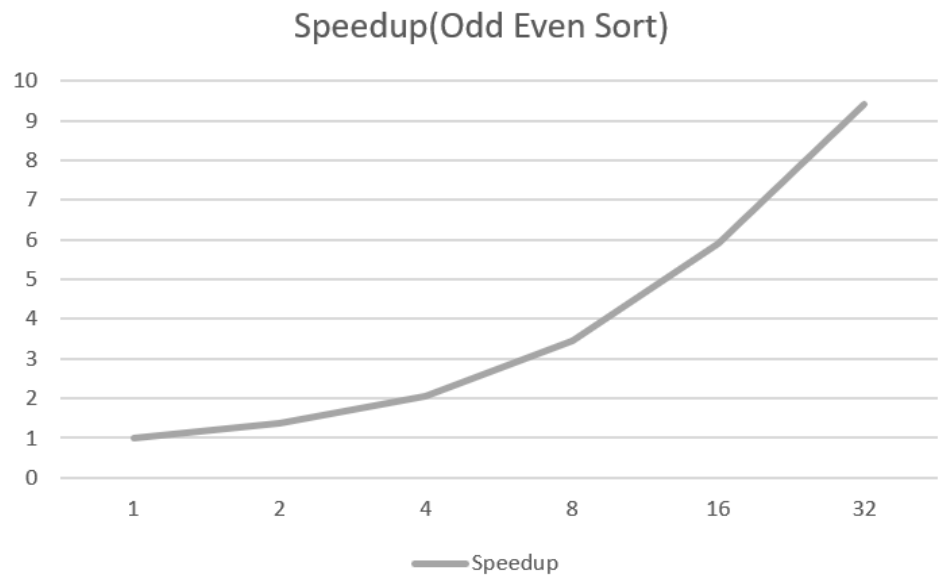


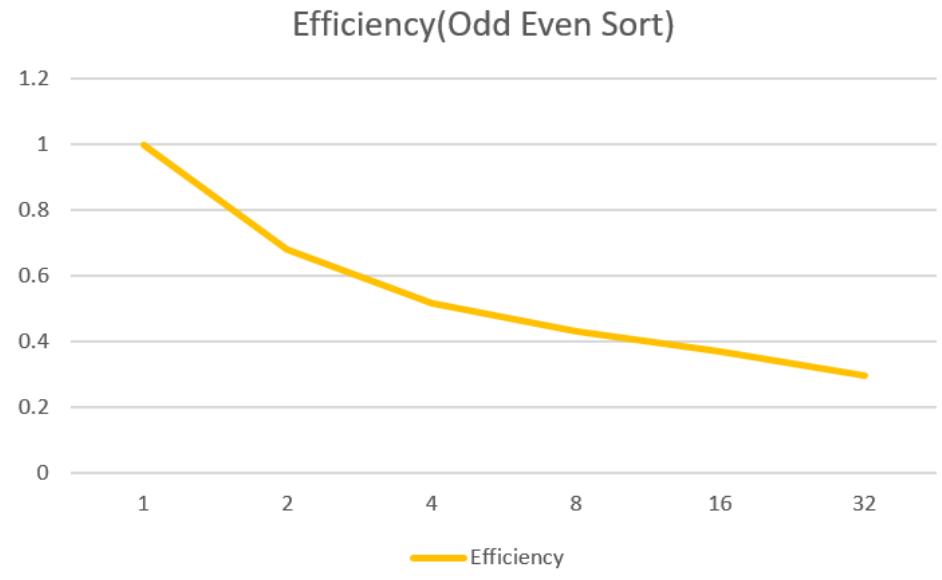
Change in Performance with Different Number of Workers

```
For task 2, the speedup keeps increasing while the number of workers increases.
But the speed up is smaller than the oddEven sort. From my points of view, the
reason is because each process needs to go through all the element in the
vector(Other two algorithm will cut the vector at first). So, the improvement of
using more process is smaller. The efficiency keep decreasing because of the
communication costs.
```

Task 3 - OddEvenSort using MPI

Workers	Time	Speedup	Efficiency
1	38003	1	1
2	27922	1.361041473	0.680520736
4	18356	2.070331227	0.517582807
8	11039	3.442612555	0.430326569
16	6424	5.915784558	0.369736535
32	4044	9.397378833	0.293668089





Change in Performance with Different Number of Workers

```
For task 3, the speedup keeps increasing while the number of workers increases.
The main burden of parallel sorting time is the communication cost. But in my
implement, the communication is limited. But the efficiency will still keep
decreasing because of the communication costs.
```

Kinds of optimizations I have tried

COMPELLING

```
I use O3 instead of O3, O3 aims to improve the efficiency of parallel computing
compared to O2. It achieves this by applying advanced optimization techniques
specifically targeted at parallel code. These optimizations include automatic
vectorization, loop unrolling, and software pipelining. Loop unrolling is a
technique where the loop iterations are reduced by executing multiple iterations
in a single loop iteration. This reduces the overhead of loop control and improves
the efficiency of parallel execution
```

Task 1 - QuckSort using MPI

```
As I have mentioned, if I directly choose the smallest data from the k vectors,
the time comlexity will be O(kn). So, I implement a "k_merge" algorithm to combine
the vectors together in an efficient way.

The basic steps of the "k_merge" algorithm are:
1. Pick the smallest(first) element in all the k vectors and form a k-size min-
heap.
2. Create a new empty vector "result" to store the result.
3. Pop the smallest(first) element in the min-heap to the "result" and check which
vector it from and if that vector is not empty, put the next element into the min-
heap.
4. Maintain the min-heap.
5. Repeat the 3~4 steps until the min-heap is empty.

Its time complexity will be "O(k) + O((n-k)lgk) + O(klgk) = O(nlgk)".
```

Task 2 - BucketSort using MPI

There are two parts of optimization I have used:

```
Part 1. Decrease the times of calculations:
    In my original design, for every element in the vectors, the code will
calculate its corresponding index of the bucket. For each process, if the bucket
index is in its bucket list, it will store this element.
    However, I changed my algorithm. For each process, it will first calculate the
codomain of its buckets and for the element in the vector, if the value is not in
the codomian, it will directly return. By this way, the calculation and comparison
times will dereasr a lot and save about 30% time.
```

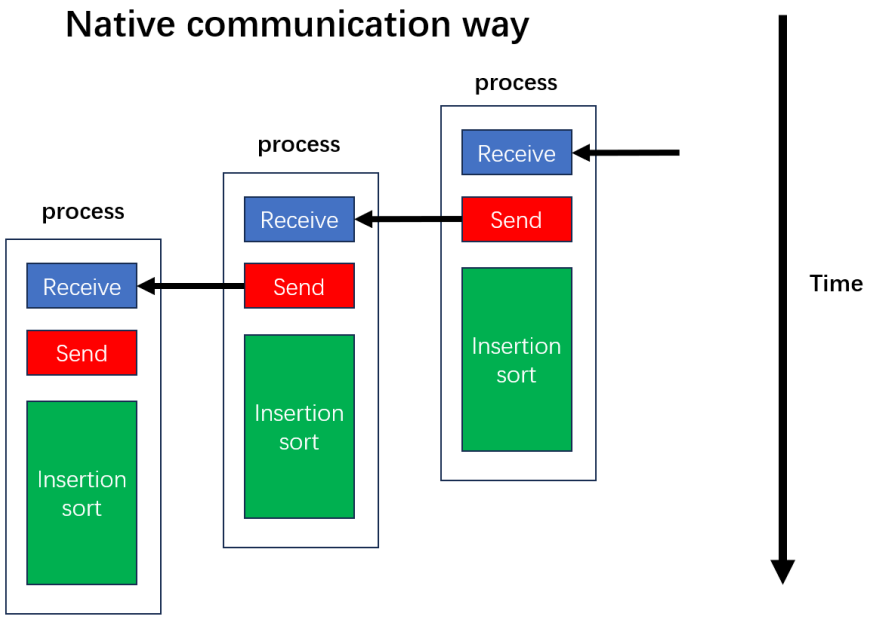
```
Part 2. The optimal buckets number:
    As the sorting algorithm used in the buckets is insertion sort(time complexity
is O(n^2)), if we denote the size of the array is S, the number of the buckets is
M, the average bucket size is N, sorting time is t, we have:
    S = M*N, t = M*(N^2) -> t = (S^2)/M
    So, typically if the number of the buckets is larger, the time is smaller. But
considering about the burden of excessive space operation, I tried many times to
find the optimal bumber of buckets and decided to use 200000(The array size is
100000000).
```

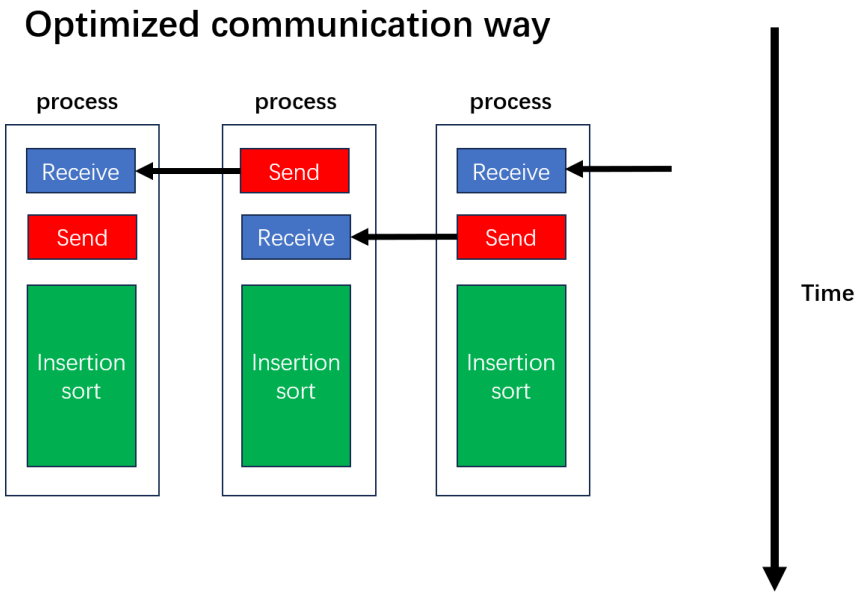
Task 3 - OddEvenSort using MPI

There are two parts of optimization I have used:

```
Part 1. Decrease the times of communications:
    Here, in every phase, the process will use the index of the start element to
determine which phease it should do (odd or even). Then, it will use the size of
its fragment to determine whether it need the data from other process to do the
boudary swap. For example, an odd-size fragment in even phase will need a element
from the next fragment(if exists) to give it its first element. What's more, Only
the fragment in odd phase needs to send its first element to the previous
fragment(if exists). By this way, the times of communications will decrease.

Part 2. Decrease the waiting time:
    I changed he order of the "send" and "receive" to save the waiting time, here
is the description:
```





Interesting discoveries I found during the experiment

```
During this experiment using MPI, I found that the core differencies of these
algorithms is how to divide the vector into smaller fragments and how they
"combine" the fragments together.
For "divide":
    Task 1 and Task 3 divide the vector according to the index of the element.
However, Task 2 divides the vector according to the value of the elements.
For "combine":
    Task 1 and Task 2 do the "combination" after all the sorting finished, Task 3
do the "combination" during the phases.
```

Profiling Results & Analysis with perf

Some perf results:

	Workers	cpu-cycles	cache-misses	page-faults
QuickSort(MPI)	squential	81,564,122,059	134,477,703	5,931
QuickSort(MPI)	1	79,548,001,051	154,908,034	12,773
QuickSort(MPI)	2	82,847,438,161	41,356,052	4,698
QuickSort(MPI)	4	72,225,887,458	25,304,762	3,665
QuickSort(MPI)	8	68,741,290,537	15,839,748	3,669
QuickSort(MPI)	16	69,170,690,221	13,571,018	3,672
QuickSort(MPI)	32	72,096,041,712	11,546,019	3,451
BucketSort(MPI)	squential	73,332,632,824	298,761,316	336,053

	Workers	cpu-cycles	cache-misses	page-faults
BucketSort(MPI)	1	67,995,166,056	338,831,653	291,947
BucketSort(MPI)	2	71,127,354,229	150,834,795	147,591
BucketSort(MPI)	4	62,432,785,762	86,808,625	78,646
BucketSort(MPI)	8	57,960,053,437	55,967,145	42,618
BucketSort(MPI)	16	56,125,532,494	42,172,560	25,111
BucketSort(MPI)	32	56,474,625,582	36,340,380	15,048
Odd-Even-Sort(MPI)	squential	119,634,874,275	114,568	2,114
Odd-Even-Sort(MPI)	1	111,078,697,746	190,477	3,762
Odd-Even-Sort(MPI)	2	80,641,147,874	77,253	3,559
Odd-Even-Sort(MPI)	4	52,820,156,059	2,922,903	2,087
Odd-Even-Sort(MPI)	8	31,635,396,982	4,430,671	1,951
Odd-Even-Sort(MPI)	16	18,171,196,153	3,877,266	1,877
Odd-Even-Sort(MPI)	32	11,551,814,759	3,140,505	1,889

Analysis:

```
Compared to linear computing, as the number of processes increases, the CPU
cycles, cache misses, and page faults of each process generally show a downward
trend. For task 1 and Task 2, the cpu-cycles decreases in slower speed than tasks.
I think the reason is the "k_merge" in Task 1 and "Operations of going through all
the element". The cache-misses in Task 3 increase a lot when the number of process
is large. It is probably because the more frequent sending and receiving data from
other process.
```